
**Тема лекции**: Техники тест-дизайна черного ящика, т.е. основные спецификации

Decision tables

Откуда вообще взялась идея этого метода? Достаточно много проектов, достаточно много продуктов, со сложной предметной областью, со сложной бизнес логикой, которая в общем не сводится к тестированию на различных наборах данных. То есть это области, где есть какие-то сложные бизнес правила, которые зависят от многих условий, от действий пользователей и тому подобного и нужно проверять различные комбинации, например настроек системы с действиями пользователя, с реакцией системы, возможно с какими-то внешними параметрами и тому подобными вещами. Исходя из этих потребностей была придумана методика тестирования, основанная на переборе вариантов сочетаний условий, то есть основная идея метода - это рассмотреть все комбинации возможных условий, от которых зависит конечный результат поведения системы и для каждого набора условий определить ожидаемый результат.

Эта методика универсальна, она применима почти к любым требованиям, она не всегда будет лучшим выбором, но всегда применима в отличие от, например, эквивалентных разбиений, которые применимы не всегда. И кроме того эта методика позволяет регулировать объём тестирования, то есть, выбирая условия на разных уровнях бизнес логики, пренебрегая возможно какими-то мало приоритетными условиями, мы можем регулировать набор тест кейсов, выбирая наиболее приоритетные и не включая в рассмотрение менее важные, если у нас например ограничено время или бюджет.

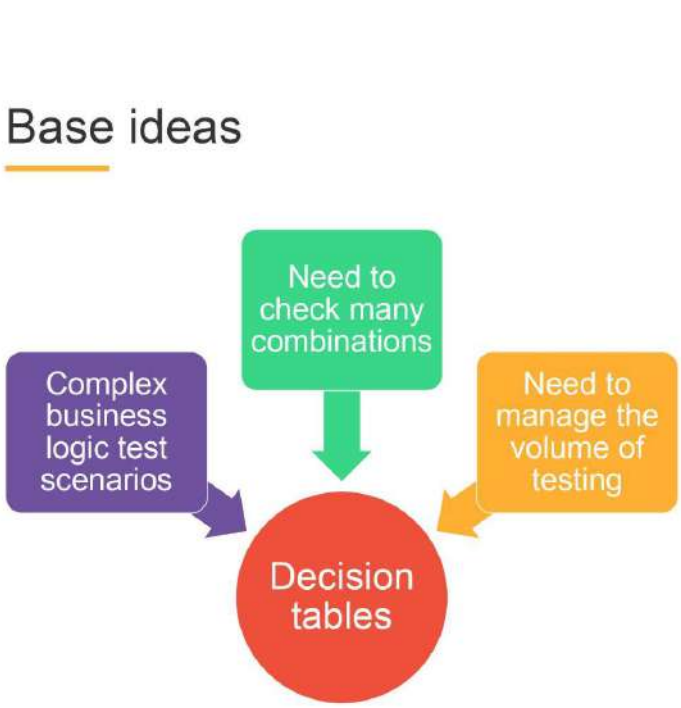
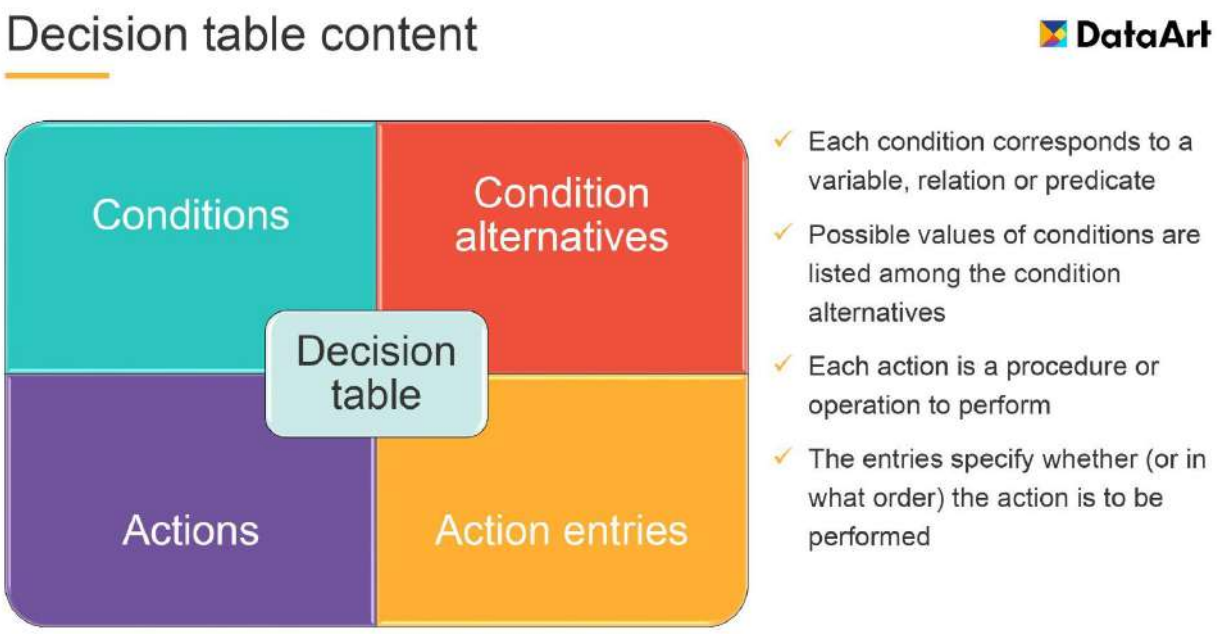



Таблица решений состоит из 4 секций, которые изображены на слайде.




В первой секции мы должны перечислить все условия, которые мы включаем в рассмотрение и от которых зависит поведение системы, затем для этого набора условий мы строим все возможные сочетания вариантов их реализации. Есть литература в которой явно прописано, что условия должны быть бинарные, но это не всегда удобно. В качестве условий можно использовать, например, набор диапазонов возрастов с 0 до 18, с 18 до 60, с 60 до 90 мы получим 3 возможных наборов значений для условия “возраст”. То есть само название этой секции conditions, оно скорее сложившееся привычное с того времени, когда по академическим учебникам использовали только условие бинарные, которые принимают значение только True или false. Надо сказать, что для них правда удобнее строить таблицу но сами эти значение выбирать не всегда удобно.

Нижние две секции таблицы: левая секция - это действие. Это те возможные действия, которые должна или может выполнить система при каких-то из этих условий. Здесь мы должны полный набор возможных действий перечислить. И наконец, применимость существования этих действий, то есть в четвертой секции мы отмечаем для каждого набора условий, какие действия соответствуют этому набору, то есть должны быть выполнены системой при соблюдении этих условий.

Пример: Предположим у нас есть некий магазин, который работает по изложенным на сайте правилам.

## The simplest example – discount table

DataArt

A company sells merchandise to wholesale and retail outlets. Wholesale customers receive a two percent discount on all orders. The company also encourages both wholesale and retail customers to pay cash on delivery by offering an additional two percent discount for this method of payment. Another two percent discount is given on orders of 50 units and tree persent discount is given on orders of 100 or more units.

То есть у нас возможны оптовые и розничные покупки. Размер скидки зависит от количества купленных товаров и от способа оплаты, то есть если оплата наличными, то у нас также дополнительная скидка за это предоставляется.

Строим таблицу

## The simplest example – discount table

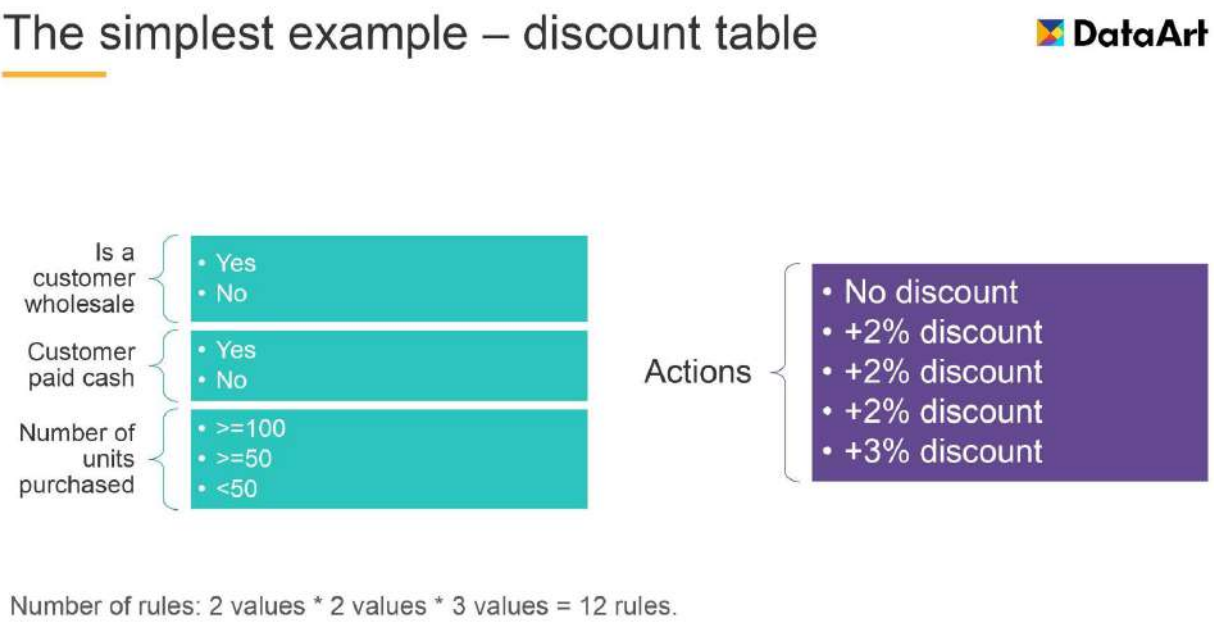
DataArt

| | | |
|---|---|---|
| Customer type | • Wholesale<br>• Retail | Actions: • No discount |
| Customer paid cash | • Yes<br>• No | • +2% discount |
| Number of units purchased | • >=100<br>• >=50<br>• <50 | • +2% discount<br>• +2% discount<br>• +3% discount |

Number of rules: 2 values * 2 values * 3 values = 12 rules.

Сначала нам нужно выбрать условия и выбрать действие. Получается у нас скидка зависит от трех факторов. Мы можем назвать их условиями. Это тип покупателя: оптовый или розничный; способ оплаты: наличными или нет; и количество купленных товаров. Количество сочетаний для этого набора условий, равно произведению количества этих условий, то есть 2 x 2 x 3=12 сочетание условий. По сути каждое сочетание условий задаёт собой некое правило бизнес логики, поэтому их так и называют - бизнес правилом или просто правилом. Набор возможных действий также перечислен на слайде, то есть у нас возможно отсутствие скидки, есть три варианта при которых мы получаем 2-ух процентную скидку дополнительную и возможен вариант с 3ех процентной скидкой. Это один из возможных способов описать действие. Другой мы рассмотрим в конце этого примера, и я расскажу какой из них лучше, а какой хуже и почему.

Итак мы выбрали условия, выбрали действия. Обратите внимание, мы можем условия немного переформулировать:




То есть мы можем сформулировать условие первое так, чтобы оно стало бинарным. В принципе это немного удобнее при оформлении таблицы и немного привычнее тем, кто привык к “олдскул” подходу, когда условия бинарные разницы для конечного результата в этом нет.

Вот наши условия и все варианты их сочетаний:

## The simplest example – discount table

DataArt

| Wholesale | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No | No | No | No |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cash | Yes | Yes | Yes | No | No | No | Yes | Yes | Yes | No | No | No |
| Units | <50 | >=50 | >=100 | <50 | >=50 | >=100 | <50 | >=50 | >=100 | <50 | >=50 | >=100 |

Мы перебираем все, что возможно, записываем действие и для каждого набора условий мы должны внести отметку

-Должно ли это действие быть выполнено для этого набора условий или не должно?

Это будет выглядеть примерно так:

## The simplest example – discount table

DataArt

| Wholesale | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No | No | No | No |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cash | Yes | Yes | Yes | No | No | No | Yes | Yes | Yes | No | No | No |
| Units | <50 | >=50 | >=100 | <50 | >=50 | >=100 | <50 | >=50 | >=100 | <50 | >=50 | >=100 |
| No | | | | | | | | | | X | | |
| +2% | X | X | X | X | X | X | | | | | | |
| +2% | X | X | X | | | | X | X | X | | | |
| +2% | | X | | | X | | | X | | | X | |
| +3% | | | X | | | X | | | X | | | X |

При таком подходе, мы должны просуммировать скидки. Было бы удобнее добавить в снизу строчку “итого”, чтобы конкретное число финальной скидки, получить для каждого набора условий. Мы могли бы действие сформулировать иначе, сказать, что у нас возможные варианты действий: нет скидки, 2% скидка, 3% скидка, 4% скидка, 5% скидка, 6% и 7%, тогда бы наши таблица выглядела вот так:

## The simplest example – discount table

DataArt

| Wholesale | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No | No | No | No |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cash | Yes | Yes | Yes | No | No | No | Yes | Yes | Yes | No | No | No |
| Units | <50 | >=50 | >=100 | <50 | >=50 | >=100 | <50 | >=50 | >=100 | <50 | >=50 | >=100 |
| No | | | | | | | | | | X | | |
| 2% | | | | X | | | X | | | | X | |
| 3% | | | | | | | | | | | | X |
| 4% | X | | | | X | | | X | | | | |
| 5% | | | | | | X | | | X | | | |
| 6% | | X | | | | | | | | | | |
| 7% | | | X | | | | | | | | | |

Почему этот вариант мне нравится меньше, хотя его очевиднее проще построить, чем предыдущий потому, что здесь сложно догадаться, что сформулировать действия стоит вот таким образом, что нужно 2% трижды упомянуть, но если всё-таки догадаться это сделать, то мы получаем возможность видеть в таблице *(вариант приведен ниже)* как формируется эта скидка:

## The simplest example – discount table

DataArt

| | | | Rule | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Wholesale | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No | No | No | No |
| Cash | Yes | Yes | Yes | No | No | No | Yes | Yes | Yes | No | No | No |
| Units | <50 | >=50 | >=100 | <50 | >=50 | >=100 | <50 | >=50 | >=100 | <50 | >=50 | >=100 |
| No | | | | | | | | | | X | | |
| +2% | X | X | X | X | X | X | | | | | | |
| +2% | X | X | X | | | | X | X | X | | | |
| +2% | | X | | | X | | | X | | | X | |
| +3% | | | X | | | X | | | X | | | X |

Откуда взялась именно такая скидка, к сожалению во втором примере этого не видно:

## The simplest example – discount table

DataArt

| Wholesale | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No | No | No | No |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cash | Yes | Yes | Yes | No | No | No | Yes | Yes | Yes | No | No | No |
| Units | <50 | >=50 | >=100 | <50 | >=50 | >=100 | <50 | >=50 | >=100 | <50 | >=50 | >=100 |
| No | | | | | | | | | | X | | |
| 2% | | | | X | | | X | | | | X | |
| 3% | | | | | | | | | | | | X |
| 4% | X | | | | X | | | X | | | | |
| 5% | | | | | | X | | | X | | | |
| 6% | | X | | | | | | | | | | |
| 7% | | | X | | | | | | | | | |

Это сложнее поддерживать, если правила скидок поменяются, нам придётся всю таблицу перерисовывать, а не одно действие какое-то поменять. Это сложнее понимать и разбирать. К примеру у нас есть тест кейс и мы нашли в нём баг, нам нужно обратное действие проделать, понять, от чего этот баг зависит, на каких наборах условии появляется и так далее, а по такой таблице нам это сделать несколько сложнее. Это не обязательно делать по таблице, можно сделать и без нее, но так удобнее. Поэтому этот вариант с точки зрения формулирования условий мне нравится меньше.

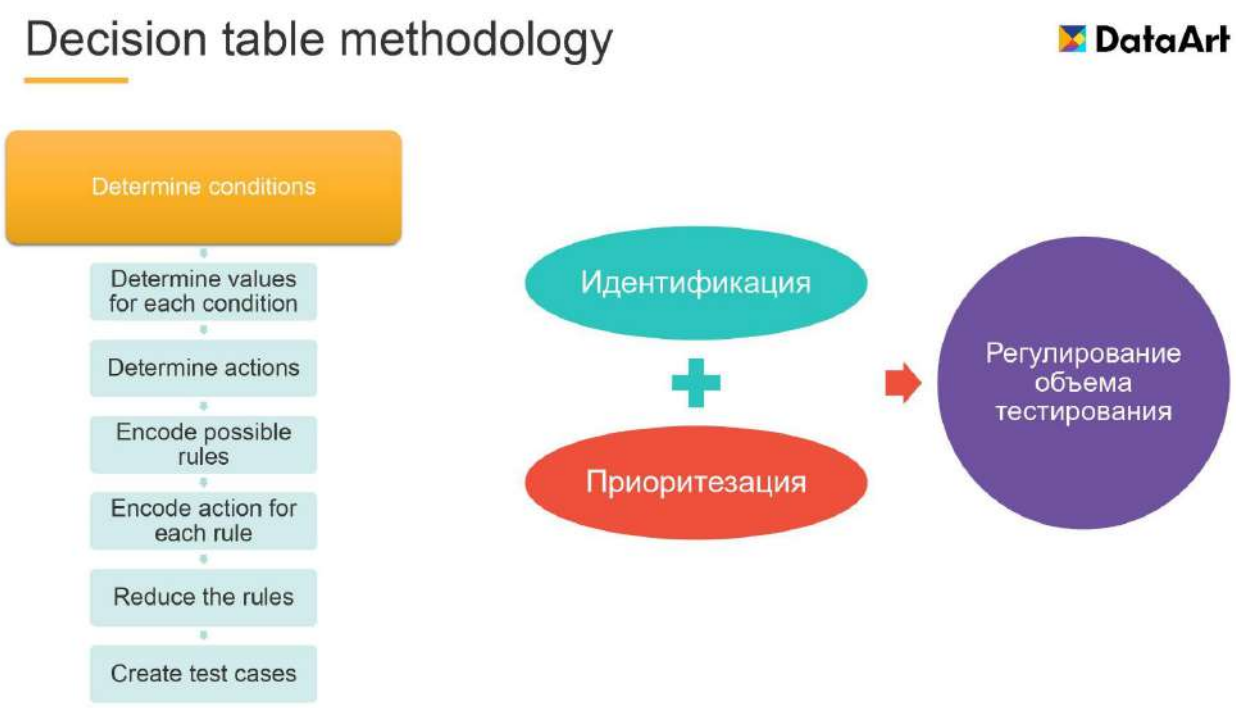
А сейчас давайте рассмотрим общий алгоритм применения этого метода, он на слайде изображён:

## Decision table methodology

DataArt

- Determine conditions
- Determine values for each condition
- Determine actions
- Encode possible rules
- Encode action for each rule
- Reduce the rules
- Create test cases

Первый этап - это определить условия.

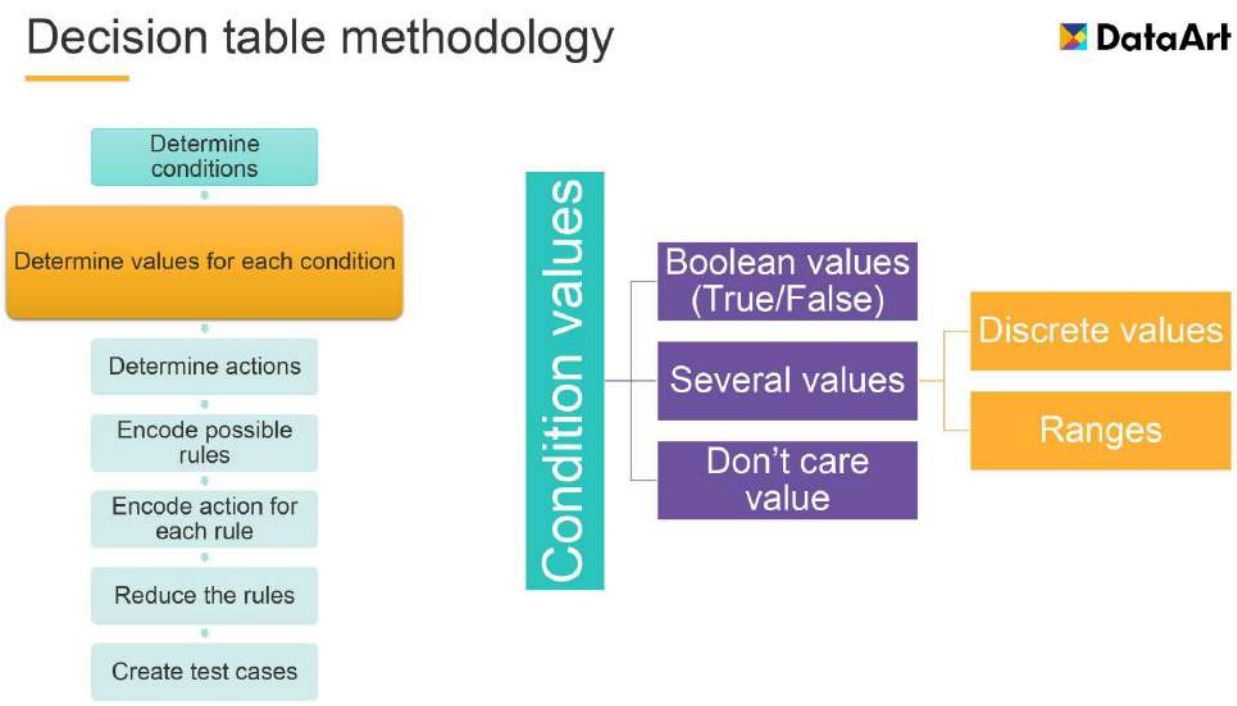



Какие на этом этапе есть особенности:

1. условия мы можем формулировать по-разному (т.е. мы можем объединять несколько условий в одно с набором нескольких значений)
2. мы можем пренебрегать менее приоритетными условиями, мы можем на разных уровнях формулировать (пример: Форму логина: мы можем сформулировать как: введён существующий логин и введён соответствующий логину пароль и у нас получается 4 возможных сочетания, т.е. фактически 4 теста) 4 теста так как 2 условия для логина (существующий и соответствующий) и для логина (подходящий и нет) 2*2 =4
3. Но, мы можем учесть и пустой логин или получится три варианта (логин пустой, логин существующий, логин несуществующий)
4. та же история с паролем (соответствует, не соответствует, пустое поле)

