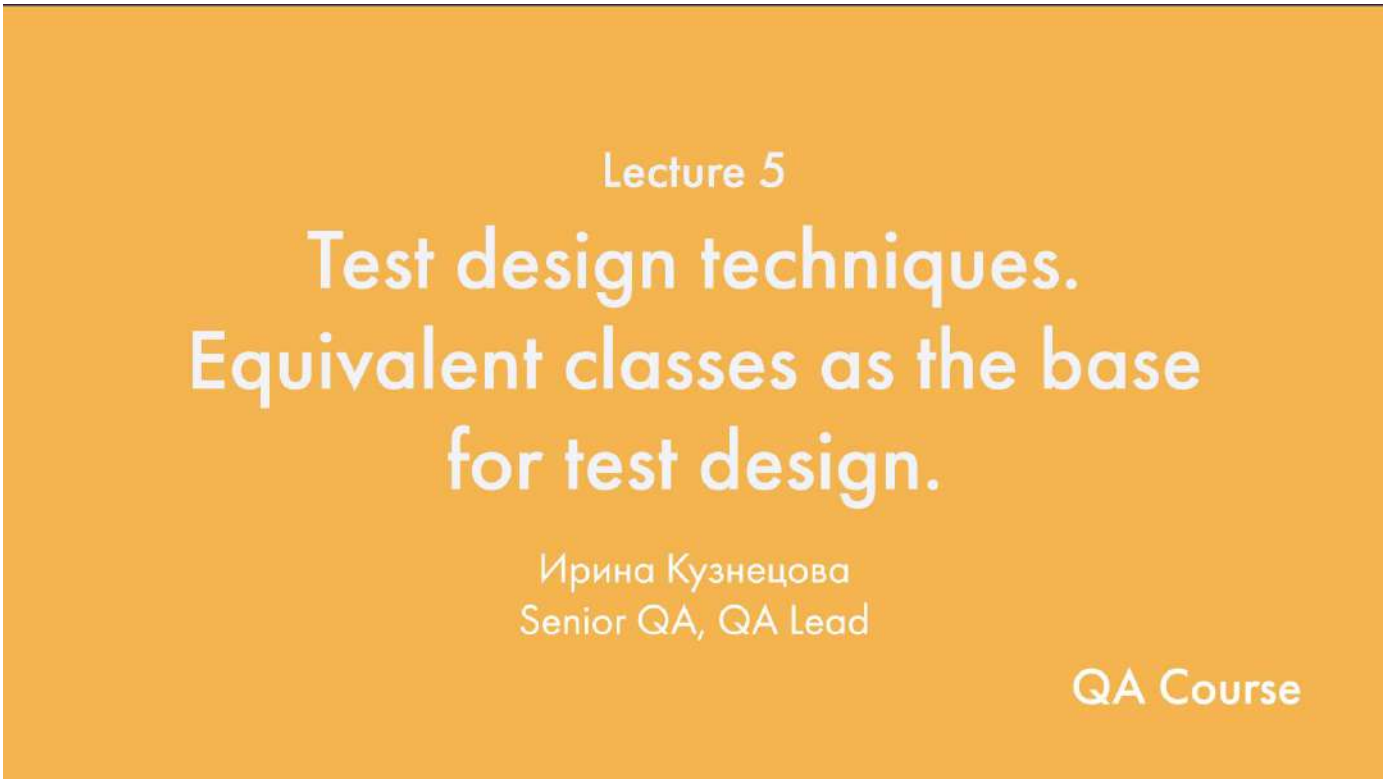


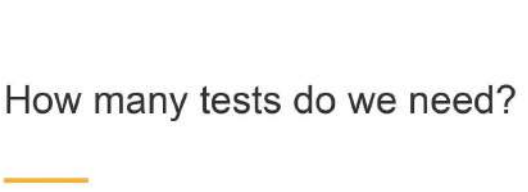
Тема лекции: Техники тест-дизайна.
Подтема: Техники тест дизайна, которые так или иначе базируется на классах эквивалентности либо связаны с ними, либо следует из этой методики.
Содержание лекции:

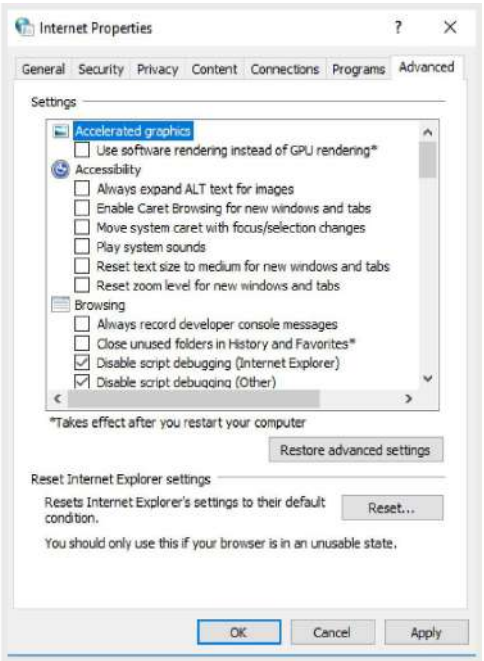
1. What is test design?
2. Equivalence class partitioning (ECP)
3. Pairwise testing
4. Boundary Value Analysis

1. What is test design?/ Что такое тест-дизайн?

Зачем нам тест-дизайн? Зачем нам нужны техники, чтобы писать тест кейсы? В первую очередь они нужны, чтобы обеспечить требуемое тестовое покрытие.

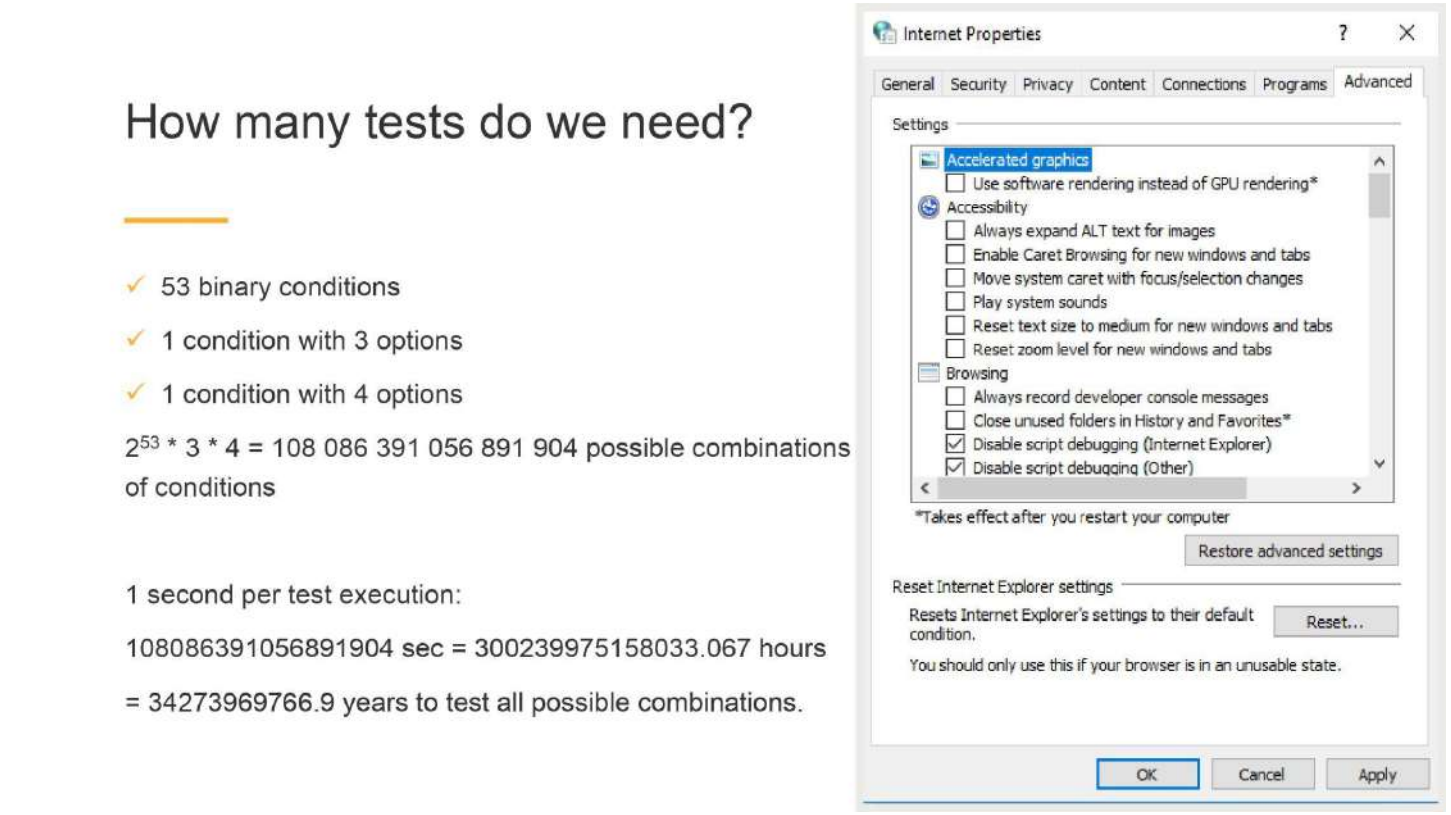
Итак давайте посмотрим на форму настроек интернет-соединения в Windows





Сколько тестов нужно чтобы проверить работу этой формы?

Здесь 53 бинарных условия, у которых по два возможных варианта True или False. Есть также одно условие с 3 возможными вариантами: включено, выключено или не указано, и одно условие, в котором 4 возможных опции. Если мы не будем использовать техники тест дизайна и просто попытаемся протестировать все возможные комбинации, то всего комбинаций таких опций получится примерно столько, сколько указано на слайде ниже. И время на тестирование также указано на слайде.



