

Введение в тестирование  
Урок 4

# Техники тест-дизайна





## Оглавление

На этом уроке	3
Техники тест-дизайна	3
Классы эквивалентности	4
Граничные значения	8
Попарное тестирование	10
Тестирование состояний и переходов	13
Таблицы принятия решений	17
Исследовательское тестирование	20
Заключение	23
Контрольные вопросы	24
Дополнительные материалы	24



## На этом уроке

1. Узнаем о техниках тест-дизайна.
2. Познакомимся с классами эквивалентности и граничных значениях.
3. Познакомимся с попарным тестированием и тестированием состояний и переходов.
4. Ознакомимся с таблицами принятия решений.
5. Познакомимся с исследовательским тестированием.

## Техники тест-дизайна

Разработка тест-кейсов — важный этап в жизненном цикле тестирования. От того, насколько правильно написаны тест-кейсы, может зависеть весь процесс тестирования. Создавать эффективные тест-кейсы позволяют техники тест-дизайна.

В начале тестирования мы анализируем требования: определяем, насколько они полные, чёткие, тестируемые. Когда требования проверены, проанализированы и приоритизированы, начинается этап написания тест-кейсов — это и есть тест-дизайн.

**Тест-дизайн** — этап тестирования ПО, на котором проектируются и создаются тестовые случаи (тест-кейсы). Они соответствуют определённым ранее критериям качества и целям тестирования.

Важно, что критерии качества и цели тестирования должны быть определены до начала написания тест-кейсов. От этих критериев и целей зависит, какими будут тест-кейсы, для каких модулей они будут описаны в первую очередь, проверка каких функций будет приоритетной.

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

**Задачи тест-дизайна** на проекте:

- максимально покрыть функциональность тестами;
- обнаружить серьёзные баги;
- сократить количество тестов, исключив непродуктивные;
- не пропустить важные тесты.



Как мы говорили раньше, нельзя провести исчерпывающее тестирование. Поэтому нужно применять разные техники, чтобы выполнить его эффективно и вовремя, избежав при этом проверки лишних кейсов. При этом вся функциональность должна быть покрыта тестами.

Кроме того, нужно попытаться составить тесты так, чтобы с их помощью можно было обнаружить критичные дефекты. Нельзя выявить все баги и убедиться в их отсутствии, но усилия и внимание тестировщиков должны быть направлены на поиск самых серьёзных дефектов.

Если времени и специалистов мало, важно исключать из работы непродуктивные тесты, которые не обнаруживают ошибок. В этом тоже помогут техники тест-дизайна.

Есть разные наборы техник тест-дизайна. И сочетания, и названия в разных источниках могут отличаться. Сегодня мы обзорно рассмотрим основные техники, а подробнее остановимся на каждой в одном из следующих курсов.

На этом уроке мы познакомимся со следующими техниками тест-дизайна:

1. Классы эквивалентности (эквивалентное разделение).
2. Граничные значения (анализ граничных значений, метод граничных значений).
3. Попарное тестирование (тестовая комбинаторика, pairwise).
4. Тестирование состояний и переходов.
5. Таблицы принятия решений.
6. Исследовательское тестирование.

## Классы эквивалентности

**Класс эквивалентности** — набор данных, которые обрабатываются одинаковым образом и приводят к одному результату.

Например, в требованиях есть условие для посещения онлайн-кинотеатра: «Возраст пользователя — от 16 лет и старше». Результат для пользователей, которые указывают возраст меньше 16 лет (не важно, 5 или 15), всегда должен быть одинаковым — сообщение «Извините, в связи с политикой сайта вы не можете пользоваться сервисом». Так же и со значениями от 16 и выше — не важно, какой возраст укажет пользователь (16, 23, 75, 99 лет), результат будет одинаковым: «Добро пожаловать в наш кинотеатр. Желаем приятного просмотра!»



Если известно, что есть группа данных, использование которых приводит систему в одно и то же состояние, нет необходимости проверять каждое значение из этой группы отдельно. Исключения возможны, но мы не можем проверять все данные, так что приходится прибегать к подобным допущениям.