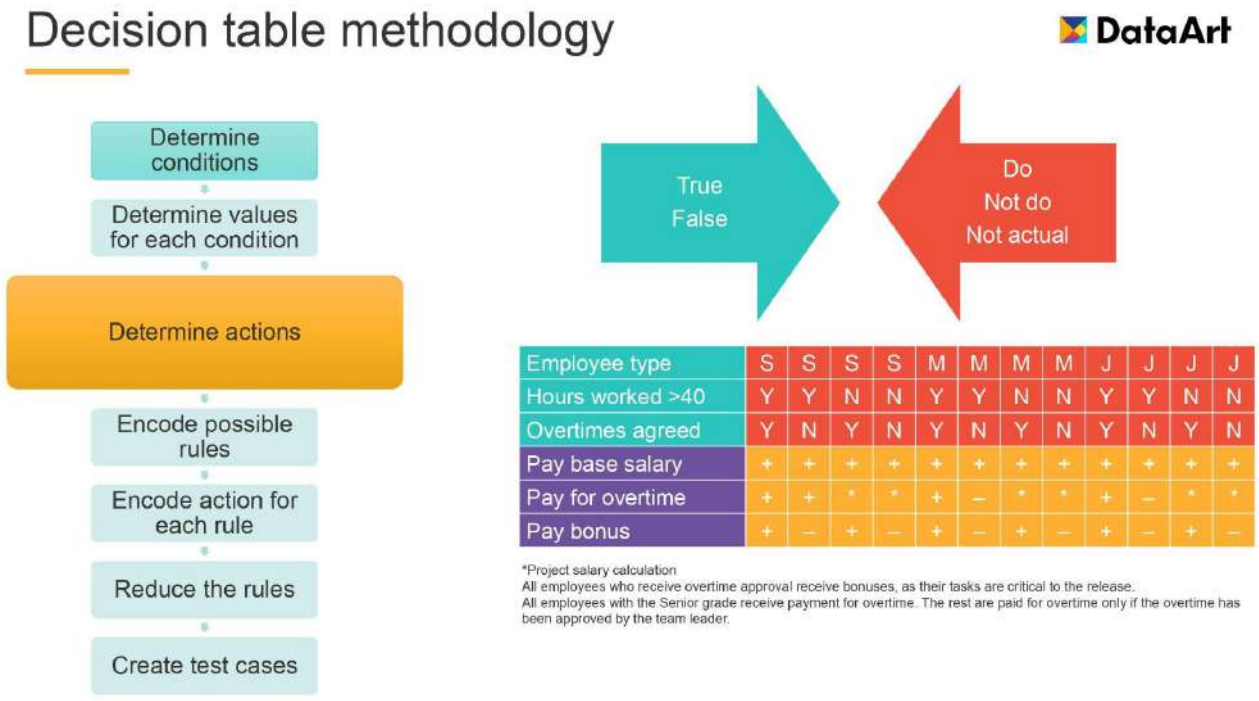
В таком случае набор тестов получается больше, покрытие лучше, но объём тестирование выше,дольше, дороже и всё что из этого следует. То есть это инструмент, которым вы можете регулировать объем тестирования. Он на стадии условий таким образом работает, с действиями оно тоже отчасти работает, но в меньшей степени. С действиями мы просто можем какие-то действия считать важным, значительным и проверять, а какие-то не проверять, это влияет не на количество написанных тестов, а на время прохождения каждого теста.

Итак следующий этап - определить значения возможные для каждого из условий.




Самый простой вариант это значение True/False. Это может быть некий набор значений, он может быть, как дискретным, так и задан набором диапазонов например. И ещё чуть позже мы рассмотрим пример, где значение означает что это условие не важно. *(Don't care value)*

Следующий этап - определение действий:




Здесь есть два подхода также. Тот пример, который мы с вами рассмотрели про скидку хорош тем, что скидка не может быть применена на половину или нас не интересует применена она или не применена, то есть для каждого набора условий каждый вариант скидки или есть или нет, строго два возможных варианта для этого действия, оно либо выполнено, либо не выполнено. Так работает не во всех случаях, бывает ситуация, в которой нам к какому-то сочетанию условий действие неприменимы,

не можем проверить выполнена она или не выполнено. Бывают ситуации, когда нам нужно просто проверять, что действие выполнено и этого достаточно, То есть вот мы взяли некое правило, первый столбец из правой части таблицы и просто проверили, что каждое действие выполнено. Вопрос, вот для следующего второго столбца, там где стоит минус, нам надо это проверить или не надо? Здесь я старалась, таким образом придумать, пусть даже несколько притянутый за уши пример, чтобы это нужно было явно проверить.

В этом примере говориться, что у нас есть зарплата условно за месяц, есть разные грейды у сотрудников (синьор, миддл, джун), у нас есть количество отработанных часов, оно либо меньше или равно 40, либо с переработками т.е. больше 40 часов за неделю и у нас есть метка согласованы или несогласованны овертаймы для этого сотрудника. И таким образом, у нас любому сотруднику платится его базовая зарплата, а дальше, по выдуманной мной логике, овертаймы сеньорам оплачиваются в любом случае, а мидлам и джунам выплачиваются только, если они согласованы.

Таким образом получается, что для второго например столбца этой таблицы, нам нужно явным образом проверить, что если овертаймы были не согласованы, то бонус этому сотруднику не оплачивается потому, что он может оказаться добавлен к зарплате, может не оказаться и это существенно для результата выполнения теста. Нам это надо явно проверить. Звёздочками отмечены действия, где мы не проверяем это действия потому, что овертаймы не оплачиваются. Таким образом, у нас действий для возможных состояний может быть не два, может быть больше, в данном примере, здесь три возможных состояния: действие должно быть выполнено, не должно быть выполнено *(мы это явно в ожидаемых результатов кейса проверяем)* и неактуально, тогда мы не проверяем.

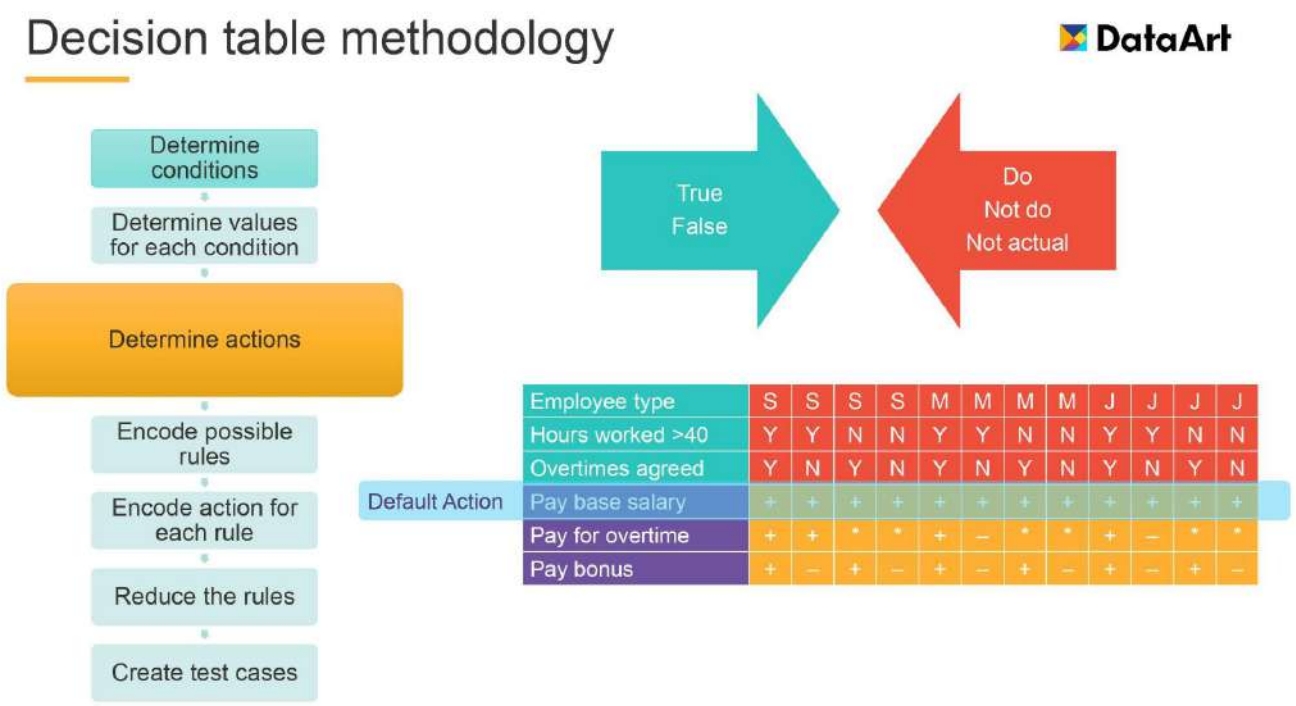
Для предыдущего случая, нам не нужно вообще ничего проверять там, где у нас не отмечено действие крестиком. То есть мы просто в ожидаемый результат проверяем ровно те действия, которое для этого сочетания условий отмечено крестиком. По остальным действиям просто не делаем ничего.

## The simplest example – discount table

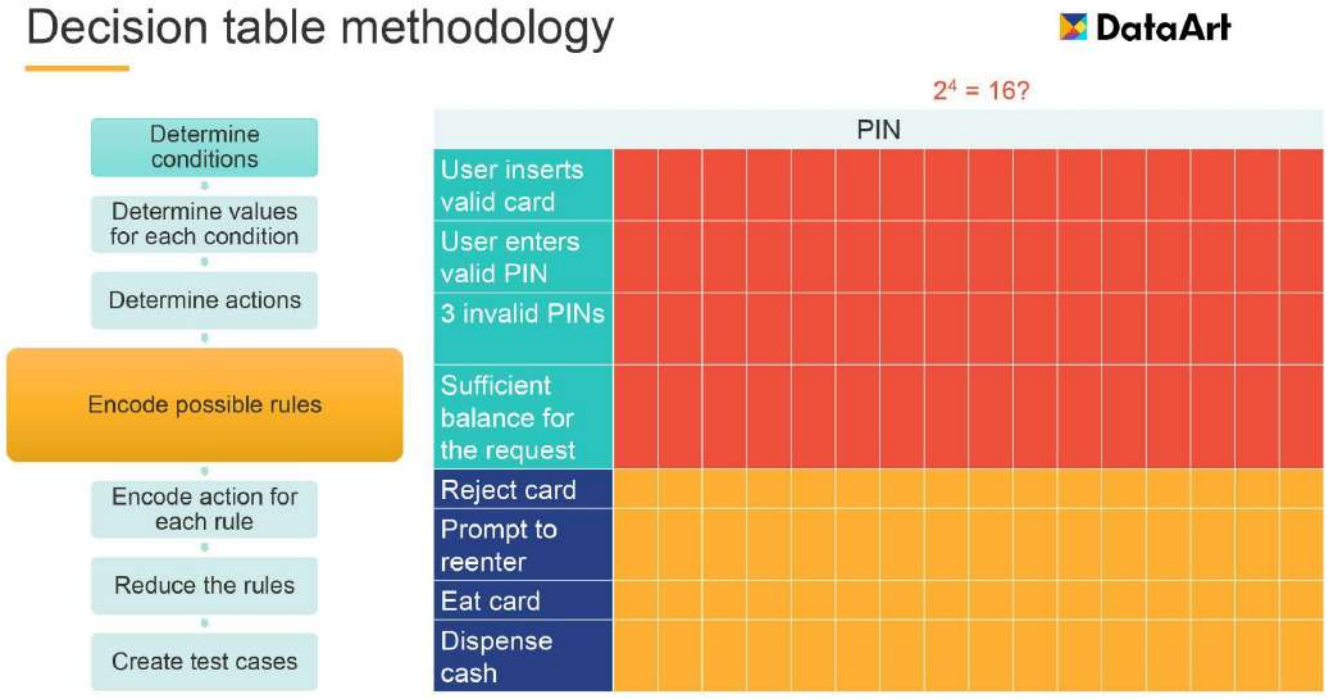
DataArt

| Wholesale | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No | No | No | No |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cash | Yes | Yes | Yes | No | No | No | Yes | Yes | Yes | No | No | No |
| Units | <50 | >=50 | >=100 | <50 | >=50 | >=100 | <50 | >=50 | >=100 | <50 | >=50 | >=100 |
| No | | | | | | | | | | X | | |
| 2% | | | | X | | | X | | | | X | |
| 3% | | | | | | | | | | | | X |
| 4% | X | | | | X | | | X | | | | |
| 5% | | | | | | X | | | X | | | |
| 6% | | X | | | | | | | | | | |
| 7% | | | X | | | | | | | | | |

Итак обратите внимание на ещё одну интересную вещь в этом примере.

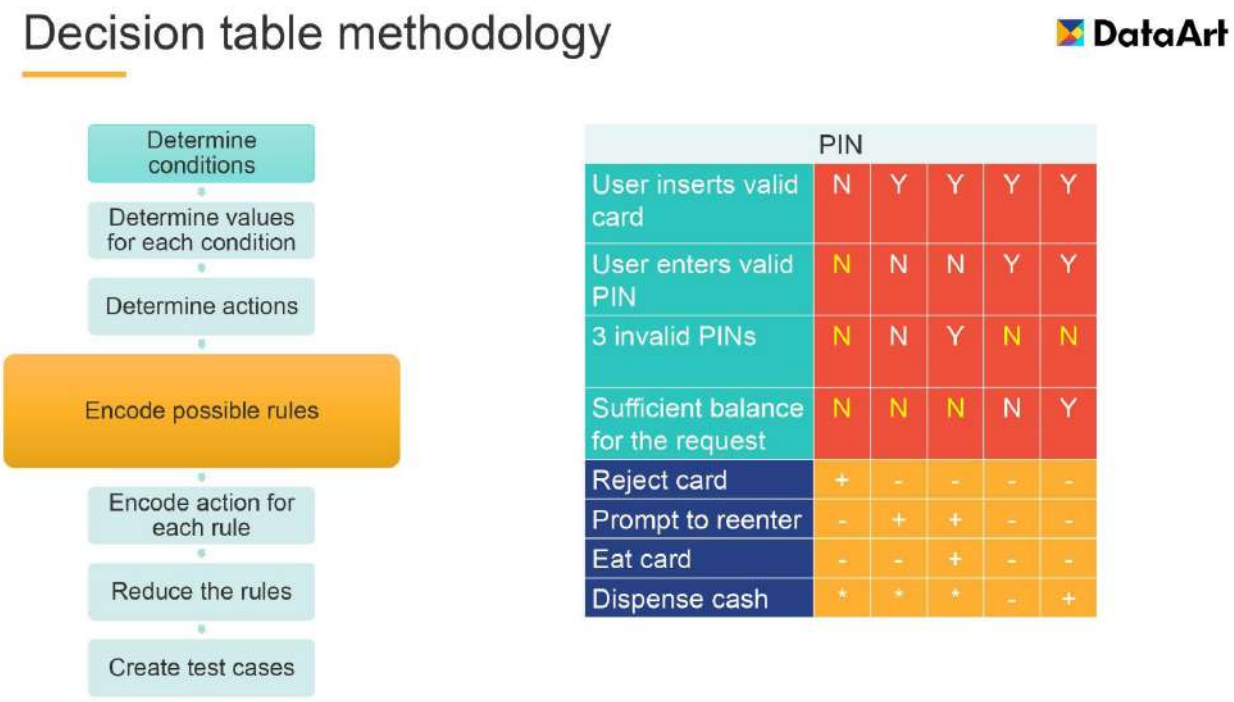



Здесь есть действие, которое выполняется для всех кейсов, так называемый Default Action. Желательно, чтобы оно было определено, если оно есть. Тогда его можно в принципе в таблицу не включать, т.к. оно может быть записано где-то еще и это немного упрощает работу, но тут пример маленький и простой, его упрощать дальше некуда. Таких действий может быть больше одного в больших примерах и такие ситуации отслеживать имеет смысл, чтобы упростить таблицу, с ней так легче работать.




Следующий этап, который необходимо выполнить после того, как мы определили условия и действия - это записать возможные правила. Что значит возможные? Не все варианты реализации условий совместимы между собой. Отбрасывая несовместимые и не реализуемые варианты сочетаний условий, мы существенно сокращаем таблицу.

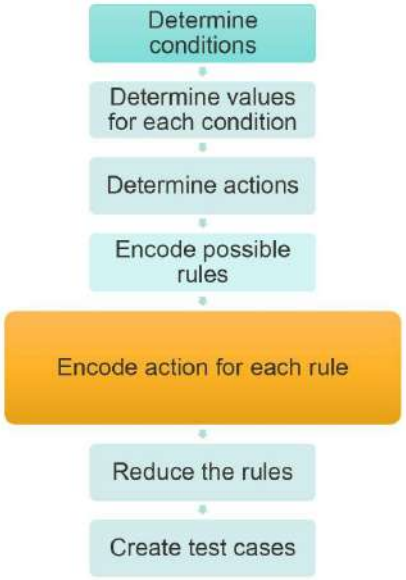
На слайде выше представлен пример, на котором у нас 4 условия, они бинарные, вроде бы подразумевается, что должно быть 2 в 4 степени = 16 правил. Однако, если посмотреть на сами условия, то можно увидеть, что например, если карта не вставлена, то мы не можем ввести, ни валидный пин, ни невалидный пин, ни тем более узнать результат запрос баланса по операции. Соответственно, если верхнее условий не выполнено, то все остальные варианты с ним не сочетаемы. Итак, мы уже сократили таблицу почти в два раза и из 16 получаем 9 кейсов, сокращая последовательно таким образом эту таблицу, мы получим такое количество кейсов как на слайде:




Берём первый столбец, тот самый, когда у нас карта невалидна и поэтому нет смысла рассматривать сочетания остальных условий. Но подобрать подходящие их правильные значения всё-таки стоит. В данном конкретном примере это неважно потому, что попросту не реализуемо, но есть примеры, в которых это важно. То есть здесь, значению false для первого условия, должно соответствовать значению false для всех остальных условий. Потому, что если в любом из них будет значение True, то такое правило становится бессмысленным. Рекомендуется так делать даже там, где это не важно потому, что это упрощает чтение таблицы, например через три месяца, для вас или для человека, который к нам пришёл в проект. Мы подразумеваем, что это не одноразовое действие, мы построили таблицу, написали по ней тест кейсы и выбросили ее в мусорное ведро. Скорее всего, вам придется ей пользоваться дальше при изменениях требований, при ревью тестов и будет полезно, если она будет читаема и поддерживаема. Желтым помечены те условия в правилах, которые несочетаемы с остальными и для которых, мы подобрали значение, адекватно отображающее бизнес логику.

## Decision table methodology

DataArt



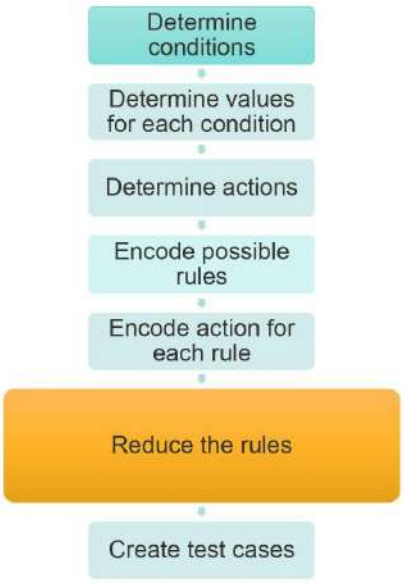

| PIN | | | | | |
|---|---|---|---|---|---|
| User inserts valid card | N | Y | Y | Y | Y |
| User enters valid PIN | N | N | N | Y | Y |
| 3 invalid PINs | N | N | Y | N | N |
| Sufficient balance for the request | N | N | N | N | Y |
| Reject card | + | - | - | - | - |
| Prompt to reenter | - | + | + | - | - |
| Eat card | - | - | + | - | - |
| Dispense cash | * | * | * | - | + |

Следующий этап - это определить действие для каждого правила. Здесь мы как раз видим тот пример, когда у нас это действие неприменимо к соответствующему набору условий. То есть мы не можем его проверить. В примере “+” значит, что мы должны проверить, что действие выполнено, “-” мы должны проверить, что действие не выполнено, “*” означает, что мы не можем проверить это действие ни в ту, ни в другую сторону.

Следующий этап - упростить правила. Помимо того, что у нас есть правила внутри которых условия несовместимы - мы их должны просто выкинуть, у нас может существовать еще несколько сочетание условий, таких, что фактический результат зависит не от всего набора условий, а от пары условий всего. Остальные в данном случае несущественны, т.е. их изменение, не влияет на конечный набор действий. Анализируя это, мы можем еще сократить таблицу и вот красивый пример на эту тему.