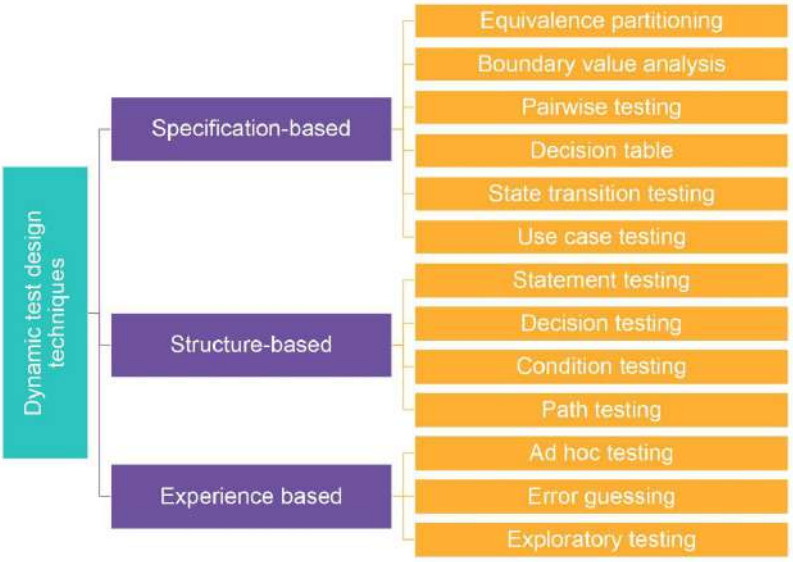
Это яркий пример иллюстрации на тему “Принципы тестирования” - Исчерпывающие тестирование недостижимо. Очевидно, что никто и никогда за это не заплатит, даже если мы захотим обеспечить работой себя и всех наших потомков на несколько тысячелетий вперед.

Как мы можем это огромное количество тестов сократить до какого-то более-менее вменяемого? Основа для тест-дизайна - это оценка приоритетов, разных сочетаний условий, разных тестов и рисков обнаружения ошибок в них. Исходя из этой оценки мы выбираем те тестовые условия, которые мы можем себе позволить покрыть. Это должны быть либо условия с наибольшим приоритетом, то есть какие-то сценарии, которые для нашего пользователя важны или возникают с большей вероятностью, либо те тестовые условия, в которых невыполнение системой своих функций или некорректное выполнение несет наибольшие риски. Это не всегда синонимы потому, что могут быть ситуации, когда сценарий достаточно маловероятный и является не очень нужным пользователю, но в случае если он не работает - это несет риск работоспособности системы. Таким образом нам нужно как-то выбрать, как нам покрыть требования и все остальные части нашего базиса тестирования.

Базис тестирования - это совокупность артефактов, которая нам рассказывает о том, какие ожидаемые результаты мы должны получить от работы системы. Основой для базиса тестирования на системном уровне тестирования являются требования. Но требованиями эта совокупность не ограничивается. Это могут быть use case, какие-то описания бизнес-процессов, должностные инструкции пользователей разных ролей и тому подобные источники. Итак, во-первых, мы используем для тест-дизайна их все: покрываем тестами как правило требования, а остальные просто учитываем в том или ином виде, потому что контрактных обязательств по ним у нас как правило нет. Тем не менее, например для валидации системы учет дополнительных артефактов необходим, т.к. на основе требований валидацию мы провести не можем.

Software testing techniques

DataArt



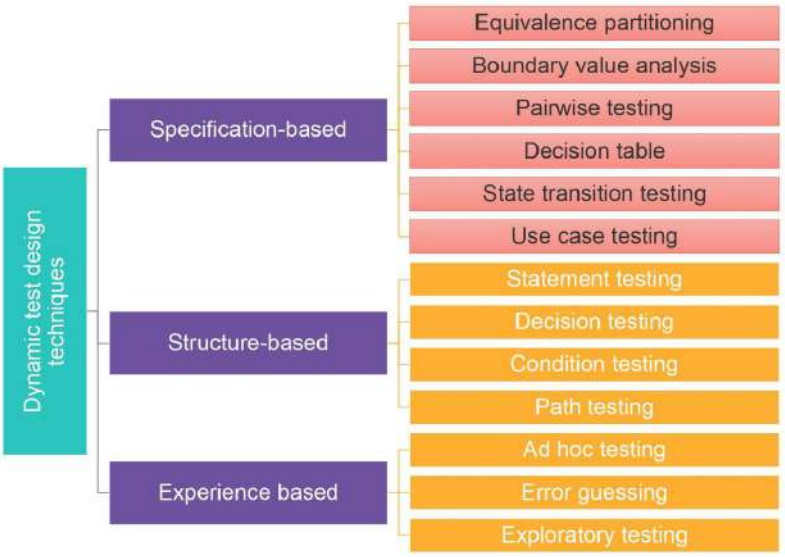
Различных техник тестирования существует великое множество. Их можно разделить на три категории, которые представлены на слайде выше:

- Техники, базирующиеся на разного вида спецификациях. Это могут быть требования, технические спецификации или артефакты, которые уже перечислялись ранее.
- Техники, базирующиеся на структуре. Это как правило методы белого ящика, то есть покрытие операторов, ветвлений, множественных ветвлений и т.п.
- Техники, основанные на опыте. Это ситуации, когда у нас нет достаточной информации о внутренней структуре программы и нет требований или они не соответствуют критериям, предъявляемым к требованиям (смотри лекцию 4 “Статическое тестирование. Тестирование требований”)

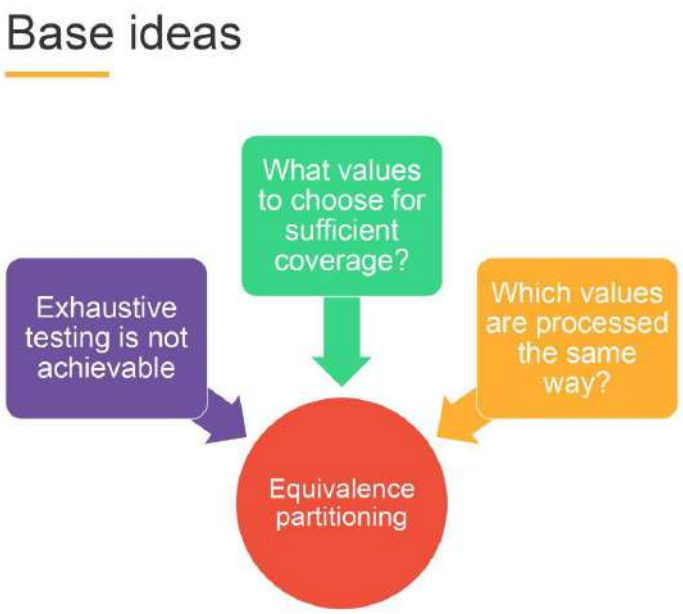
В рамках нашей пары лекций мы будем рассматривать только первый блок - методы, основанные на спецификациях. Потому что для них есть формальные методики, и мы будем изучать как этими формальными методами пользоваться.

Software testing techniques

DataArt



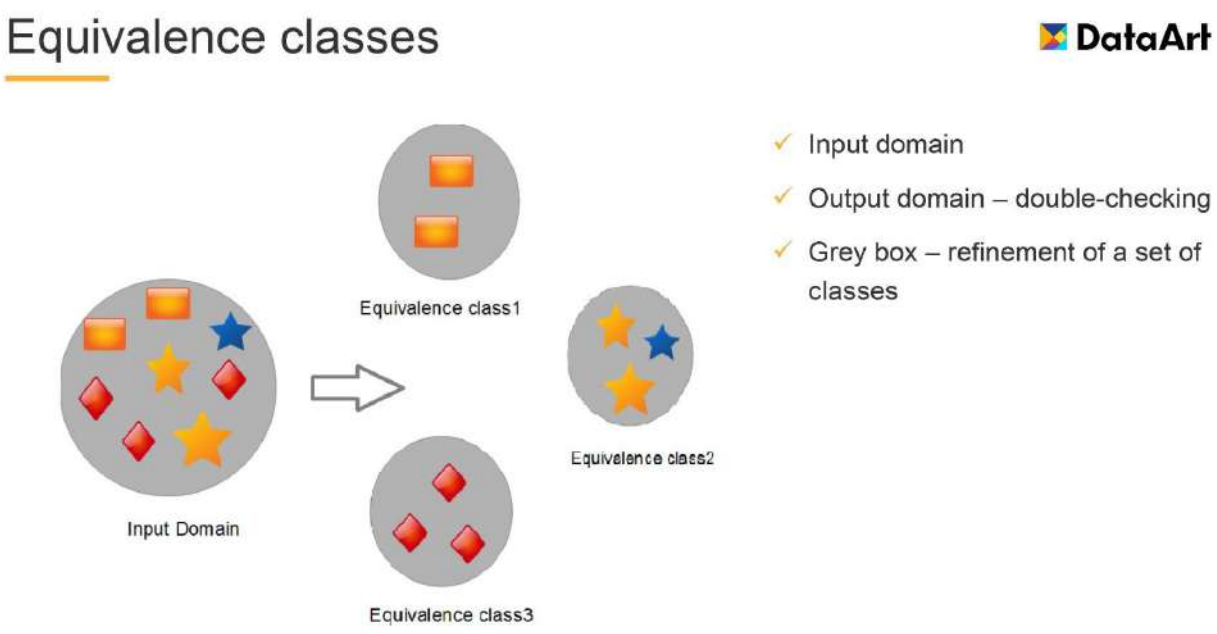
2. Equivalence class partitioning (ECP)/ Метод разбиения на классы эквивалентности.



Как мы уже ранее отметили, исчерпывающее тестирование недостижимо. Поэтому нам нужно выбрать какие именно входные значения из огромного множества мы хотим использовать. При этом мы хотим максимально покрыть требования так, чтобы не осталось не протестированных веток, случаев и т.д. Одним из возможных подходов является метод разбиения на эквивалентные классы.

Основная идея этого подхода - разделение множества входных значений на классы таким образом, чтобы для каждого класса мы ожидали одинаковую работу программы. В этой ситуации мы можем выбрать по одному из значений для каждого класса и протестировать на них, опираясь на то, что если программа работает одинаково для этого класса, значит все остальные значения дают аналогичные результаты при работе программы. То есть мы выбираем одно любое значение и считаем, что оно представляет всю выборку.

Что можно делить на классы эквивалентности? В первую очередь мы делим на классы эквивалентности входные значения. То есть для определенной группы входных значений мы должны ожидать определенное поведения системы, для следующих групп другое поведение системы и так далее.



Что нам может помочь разделить на классы эквивалентности правильно? Мы можем рассматривать не только класс эквивалентности входных значений, но и классы эквивалентности выходных значений. С одной стороны это нам помогает перепроверить себя. То есть если мы видим, что для какого-то одного класса эквивалентности выходные значения можно разделить на два разных класса эквивалентности, то получается, что мы изначально неправильно выделили этот класс эквивалентности. Значит его нужно дробить на еще какие-то подмножества. При этом мы можем увидеть, что для некоторых разных классов эквивалентности у нас выходные значения попадают в один класс эквивалентности. Это означает, что возможно мы можем объединить эти классы эквивалентности и на этом сэкономить время на написание и проведение тестов.