Тестирование на основе классов эквивалентности (equivalence partitioning) — техника тест-дизайна на основе метода чёрного ящика: специалист не знает, как устроена система, и проходит все шаги тестов, используя только те инструменты, которые доступны пользователю.

Цель техники — обеспечить максимальную проверку всех требований тестами. Разделяя данные на классы эквивалентности и выбирая лишь несколько значений из каждого, можно существенно повысить эффективность и скорость тестирования, разрабатывать и выполнять меньше тест-кейсов.

Есть два признака, что данные в тесте относятся к одному классу эквивалентности:

1. **Если один тест выявит ошибку, остальные, скорее всего, тоже это сделают.** Если в тестах используются значения из одного класса эквивалентности, и один из тестов выявляет ошибку, остальные тесты, построенные на данных из этого класса, тоже должны обнаружить эту ошибку. Например, если онлайн-кинотеатр позволяет пользователю в возрасте 14 лет зарегистрироваться на сайте, то, вероятнее всего, регистрация будет возможна и для пользователей, указавших возраст 5, 10, 12 лет. А по требованиям это ошибка.
2. **Если один из тестов не выявит ошибку, остальные, скорее всего, тоже этого не сделают.** Если пользователю, указавшему возраст 15 лет, было отказано в регистрации на сайте, то нет смысла перебирать все значения от 0 до 15 лет. Вероятнее всего, они тоже обработаются корректно.

Так как в тестировании нельзя быть уверенным в наличии или отсутствии ошибок, в описаниях часто встречаются комментарии «скорее всего», «с большой долей вероятности».

Рассмотрим несколько примеров определения классов эквивалентности.

В требованиях о найме у HR-отдела есть условие, которое автоматически распределяет резюме кандидатов в разные категории:

Возраст кандидата, лет	Статус резюме
------------------------	---------------



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

Посмотрим, как это могло бы выглядеть в коде приложения. Обратите внимание на знаки `>=` и `<=`. Они важны при определении класса эквивалентности и граничных значений.

```
If (applicantAge >= 0 && applicantAge <16)
    hireStatus="NO";
If (applicantAge >= 16 && applicantAge <18)
    hireStatus="PART";
If (applicantAge >= 18 && applicantAge <65)
    hireStatus="FULL";
If (applicantAge >= 65 && applicantAge <=99)
    hireStatus="NO";
```

Если проводить исчерпывающее тестирование и проверять все варианты, количество тестов составит минимум 100 (без учёта проверок отрицательных значений, значений больше 99, символов, пустого ввода и прочего). Выполнить их невозможно, поэтому нужно:

1. Разделить данные на классы эквивалентности.
2. Выбрать хотя бы одно значение из каждого класса эквивалентности для проверки.

Получаем следующие проверки:

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

Почему лучше выбрать именно эти значения, разберём позже, когда речь пойдёт о граничных значениях.

Из каждого класса эквивалентности мы выбрали значения, чтобы сократить количество тестов (теперь их 4 вместо 100). Любые другие значения из класса эквивалентности должны давать те же результаты, что и выбранные.



Аналогичные действия проводятся и с другими данными, которые используются системой, например, со временем.

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

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

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

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

Рассмотрим другой пример. Тестируется система таможенного контроля, которая обрабатывает паспортные данные выезжающего за границу и определяет, есть ли он в базе лиц, для которых выезд ограничен (например, из-за финансового долга). Здесь есть два класса эквивалентности:

1. Валидный — данных пересекающего границу нет в базе лиц, которым выезд запрещён. Выезд разрешён.
2. Невалидный — данные пересекающего границу внесены в базу. Выезд запрещён.

Так как в классах эквивалентности в этом случае будет много вариантов, нельзя ограничиваться только несколькими наборами тестовых данных из каждого класса. Для более эффективных проверок нужно разбить каждый класс эквивалентности на несколько подклассов: например, по полу, возрасту, размеру финансового долга,



времени внесения в базу. Затем следует выбрать из каждого подкласса несколько вариантов и провести тесты.