## Decision table methodology

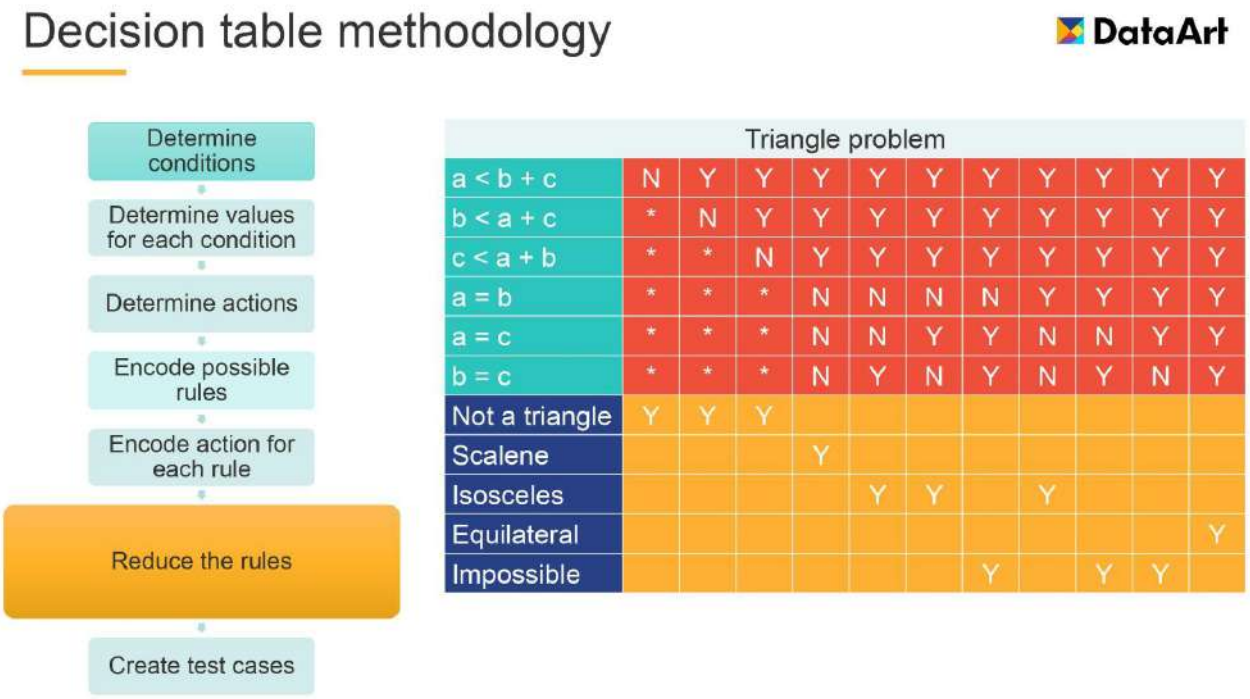
DataArt




$2^6 = 64?$

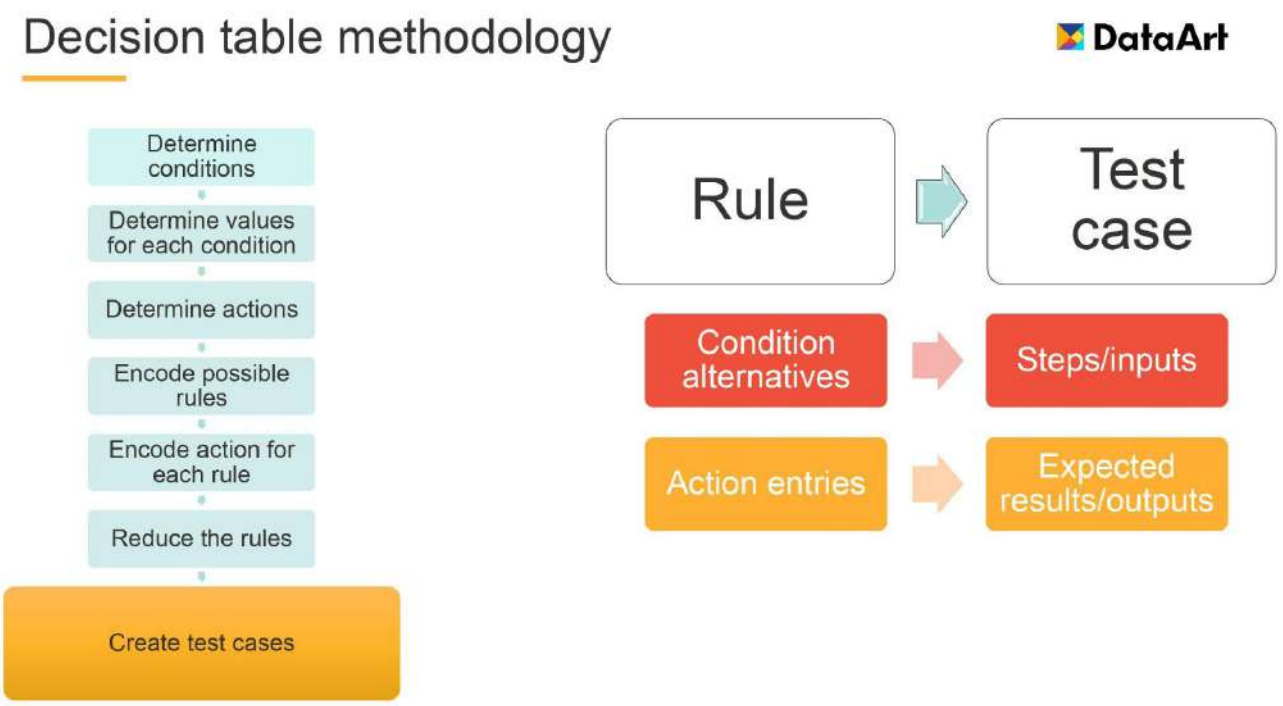
| Triangle problem | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a < b + c | | | | | | | | | | | | |
| b < a + c | | | | | | | | | | | | |
| c < a + b | | | | | | | | | | | | |
| a = b | | | | | | | | | | | | |
| a = c | | | | | | | | | | | | |
| b = c | | | | | | | | | | | | |
| Not a triangle | | | | | | | | | | | | |
| Scalene | | | | | | | | | | | | |
| Isosceles | | | | | | | | | | | | |
| Equilateral | | | | | | | | | | | | |
| Impossible | | | | | | | | | | | | |

Правило треугольника даны A, B и C. Необходимо проверить существует ли треугольник с такими сторонами. Длина каждой стороны должна быть меньше суммы длин других сторон. И если стороны равны то это вырожденный случай и это прямая, а не треугольник. Также здесь мы определяем тип треугольника: обычный треугольник, равнобедренный, равносторонний и невозможность сочетания. Итак, в примере есть 6 условий, сами условия бинарные и это подразумевает 64 теста.

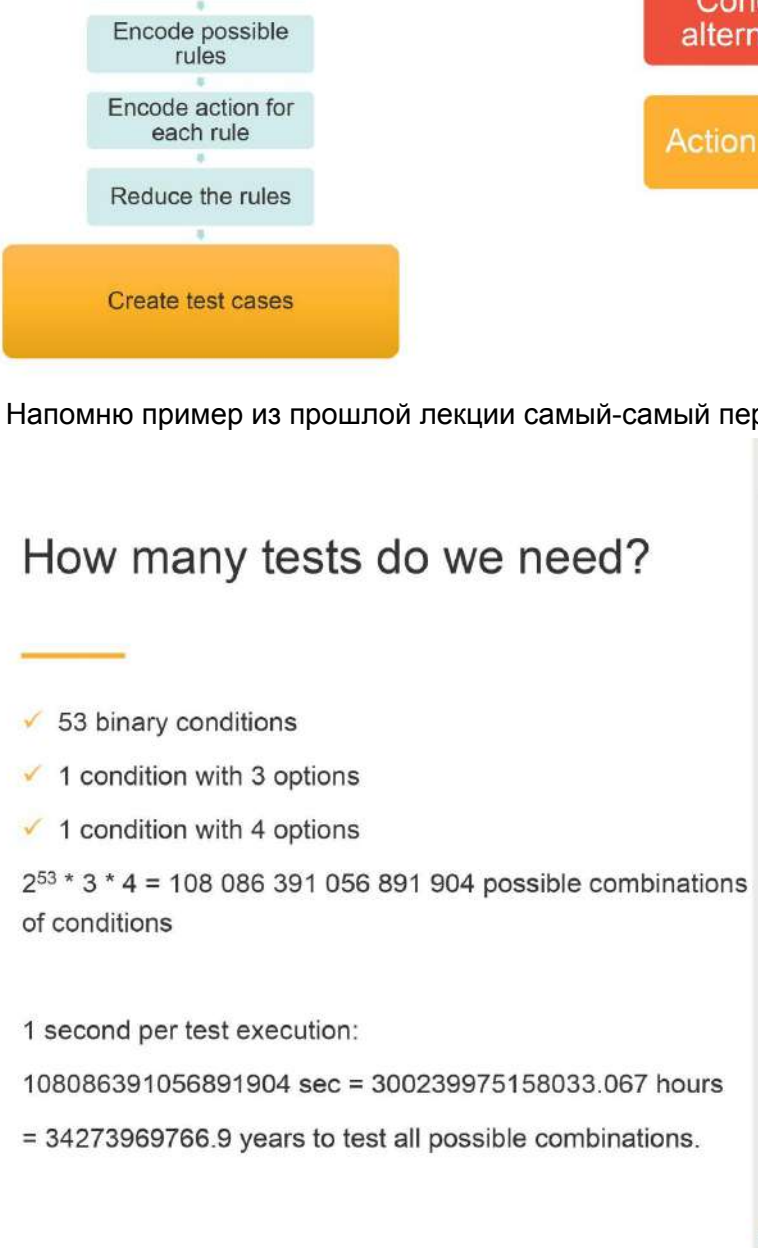


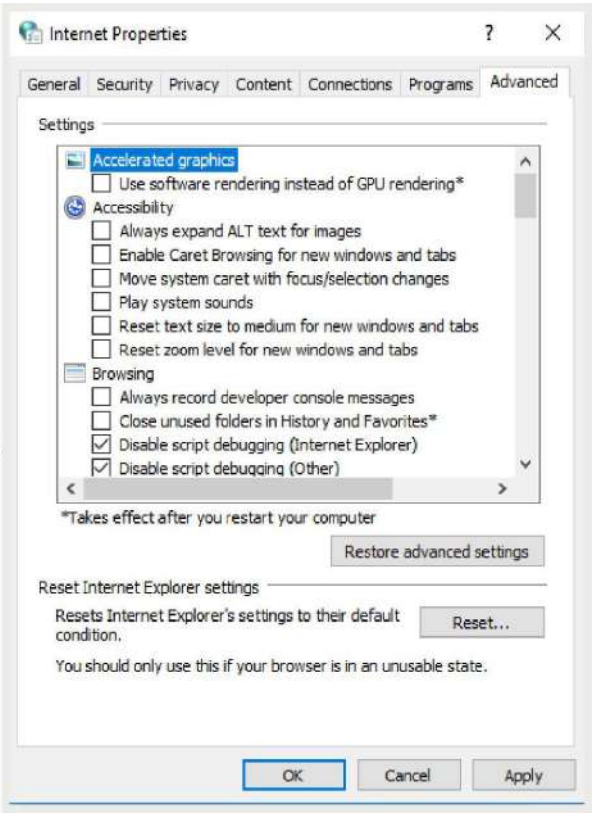

Более подробно можно рассмотреть на слайде. Если у нас длина стороны превышает сумму длин двух других сторон, нужно ли нам что-то ещё для того, чтобы определить является ли фигура треугольником? Нет, не нужно! То есть нам не нужно рассматривать остальные сочетания, нам хватит знания о том, что это правило нарушено. Соответственно результат конечный - этот набор действий, в котором мы определяем, что это не треугольник, зависит только от первого условия и больше ни от каких. Аналогично следующие пункты. Таким образом вместо огромного набора тестов мы получаем 11 тестов.

Последний этап - это написание тест кейсов по таблице на слайде выше. Каждый столбец таблицы - это тест кейс, то есть он кодирует бизнес правило, которое мы проверяем. Набор сочетания вариантов реализации условий - кодирует входные данные, то есть мы должны подобрать входные данные и сформулировать шаги таким образом, чтобы выполнился именно этот набор условий. А соответственно и нижняя половина существования действий - кодирует выходные данные. То есть то, что мы будем указывать в ожидаемом результате тест кейса. При наличии данной таблицы в данном конкретном случае мы получаем 11 тест кейсов и знаем примерно, что должно быть в шагах и что должно быть в ожидаемых результатах.




Напомню пример из прошлой лекции самый-самый первый.

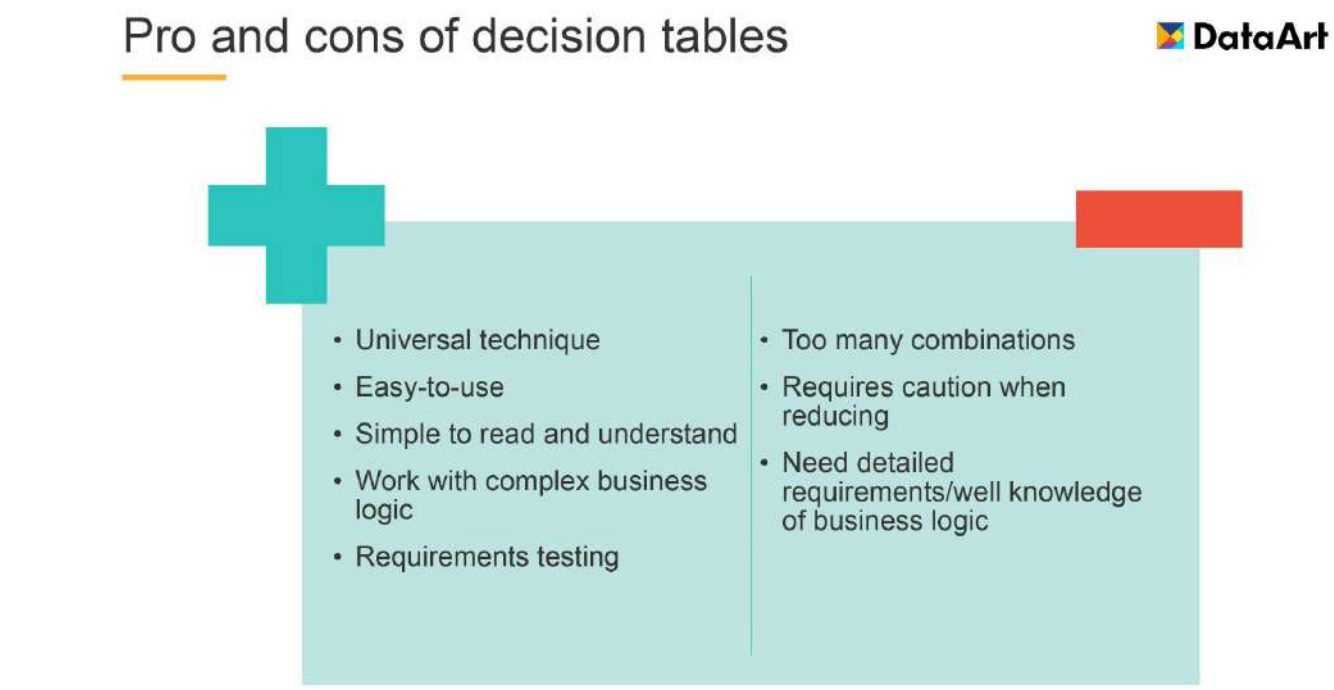



Пример с окошком сетевых настроек Windows. Когда мы рассматривали это окошко, ни один из методов, которые мы рассматривали на прошлой лекции не позволял нам сократить огромное количество тестов в данном случае, то есть pairwise сократил бы его наверное раза в три, что не очень нам помогает в данном случае. Эквивалентное разбиение заставило бы нас всё также проверять все сочетания, анализ граничных значений здесь не применим, а pairwise хорошо работает, когда у нас есть большие наборы условий, а здесь у нас все условия бинарные и перебор пар нас привёл бы почти к такому же результату, так как у нас не бинарных условий всего 2.

Как к этому подойти? Скорее всего, условия в этом окошке, не то чтобы взаимосвязаны. Скорее всего их как раз достаточно легко разбить на группы и конечные результаты *(действия)* будут зависеть

в 99% случаев не от всего набора условий, а от какой-то части. И именно таблица решений нам позволит это выявить и сократить набор тест.




Этот метод на мой взгляд самый полезный, он просто в применение, он подходит почти к любой системе, почти к любой функциональности и в общем-то он позволяет регулировать объем тестирования. Еще один важный момент, рассматривая сочетания условий, вы для многих подобных сочетаний ожидаемого результата в требованиях не найдёте. То есть применение этой техники тест-дизайна - это неплохой способ тестирования требований сам по себе. Вы сразу обнаруживаете пробелы в требованиях, не находя для какого-то набора условий собственного ожидаемого поведения. Но у него есть ряд особенностей, которые нужно учитывать применяя этот метод (есть также и минусы)

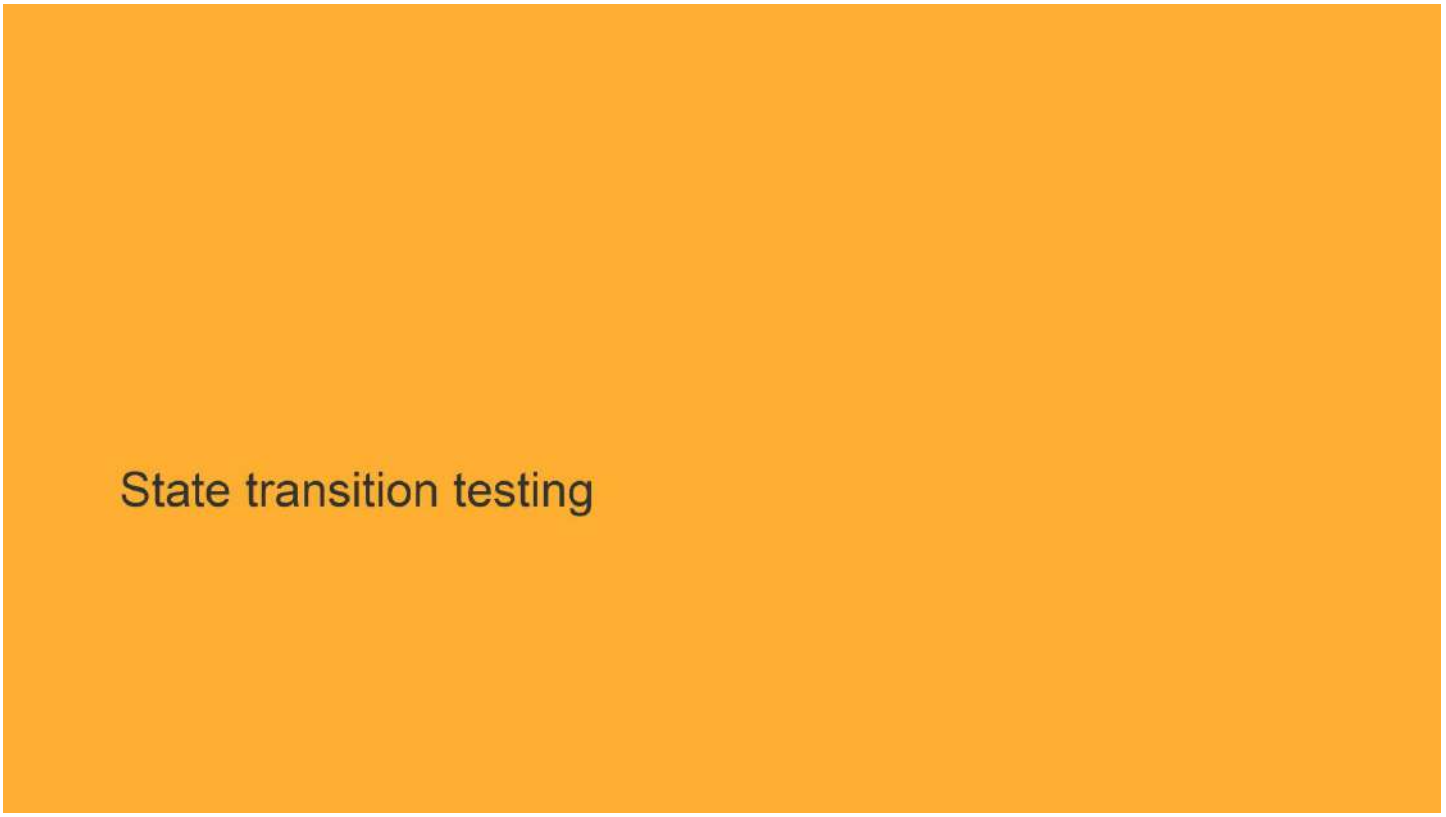
Во-первых, комбинаций может быть очень много. Из реального опыта, когда мы эти таблицы рисовали, они были огромными - это была эквайринговая система, там важно было учесть все возможные условия, потому, что это банковская система и в ней выкидывать тест-кейсы нельзя. Поэтому количество этих столбцов в таблицы уходило за две сотни, уже после всех сокращений, выкидывания несовместимого и прочего. Собственно используя этот инструмент, мы получили полный набор тестов и были достаточно уверены в надёжности системы, чтобы согласовать это с заказчиком. Тем не менее, количество комбинаций огромное, и если условия вам одинаково важны или вы не можете их приоритизировать, то у вас будет проблема в объёме тестирования.

Когда мы пытаемся сократить таблицу, мы иногда теряем что-то важное, то есть когда мы пытаемся приоритизировать и убирать условия, мы очевидно пренебрегаем какими то кейсами. То есть мы предполагаем, что они мало приоритетные, но тут есть важная тонкость. Приоритеты тест кейсов, не всегда гарантируют вам, что дефекты, которыми мы в этом случае пренебрегли, окажутся не значимыми для пользователя. Мы стараемся выстраивать приоритеты так, чтобы баг, на проверку отсутствия которого нацелен тест кейс и приоритет тест-кейса, были более или менее равнозначны. То есть, если выполнение этого сценария критически важно, значит у теста будет высокий приоритет, если выполнение того или иного сценария для пользователей менее важно, приоритет будет соответственно ниже. Но, это не означает, что ни один пользователь на это не наткнется, и мы не знаем как проявится дефект в этом сценарии, т.е. мы предполагаем, что пользователь не сможет выполнить этот сценарий, а что при этом произойдет в системе, мы не проверяли, раз мы пренебрегли этим тест кейсом, может быть система упала, может больше не поднимается потому, что там база данных “побилась”. То есть кейс может быть маловероятным, но масштабы бедствия огромные. И при написание тест-кейсов мы это не можем учесть, то есть мы можем учесть насколько тот или иной сценарий важен для пользователя, мы можем учесть насколько сценарий важен для системы. Но мы не знаем по какой

причине дефект появится, поэтому не можем предугадать масштабы бедствия в результате возникновения дефекта в этом сценарии. То есть мы знаем про этот сценарий только то, что пользователь не сможет какую-то свою цель выполнить при этих условиях. На этой основе можно выставлять приоритеты, остальное мы не знаем, соответственно всегда, когда мы пренебрегаем какими-то тестами возникают риски и про них надо помнить и их надо учитывать.

Ну и наконец для того, чтобы построить хорошие таблицы решений, вам необходимы качественные требования или какой-то другой источник информации. Качественные требования в современных проектах скорее редкость потому, что у нас Agile у нас же люди и их взаимодействие важнее документации. Но если у вас не качественные требования, вам нужно использовать взаимодействие  и это существенно усложняет использование этого метода потому, что вам требуется постоянно общаться либо с аналитиком, либо заказчиком, либо с ними со всеми вместе и в общем-то это оправданно и правильно, но это делает время написание тест-кейсов плохо прогнозируемым.