В какой ситуации так может быть? Предположим у вас есть какая-то форма с кучей валидаций для разных полей этой формы. Как правило, это очень тяжелый случай для применения классов эквивалентности, потому что возможных сочетаний значений очень много. Но при этом, если во всех негативных сценариях мы получаем одно и то же одинаковое для всех сообщение об ошибке, то сочетания возможных различных негативных сценариев нам проверять не надо. То есть мы проверяем по отдельности каждый кейс с ошибкой, а вот сочетание полей с ошибкой проверять не стоит. Мы можем себе позволить на этом сэкономить, потому что выходной результат от этих сочетаний не зависит.

В данном случае нам желательно отойти немного от чёрного ящика, уйти к серому и понимать порядок проверок. Это поможет ещё больше сэкономить. Но у нас не всегда есть информация о внутренней структуре программы. И в том, что мы на неё полагаемся, всегда есть определённые риски. Потому что после того, как эта информация к нам поступила, внутренняя структура может поменяться. И не факт, что эта новая информация к нам тоже попадёт во время, чтобы мы могли наши тест кейсы поправить. Поэтому серый ящик позволяет сделать тестирование более эффективным, но при этом требует большого внимания и учета рисков возможных неучтенных изменений внутренней структуры программы.

Пример

У нас есть некая система, рассчитывающая сколько денег вернется, если отменить оплаченное бронирование авиабилета. Работает она по правилам, которые представлены на слайде.

Example

DataArt

Example. Calculation of the commission when canceling air tickets:

- ✓ 5 days before departure, the commission is 0%
- ✓ Less than 5 days, but more than 24 hours - 50%
- ✓ Less than 24 hours, but before departure - 75%
- ✓ After departure - 100%

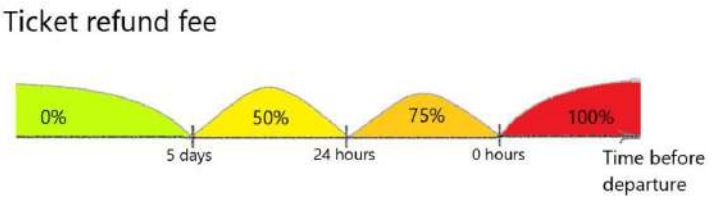
Для этого примера можно выделить 4 класса эквивалентности, представленных на слайде.

Example

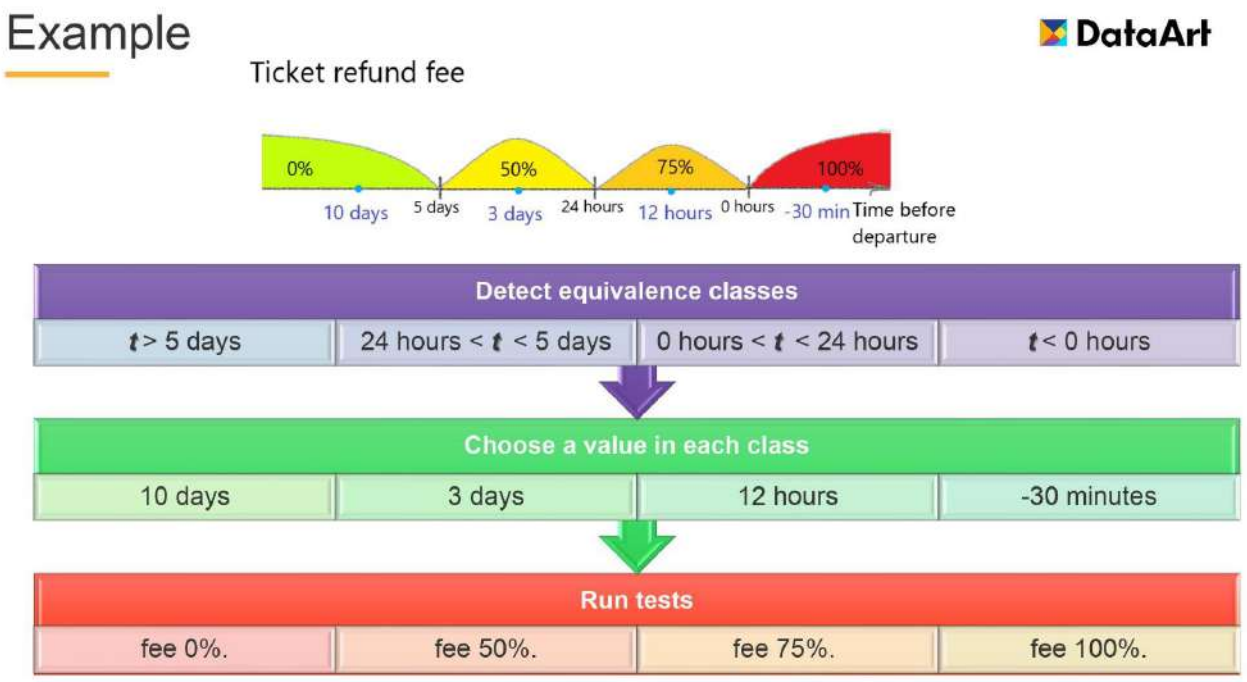
DataArt

Example. Calculation of the commission when canceling air tickets:

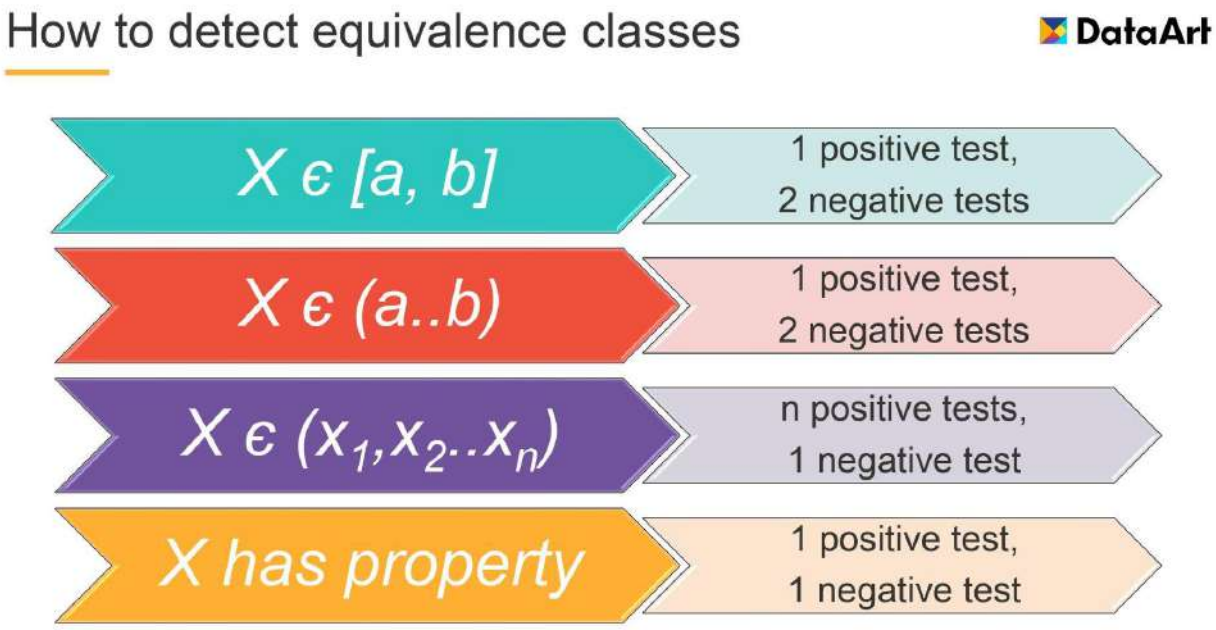
- ✓ 5 days before departure, the commission is 0%
- ✓ Less than 5 days, but more than 24 hours - 50%
- ✓ Less than 24 hours, but before departure - 75%
- ✓ After departure - 100%



Далее для каждого класса эквивалентности выбираем какое-то одно значение. Для выбранного значения проводим тест и сравниваем результат с ожидаемым.



Сложная часть этого метода - это разделения на классы. А дальше метод применить достаточно просто. Но стоит заметить, что применение метода простое только в той ситуации, когда у вас есть числовая прямая. На ней можно сделать засечки и разделить все на классы эквивалентности. В реальной жизни таких входных параметров немного. Входные параметры могут быть разными. На слайде ниже есть некоторый список, который мы можем рассмотреть.



1) Классы эквивалентности представлены отрезками. В этом случае граничные значения тоже входят в множество допустимых значений.

2) Классы эквивалентности представлены интервалами. Здесь уже граничные значения не входят в множество допустимых значений.

Применение метода для этих двух вариантов похоже на рассмотренный нами ранее пример с отменой авиабилетов. И в том и в другом случае с точки зрения метода эквивалентного разбиения существенных отличий не будет. Единственная разница в том, что для отрезка мы можем использовать граничные значения как представителей класса эквивалентности, а для интервала нет. В остальном метод будет работать будет одинаково: один позитивный тест на принадлежность интервалу/отрезку и два негативных по правую и по левую сторону.

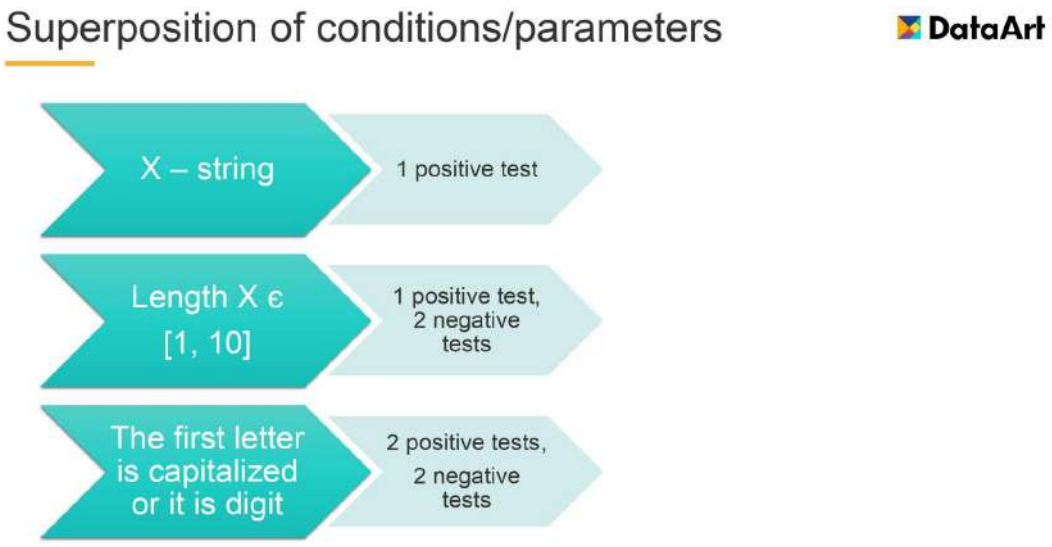
3) Чаще бывает, что есть некое дискретное множество значений (на уровне типов данных в компьютере). Например поле “пол” или выбор страны в форме на регистрацию. Здесь будет столько позитивных тестов, сколько есть значений, а также будет один негативный тест на проверку значения не из списка допустимых. То есть будет n+1 тест для того, чтобы покрыть эти тестовые ситуации.