Кроме чисел, на классы эквивалентности можно разбить:

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

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

## Граничные значения

Когда тестировщик работает с линейными классами эквивалентности (диапазонами значений), может потребоваться определить границы диапазона, чтобы точно отнести значение к конкретному классу эквивалентности.

У каждого диапазона будет начальная и конечная граница — это места повышенного риска ошибок, так как разработчик может указать некорректный знак неравенства или задать ошибочную границу диапазона.

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





**Техника анализа граничных значений** (boundary value testing) — проверка поведения продукта на граничных значениях входных данных.

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

Алгоритм определения граничных значений:

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

Рассмотрим применение техники анализа граничных значений на знакомых примерах.

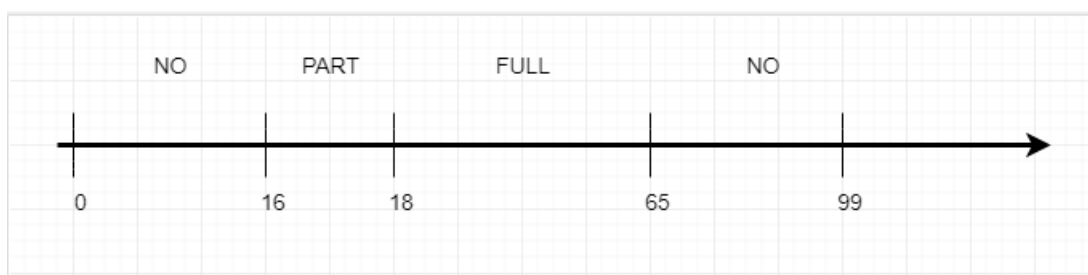
### Требования

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

1. Определяем граничные классы эквивалентности:

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

2. Выделяем граничные значения:



3. Определяем, к какому классу относится каждая граница:

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

4. Для каждой границы выделяем три значения:

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

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

На основании этих данных можно проводить тестирование.

## Попарное тестирование

Техники эквивалентного разбиения и анализа граничных значений — самые используемые в тестировании. На их основе формируется большинство проверок и тестов.

Следующие техники тест-дизайна, которые мы рассмотрим в этом и следующем уроке, не так популярны. Их применение зависит от особенностей тестируемого проекта.

Первая — попарное тестирование (pairwise). Рассмотрим несколько определений.

**Попарное тестирование** (pairwise testing) — техника формирования наборов тестовых данных, при которой каждое тестируемое значение каждого из



проверяемых параметров хотя бы раз сочетается с каждым из тестируемых значений всех остальных проверяемых параметров.

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

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

Техника применяется на проектах, где много параметров и их значений. Для примера возьмём сайт по поиску автомобилей.

Марка  
Марка

Модель  
Модель

Цена  
Цена от до, руб.

Тип автомобиля  
Все С пробегом Новые

Год выпуска  
от 1960 до 2019

Коробка передач  
☐ механика ☐ автомат

Ещё

Тип кузова  
седан хетчбэк универсал  
внедорожник кабриолет купе

Ещё

Марка  
Марка  
Bajaj  
Barkas  
BAW  
Bentley  
BMW  
Brilliance  
Bufo  
Bugatti  
Buick  
BYD

Коробка передач  
☐ механика ☐ автомат

Ещё

Тип кузова  
седан хетчбэк универсал  
внедорожник кабриолет купе

Ещё

Для поиска автомобиля пользователь может указать марку, модель, цену, тип автомобиля и другие параметры. У каждого из них множество возможных значений. У параметра «Коробка передач» — четыре: механика, автомат, робот, вариатор. У «Типа кузова» — седан, хетчбэк, универсал и другие. У параметров «Марка» и «Модель» несколько десятков значений. Если и другие параметры, проверить все варианты и их сочетания невозможно.

Есть теория, что большинство дефектов возникают при комбинации двух параметров. Если проверка будет состоять из параметров BMW + X6 + от 1 500 000



руб. до 4 000 000 руб. + с пробегом + от 2001 до 2019 года + автомат + хетчбэк + бензин, то ошибка с большей вероятностью возникнет из-за сочетания только двух из вышеперечисленных параметров.