**Метод Таблица Переходов**

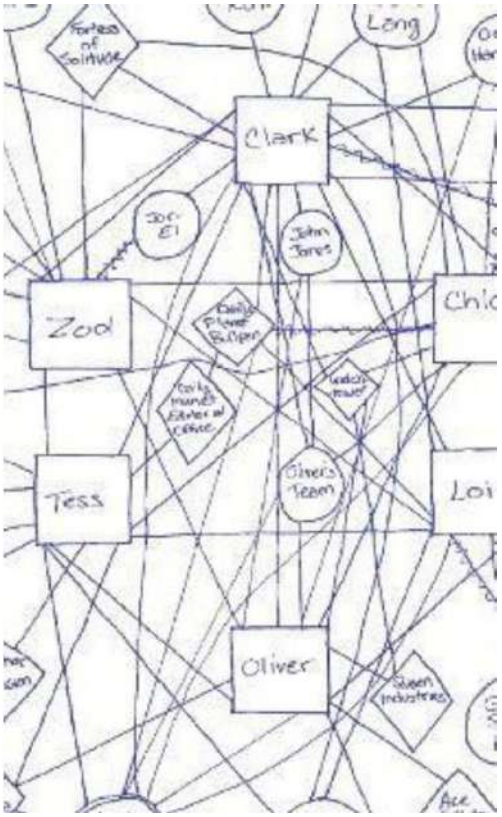



Следующая методология тестирования, которую рассмотрим - это таблицы переходов. Это метод, основанный на конечных автоматах. Если работу системы или подсистемы можно представить в виде конечного автомата, то этот метод применим и будет достаточно полезен.

## Base ideas

The operation of the system can be represented in the form of a state machine

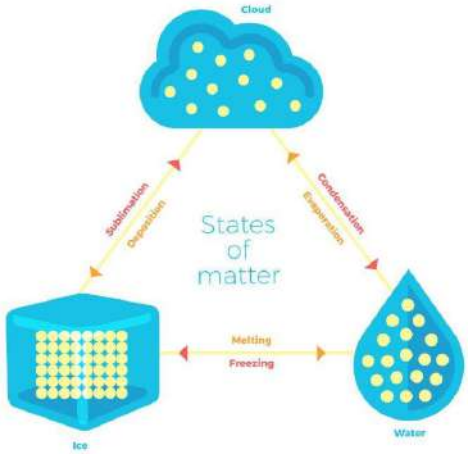
State transition tables




Что такое конечный автомат? Конечным автоматом называют систему, работу которой можно представить в виде конечного набора состояний, с конечным набором переходов между ними. То есть метод применим тогда, когда мы можем выделить некий набор состояний нашего объекта тестирования: система, подсистема или объект в системе, и у нас есть информация о правилах перехода из одного состояния в другое.

## What do we have to know about state machine?

DataArt




**States**
- What state does the software might get?

**Transition**
- Changing a system state from one to another

**Events**
- Reasons for change of state (internal & external)

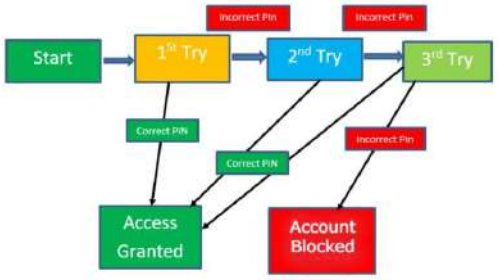
**Actions**
- Operation initiated as a result of a state change

Например, на слайде представлено состояние вещества, то есть вода может превращаться в лед, может превращаться в пар и у нас есть условия, при которых этот переход осуществляется. В рамках конечного автомата *(в рамках* Statechart *диаграммы)*, есть следующий набор объектов: это непосредственно сами состояния, переходы из состояния в состояние, условия, при которых осуществляется переход и действия, которые выполняет система при возникновении этого перехода. Что отсутствует на картинке из этого списка? У нас есть набор объектов необходимых для того, чтобы

схема была корректной диаграммой переходов Statechart диаграммой. У нас есть картинка, является ли она корректной Statechart диаграммой? Тут нет event'ов *(условий)*, состояния обозначены, стрелочки-переходы обозначены, действия (таяние, замерзание) здесь также обозначены. А вот условий нет, а это на самом деле второе по важности после самих состояний.

## How to represent a state machine?

DataArt




| Initial state | Event | Final state | Action |
|---|---|---|---|
| 1st Try | Correct PIN | Access | Home screen |
| 1st Try | Incorrect PIN | 2nd Try | PIN Error |
| 2nd Try | Correct PIN | Access | Home screen |
| 2nd Try | Incorrect PIN | 3rd Try | PIN Error |
| 3rd Try | Correct PIN | Access | Home screen |
| 3rd Try | Incorrect PIN | Blocked | Blocked Error |
| Access | Correct PIN | - | - |
| Access | Incorrect PIN | - | - |
| Blocked | Correct PIN | - | - |
| Blocked | Incorrect PIN | - | - |

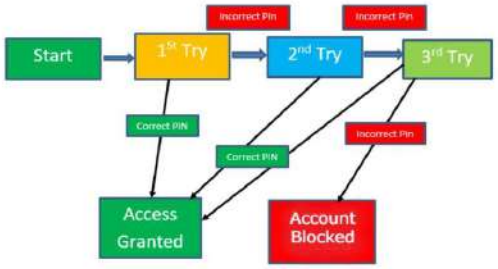
Второй пример. Здесь у нас нет действий, но это как раз не обязательная часть диаграмм. Конечный автомат можно представить двумя способами:

1. в виде графа *(на слайде это картинка слева)*, на котором обозначены состояния, стрелками обозначены переходы и подписями к этим стрелочкам условия и действия.
2. в виде таблицы, которую можно представить двумя способами:
    - первый способ мы рассматриваем все возможные пары "состояние - событие", и для каждой пары "состояние - событие", указываем новое состояние, в которое система переходит при возникновении этого события и выполняемые действия.
    - второй способ на слайде ниже, когда мы строим матрицу по вертикали и по горизонтали, соответственно события и состояния, а на пересечении указываем новое состояние и действие.

Мы потом поговорим потом обговорим какой вариант лучше, а сейчас поговорим о том, как с этим работать.

## How to represent a state machine?

DataArt



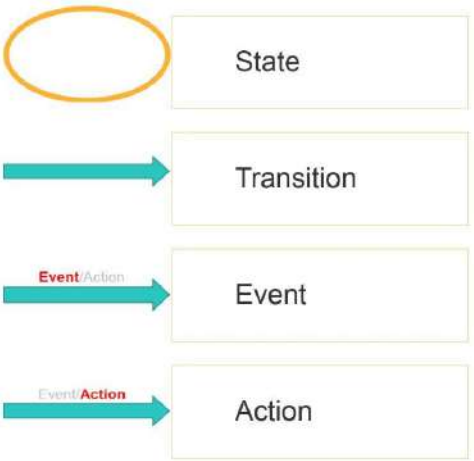

| | Correct Pin | Incorrect PIN |
|---|---|---|
| 1st Try | **Access** Home screen | **2nd Try** PIN Error |
| 2nd Try | **Access** Home screen | **3rd Try** PIN Error |
| 3rd Try | **Access** Home screen | **Blocked** Blocked Error |
| Access | - | - |
| Blocked | - | - |

Итак как если мы хотим нарисовать диаграмму, то есть прекрасный язык UML предназначенный для описания в виде диаграммы - поведение сложных программных систем.

## How to draw the diagram?

DataArt

- ✓ Choose an object
- ✓ Identify states and draw them in circles
- ✓ Define transitions and connect states with arrows
- ✓ Identify events
- ✓ Sign them under/above the arrows.
- ✓ Define actions for each event.
- ✓ Sign them next to the corresponding event




Один из типов UML диаграмм это как раз Statechart диаграммы и они оформляются по перечисленным на слайде правилам. То есть кружочками обозначаем состояние, стрелочками - переходы, и к переходам подписываем через разделительную черту, события и действия.

# Typical mistakes

DataArt


Вместо объекта — GUI

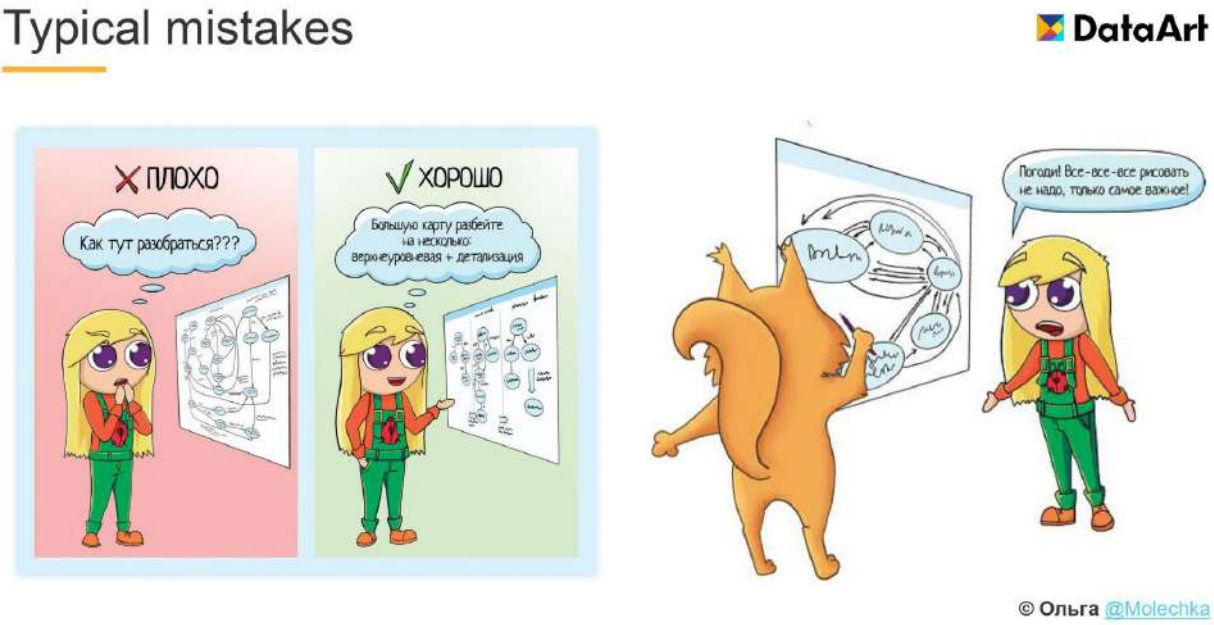

Несколько объектов


© Ольга @Molechka

Когда мы рисуем Statechart диаграмму, какие типичные ошибки мы совершаем? Когда мы вместо объекта в системе описываем пользовательский интерфейс Например у нас есть оформление заказа в интернет-магазине. Его возможные состояния: создан, оплачен, доставлен и, возможно, ещё какие-то. Вот они именно так и должны обозначаться и не должно быть обозначений, по типу: переход к форме выбора способа оплаты, переход к форме выбора способа доставки или что-то в этом роде - это типичная ошибка и если мы пытаемся опираться на интерфейс, то дальше мы фактически рисуем диаграмму переходов для интерфейсов. Это полезно для UI тестирования, но бесполезно для тестирования бизнес логики потому, что в принципе мы как раз это и должны проверять, что в правильное окошко переходим, а мы это как условие задаем.

Вторая ошибка - это когда мы путаем разные объекты. Вот пример на слайде с заказом пиццы. То есть, когда мы рисуем диаграмму для 1 объекта в системе, мы не можем нарисовать диаграмму, в которой объект изменяется по пути, то есть мы для одного какого-то объекта фиксируем набор состояний, иначе этот подход не работает.

И наконец еще одна типичная ошибка - это когда мы дублируем состояния для заказов. Нам важно один товар добавлен или несколько? Нет! В этот момент заказа еще не существует. Поэтому состояние “товары добавлены в корзину” к заказу, к сущности заказа, к объекту заказа, отношения не имеет - заказа нет. И пока пользователь не нажал кнопку “оформить заказ” - заказа нет. Частая ошибка - это придумать кучу состояний, например в корзине нет товара, в корзину добавлено столько-то товаров - это эквивалентное с точки зрения объекта заказа состояние. Когда мы попытаемся потом между этими состояниями как-то фиксировать переходы, а потом начнём всё это собирать в одну кучу и нам будет очень неудобно. У нас будут получаться странные сочетания состояний и событий потому, что мы рассматриваем состояния и события по сути для разных объектов. Этот пример иллюстрирует сразу две ошибки, здесь мы смешали сразу два объекта: корзину и заказ, и тут мы с точки зрения заказ сделали несколько эквивалентных состояний.

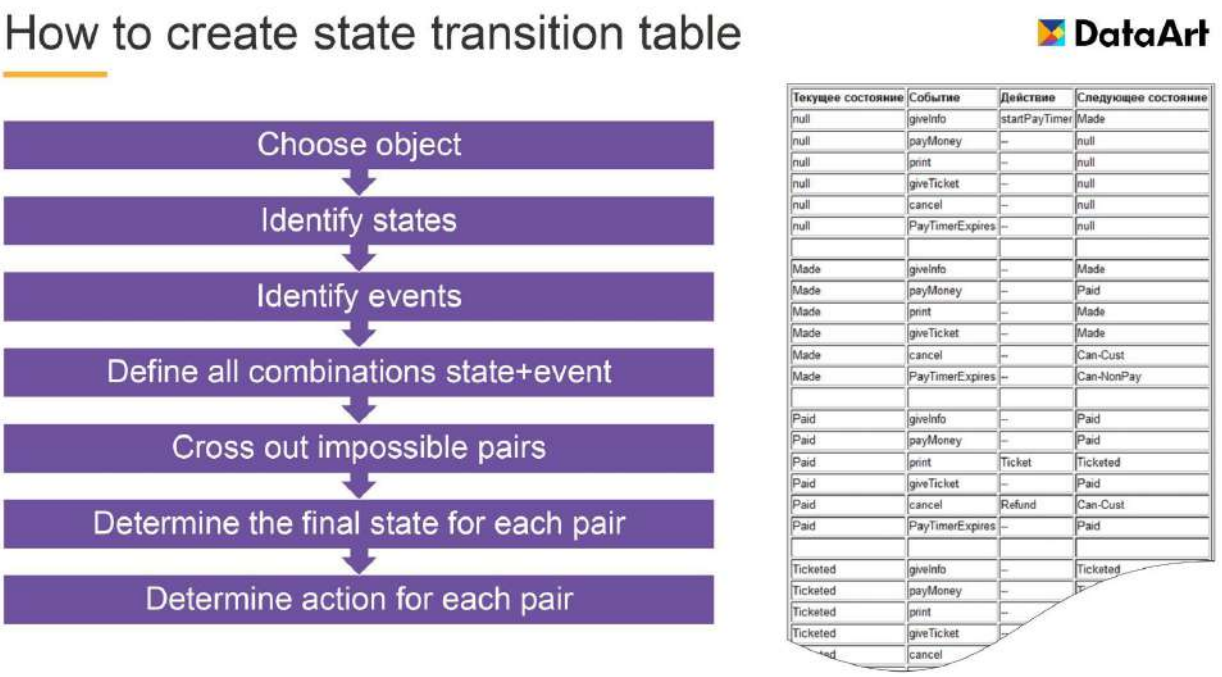



Ещё про ошибки! Для сложных бизнес-процессов, набор состояние в системе может оказаться большим и тогда диаграмма становится чрезмерно сложной. Работать с ней неудобно и чаще всего не очень полезно, то есть вряд ли вы будете строить тесты, которые будут пробегать такие сложные наборы состоянии за один раз, т.е. можно взять не самого нижнего уровня объекты для рассмотрения, а взять некую группирующую сущность и для неё построить свой Flow. Построить иерархию сущности и для каждой сущности построить свой Flow, свою диаграмму состояний.

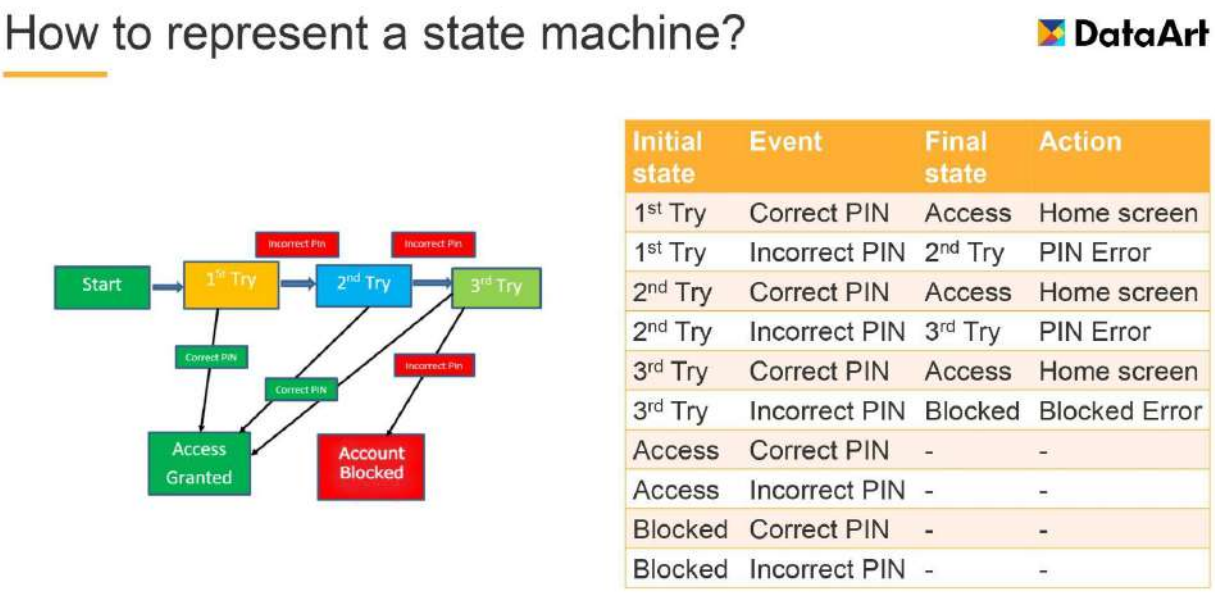
*Пример из жизни. Работала я с системой, для которой мне нужно было тестировать отказоустойчивость. Это очень сложная штука, потому, что зависит от состояние системы зависит от очень многих вещей. Например: количество памяти, возникновении каких-то событий, сбоев в системе, а сбои бывают разных типов, у нас может не отвечать один сервис или может не отвечать другой сервис может не отвечать СУБД. Это все огромное количество возможных событий и довольно большое количество состояний. А если разбить всё это на несколько уровней иерархии, рассмотреть отдельно состояние связанное с базой данных например, то есть отдельно рассмотреть состояние СУБД, отдельно состояние серверов приложений, отдельно состояние подсистемы создания бэкапов (подсистемы резервирования), а потом отдельно состояние системы в целом, то для каждого отдельного случая у нас получится гораздо меньший набор состояний и переходов, но при этом у нас сохраняется возможность делать сквозные большие тесты, если мы понимаем, что общая диаграмма состояний системы, в которой каждому состоянию соответствует ни одно состояние, а целая диаграмма или часть диаграммы нарисованная для другого объекта. Достаточно удобно и просто с этим работать, на финальный результат тест кейсов при аккуратном исполнении не влияет, но аккуратно исполнить такое разбиение конечно чуть сложнее.*

И еще одна типичная ошибка, которую мы рассмотрим - это попытка учесть слишком много возможных событий. Метод с использованием диаграммы состояний не претендует на полноту тестирование. Мы в данном случае не говорим о полноте тестирования. Это способ протестировать жизненный цикл неких объектов в системе, при этом внутри каждого состояний. *Например, учётная запись пользователя и ее состояния: незарегистрированный пользователь, зарегистрированный пользователь с неподтвержденной почтой, зарегистрированный подтвержденный пользователь.* Мы проверяем переходы из состояния в состояние, а процедуру регистрации, заполнения полей и прочее, мы в рамках этой программы в принципе не рассматриваем. Данный метод на полноту не претендует, поэтому не нужно пытаться засунуть в него всевозможные события, которые в системе могут произойти и не надо пытаться в этот flow, например с регистрацией, засунуть событие -

*пользователь закрыл браузер без сохранения, нажал F5 или сделал еще что-нибудь такое странное*, эти кейсы нужны, но вам неудобно будет писать эти кейсы по этой диаграмме и вам неудобно будет писать те другие кейсы, для которых эта диаграмма предназначена, потому что получится каша.




Давайте теперь рассмотрим порядок применения этого метода причём именно в примере вот такой таблицы.