4) Довольно часто задаётся условие в виде X - входное значение, обладающее собственным свойством. Например строка начинается с большой буквы. Здесь будет два класса эквивалентности: строки, начинающиеся с большой буквы, и строки, не начинающиеся с большой буквы. Особо въедливые в этой ситуации могут выделить третий класс: строки не начинающиеся с буквы вообще, а начинающиеся с какого-то другого символа. Это уже к вопросу о том, что результат применения метода сильно зависит от того, насколько разумно мы выделяем классы эквивалентности. Потому что нельзя точно сказать, как правильно поступить в этой ситуации. Все зависит от конкретных условий, от предметной области, от того, где и что мы рассматриваем и возможно от знания внутренней реализации системы. Например у нас может быть невозможен ввод чего-либо другого кроме букв, тогда зачем нам рассматривать этот третий класс эквивалентности.

В реальной жизни у нас очень редко бывают ситуации, когда мы обрабатываем один параметр, чаще мы проверяем совокупность нескольких параметров. И даже если у нас на форме есть только одно поле, важно понимать, что одно поле не равно один параметр.

Рассмотрим пример, в котором у нас есть одно поле, куда вводятся некие данные с ограничением по длине и с ограничением на первый символ.

У нас есть три условия. Фактически это три разных параметра и их необходимо рассматривать отдельно. По каждому из них нужно построить классы эквивалентности и получить набор классов. На слайде классы обозначены просто как позитивные или негативные.



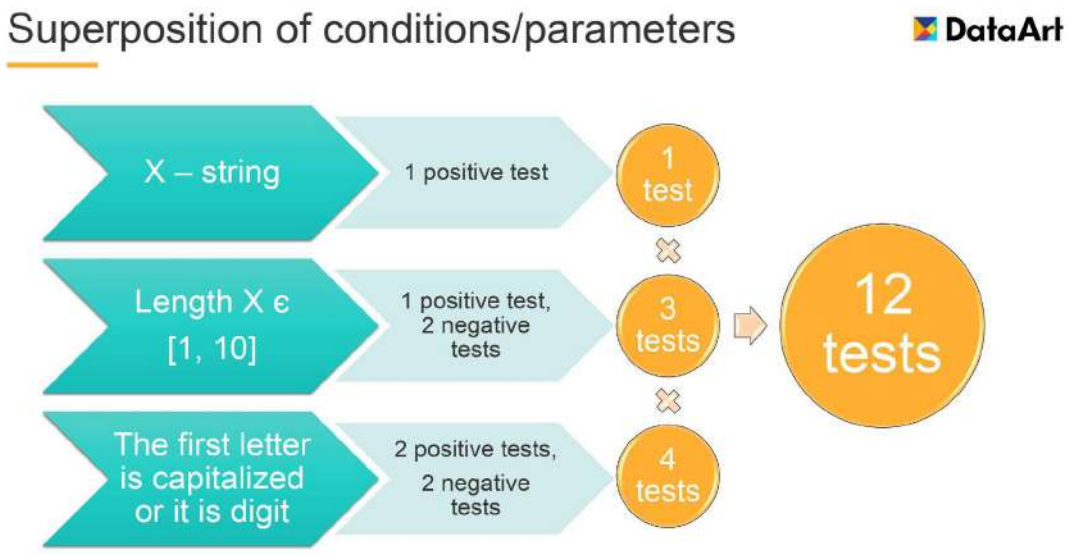
1) Почему в первом случае один позитивный тест? Потому что наше условие X - строка, и невозможно заставить компьютер не интерпретировать что-то как строку. То есть ввести не строку вы не можете. Поэтому здесь всего один класс эквивалентности.

2) По длине всё по классике - три класса эквивалентности

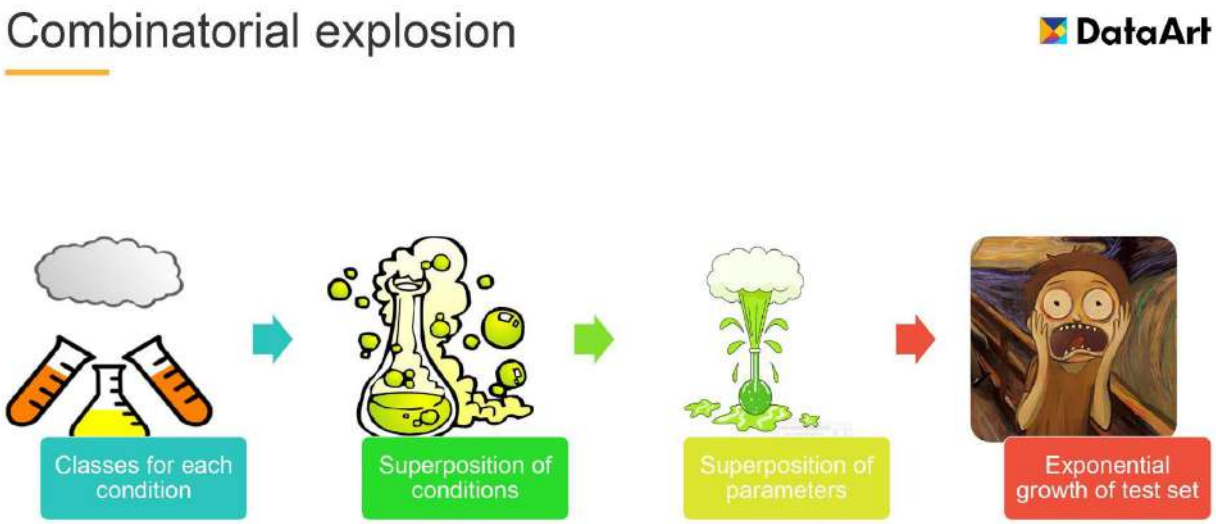
3) По символам - четыре класса эквивалентности:

- первый символ заглавная буква,
- первый символ цифра,
- первый символ не заглавная (строчная) буква,
- первый символ не буква и не цифра вообще.

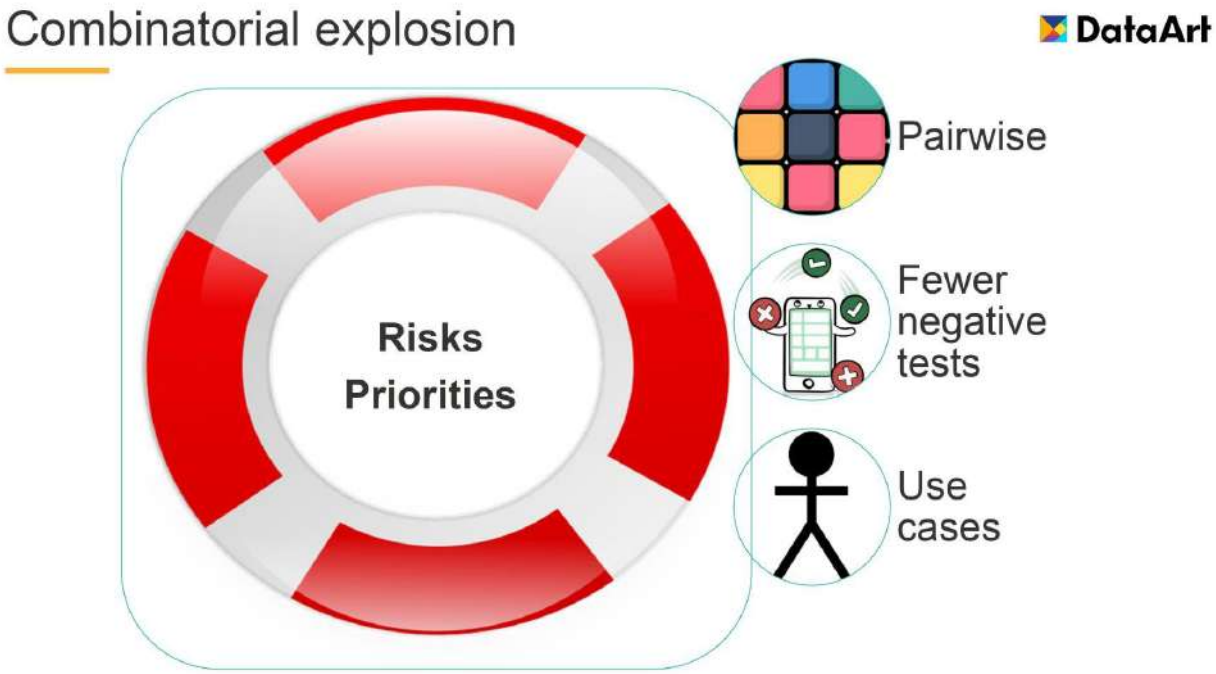
Дальше мы должны построить так называемую суперпозицию, она же декартово произведение, то есть сочетание всех классов со всеми. И в этой ситуации получим 12 тестов на проверку работы этого простого поля ввода.



Теперь давайте представим, что у вас несколько полей и по каждому такой же набор условий. Вы строите суперпозицию для этих полей и получается то, что называется комбинаторные взрыв, когда у вас количество тестов начинает уходить за любые разумные границы.