Например, при совпадении пары «хетчбэк + бензин», а не из-за сочетания всех параметров одновременно. Поэтому есть смысл проверять сочетания двух значений разных параметров. Это сократит количество тестов и увеличит вероятность выявления дефектов.

Рассмотрим пример: нужно проверить форму, содержащую 20 чекбоксов (элементов страницы с двумя значениями — выключен/включен). Если проводить полный перебор для проверки сочетания всех значений, может потребоваться 1 048 576 тестов. Выполнить столько физически невозможно. Попарное тестирование позволяет сократить количество проверок до 10.

	чек-бокс1	чек-бокс2	чек-бокс3	чек-бокс4	чек-бокс5	чек-бокс6	чек-бокс7	чек-бокс8	чек-бокс9	чек-бокс10	чек-бокс11	чек-бокс12	чек-бокс13	чек-бокс14	чек-бокс15	чек-бокс16	чек-бокс17	чек-бокс18	чек-бокс19	чек-бокс20
1	on	off	off	off	off	off	off	on	off	off	on	on	on	off	off	off	on	on	on	off
2	off	on	on	on	on	on	on	off	on	on	off	off	off	on	on	on	off	off	off	off
3	off	off	off	on	off	on	off	off	on	off	off	on	on	on	on	on	on	off	on	on
4	on	on	on	off	on	off	on	on	off	on	off	off	off	off	off	off	off	on	off	on
5	off	off	on	off	off	on	on	on	on	on	on	on	on	off	off	on	off	on	off	on
6	on	on	off	on	on	off	off	off	off	on	on	off	on	on	off	on	off	on	on	off
7	off	off	off	off	on	on	on	off	off	off	off	on	on	off	on	on	off	on	off	on
8	off	off	on	on	on	off	off	on	on	off	on	off	off	on	off	off	on	off	on	on
9	on	on	on	off	on	off	off	off	off	off	on	on	on	on	on	off	on	off	off	on
10	on	on	off	on	off	on	on	on	on	off	on	off	off	off	off	on	off	on	on	off

В примере хорошо прослеживаются особенности техники попарного тестирования: она наиболее эффективна при большом количестве параметров, которые имеют ограниченное количество значений. В примере 20 чекбоксов, по два значения для каждого — это позволило ощутимо сократить количество тестов.

Если у параметра очень много значений, как у «Марки» в примере с сайтом по поиску автомобилей, сокращение проверок при применении техники попарного тестирования может быть незначительным. В этом случае удобнее разделить значения на два класса эквивалентности — валидный и невалидный. К невалидному классу будет относиться пустое поле или некорректные значения, а к валидному — все корректные значения.

Учитывая огромное количество тестируемых параметров и значений в приложениях, сгруппировать их по технике pairwise вручную невозможно либо очень трудозатратно. Для этого есть специальные инструменты, и один из популярных — [PICT](#).

В этом уроке мы рассмотрели самые популярные техники тест-дизайна, без которых не обходится практически ни один тестовый процесс. На следующем занятии



разберём техники, благодаря которым процесс тестирования может стать более структурированным и быстрым.

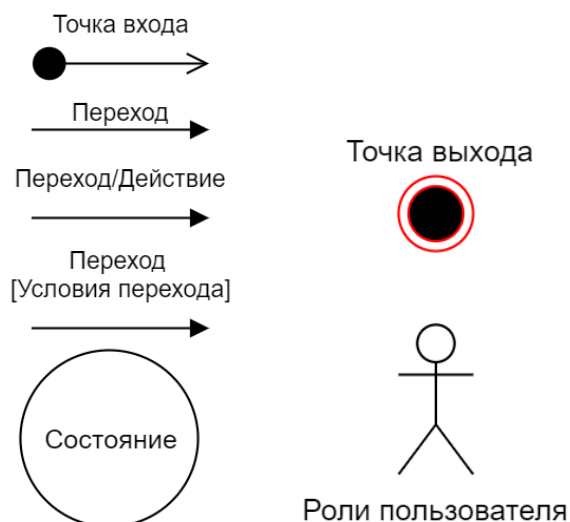
## Тестирование состояний и переходов

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

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

Чтобы избежать таких ошибок, можно использовать технику тест-дизайна «Тестирование состояний и переходов». Она позволяет составлять тестовые сценарии, основываясь на визуальном представлении состояний и переходов системы.