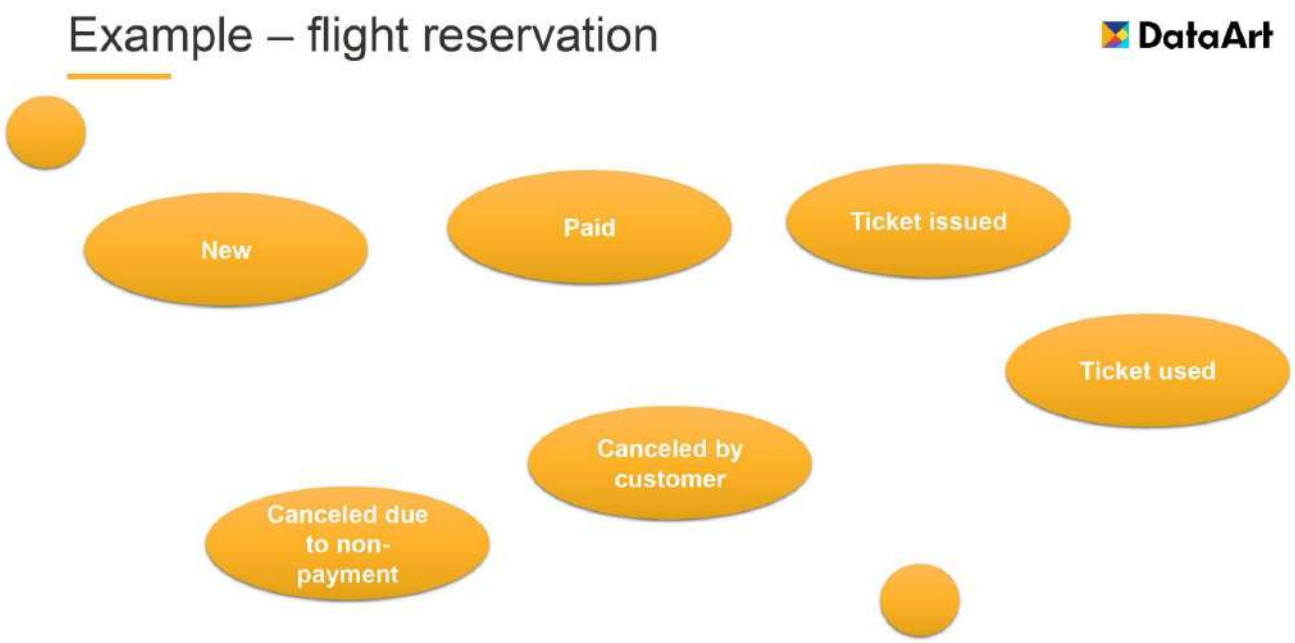



Первое, что мы должны сделать это выбрать объект. Чётко определиться состояние какого объекта мы хотим рассмотреть и Flow жизненный цикл какого объекта мы собираемся протестировать. Дальше нужно идентифицировать состояние, идентифицировать возможные события, дальше мы строим все возможные пары “событие + состояние”, убираем несовместимые пары событие + состояние. То есть у некоторых состояний какое-то событие возникать не может или мы не можем это фиксировать. Соответственно в этих ситуациях как правило это событие не влияет на поведение.

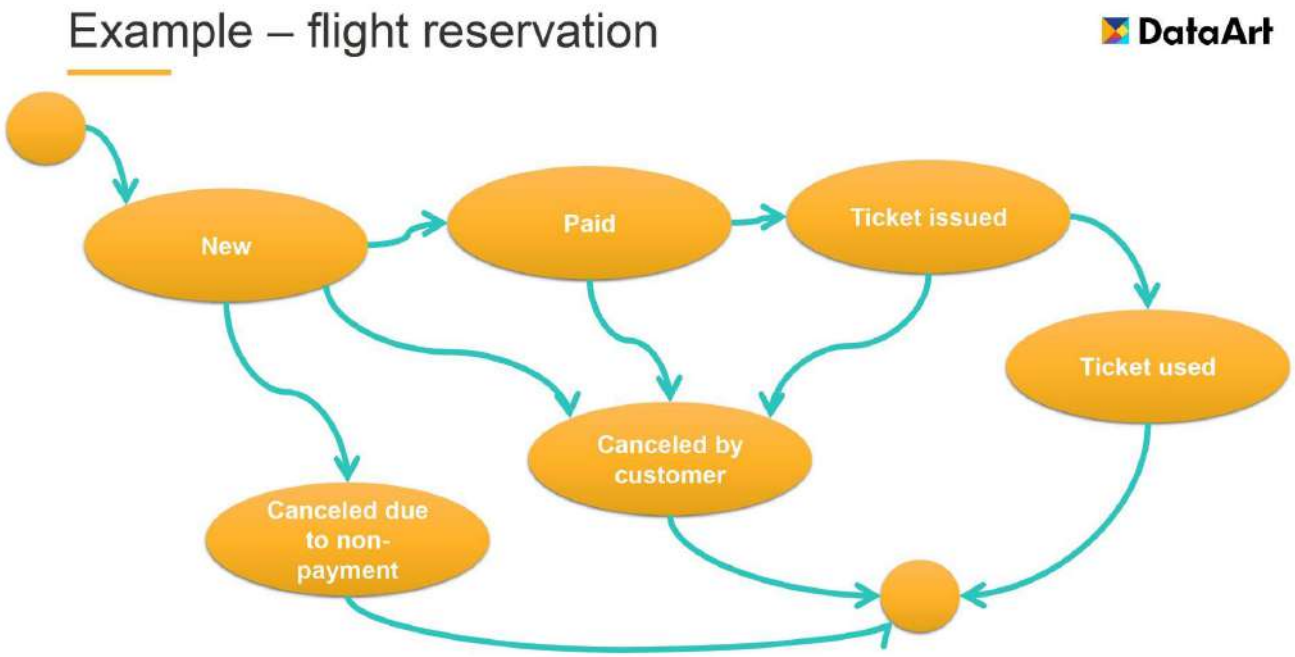
Наконец, определяем для каждой пары финальное состояние и необходимые действия. Обратите внимание, где здесь пункт нарисовать диаграмму? Его нет, то есть сама по себе диаграмма хороша с точки зрения иллюстрации, более того, иногда она у нас заранее есть. То есть её нам нарисовали архитекторы, когда проектировали системы. Удобно, но необязательно, это во-первых, а во-вторых даже, когда архитекторы проектируют систему и нарисовали нам диаграмму, я рекомендую способ из слайда *(How to create state transition table)*.

На примере слайда ниже, давайте попробуем применить этот метод, построить диаграмму и построить тест кейсы с помощью метода таблицы переходов.

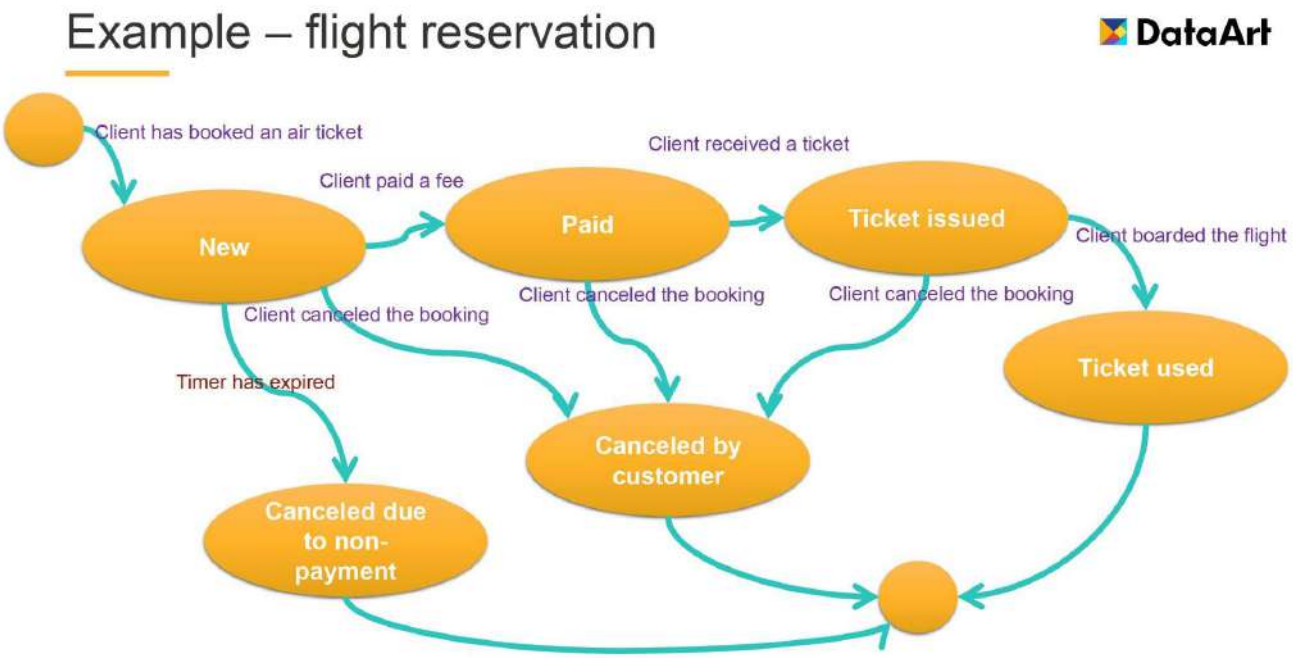

Итак предположим мы рассматриваем систему бронирование авиабилетов и для бронирования у нас есть, приведенный на слайде, набор состояний. Т.е пользователь создал на сайте бронирование - это состояние New, после этого пользователь должен эту бронь оплатить, также пользователь может отменить бронь, также бронь будет отменена, если пользователь вовремя не внес оплату. На основании бронирования, когда пользователь приедет на рейс ему будет выдан билет ну и наконец, когда пользователя посадят в самолет билет считается использованным.




К этим состоянием согласно нотации UML положено добавлять начальное и конечное состояние. Начальное состояние добавлять всегда имеет смысл потому, так как это является входом в этот flow, как правило вход инициируется каким то событием. Конечное состояние нужно обязательно в том случае, если у вас с ним связаны какие-то дополнительные действия, в данном случае - нет. Когда мы нарисуем всю диаграмму, я попробую придумать пример, в котором действия могут быть связаны.

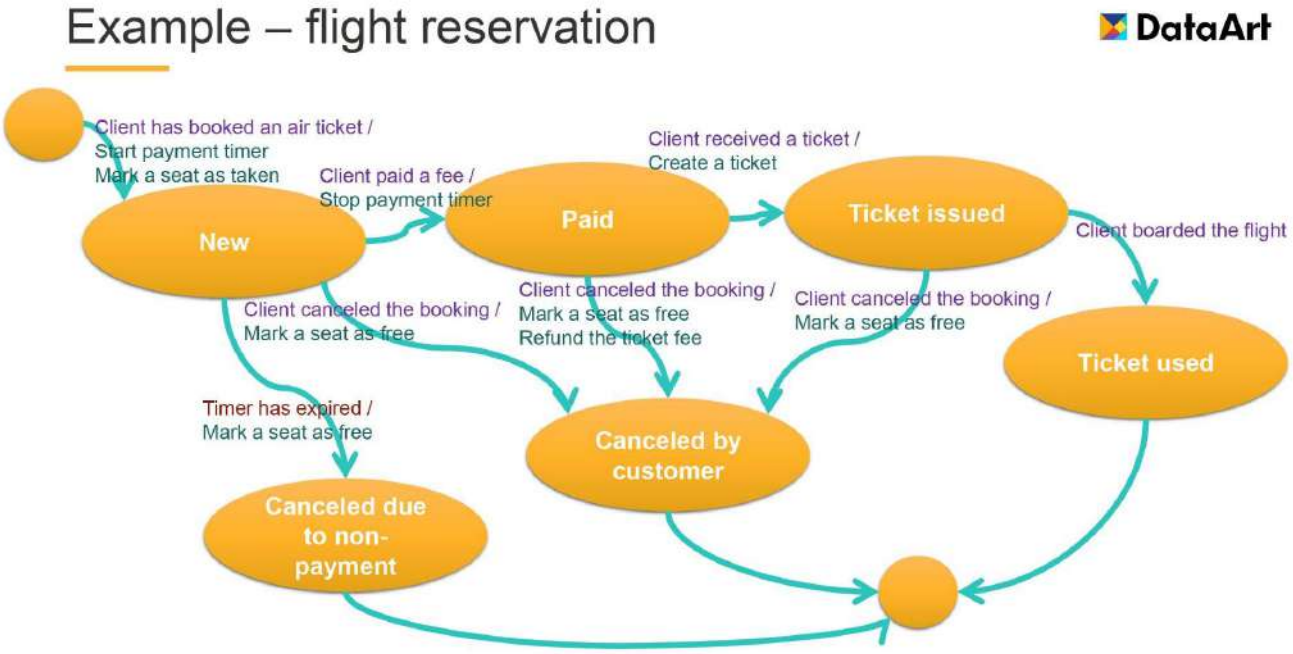



Возможные переходы представлены на слайде. То есть стандартный flow: бронь создана, оплачена, выдан билет, билет использован. Если бронь создана, но вовремя не оплачена, то она отменяется системой, также пользователь до момента посадки в самолет, в любой момент может отменить покупку. Переходы мы зафиксировали. Теперь нам нужно определиться с событиями, которые связаны с этими переходами.




Т.е. каждый переход инициируется событием, в нашем случае одним, но вообще не обязательно, что событие, которое инициирует переход может быть лишь одним. Переходы могут инициироваться одним из нескольких событий, либо набором событий сразу, т. е если возникает сочетание событий. И тот и другой вариант возможны и про эти оба варианта нужно помнить, чтобы их правильно учесть.

Обратите внимание, на одно из событий, оно почему-то покрашено другим цветом, как вы думаете почему? События бывают внутренние и внешние, и все они в общем-то должны быть учтены в системе.




Итак, после того, как мы определили события нам нужно определиться с возможными действиями. Когда клиент создает бронирование, система должна стартовать таймер и отметить посадочное место как занятое. Далее, если клиент оплатил заказ, то таймер останавливается, если

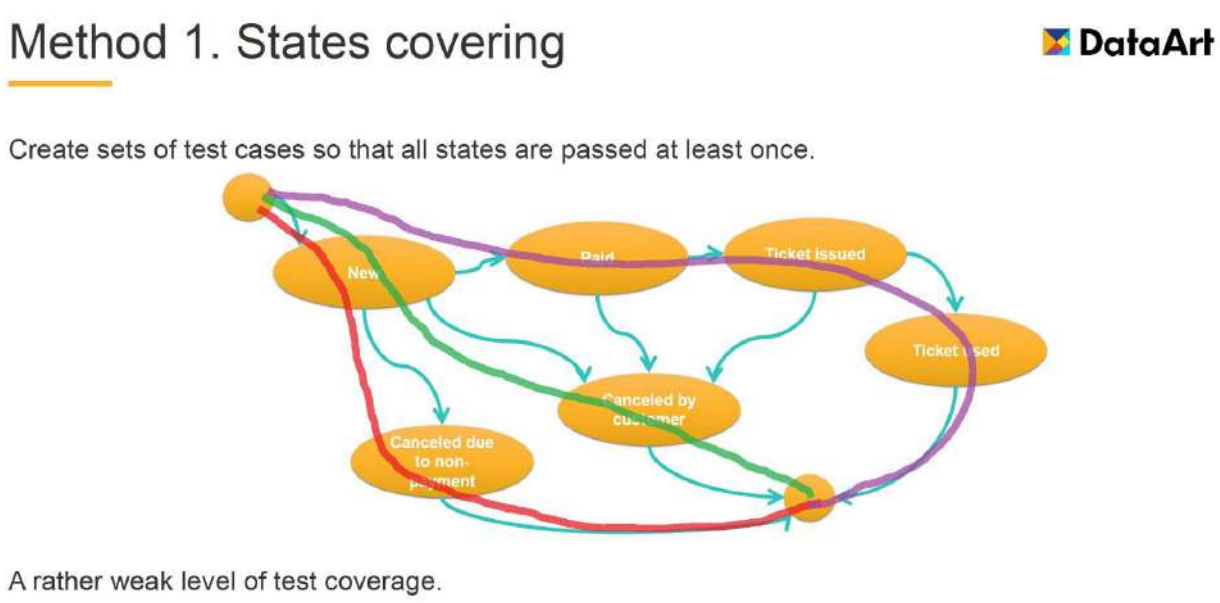
таймер истек, то система отменяет бронирование, соответственно она отмечает то занятое место как свободное.

Если клиент отменяет бронирование? Пока бронирование не оплачено, всё, что нам нужно сделать - это отметить место как свободное. Мы во всех трех случаях должны указать, что место сново свободное, когда состояние - canceled by customer. Если клиент оплатил бронирование, мы возвращаем ему стоимость исходя из диаграммы. Этим методом мы с вами на прошлой лекции тестировали ситуацию, в которой у нас рассчитывается стоимость возврата за билет, в зависимости от сроков возврата, но Statechart диаграмма эту часть функциональности не покрывает. Более важен сам факт события - возврат денег. Если же билет уже выдан, мы считаем, что деньги уже не возвращаются. Когда клиент проходит посадку в самолёт, то мы переходим в состояние билет использован.

Например, если бы нам нужно было бы вести какую-нибудь статистику, которая должна была бы записываться в какую-нибудь подсистему аудита, например, статистику по тому, как часто отменяются заказы, как часто клиенты не оплачивают заказы, то при переходе в конечное состояние у нас бы возникли какие-то дополнительные действия. Например, внести какую-нибудь запись в подсистему аудита. В нашем примере их нет, поэтому стрелочки пустые, поэтому конечное состояние можно убрать, но все же удобнее оставить.

А сейчас давайте поговорим о том, что мы делаем дальше после того как нарисовали диаграмму?

Итак, самый простой и примитивный способ тестирования - это покрытие состояний.



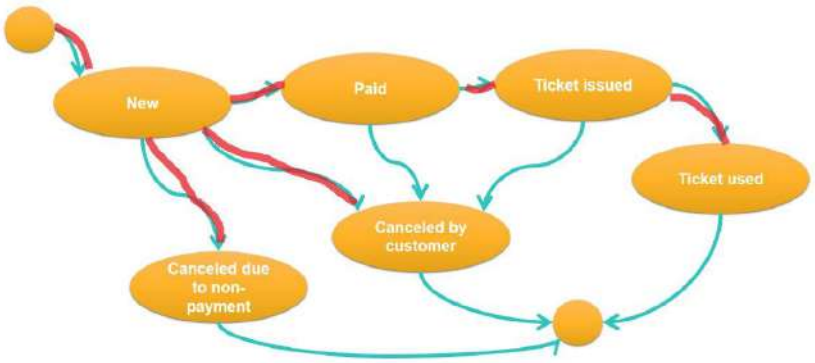

То есть, нам нужно, чтобы система побывала в каждом состоянии хотя бы один раз. Для этого примера мы должны создать в этой ситуации - три теста. Для проверки всех состояний. Для какого-то поверхностного smoke тестирования этот подход может подойти, но в целом покрытие в этом случае получается очень небольшим.

Следующий подход - покрытие событий.

## Method 2. Events covering

DataArt

Create sets of test cases so that all events are triggered at least once.




Test cases that cover all events at the same time cover all states.

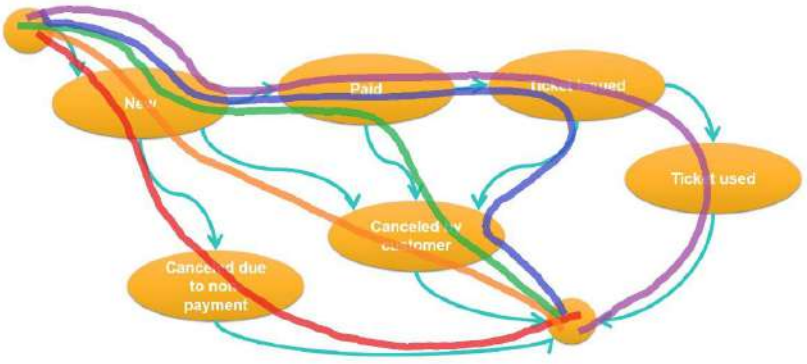
Again, the level of test coverage is weak.

Мы пытаемся проверить возникновение каждого события. Обратите внимание, что события приводящие к переходу в canceled by customer состояние, одинаковые. То есть клиент отменил бронирование, исходя из этого у нас нет прямой необходимости проверять все три возможных ситуаций, мы должны проверить реакцию на событие. Покрытие получается чуть получше, чем в предыдущем случае и покрытие состояний мы в этом случаем также получаем. В каждое состояние мы всё-равно пришли,  но при этом тестирование по-прежнему остается поверхностным, лишь чуть более детальным, чем в предыдущем случае.

## Method 3. Path covering

DataArt

Create sets of test cases so that all paths are traversed at least once.



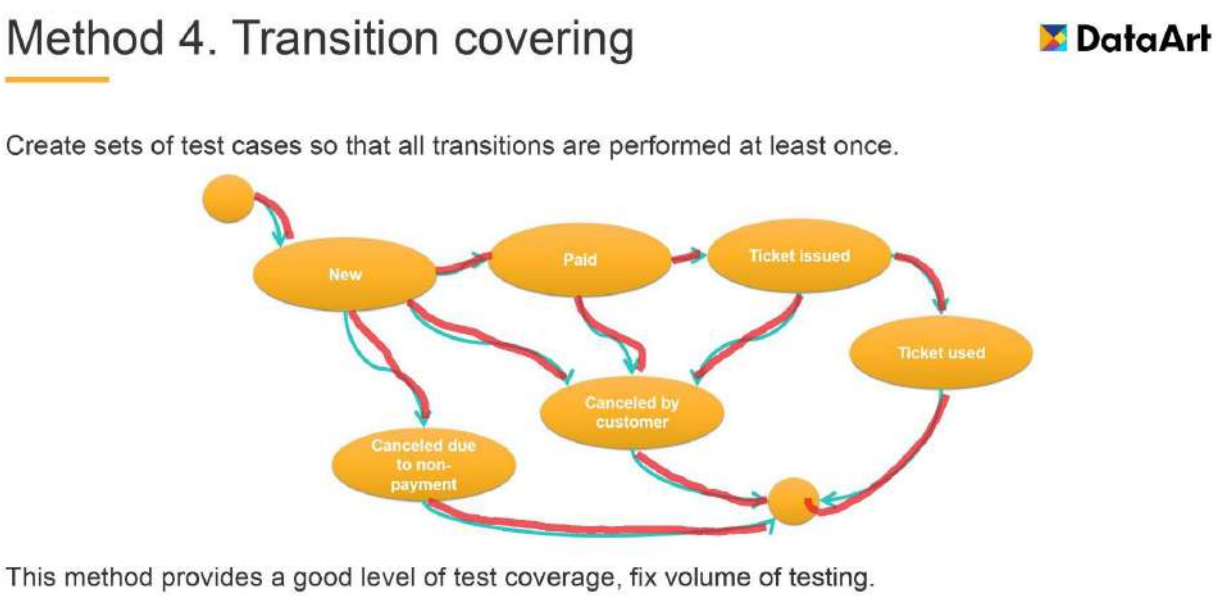

Good test coverage, but practically impossible sometimes. If the diagram has cycles, then the number of possible paths can be infinite.