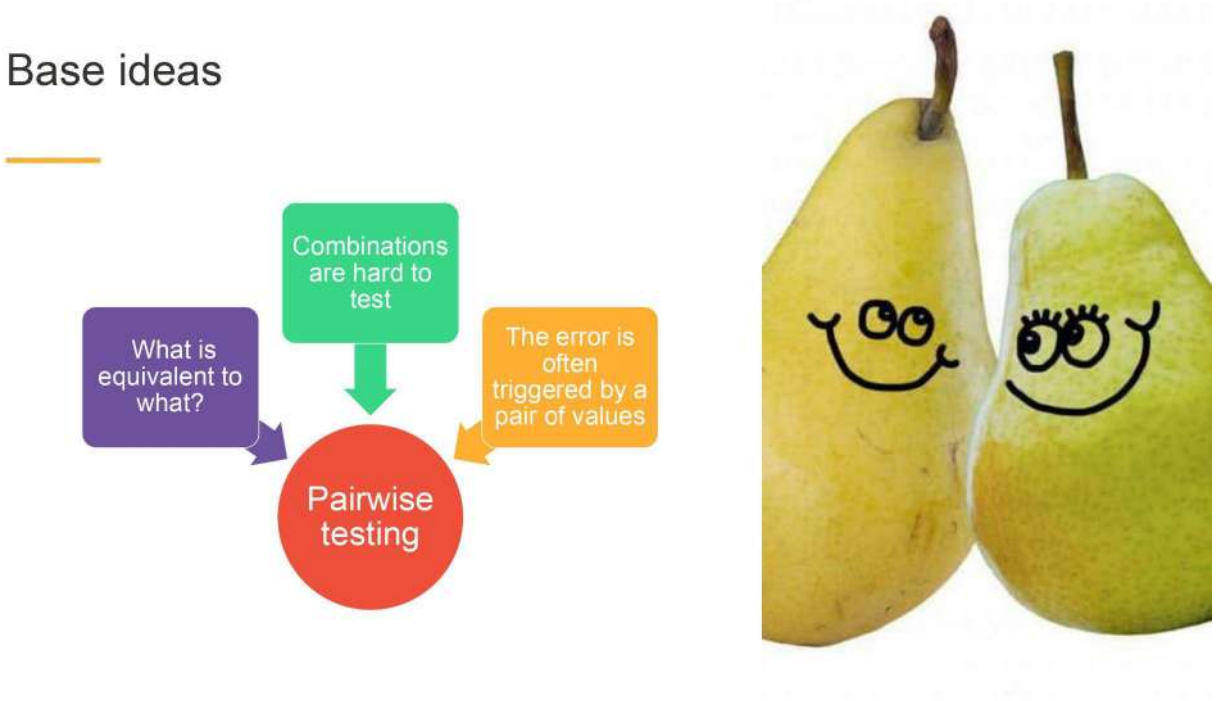
Собственно это и есть главное ограничение применения метода эквивалентных разбиений, потому что такое количество тестов потянуть достаточно сложно. Чаще всего вы будете применять метод ограничено, то есть для каких то отдельных полей, но не строить сложные суперпозиции. Это один из вариантов сокращения множества возможных тестов. Но это вариант не очень надёжный, потому что здесь вы эвристически учитываете приоритеты и можете упустить какие то важные кейсы.



Есть и более обоснованные способы сократить эти огромные тестовые наборы. Один из них - техника попарного тестирования Pairwise. По сути это отдельный метод тестирования, и мы его будем рассматривать далее. Кроме того, как уже говорилось ранее, можно поработать с негативными тестами. То есть сочетание негативных по разным параметрам тестов исключить и оставить только те тесты, в которых негативный тест будет только по одному из параметров. Это на самом деле огромное количество тестов, потому что в большинстве случаев негативных классов эквивалентности будет больше чем позитивных. То есть это сокращение тестового набора даже не в два раза, а в несколько.

Ну и наконец, мы можем использовать use case, чтобы понимать какие из наших тестов ближе к стандартным пользовательским сценариям, а какие от них дальше, и на этом основании расставлять приоритеты. Главное в сокращении тестового набора, чтобы наш подход действительно опирался на приоритеты и риски. То есть, чтобы он минимизировал возможность исключения теста, непрохождение которого, несет высокие риски или высокую вероятность ошибки.

3. Pairwise testing / Метод попарного тестирования.



Сам по себе метод эквивалентных разбиений достаточно сложен в применении. Мы не всегда хорошо понимаем, как разделить множество входных значений на классы эквивалентности. Особенно так бывает с какой-нибудь сложной предметной областью.

Например: Я долгое время работала в компании, которая производит радио контрольное оборудование. Чтобы тестировать софт, который работает с радио контрольным оборудованием, нужно хотя бы примерно понимать, что такое радиоконтроль, какие стандарты связи бывают. Также нужно хорошо понимать, что при переходе частоты из одного диапазона в другой мы меняем стандарт связи и у него совсем другие характеристики сигналы, которые надо мерить. То есть все эти вещи нужно знать, чтобы правильно разделить на классы эквивалентности например поле для ввода частоты. Если мы этого не знаем, то метод эквивалентных разбиений даёт очень невысокие результаты. Кроме того у нас есть комбинаторный взрыв, и с этим мы сделать ничего не можем, даже при правильном применении метода.

Исследования выявили, что в большинстве случаев триггером для ошибки является комбинация одного или двух параметров значений. Вероятность того, что триггером ошибки является сочетание трех параметров, на порядки меньше. Соответственно отсюда и появилась идея попарного тестирования. Идея в том, что если рассмотреть все возможные пары входных параметров при каких-то валидных значениях остальных параметров, то таким образом мы покроем достаточное количество тестовых случаев, чтобы считать, что тестирование достаточно глубокое и хорошее и мы не упускаем высоко-вероятный сценарий с высоко приоритетными ошибками.

На слайде представлена таблица из статьи, обосновывающей Pairwise тестирование

How good does it work?

DataArt

Table 2. Fault classification for injected faults

Fault Type	LAS	DMAS
2-way	30	29
3-way	4	12
4-way	7	1
> 4-way	7	3
Not Found	34	43

D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design," IEEE Transactions on Software Engineering, vol. 23, no. 7, pp. 437-444, 1997

Таблица показывает количество ошибок для двух тестовых систем зависящих от:
1-ая строка - от двух параметров, 2-ая строка - от трёх параметров и так далее. Мы видим, что различие достаточно существенное.

Пример.

У нас есть фонарик в смартфоне и работает он по правилам, которые указаны на слайде:

- приложение работает под Android или iOS,
- у него есть 3 режима яркости
- умеет гореть постоянно или мигать.

Example

DataArt

Flashlight mobile application

- ✓ works with iOS and Android
- ✓ has three brightness modes
- ✓ allows you to light constantly or flash in strobe mode.

На слайде ниже представлен набор тестов, которые мы получим методом эквивалентного разбиения для этого случая *(левая таблица)* и набор тестов, которые мы получим методом попарного тестирования, т.е сочетание всех возможных пар *(правая таблица)*.

Example

DataArt

	OS	Brightness	Light type
1	iOS	1	Constant
2	iOS	1	Strobe
3	iOS	2	Constant
4	iOS	2	Strobe
5	iOS	3	Constant
6	iOS	3	Strobe
7	Android	1	Constant
8	Android	1	Strobe
9	Android	2	Constant
10	Android	2	Strobe
11	Android	3	Constant
12	Android	3	Strobe

	OS	Brightness	Light type
1	iOS	1	Constant
2	iOS	2	Strobe
3	Android	2	Constant
4	Android	3	Constant
5	Android	1	Strobe
6	iOS	3	Strobe
7	iOS	2	Constant

В данном случае у нас всего три параметра, поэтому различия не радикальны, но тем не менее разница почти в 2 раза. Соответственно, чем больше у нас параметров, тем больше будет разница между количеством тестов, проходящих все возможные сочетания, и количеством тестов, полученных методом Pairwise.

Одно время метод попарного тестирования был очень популярным. Сейчас его наконец начали использовать меньше и в основном в тех случаях, когда его правда нужно использовать. Это не универсальный метод. В нем точно также как и в методе эквивалентных разбиений важно правильно собрать наборы параметров. По своей сути Pairwise чаще всего основывается на тех же классах эквивалентности, что и метод эквивалентных разбиений, с теми же сложностями правильного выбора классов эквивалентности

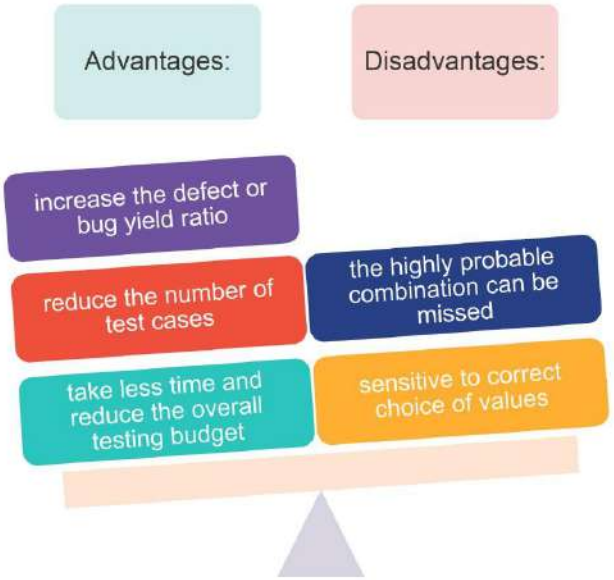
Также результатом построения сочетаний всех возможных пар может быть достаточно большое количество тестов. Их будет например не 800, как в методе эквивалентных разбиений, а 120. Но и это количество для вас тоже может оказаться большим в некоторых случаях.

Рассмотрим другие преимущества и недостатки метода попарного тестирования. В качестве преимущества мы получаем лучшее соотношение количества тестов к количеству багов. То есть количество тестов необходимых для того, чтобы найти хотя бы один баг, у нас уменьшается и общее количество тест-кейсов тоже уменьшается. Соответственно мы экономим время и бюджет проекта.

При этом мы можем упустить какие-то сложные кейсы, которые всё-таки появятся в продакшене. То есть необходимо понимать, что любое снижение количества тестов автоматически означает повышение рисков появления багов. Исчерпывающее тестирование невозможно, значит все равно останется что-то не протестированное, значит все равно баги в системе будут. Всё что мы можем, это минимизировать вероятность того, что пользователь на них наткнется. И второй недостаток - чувствительность к выбору корректных значений, которые мы потом будем попарно тестировать. Если выбор не корректный, то как и с методом эквивалентных разбиений результат получится достаточно скромным.

Pros and cons of pairwise testing

DataArt



Раньше метод попарного тестирования, был достаточно сложен в применении из-за отсутствия инструментов, для построения таблиц. Сейчас же этих инструментов полно. Часть из них представлена на слайде ниже.