Прежде чем рассматривать эту технику, познакомимся с основными понятиями, которые используются при составлении диаграмм переходов и состояний.



**Точка входа** — старт работы системы или приложения.

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

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

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

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

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

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

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

Классический пример — бронирование авиабилетов. Начнём с позитивного сценария: пользователь успешно проходит весь цикл бронирования, включая оплату и использование билета. Всегда стоит начинать с позитивных проверок, чтобы убедиться, что система работоспособна и выполняет ключевые функции.

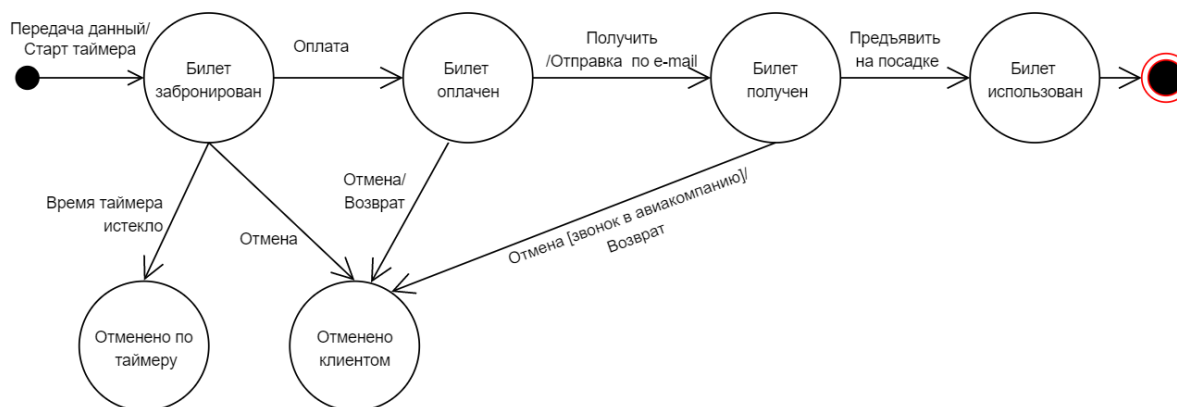


Если это не так, дальнейшее тестирование не имеет смысла до устранения дефектов.



1. Точкой старта будет вход в систему бронирования и выбор нужного билета. Затем пользователь передаёт информацию, нужную для бронирования (имя и фамилию, паспортные данные), и нажимает кнопку «Забронировать». Нажатие можно считать событием. После него стартует таймер до окончания срока оплаты. Система перешла в первое состояние — «Билет забронирован».
2. Дальнейшее событие — «Оплата билета». Переводит систему в следующее состояние — «Билет оплачен».
3. Затем по событию «Получить билет» система должна выполнить действие «Отправка билета по email» и перейти в другое состояние — «Билет получен».
4. Последним звеном в этой цепочке будет событие «Предъявление билета при посадке» (в примере часть событий пропущена — в реальных проектах их, конечно, может быть намного больше). Состояние «Билет использован» — цикл бронирования успешно завершён, система попадает в точку выхода.

Рассмотренный сценарий — позитивный, он не предполагает дополнительных действий пользователя. Это, конечно, невозможно, так как не всегда бронирование билета должно заканчиваться его использованием. Пользователь может не оплатить билет или оплатить, но потом отменить и прочее. Эти состояния также нужно отразить на диаграмме.



Так как пользователь может забронировать билет, но не оплатить его, бронь отменится, когда время на оплату истечёт. В таком случае нужно добавить в диаграмму состояние «Отменено по таймеру». А также «Отменено клиентом», в которое система может перейти из трёх предыдущих: «Билет забронирован», «Билет оплачен», «Билет получен».