Метод покрытия путей. Фактически мы проходим все возможные пути от начального состояния до конечного. Вот почему конечное состояние удобно всё-таки рисовать, чтобы потом аккуратно отследить покрытие путей потому, что если мы не рисуем конечное состояние, как нечто отдельно,то мы получаем три конечных состояний. В нашей маленькой диаграмме - это достаточно очевидно, то есть

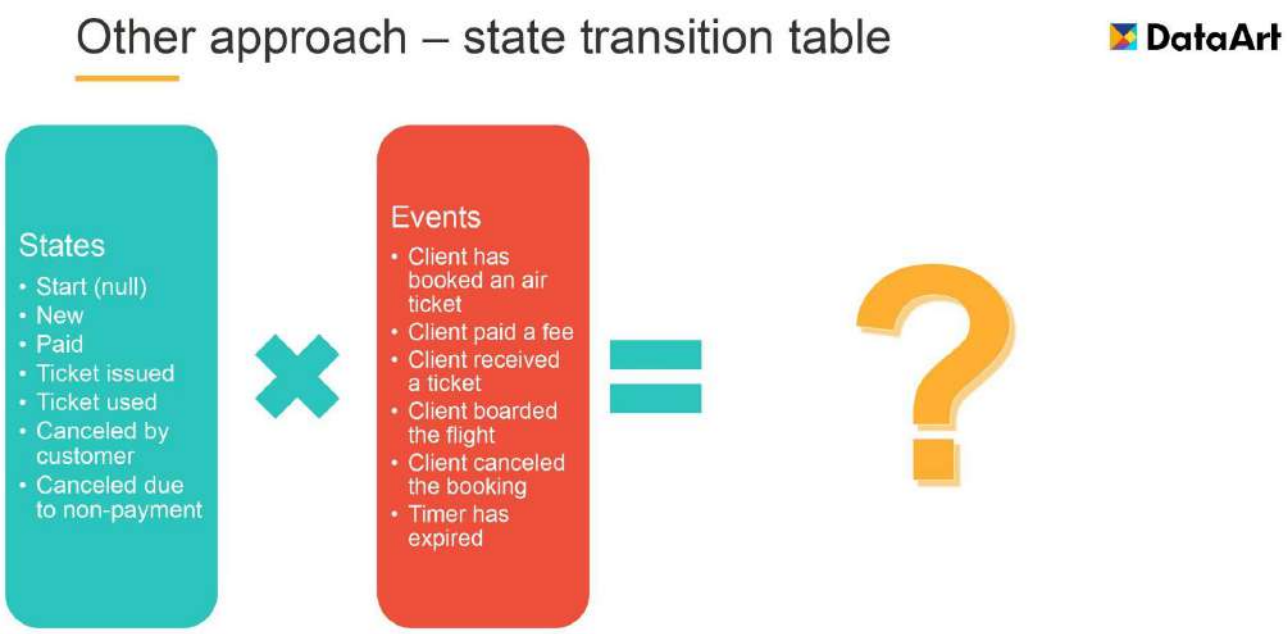
конечное у нас будет любое состояние, из которого не выходит ни одной стрелочки, а если диаграмма большая и развесистая, то отслеживать какие состояния конечные довольно неудобно. Когда есть одно, в которое потом от них все стрелочки сойдутся, то по этим стрелочкам, в обратную сторону мы все конечные состояния системы легко отлавливаем.

В этой ситуации интересно то, что фактически, каждый путь в системе представляет отдельный use case, здесь очень сильно пересекается метод Statechart диаграмм и метод тестирования use case'ов и в общем-то выявлять use case'ы на основе диаграммы вполне себе можно. И обнаружить например, что ваши аналитики не все use case'ы в системе предусмотрели и расписали.

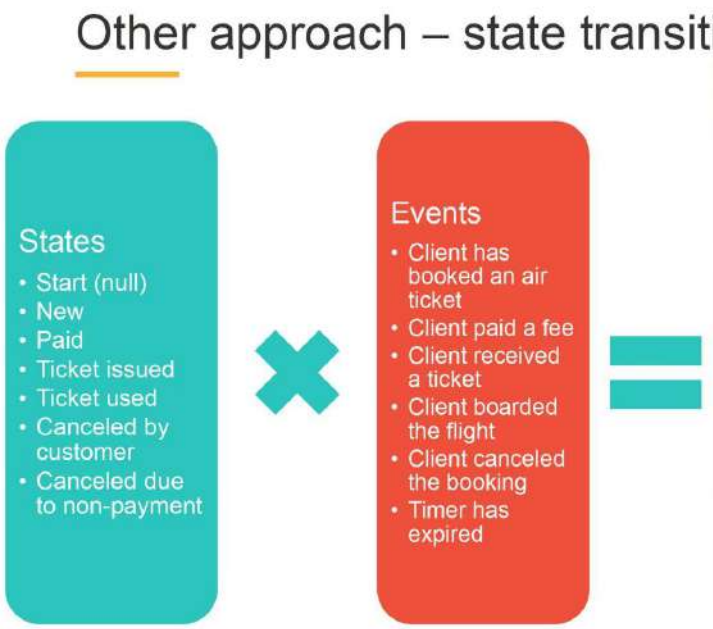
Метод покрытия переходов.




В этой ситуации мы тестируем все возможные переходы из всех возможных состояний. По тесту на каждую стрелочку. Если мы тестируем на основе диаграммы состояний, то это золотой стандарт. Потому, что в этом подходе покрытие получается наиболее полным. Слайды с методом 3 и методом 4 эквивалентны в некоторых случаях, но не во всех. Поэтому, если вы пытаетесь опираться именно на диаграмму, то имеет смысл тестировать по последнему методу или используя use case'ы, если вам этого достаточно с точки зрения глубины и детальности тестирования.




Тот алгоритм, который мы рассматривали это несколько другой подход, диаграмма нам не нужна. Мы взяли всевозможные события, всевозможные состояния и строим всевозможные их сочетания. Очевидно, что их получится очень много. 7x6=42




| State | Event | New state | Action |
|---|---|---|---|
| null | Client has booked an air ticket | New | Start timer<br>Mark a seat as taken |
| null | Client paid a fee | - | |
| null | Client received a ticket | - | |
| null | Client boarded the flight | - | |
| null | Client canceled the booking | - | |
| null | Timer has expired | - | |
| New | Client has booked an air ticket | - | |
| New | Client paid a fee | Paid | Start timer |
| New | Client received a ticket | - | |

После заполнения первых двух столбцов в таблице, анализируем каждую пару состояние + событие. Либо определяем новое состояние и набор действий при необходимости. Действие же не обязательно в этом случае, когда состояние обязательно. Либо обнаруживаем, что эта пара состояние + событие не сочетаемо.

Такой способ помогает увидеть не пропустили ли мы или архитекторы какую-то стрелочку. А так мы видим, что пара состояние+событие совместимы и в системе могут быть реализованы. Либо, мы понимаем, что они не совместимы, но видим в интерфейсе такую возможность, т.е. имеем возможность найти баг еще не начав писать тест кейсы. Тоже вполне возможный вариант. Т.е проанализировать

совместимы ли эти события, посмотреть на макеты интерфейса в этот момент, посмотреть на требования, это само по себе полезное статическое тестирование. Картинки хороши для иллюстраций и презентаций, а для тест-дизайна лучше подойдут таблицы.

Мы проанализировали эти события и указали их в виде прочерков в таблице, где они несовместимы, что делать дальше? Дальше мы их из таблицы вычеркиваем. И вычеркивая их из таблицы, мы сокращаем ее размеры в несколько раз. Этим удобнее такие таблицы, чем матрицы, потому, что матрицу так не сократить, она все равно останется большой. В данном примере 6x7.

## Other approach – state transition table

DataArt

States
- Start (null)
- New
- Paid
- Ticket issued
- Ticket used
- Canceled by customer
- Canceled due to non-payment

Events
- Client has booked an air ticket
- Client paid a fee
- Client received a ticket
- Client boarded the flight
- Client canceled the booking
- Timer has expired

| State | Event | New state | Action |
|---|---|---|---|
| null | Client has booked an air ticket | New | Start timer<br>Mark a seat as taken |
| null | Client paid a fee | - | |
| null | Client received a ticket | - | |
| null | Client boarded the flight | - | |
| null | Client canceled the booking | - | |
| null | Timer has expired | - | |
| New | Client has booked an air ticket | - | |
| New | Client paid a fee | Paid | Start timer |
| New | Client received a ticket | - | |

Выделили несовместимые и теперь вычеркнем их из таблицы. Таким образом дополнительно к тест дизайну, этот метод позволяет проанализировать то, что перечислено на слайде ниже.

## Other approach – state transition table

DataArt

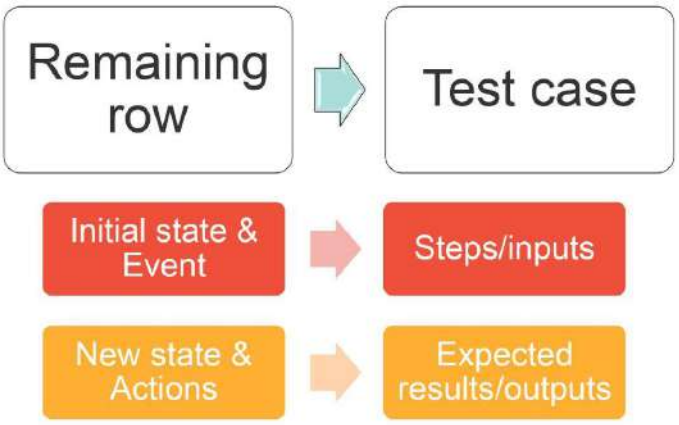
Analyze
- ✓ completeness of requirements
- ✓ correctness of the user interface
- ✓ if transitions are skipped

| State | Event | New state | Action |
|---|---|---|---|
| null | Client has booked an air ticket | New | Start timer<br>Mark a seat as taken |
| null | Client paid a fee | - | |
| null | Client received a ticket | - | |
| null | Client boarded the flight | - | |
| null | Client canceled the booking | - | |
| null | Timer has expired | - | |
| New | Client has booked an air ticket | - | |
| New | Client paid a fee | Paid | Start timer |
| New | Client received a ticket | - | |

Что мы делаем с этим дальше? Мы строим тест кейсы.

## Other approach – state transition table

DataArt



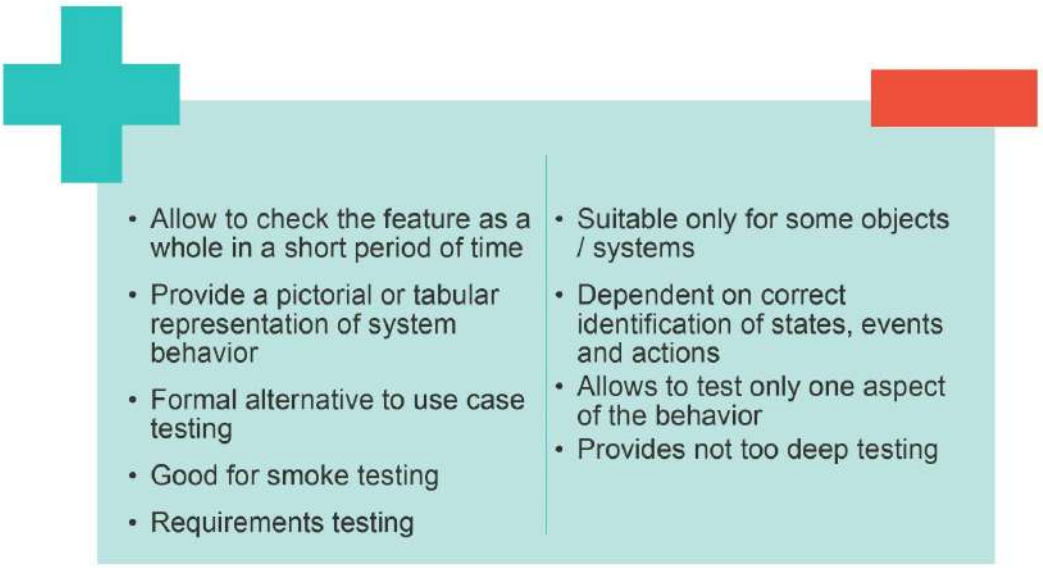

| State | Event | New state | Action |
|---|---|---|---|
| null | Client has booked an air ticket | New | Start timer<br>Mark a seat as taken |
| null | Client paid a fee | - | |
| null | Client received a ticket | - | |
| null | Client boarded the flight | - | |
| null | Client canceled the booking | - | |
| null | Timer has expired | - | |
| New | Client has booked an air ticket | - | |
| New | Client paid a fee | Paid | Start timer |
| New | Client received a ticket | - | |

На каждую из оставшихся строк таблицы в этом варианте, либо на каждую заполненную ячейку в матричном варианте, мы должны написать тест кейс. При этом первые два столбца задают входные условия для нашего тест кейса *(шаги и входные данные)*, а на основе последних двух столбцов мы формируем ожидаемый результат. Само построение таблицы достаточно сложное, скорее сложно выявить состояние и событие ничего не упустив. А дальнейшая работа с ней достаточно простая.

Плюсы и минусы этого подходы.

## Pro and cons of state transition testing

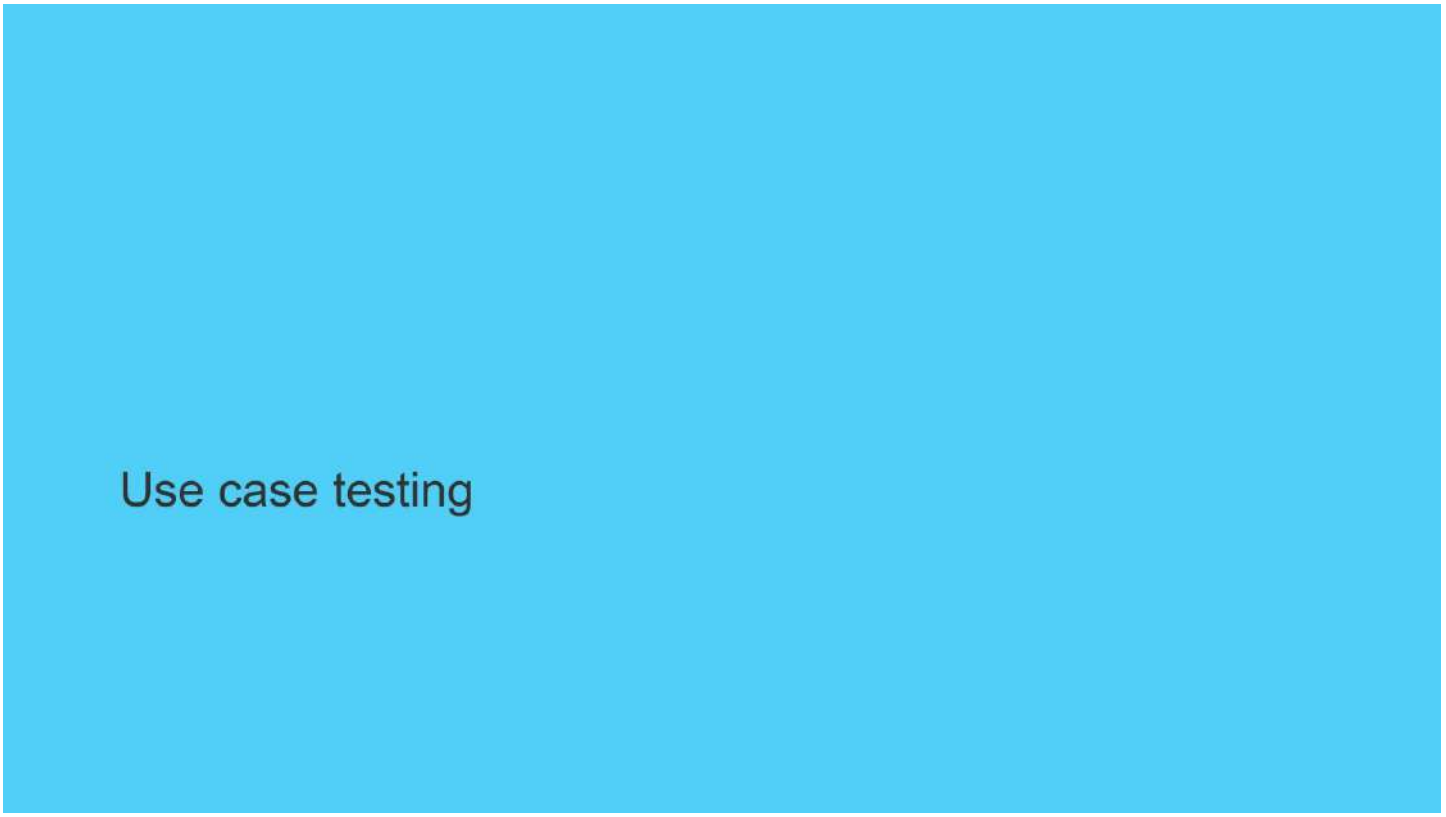
DataArt




Этот подход не дает высокого покрытия для функциональности. Он предназначен тестировать жизненный цикл, какого-то объекта в системе. При этом если у вас в системе таких объектов нет, то для вашей системы он не применим, ну или ограниченно применим, если это не основные части вашей

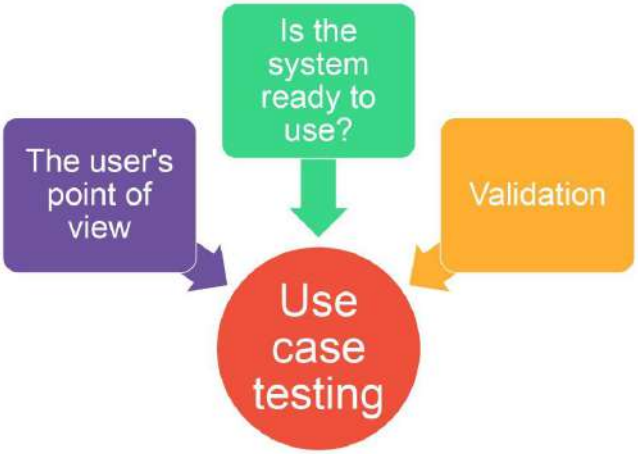
системы. Зато, если у вас есть достаточно сложные flow в системе, то вы можете протестировать глубоко во всех возможных сочетаниях. И кроме того, этот подход хорош для smoke тестирования. Если есть некий flow - этот метод хорошая база для тестирования smoke функциональности.

Этот подход не плохо пересекается с тестированием use case. Он лучше, чем тестирование с помощью use case, потому что он более формальный. Потому что use case это словами написанный набор действий, а все, что написано словами - трудно формализуется и трудно проверяется. А здесь при правильном построении flow получается формальный метод, где нам не нужно дополнительно думать *(А что же здесь имеется в виду, в этом пункте 3.а, какие варианты, как трактовать?)*. Тут все просто, однозначно и формально. Этим удобнее, чем аналогичные операции выполнять на основе use case.




Техника use case тестирования. Только этот подход к тестированию позволяет нам смотреть на приложение с точки зрения пользователя. В предыдущем случае мы проверяли состояния и переходы, здесь же пользователь не знает какие есть состояния у заказов и ему это не интересно. И что мы считаем событиями, а что не считаем, ему тоже не интересно. Ему нужно, чтобы система была удобной, чтобы он понимал как с ней работать, чтобы действия, которые ему в ней нужно выполнить, выполнялись быстрее и с минимальным количеством кликов и тому подобные вещи.

## Base ideas

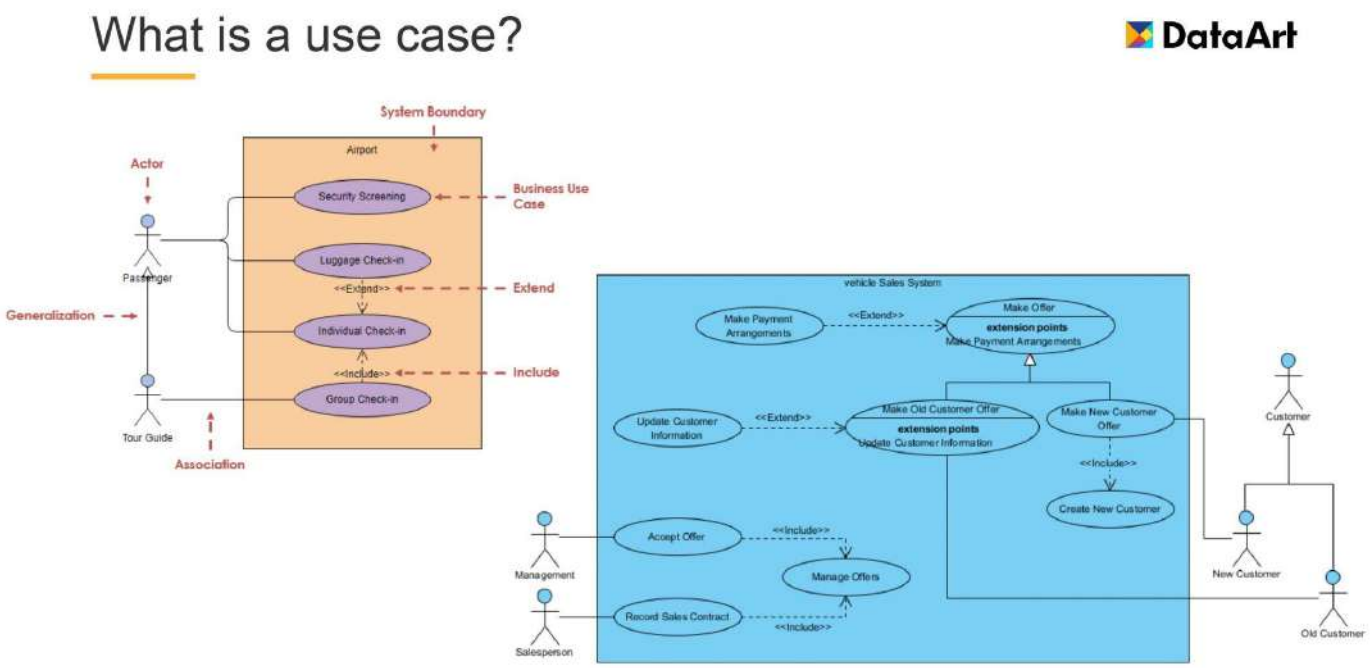



Этот подход к тестированию позволяет взглянуть на систему с точки зрения пользователя и это, пожалуй, единственный подход, в рамках которого мы можем изобразить какие то зачатки валидации.

Валидация - это проверка соответствия ожиданиям пользователя.