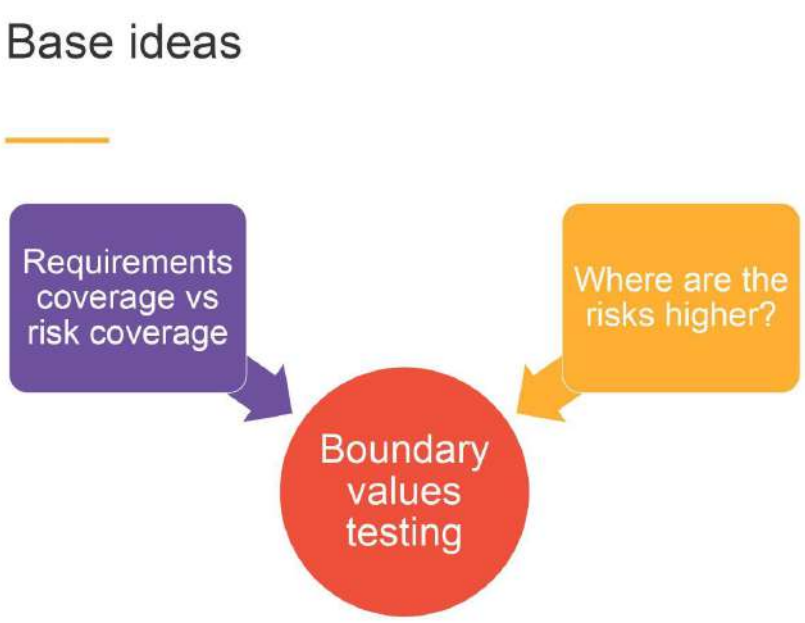
Pairwise tools

DataArt

- ✓ **PICT** — *'Pairwise Independent Combinatorial Testing', provided by Microsoft Corp.*
- ✓ **IBM FoCuS** — *'Functional Coverage Unified Solution', provided by IBM.*
- ✓ **ACTS** — *'Advanced Combinatorial Testing System', provided by NIST, an agency of the US Government.*
- ✓ **Hexawise**
- ✓ **Jenny**
- ✓ **Pairwise** *by Inductive AS*
- ✓ **VPTag free All-Pair Testing Tool.**

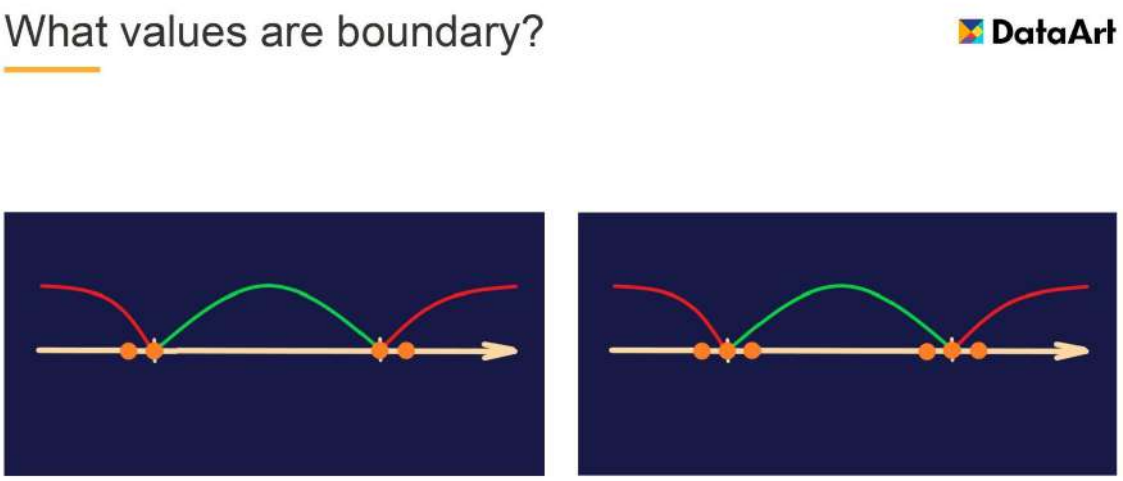
PICT используется шире всего, потому что это онлайн инструмент. В нем вы вводите ваш набор параметров и возможные значения для каждого из параметров, и он просто выдаёт вам таблицу.

4. Boundary Value Analysis/ Анализ граничных значений.



Метод появился из простого наблюдения, что на границах классов эквивалентности баги возникают существенно чаще, чем на значениях из середины класса, если мы берем некое последовательное множество. Почему так? Потому что программисты и аналитики могут перепутать знаки, например вместо знака “<” поставить знак “>”. Потому что достаточно много краевых условий заложено в алгоритмы, которыми мы пользуемся, когда берем их из сторонних библиотек и трудно уследить за ними всеми. Соответственно это повышает вероятность ошибок в каких-то пограничных ситуациях и означает, что довольно эффективно находить ошибки мы будем, тестируя граничные значения.

Метод анализ граничных значений основывается на эквивалентных разбиениях и предполагает, что мы тестируем границы классов эквивалентности и, как часто формулируется в разных учебниках, соседние с ними значения. Что такое соседние значения? Рассмотрим следующий слайд



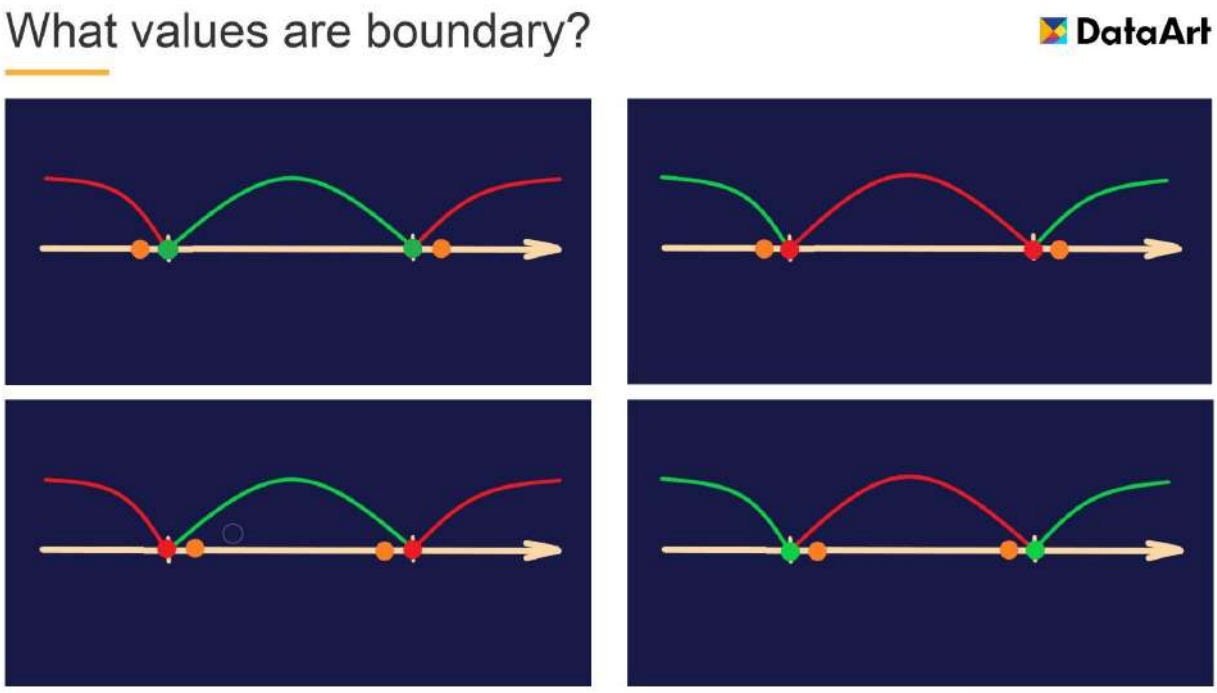
Есть источники, где будет написано, что мы тестируем граничные значения и соседние с ним, лежащие за пределами классы эквивалентности (левая картинка) Есть источники, в которых будет написано, что мы тестируем оба соседних значения (правая картинка). Какой же вариант правильный?

Верным ответом будет левая картинка - мы тестируем граничные значения и соседние с ним, лежащие за пределами классы эквивалентности. Анализ граничных значений опирается на метод эквивалентных разбиении и мы хотим покрыть классы эквивалентности. То есть мы берём граничные значения и соседнее значение с той стороны, где мы переходим через границу классов эквивалентности и уже попадаем в другой класс.

Тестирование, когда мы берём значение в обе стороны - это более простой подход, но оно избыточное. Если у вас таких кейсов лишних получается десять, то ладно, но если их сто, то это становится существенно. Поэтому в каких-то мелких частных случаях можно позволить себе избыточное тестирование. Такое избыточное тестирование называется проверкой робастности - проверкой устойчивости системы. Она не всегда нужна, но иногда проще написать эту пару лишних кейсов, чем вообще задуматься, нужна она или нет. Потому что пара лишних кейсов в случае, когда их у вас несколько сотен - это не очень много. А вот когда количество таких кейсов становится большим, разница более существенная, и важно выбирать граничные значения правильно.

В рассмотренном выше примере на левой картинке подразумевается, что допустимое значение это отрезок. А если у нас интервал и граничные значения недопустимое? А если у нас обратная ситуация, когда у нас отрезком заданы негативные значения, а остальные позитивные?

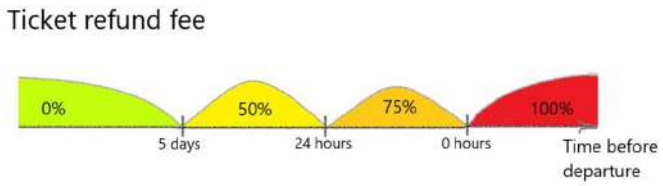
Четыре кейса для четырех разных случаев представлены на слайде. Они покрывают все возможные варианты. Красной линией обозначены негативные значения, а зеленой позитивные.



Давайте теперь вернемся к нашему примеру из начала лекции с возвратом денег при отмене авиабилетов. И рассмотрим, сколько же нам нужно отступить от границы? Один день? Час? Минута? Какое значение будет соседнее?

Above and below the minimum and maximum
What does it mean?

DataArt



1 day?

1 hour?

1 minute?

1 second?

В идеале мы опираемся на соседнее значение с точки зрения того, как данные хранятся. То есть, если у нас данные хранятся в секундах и обрабатываются в секундах, то мы берём как интервал - секунду, если в минутах соответственно берём интервал - минуту. Потому что на интервале в полчаса, если данные хранятся в секундах, то ошибку > или >= мы не поймаем.