При переходе из «Билет забронирован» в «Отменено клиентом» пользователю достаточно просто произвести событие «Отмена». Если билет уже был оплачен, действием системы должен стать возврат денежных средств — «Возврат».

Предположим, что для перехода системы из состояния «Билет получен» в «Отменено клиентом» помимо отмены билета пользователь должен выполнить условие перехода — позвонить в авиакомпанию. Это тоже нужно отразить на диаграмме. Условие перехода указывается над стрелкой в квадратных скобках.

Так визуализируется работа системы бронирования. Можно сразу увидеть состояния, которые принимает система, и условия для их изменения. Такая визуализация актуальна для сложных проектов с множеством состояний, переходов и условий для них. Она позволяет не пропустить важные звенья системы и наиболее полно описать тестовые сценарии для проверки. Например, первым сценарием может стать проверка полного цикла работы системы от входа в неё до точки выхода. Затем можно выполнять тестирование более детально, добавляя новые сценарии на основании диаграммы переходов и состояний.

При построении диаграмм состояний и переходов важно:

- не допускать пересечения линий переходов — это усложняет визуальное восприятие диаграммы и может привести к ошибочному переходу;





- сложные процессы лучше представлять в виде нескольких диаграмм — если охватить всё одной схемой, она будет слишком трудной для понимания;
- главную последовательность состояний следует размещать на одной горизонтальной линии, чтобы прослеживался позитивный сценарий работы системы. Дополнительные состояния можно представить в виде ответвлений и разместить по бокам от основной последовательности.

Плюсы диаграмм состояний:

- позволяют визуализировать состояния продукта;
- демонстрируют варианты переходов, которые можно пропустить;
- помогают отследить дефект, сужая его локацию до конкретного перехода;
- показывают внутреннюю механику продукта.

Минусы:

- можно пропустить неочевидные переходы;
- при слишком сложной структуре продукта диаграммы могут стать громоздкими и запутанными;
- являются только основой к применению других методов;
- бесполезны при плохом знании продукта.

## Таблицы принятия решений

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

**Таблицы принятия решений (таблицы решений)** — способ компактно представить модели со сложной логикой. А ещё это техника тестирования чёрного ящика, которая применяется для систем со сложной логикой.

Таблицы принятия решений используют, чтобы упорядочить и задокументировать сложную логику приложения, а также протестировать все комбинации условий и состояний.



Рассмотрим сущности, из которых состоят таблицы.

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

**Действия (actions)** — чёткое описание ожидаемого результата, действия системы. Формулировка действия — утвердительное предложение. Одно предложение обязательно описывает только одно действие. Например: «Данные заполнены автоматически», «Сообщение об ошибке отображается на экране».

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

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

Рассмотрим составление таблицы на примере.

*Требование:* для поддержания системы лояльности провести информационную рассылку постоянным клиентам.

Содержание писем зависит от следующих условий:

1. Клиенты типа А, В получают стандартное письмо.
2. Клиенты типа С получают специальное письмо.
3. Клиентам, совершившим пять и более покупок или купившим на сумму более 500 долларов, в письме сообщается о дополнительной скидке 20% на следующий заказ.

Начнём составлять таблицу по плану:

1. Разбить требование на условия.
2. Посчитать количество возможных правил (комбинаций).
3. Составить таблицу принятия решений.
4. Исключить лишние комбинации, если они есть.
5. Создать тесты.

Теперь разберём каждый пункт.

### **1. Разбить требование на условия.**

Можно выделить три условия:



- тип клиента;
- пять и более покупок;
- сумма больше 500 долларов.

## 2. Посчитать количество возможных правил (комбинаций).

Расчёт можно выполнить по формуле  $X = Y_1 \cdot Y_2 \cdot \dots \cdot Y_n$ , где:

- $X$  — вычисляемое количество комбинаций;
- $Y_1 \dots Y_n$  — количество вариантов каждого условия;
- $N$  — количество условий.