Когда считают, что мы можем провести валидацию в системе, в общем случае это из серии - “возомнить себя богом”. В большинстве случаев система имеет свою предметную область, в которой мы не компетентны, но use case - это окошко в жизнь пользователя, в рамках нашего представления о нем. И если сценарии use case’ы написаны не на основе уже реализованной системы, как это к сожалению часто бывает, а на основе реальных жизненных задач пользователя, то это хорошо. *Например: Я как пользователь хочу купить красное платье, для этого я хочу отфильтровать все красные платья по длине, по ткани, по цене и популярности.* Здесь нет ни слова про интерфейс и как это будет сделано. Если use case написан так, то это максимум, который мы сможем достать из валидации. Все остальное будет про тестирование на стороне заказчика, экспериментальное использование, это уже будет на этапе внедрения. Опять же в Agile - этап внедрения, является условным. Но до завершения фазы MVP (первого разворачивания на продакшен), мы никакой обратной связи не получим и все, что у нас есть это use case.

На следующем слайде представлены примеры use case диаграмм.




Use case тоже можно изображать диаграммами, это один из типов диаграмм аннотации UML. Вариант диаграмм не очень удобен. Другой способ представления use case'ов - это текстовое представление. На диаграмме удобно выделяются типы пользователей с возможной иерархией. *Например, если у нас есть (Actor) - клиент и (Actor) - VIP клиент, то VIP клиент, может все тоже, что и клиент, но плюс ему доступны дополнительные опции, например доступ к распродажам или дополнительные скидки.*

Есть Actor - типы пользователей, есть Use case - бизнес кейсы, т.е. некие действия, которые эти пользователи могут выполнять. И между этими действиями есть сложные взаимоотношения, т.е *например, некое большое действие, как регистрация в системе и она может быть детализирована (ввести логин, ввести пароль) и может быть расширенной (ввести пароль уже зарегистрированного пользователя).*

Диаграммы - это тоже способ формализации, в данном случае им достаточно тяжело пользоваться в отличии от Statechart диаграммы, но зная способ формализации, вы лучше и качественнее напишите Use case в обычной текстовой форме.

## What is a use case?

DataArt

| | Step | Description |
|---|---|---|
| Main Success Scenario<br>A: Actor<br>S: System | 1 | A: Inserts card |
| | 2 | S: Validates card and asks for PIN |
| | 3 | A: Enters PIN |
| | 4 | S: Validates PIN |
| | 5 | S: Allows access to account |
| Extensions | 2a | Card not valid<br>S: Display message and reject card |
| | 4a | PIN not valid<br>S: Display message and ask for re-try (twice) |
| | 4b | PIN invalid 3 times<br>S: Eat card and Exit |

*A Partial Use Case for PIN Entry*

| | | |
|---|---|---|
| **USE CASE 5** | Buy Goods | |
| **Description** | Buyer issues request directly to our company, expects goods shipped and to be billed. | |
| **Used by** | Manage customer relationship (use case 2) | |
| **Preconditions** | We know Buyer, their address, and needed buyer information. | |
| **Success End Condition** | Buyer has goods, we have money for the goods. | |
| **Failed End Condition** | We have not sent the goods, Buyer has not spent the money. | |
| **Actors** | Buyer, any agent (or computer) acting for the customer.<br>Credit card company, bank, shipping service | |
| **Trigger** | purchase request comes in. | |
| **DESCRIPTION** | **Step** | **Action** |
| | 1 | Buyer calls in with a purchase request |
| | 2 | Company captures buyer's name, address, requested goods, etc. |
| | 3 | Company gives buyer information on goods, prices, delivery dates, etc. |
| | 4 | Buyer signs for order. |
| | 5 | Company creates order, ships order to buyer. |
| | 6 | Company ships invoice to buyer. |
| | 7 | Buyers pays invoice. |
| **EXTENSIONS** | **Step** | **Branching Action** |
| | 3a | Company is out of one of the ordered items:<br>3a1. Renegotiate order. |
| | 4a | Buyer pays directly with credit card:<br>4a1. Take payment by credit card (use case 44) |
| | 7a | Buyer returns goods:<br>7a. Handle returned goods (use case 105) |
| **VARIATIONS** | | **Branching Action** |
| | 1 | Buyer may use<br>phone in,<br>fax in,<br>use web order form,<br>electronic interchange |
| | 7 | Buyer may pay by<br>cash or money order<br>check<br>credit card |

Если вы знаете UML и понимаете какие там варианты с действиями могут быть изображены, то скорее всего вы будете об этом думать, когда будете их описывать текстом, так как особой разницы нет. Поэтому как упражнение для начинающих - графическое изображение, является хорошим способ для практики.

На слайде выше представлены два варианта описания Use case один простой *(слева)* и второй *(справа)* более подробный.

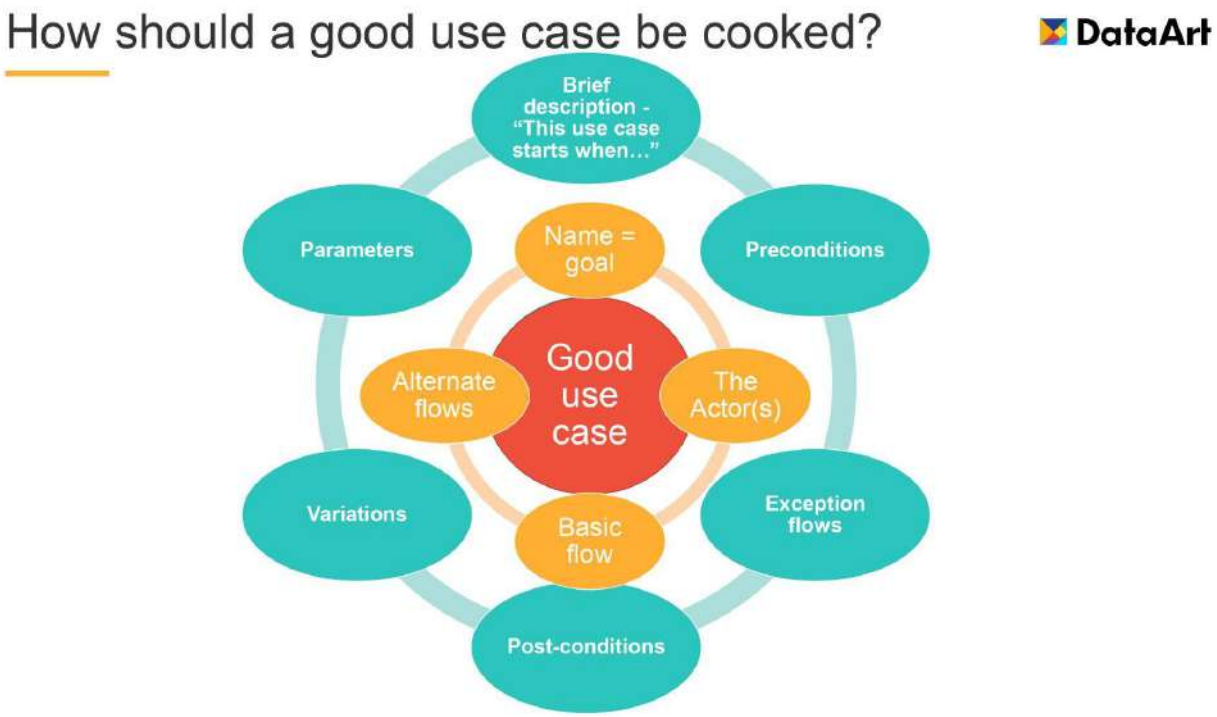
Что должно так или иначе в Use case входить?

## How should a good use case be cooked?

DataArt


У Use case должны быть:

- заглавие, которое отражает цель пользователя,
- должен быть обозначен пользователь, т.е. некая роль в системе,
- описан базовый сценарий,
- рассмотрены альтернативные сценарии.



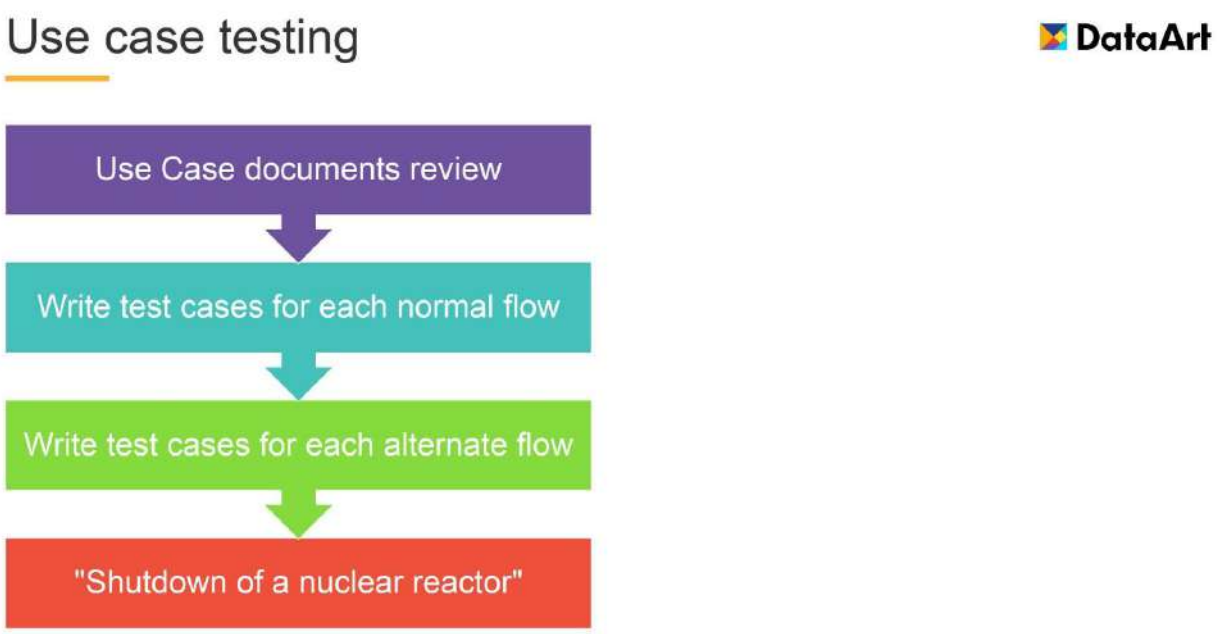

Дополнительно, может быть описано много другого интересного:

- предусловие и постусловие *(откуда вообще пользователь пришел к этим задачам и какие последствия могут быть у выполнения этого кейса)*
- некое общее описание *(в какой ситуации пользователь начинает этот сценарий, что инициирует этот сценарий)*
- возможные дополнительные варианты исключения *(какие exception flows могут быть интересны, например если у пользователя аварийно прервано подключение, это не пользовательский flow, т.к его инициирует не пользователь, но оно влияет на пользовательский flow)*
- влияние параметров
- вариации сценариев

Чего не должно быть в хорошем Use case?

1. Прежде всего каждый use case  должен описывать выполнение пользователем, некой его реальной жизненной цели, т.е. цели, которые у него есть в его реальности, а не в вашей программе. Потому, что в его реальности, у него нет use case восстановления пароля, это один из альтернативных  flow use case авторизации, если пользователь забыл пароль, то дальше он вынужден его восстановить. Это важно когда вы хотите произвести валидацию. С точки зрения написания тест кейсов на основе use case, для какого нибудь приемочного тестирования мы можем выделить восстановление пароля в отдельный flow или можем рассмотреть его в рамках use case'ов авторизации - это неважно, а вот с точки зрения валидации - это будет важно.
2. Use case должен быть инициирован пользователем, т.е если вы читаете use case, а его инициирует система, то это что-то странное. Скорее всего это плохо написанный use case
3. Внутри use case, система не должна предугадывать действия пользователя, т.е use case это поведение с точки зрения пользователя. *Есть такая метафора "хороший use case похож на пинг понг" т.е пользователь сделал одно, система ему ответила второе, тогда пользователь сделал другое и ситема ему ответила по другому. Если use case описывает вариант взаимодействия "пользователь - пользователь - пользователь", то скорее всего это неполный use case. Если use case описывает вариант взаимодействия "пользователь - система - система - система", то это тоже плохой use case. Пользователь ожидает 1 действие от системы*
4. У нас есть альтернативные сценарии, это когда пользователь сделал не то, что система ожидала и система, как то на это отреагировала*(например, пользователь ввел неверный пароль три раза и система заблокировала учетную запись этого пользователя, а далее мы пишем, что следующим шагом пользователь должен восстановить учетную запись через администратора или какой-то другой вариант - это способ устранить проблему, а не действие пользователя).* Альтернативный сценарий - это, когда пользователь пошел туда-то и сделал то то.
5. В реальных пользовательских сценариях не будет детального описания, по типу, *какую кнопочку нажать, по какой ссылке перейти и т.д.* Соответственно в use case эти вещи тоже лишние.
6. Use case должны быть атомарные. Одна пользовательская цель и один flow. С use case, в который объединили несколько альтернативных  flow неудобно работать (авторизация и регистрация не объединятся)
7. Частая ошибка, когда в альтернативных flow все просто описывается заголовком. *Например, в основной flow пользователь ввел все корректно, в альтернативный flow пользователь ввел некорректные данные. Далее никаких объяснений.*

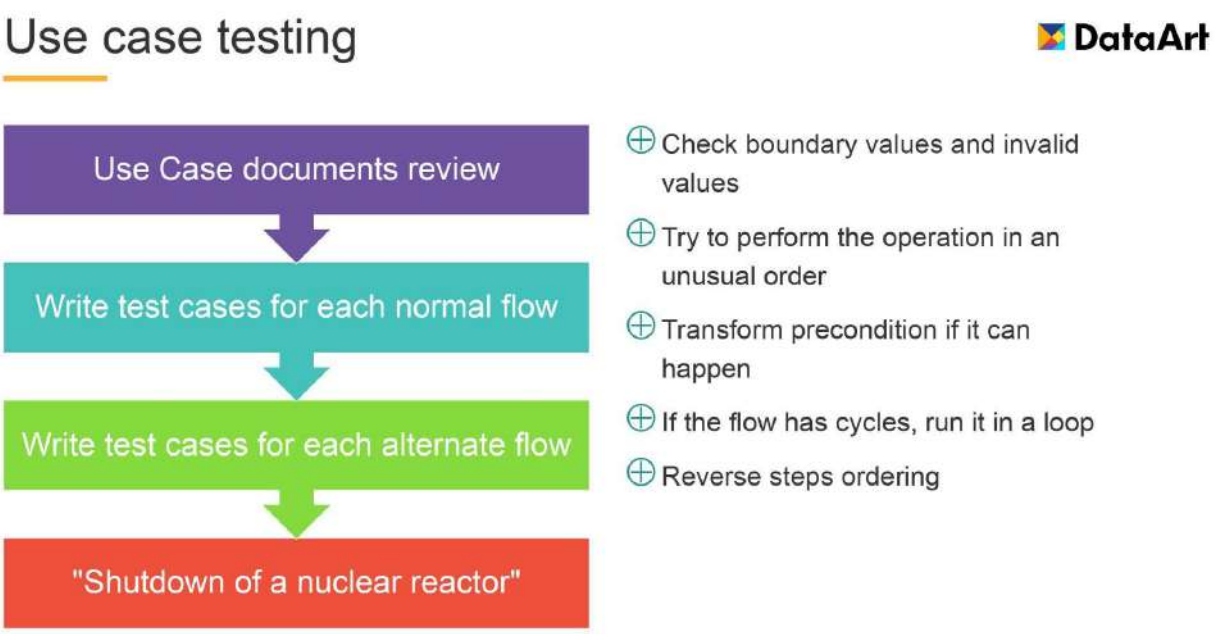
Почему мы так подробно разбираем Use case, потому, что если их брать за базис для тестирования, то это должны быть хорошо продуманные и написанные use case'ы - как сильный инструменты для тестировщика. Все, о чем мы говорили ранее, позволяет вам оценить качество Use case. Для написания хорошего use case нужно плотное взаимодействие с заказчиком потому, что если у нас есть развесистое ТЗ, а мы по нему пытаемся писать тест кейсы, то появляется вопрос (*а зачем мы это делаем*?). Иногда, когда в проекте имеются очень сложные кейсы с очень сложной бизнес логикой, например, *мы разрабатываем систему документооборота* - то в этом случае будет громадное количество use case'ов. Написание use case'ов больше относится к работе аналитиков, но QA инженер, также должен уметь писать use case, QA должен уметь оценивать их качество и просить доработки, если они недостаточно качественны, все это необходимо для того, чтобы применять этот метод с хорошими результатами.




Метод Use case тестирования.

Проводим ревью use case, если качество use case нас устраивает, то мы пишем тест кейс на каждую ветку основного flow, и на каждую ветку альтернативного flow. Дополнительно рекомендуется рассматривать всякие маловероятные ситуации, которые критично важны для системы. Классический пример из учебника на эту тему это “Остановка ядерного реактора” (т.е. этот вариант произойдет с небольшой вероятностью, но ошибка данного сценария будет критична для системы). Такие сценарии не всегда в use case’ах учтены, когда они учтены такие use case или такие ветки как правило имеют маленький приоритет. Потому, что приоритет часто выставляется на основе вероятности того, что пользователь по этому сценарию пройдет. Так вот, это, в принципе глобально не правильный способ расстановки приоритетов, не только в этом случае, но и вообще для любых тест кейсов и любого тестирования.

Правильный способ - это оценка приоритета на основе риска. А риск складывается из двух составляющих: вероятность возникновения события и масштабы бедствия, в случае если это событие возникает. Когда мы будем оценивать приоритет на основе вероятности, такого рода сценарии мы скорее всего учитывать не будем вообще, где то они будут с очень маленьким приоритетом болтаться в хвосте. Но если мы включим в оценку масштабы бедствия в случае если этот сценарий осуществится, то наши приоритеты будут гораздо корректнее, и мы упустим меньше критических ошибок.

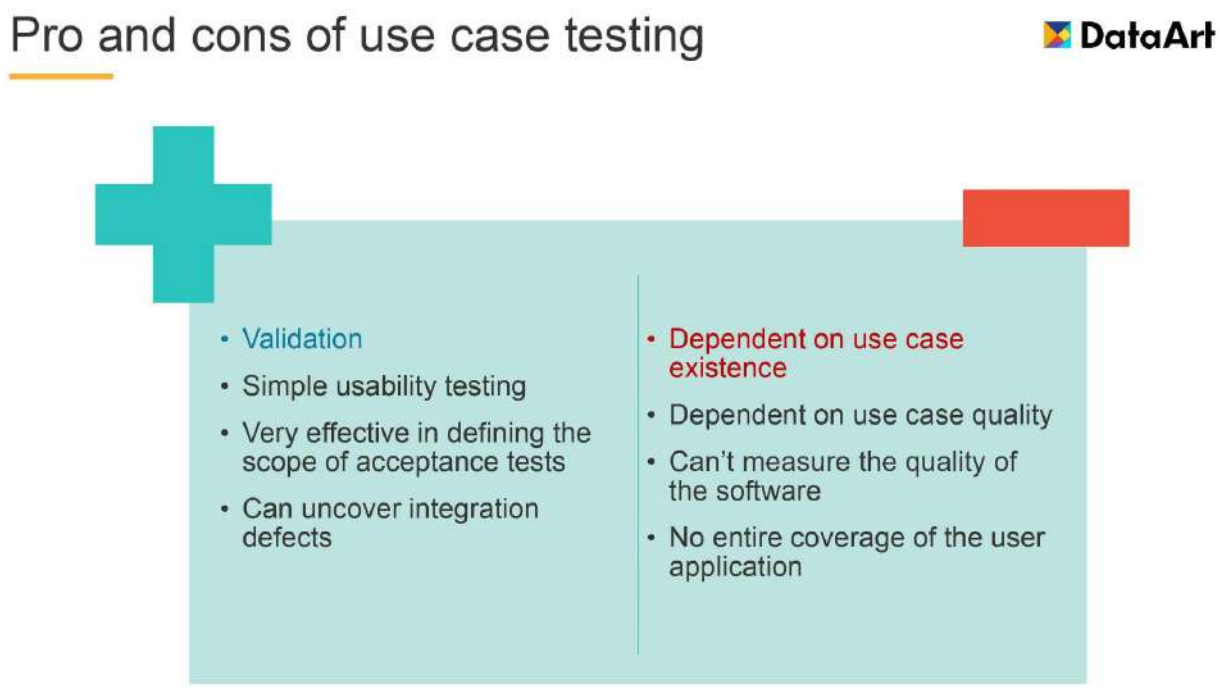



Тестирование на основе use case само по себе - это очень поверхностное тестирование, скорее всего даже более поверхностное, чем мы говорили про statechart диаграмму. Чем можно его дополнить, чтобы получить более менее разумно покрытие на том уровне, на котором оно применяется. Например, use case это хорошая база для acceptance тестирования, когда у нас нет задач доказать, что система работает правильно во всех возможных кейсах и во всех нюансах, *чаще всего для большинства заказчиков acceptance тестирование будет удовлетворительным если им продемонстрируют, что каждый пункт ТЗ действительно реализован и реализован правильно. Т.е фактически достаточно продемонстрировать работу каждой функции.* Соответственно use case неплохо решают эту проблему и делают тесты достаточно понятными нашему заказчику, но чем их стоит дополнить в этой ситуации.

Примеры перечислены на слайде выше. Во-первых проверить граничные условия. Где это возможно в рамках use case. Попробовать поменять местами шаги use case, если это возможно. Если use case содержит какие-то циклы, попробовать пройти их несколько раз. Попробовать поменять предусловия, например, поискать другие точки входа для сценариев, либо если точка входа только одна, но она зависит от каких-то условий, попробовать разные сочетания этих условий перебрать.

Если нам этого достаточно. То приемочное тестирование вполне можно на этом построить.

Плюсы и минусы этого подхода перечислены на слайде ниже.




К плюсам можно отнести валидацию и это главный плюс. Кроме того, use case предлагают нам возможность получить оценку о удобстве использования системы, *т.е мы прошли путь пользователя, если мы считаем, что этот путь достаточно удобен, то нам это было удобно, то и пользователю это скорее всего будет удобно*. Надо понимать, что ценность результатов по этому пункту будет зависеть напрямую от того, насколько мы претендуем на звание UX экспертов, соответственно, чем опытнее тестировщик, тем больше знаний и навыков у него в этой области будет.