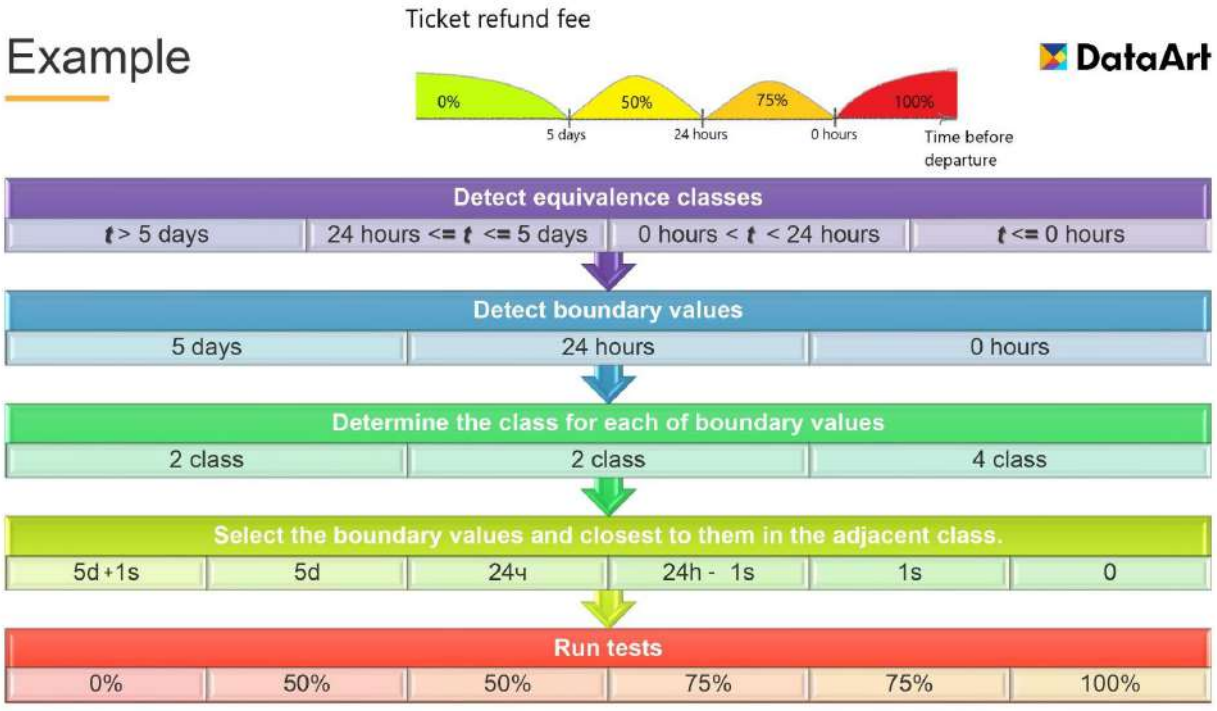
Если у нас данные хранятся и обрабатываются в секундах, а в интерфейсе вводятся в минутах, например, то мы технически не можем задать такие значения и берём те которые можем. То есть мы берём минимальный шаг, который можем себе позволить.

А если у нас входной параметр это некое число с плавающей точкой с бесконечным количеством знаков после запятой? У нас никогда не будет поля в интерфейсе, где мы сможем ввести бесконечное количество знаков после запятой. Там всегда будет ограничение. Мы опираемся на это ограничение и берём ближайшее соседнее значение, которое можем ввести. Если мы тестируем API, то мы можем ввести любое достаточно малое значение.

Общий алгоритм работы с анализом граничных значений представлен на слайде ниже

Example

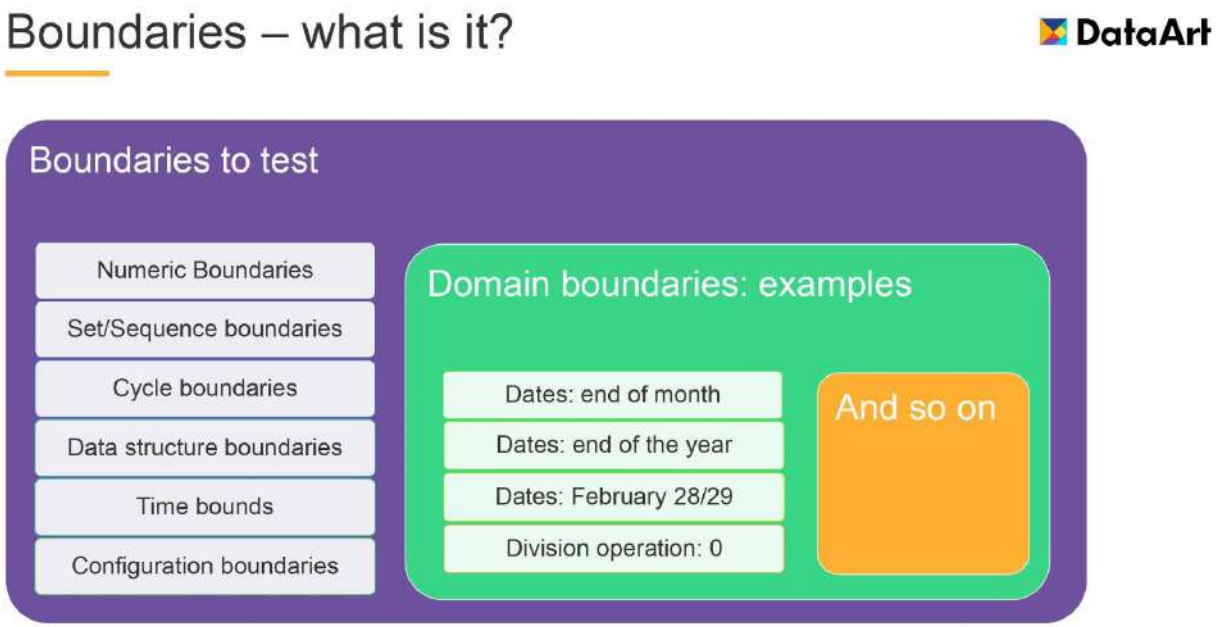
DataArt



Обратите внимание на второй шаг, где мы выбрали классы эквивалентности. В методе эквивалентных разбиений > или >= нас не интересовали, а здесь они начинают нас интересовать. И очень часто уже на этом этапе, даже не начав писать тест кейсы, мы оформляем баги и идем с ними к аналитикам, потому что в ТЗ не написано, как должно быть правильно. Так или иначе мы выясняем эти требования, пишем на них тест кейсы. А потом заводим на них новые баги, потому что раз в ТЗ было не написано, то разработчик сделал как хотел и как считал нужным. И соответственно это будет вторая порция багов. И на этом примере мы видим, как полезно тестировать требования. Потому что если бы мы протестировали требования достаточно хорошо, мы бы выявили неточности до того, как разработчик написал код.

Итак мы определили класс эквивалентности, выбрали граничные значения и определили третий шаг - к какому из классов эти граничные значения относятся. Ну и наконец выбираем соседние значение и запускаем тест.

Далее давайте немного поговорим о том, что делать если у вас всё-таки не числовая прямая. Какие вообще граничные значения бывают в этом случае? На слайде представлены и описаны некоторые примеры границ. Самый красивый пример - с датами, потому что там одно маленькое поле, а граничных значений полно.



http://testingchallenges.thetestingmap.org/challenge6.php

По ссылке можно найти тренажёр для тестирования граничных значений. Там есть одно поле с вводом даты и предложено найти в нем все баги. В нем есть дата и время, а также 18 багов на граничных значениях. Например это 29 февраля в Високосный год, 30 февраля Високосный год, граничные значения времени, граничные значения числа, месяца, то есть 13 месяц, 0 месяц и т.п. То есть это такое огромное поля для применения метода, который мы с вами сейчас рассматриваем."

Какие еще границы можно рассматривать? Довольно интересными являются границы структур данных, то есть если у вас вводится некое число и в требованиях нет ограничений на значение этого числа *(в любом случае ограничение типа присутствует)*. Надо или не надо его тестировать? Приоритетны такие цели или не очень? Это отдельный вопрос, но технически - это граничное значение и в любом случае тут будет какое-то неожиданное поведение.

Границы конфигураций тоже достаточно интересная тема, как и тестирование конфигураций. Мы далее немного поговорим про классическое конфигурационное тестирование, а сейчас я расскажу про частный случай.

Например, у вас в системе есть какие-то настройки, в которой вы инсталлируйте систему и у вас есть какой-то набор настроек у инсталлятора. Может быть достаточно большое количество корнер кейсов, то есть граничных условий: по занимаемой памяти, по настройкам системы, по значениям параметров, которые вы задаете. Даже если возьмем операционную систему Windows - в ней есть ограничение на длину адреса, длиннее которого адрес быть не может.

Есть и более сложные случаи. Например у вас есть какой-то клиент-серверное приложение и у сервера есть конфигурационный файл. Какие параметры в нем возможны и допустимы? В принципе, если это интернет-магазин, который устанавливается один раз, разворачивается на сервере и дальше в таком виде живёт, то это тестировать не обязательно. Если же например вы пишете какую-то свою СУБД, то предполагается, что её будут пытаться поставить на свою локальную машину студенты, школьники, да кто угодно. И вот тогда подобные вещи становятся важными. А какие значения параметров конфигурации можно задать? Какое значение количества памяти на одно подключение к базе данных? Когда это число совсем не адекватное, важно чтобы система даже не поднялась, потому что работать она нормально всё равно не сможет. То есть количество таких возможных кейсов будет тем больше, чем более сложную конфигурацию имеет ваша система, и чем большее число параметров вы предоставляете своим пользователям для настройки и для регулирования.

Конфигурационное тестирование - классическое.

Рассмотрим пример с мобильным приложением, которое вы должны тестировать на разных девайсах. Это самый частый и самый очевидный кейс, применения метода граничных значений. На что здесь мы можем смотреть? Самая очевидная вещь: на каких версиях Android вы будете тестировать мобильное приложение? Как правило, у нас есть минимальная версия, которую мы должны поддерживать, соответственно мы точно тестируем на ней и на последней версии. Собственно это и есть граничные значения. Проводить тестирование на всех поддерживаемых версиях может оказаться слишком дорого. Как правило, количество конфигурационных тестов для каждой из версии - весьма заметное. То есть на каждую версию у вас уйдет полчаса-час. Бывают приложения, где конфигурационное тестирование более сложное. Тут все зависит от того насколько сложное приложение и насколько оно зависимо от ресурсов системы.

Если у вас такой большой разбег по версиям, то имеет смысл применить метод эквивалентных разбиений, а не анализ граничных значений отдельно от него. И понять, что в той части, которая нас интересует, SDK менялась столько-то раз. Помимо крайних версий мы получим два класса

промежуточных, в которых поведение операционной системы отличается и от младшей версии и от старшей версии. И тогда мы берём по одной версии операционной системы из промежуточных.

Тоже самое касается тестирования девайсов, которые написаны на Андроиде.

Если кто не знает, то операционная система Android как таковая не существует - это огромный зоопарк операционных систем в рамках одной базовой версии, для которой каждый производитель девайсов что-нибудь своё изобретает. То, что смастерил какой-нибудь скажем китайский производитель телефонов, вполне может заставить вас предложения сломаться. Как правило китайские производители часто перестают поддерживать свою операционную систему очень быстро. К примеру год поддерживают, а затем прекращают. Это означает, что часть пользователей, которые раз в год телефоны не меняют, останутся со старыми версиями систем. И если вы ориентируетесь на эту целевую аудиторию, то эти граничные значения у вас появляются условно новые, даже когда вы опираетесь просто на статистику использования телефонов.