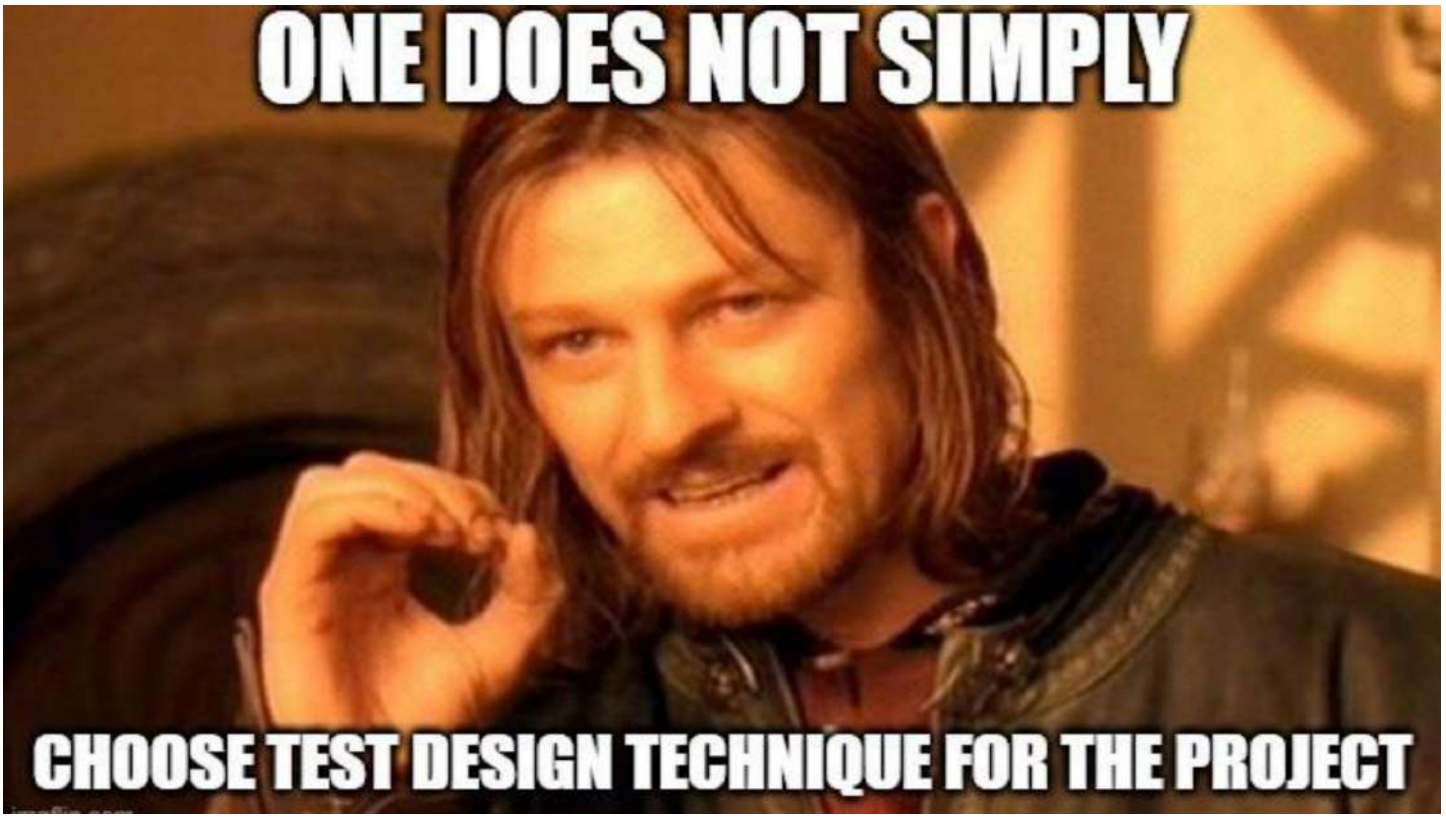
Use case являются хорошей основой для acceptance tests. Кроме того use cases как правило покрывают проверку интеграции подсистемы в систему и интеграцию с внешними системами. Т.е интеграции на верхнем уровне, например интеграция системы с соц.сетью “вконтакте” при авторизации, мы проходим flow пользователя, а не проверяем какие то отдельные кейсы.

Главное ограничения этого подхода - это наша зависимость от качества use case, от того, что нам предоставили аналитики. Кроме того - это не способ оценки качества системы в целом. С верификацией в этом подходе есть большие проблемы потому, что покрытие получается очень поверхностным и мы не можем быть уверены в качестве системы.

How to choose the best technique?

**Выбор техник тестирования.**

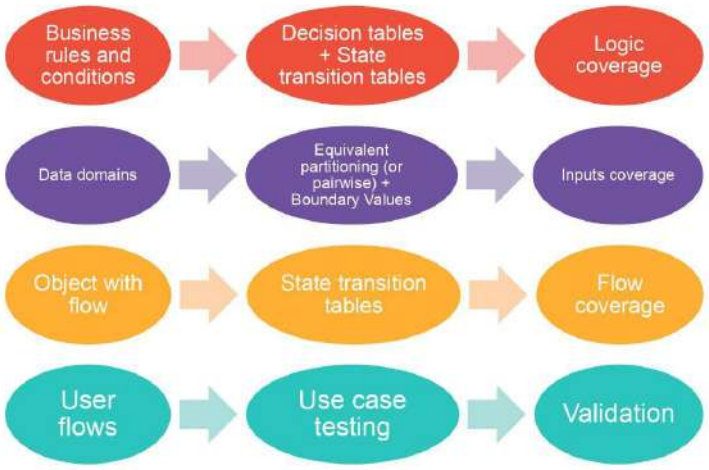
К сожалению, в жизни нет четкого алгоритма выбора техник тестирования, и вы не обойдетесь 1-2 подходами в большом проекте.




Мы использовали интернет магазин для рассмотрения каждой техники. Если попробовать очень сильно упростить выбор подхода, то он представлен на слайде ниже. И выглядит достаточно просто:

## Try to keep it simple

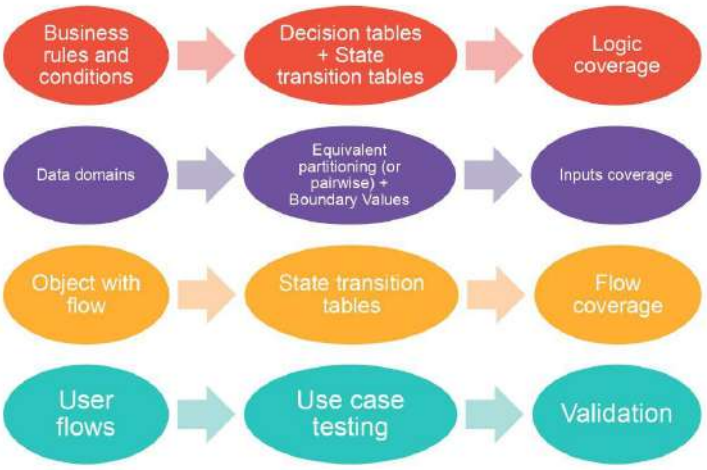
DataArt




1. Если у вас есть сложная бизнес логика, со сложными правилами и условиями. Соответственно выбираем таблицы решений, так как они не проверяют краевые условия, поэтому дополняем их краевыми условиями. И получаем хорошее покрытие бизнес логики системы.
2. Если у нас есть сложные входные данные, то мы используем их в разных сочетаниях в зависимости от ситуации. Эквивалентные разбиения, либо заменяем их на pairwise, если эквивалентные разбиения для нас получаются слишком накладными. И дополняем граничными условиями и получаем хорошее покрытие входных данных и их обработки.
3. Если у нас есть в системе некие объекты со сложным flow, применяем state transition и получаем покрытие этого flow.
4. Если мы хотим проверить пользовательские flow, то используем use case и на выходе получаем оценку системы с точки зрения пользователя и что-то похожее на оценку валидации системы.

## Try to keep it simple

DataArt





В чем подвох? Подвох в том, что если вы к каждому кусочку системы подберете подходящий способ тестирования и протестируете их, то как вы это будете собирать вместе?

Разные методы дают разную глубину покрытия, где то они будут пересекаться, где то будут оставаться серые зоны, все это нужно будет аккуратно собирать вместе, анализировать, убирать пересечения, закрывать серые зоны и т.д.

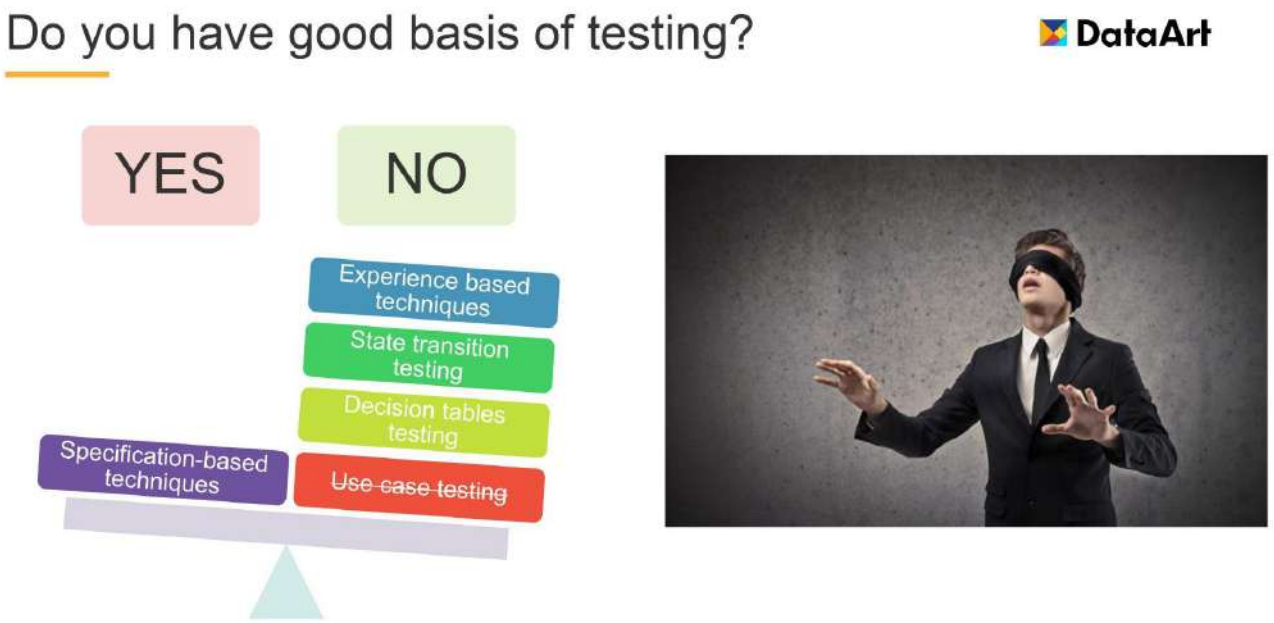
## What to consider?

DataArt

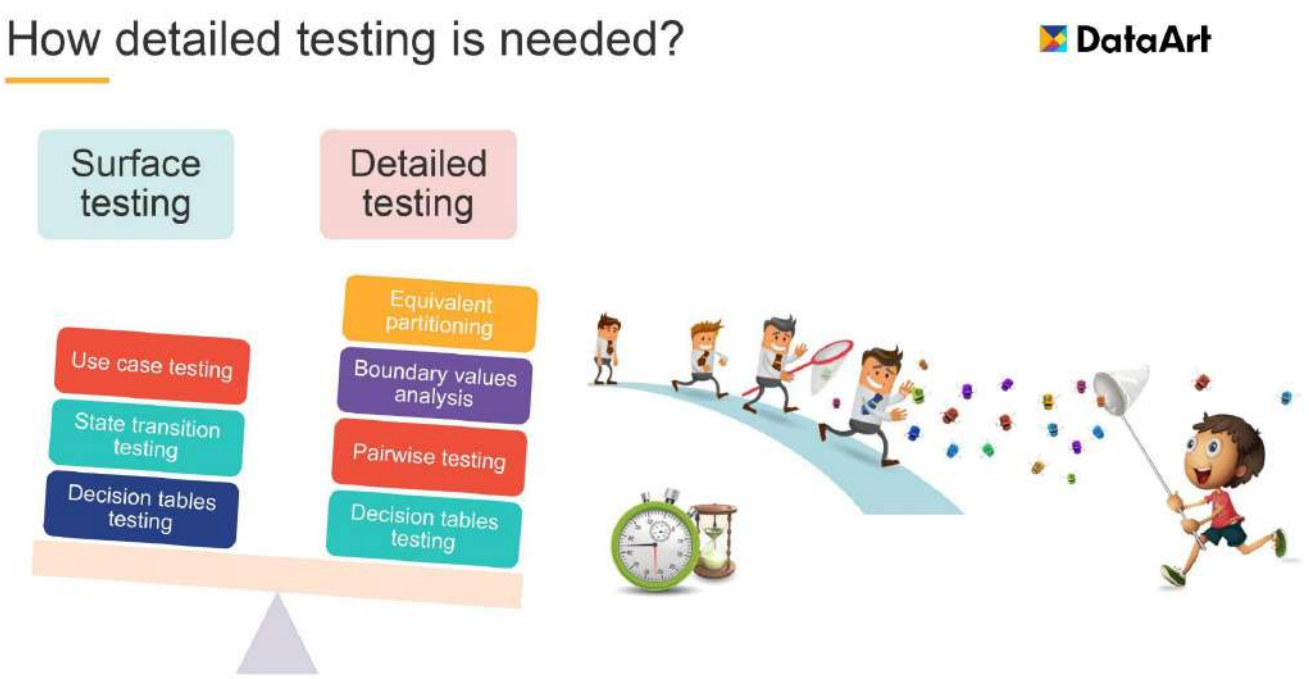

Отчего может зависит выбор техник тестирования? На слайде не весь список. Разберем несколько примеров для общего понимания как это использовать.

Во-первых, у нас есть наш базис тестирования, если есть use case, то грех их не использовать для тестирования, если их нет, то вопрос, стоит ли тратить время, а значит деньги чтобы их написать? *Так ли нужны тесты основанные на use case?.* Если у нас например сильно начинающие

тестировщики у них может не хватить знаний и понимания системы и он не сможет нарисовать хорошую диаграмму.
*Например, при тестировании важен тип системы, как ранее мы упоминали банковскую систему для которой 200 кейсов это были мелочи, хотя для среднестатистического интернет магазина это скорее всего будет много. Т.е цена ошибки в разных системах разная, уровень требований к качеству соответственно тоже разный.*

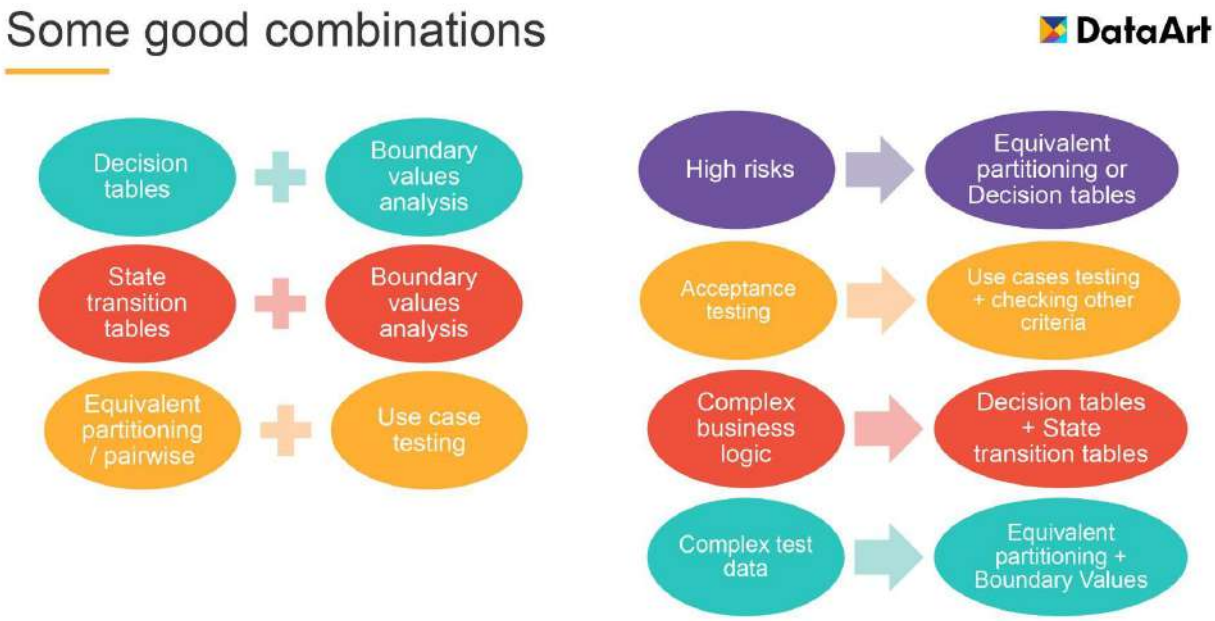



Что касается базиса тестирования, в принципе во многом определяет выбор техник. *Есть ли у нас вообще требования?* Какие то требования конечно есть в любом проекте. Почти не бывает, чтобы их не было совсем, но это могут быть требования настолько высокоуровневые и плохо проработанные, что использовать их как базис для тестирования, для приемочного еще возможно будет, *если заказчика это удовлетворяет*, а для системного уже нет. И тогда нам лучше не ориентироваться на те методы, которые мы изучали, а использовать методы основанные на опыте, на структуре, но при этом кое-что из того, что мы изучали вполне может пригодиться. Например state transition диаграммы вы можете рисовать с вашим архитектором не на основе требований, а на основе того что он вам расскажет. Или с заказчиком, если мы рассматриваем уровень не внутренних объектов системы и их flow, а более высокоуровневые требования. Т.е. вы фактические прорабатываете требования с заказчиком рисуя statechart диаграмму. Точно также можно рисовать decision tables, его можно использовать и в исследовательском тестировании, а не только в тестировании основанном на требованиях. Use case тестирование применить не получится потому, что заказчик не сможет дать корректную информацию о действиях пользователя из-за сложности предметной области. А выдумывать на пустом месте Use case - это путь в никуда. Вероятность того, что это будет соответствовать потребностям пользователей - очень мала.




Необходимая глубина тестирования.

С одной стороны разные методики дают нам разную глубину тестирования, с другой стороны в разных системах требуется разная глубина тестирования. Потому, что требуется разный уровень качества. Соответственно это тоже хорошее основание, для выбора методик тестирования.




На слайде изображены варианты хороших комбинаций. Как мы уже говорили тестирование граничных значений хорошо дополняет таблицы решений. Точно также хорошо дополняет тестирование граничных значений - подход state transition tables. И когда мы используем такие объемные подходы, как эквивалентные разбиения, часто бывает, что мы упускаем общий user flow. Концентрируясь на маленьких детальных тестах, мы очень часто пренебрегаем end to end тестами, считая что у нас с большим запасом покрыт каждый этап этого end to end теста. Поэтому когда у нас есть  глубокое

детальное тестирование, такое как с помощью эквивалентных разбиений, неплохо бы добавлять его end to end тестами на основе use case.

Набор решений для ситуаций в проекте:

- если у нас есть проект с высокими рисками нам нужно глубокое тестирование, его нам дадут методы эквивалентных разбиений и таблица решений.
- для acceptance тестирования, как мы говорили хороши use case, но часто их недостаточно, потому, что обычно acceptance тестирование подразумевает проверку acceptance критериев, основанных на требованиях, требования бывают как функциональные, так и не функциональные и их use case покрывать не будут с одной стороны. С другой стороны acceptance критерии могут быть не связаны с поведениями системы, а могут быть основанными на качестве сопроводительной документации, на каких то метриках по разработке. Об этом нужно помнить, когда вы планируете и согласуете ваш план acceptance тестирования с заказчиком. А он всегда должен быть согласован с заказчиком. Чтобы потом не возникло проблем с ним.
- если у нас есть сложная бизнес логика, то ее мы проверяем там где возможно, с помощью state transition диаграмм /таблиц и где это нужно с помощью таблицы решений.
- если у нас есть сложные тестовые данные, их мы проверяем эквивалентными разбиениями и анализом граничных значений.

## Develop a testing approach

DataArt

Есть отдельный раздел в бизнес плане, это “Поход к тестированию”. Подход к тестированию - это раздел тест плана, в котором вы должны собрать в кучу все методы, проанализировать какую часть вы тестируете каким методом, как вы все это собираете, как закрываете серые зоны, какие у вас есть ограничения и т.д.

Этот раздел требует детальной проработки для сложных систем не просто так. Подход к тестированию - это часть Стратегии тестирования

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

Boundary Values Analysis training

Ссылка на Тренажер для тестирования граничных значений.

Questions


1) Как можно под запись “рыбу” use case? Части которые должны присутствовать при составлении use case обязательно?
   Заголовок use case - описывает цель пользователя (как авторизированный пользователь, я хочу купить товар) .
2) Чем statechart диаграмма отличается от mindmap?
   Statechart диаграмма - это диаграмм, которая описывает состояния какого то объекта, mindmap - это способ оформления какой-то информации в виде дерева с возможными связями.

3) Над проектом обычно работает команда тестировщиков, каждый выбирает сам метод тестирования?
   Как правило в команде есть QA lead, который верифицирует, ревьюит то, что придумали другие участники команды, либо разрабатывает сам стратегию и подход к тестированию в зависимости от того насколько QA lead и другие члены команды различаются по квалификации.
4) Вопрос про state transition. Событие вызывает новое состояние и новое состояние вызывает действие системы или событие провоцирует действие системы и из-за этого система переходит в новое состояние?
   Событие инициирует переход из одного состояния в другое и этот переход может сопровождаться действием или набором действий. Отметки действия, строго во время до или после перехода, здесь все эти действия мы учитываем.
5) Был пример с водой и вы сказал, что таяние это действие, а на мой взгляд это переход, тогда чем отличается действие от перехода?
   Здесь различие сугубо терминологическое. Переход сопровождается событиями и действиями. Переход - это смена состояния. Т.е. у нас состояние с одного на другое изменилось - это и есть переход. Переход - это сам факт смены состояния.