Телефонов существует огромное количество, разных по своим параметрам, например по размерам экранов, разрешению экранов. Что делаем в таком случае? Точно также мы берём минимально поддерживаемые разрешения, которое оно должно быть указано в требованиях. Это тоже существенный момент, который должен учитываться при тестировании требований. Причем учитываться при тестировании требований как к мобильному приложению, так и к веб и к десктопным приложениям. Хоть в десктопах нет такого зоопарка экранов, но там есть волшебная возможность уменьшать и увеличивать окошко. Вот поэтому минимальное поддерживаемое разрешение обязательно учитываем. Размер экрана и разрешение экрана очевидно связаны, но как раз для телефонов у нас могут возникать интересные ситуации, когда у нас размер маленький, а разрешение 4К. И вот такие кейсы тоже лучше тестировать. То есть минимальный размер с максимально возможным для этого размера разрешением.

Что касается верхней границы, то тут всё несколько сложнее. Потому что, во-первых, у вас не всегда будет доступен максимальный размер экрана. А, во-вторых, вы можете начать тестирование сегодня, а через месяц появится девайс с экраном или с разрешением большим, чем вы взяли. Но на самом деле максимальный размер вам не нужен, вам нужен достаточно большой размер экран. Потому что баги вёрстки, которые мы в этой ситуации ловим, будут возникать на любом достаточно большом экране. С появлением экранов 4К все подобные эффекты стали заметны. Если кто-то себе покупал подобный девайс, то наверняка натыкался на неудобную верстку, которая рассчитана под стандартное разрешение. В таком случаем происходило следующее - на экране много места, а кнопки на экране крошечные, буквы совершенно крошечные, попасть в это невозможно. Эта проблема актуальна не только для мобильных устройств, но на мобильных устройствах появляется гораздо чаще.

Ещё один параметр, к которому анализ граничных значений мы всегда применяем - это тестирование систем с разными ресурсами. Если вы разрабатываете приложение чувствительное к ресурсам, например игру. Для игры у вас обязаны быть минимальные системные требования, то есть минимальный объем оперативной памяти, процессора и т.д. И естественно на этой минимальной конфигурации вы проводите тестирование. Здесь же встает тонкий вопрос тестирования производительности: какие ещё конфигурации вас будут интересовать? Если в принципе кроме минимальных системных требований никаких других различий на разных ресурсах для системы нет, то тестирования минимальных и каких-то более или менее стандартных будет достаточно. Но Performance тестеры очень часто находят всякое интересное на каких-нибудь промежуточных вариантах, потому что там подобные ошибки часто вылезают.

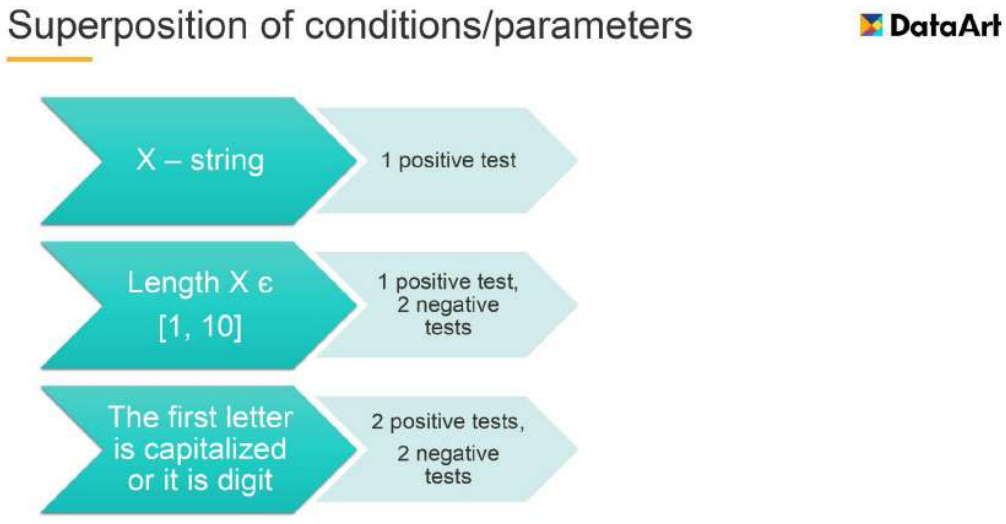
Бывают игры, которые подстраиваются под ресурс компьютера, например графику чуть-чуть упрощают, если у вас не хватает ресурсов в полном объеме для ее использования, еще какие то оптимизации и упрощения делают. И эти кейсы стоит проверять, но это уже не про анализ граничных значений. Про анализ граничных значений - это то, что мы всегда тестируем минимальное. В данном случае, в случае с ресурсами, максимальное тестировать не надо, так как это не актуально.

Questions

1. Правильно ли я понимаю что негативный тест это проверка того чего не должно быть?
Неправильно! Негативный кейс - это кейс, в котором мы ожидаем от системы отказа от выполнения той или иной функции.

2 Нужно ли обрабатывать по другому отсутствие символа (т.е пробел) при разбиение на классы эквивалентности?

Из тех требований, которые мы здесь сформулировали нет, но требования могут быть сформулированы по-разному. Пробел - это символ и если его нужно обрабатывать отдельно, то на это должно быть отдельное требование и тогда мы это учитываем в наших тест кейсах. Если отдельных требований на это нет, то пробел это точно такая же строка и она попадает в одну из групп негативных тестов.



3. Вопрос про учет ввода символов

Если мы знаем, что у нас в системе всё в Unicode, то необходимости тестировать какие-то спецсимволы, нет. Если у нас такого специального требования нет, то посмотреть как система на какую-нибудь экзотику отреагирует в принципе интересно. Интересно с точки зрения QA специалиста которому хочется всё сломать, с точки зрения практического смысла это нужно если наша целевая аудитория в принципе такое подразумевает. То есть если мы разрабатываем приложение для оплаты парковок в городе Воронеже, то скорее всего не самый актуальный кейс. Если мы разрабатываем систему которую предполагается что могут быть пользователи с экзотическими всякими клавиатурами то имеет смысл конечно надо включить т.к приоритет этого кейса существенно повышается достаточно, чтобы его на регулярной основе запускать.

4. Если в тесте первый символ является буквой и одновременно этот же тест, это тест того, что первый символ это не цифра?

У нас нет кейса не цифра а буква, у нас есть кейс, не буква и не цифра, у нас есть группа строк, которая начинается с цифры, начинаются с заглавной буквы, начинается со строчной буквы. У нас четыре этих класса, куда любая строка в какой-то из них попадает.

Мы негативными считаем те кейсы, в которых нарушаются требования и ожидаем от системы, что она как-то проинформирует пользователя о том, что он неправ в этих кейсах. Остальные кейсы у нас считаются позитивными, а на классы мы их делим потому, что но они разные и нам нужно убедиться что все работает корректно.

5. Как в реальности производится тестирование на разных версиях Android, то есть мы каждый раз меняем прошивку телефона?

Чаще всего нас интересует не только разные версии телефонов, но и разные размеры и разрешение экрана. Если у нас приложение задействует какие-то ещё функции телефона, сканер отпечатка пальца например, то нас как правило будет интересовать еще и телефон с отпечатком сканером и без сканера. Соответственно вот из этих разных наборов параметров мы собираем некий конфигурации. То есть у нас несколько телефонов есть в тестировании. А если у нас нет бюджета, чтобы их купить, можно пользоваться эмуляторами и симуляторами

6.Какое практическое применение техник тест дизайна к проетку?

Вы берёте требования и вы понимаете, что для тестирования определенной части приложения удобно ложится такой-то метод, вот вы его и применяете. Не бывает такого, что мы весь проект тестируем методом эквивалентного разбиений. Это неудобно, и в некоторых случаях не то чтобы невозможно, но чаще всего не нужно. То есть мы выбираем те области, где каждый конкретный метод можно применить. Для тестирования некоторых форм метод эквивалентных разбиений вполне себе важен и нужен, например для всяческих банковских расчетов по типу эквайринга. На них может быть очень много параметров, и метод эквивалентных разбиений даст огромное количество тестов. Но это тот случай, когда стоимость ошибки это непосредственные деньги наших клиентов. Риск от ошибки репутационный и юридический очень велик. Поэтому большое количество кейсов в данных областях применимо. Если вам для тестирования какой-то функции не подходит метод эквивалентных разбиений, всегда можно выбрать какой-то другой метод.

7.На различных операционных системах мы тестируем одно и тоже? Или можно на одном устройстве всё проверить, а на остальных только визуальные дефекты?

На одном устройстве на некой дефолтной конфигурации мы проводим всё тестирование безусловно. А дальше вопрос выбора тестов для конфигурационного тестирование - это отдельная история, потому что гонять на всех девайсах или на всех конфигурациях все тесты это очень накладно, и мы никогда так не будем делать. Визуальных тестов, о которых шла речь в вопросе, может быть недостаточно. То есть общие правило простое: мы выбираем для конфигурационного тестирования те тесты, результат выполнения которых, зависят от параметров конфигурации. Дальше если параметр конфигурации это размеры и разрешение экрана, то мы проводим визуальные тесты. Если параметр конфигурации у нас сканер отпечатка пальца в телефоне, то мы выбираем те функциональные тесты, которые задействуют сканер отпечатков пальцев. Если параметр конфигурации у нас аппаратные

ресурсы, так как предполагается тестирование игры, то мы будем гонять перфоманс тесты на разных конфигурациях, чтобы оценить эту характеристику работы системы. При этом, иногда в сложных случаях мы будем делить параметры конфигурации на группы, потому что параметров может быть много. Тестов зависящих от какого-то одного из параметров тоже может быть много, но он при этом может не зависеть от остальных параметров. Тогда мы будем группировать параметры так, чтобы на каждом наборе конфигурации гонять минимальный набор тестов, который зависит от этих параметров. В простых случаях, когда тестов не очень много, так не делают, потому что это организационно довольно сложно и нужно ничего не перепутать. Затем может оказаться так, что набор конфигурации поменяется и нужно будет снова ничего не перепутать. Поэтому если у вас тестирование одной конфигурации укладывается условно в час, а самих конфигураций в пределах 3-5 шт. (как чаще всего бывает) - это не нужно даже при большом наборе параметров. Если у вас более сложная ситуация, то стоит изучить вопрос о том, как правильно параметры поделить на группы так, чтобы минимизировать прогоны тест.