Таким образом, получим:

- $Y_1 = 4$  (четыре значения для условия «Тип клиента» — «A, B, C, D»);
- $Y_2 = 2$  и  $Y_3 = 2$  (по два значения для условий «Пять и более покупок» и «Сумма больше 500 долларов» — «YES/NO»);
- $N = 3$  (требование содержит три условия);
- $X = 4 \cdot 2 \cdot 2$ ;
- $X = 16$  правил (комбинаций условий).

## 3. Составить таблицу принятия решений.

Заносим в таблицу условия, значения и правила следующим образом:

Условия	Значения	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Тип клиента	A, B, C, D	A	A	A	A	B	B	B	B	C	C	C	C	D	D	D	D
5 и более покупок	Y, N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Сумма больше 500 дол	Y, N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
<b>Действия</b>																	
Стандартное письмо		X	X	X	X	X	X	X	X					?	?	?	?
Специальное письмо										X	X	X	X	?	?	?	?
Сообщение о скидке		?	X	X		?	X	X		?	X	X		?	?	?	?
Не получают письмо														?	?	?	?

В таблицу добавили тип клиента «D» — это все остальные типы клиентов, если будут выявлены те, которые не подпадают под характеристики для клиентов типа «A, B, C».

Для правил, которые не отражены в требованиях, использован «?» (в требованиях не указано, какое письмо должно быть отправлено, когда сочетаются условия «более пяти покупок» и «сумма больше 500 долларов», а также как поступить с клиентами типа D). Ситуации, помеченные знаком вопроса, надо прояснить с аналитиком или заказчиком.



Первая строка в таблице формируется так: количество всех правил (комбинаций) делится на количество значений первого условия. То есть 16 (число правил) делим на 4 (число значений для условия «Тип клиента»). Получаем ряд из четырёх одинаковых значений подряд (см. таблицу выше). Заполняя остальные строки, нужно соблюдать последовательность: каждая следующая строка — это предыдущая строка, разделённая пополам. То есть в первой строке каждое значение повторялось четыре раза подряд, во второй — два раза, в третьей происходит чередование значений. Если бы в таблице было ещё одно условие, то в следующей строке каждое значение снова повторялось бы четыре раза, потом два раза и так далее.

Также в таблице указываем действия, которые произойдут при совпадении тех или иных условий. И отмечаем, какое именно действие выполняется при совпадении условий. В данном случае выделим четыре действия: «Стандартное письмо», «Специальное письмо», «Сообщение о скидке», «Не получают письмо».

#### 4. Исключить/добавить комбинации.

В этом случае были добавлены комбинации для дополнительного типа клиента «D». Могут возникать ситуации, когда в таблице появятся комбинации условий, которые на практике невозможны. Например, если тестируется форма с двумя кнопками «Сохранить» и «Отменить», каждая кнопка имеет два значения: «Нажата» / «Не нажата». Одновременно обе кнопки не могут принимать значение «Нажата» — значит, такая комбинация должна быть исключена из таблицы.

#### 5. Создать тесты.

В результате получаем тестовые сценарии, которые можно либо перенести в тест-кейсы, либо оставить в таблице и добавить строку с результатом проверки.

В этом случае:

- **Draft** — тест, который требует уточнения условий;
- **Failed** — тест, который прошёл неуспешно, например, был выявлен дефект;
- **Passed** — тест, который прошёл успешно, функциональность работает.

Подробнее о статусах тест-кейсов поговорим в следующем уроке.

Повторим: таблицы принятия решений хорошо работают для систем со сложной логикой, в которых используется много условий типа «если ..., то ...».

Плюсы таблиц принятия решений:



- помогают быстро составлять тестовые сценарии;
- позволяют выявить неполноту требований;
- их можно использовать при отсутствии требований;
- можно быстро проверить покрытие требований тест-кейсами;
- позволяют предугадывать возможные ошибки.

Минусы:

- при большом количестве условий таблицы могут быть громоздкими — их сложно составлять и использовать;
- сложность в корректном определении условий, действий и значений при первоначальном проектировании.

## Исследовательское тестирование

Исследовательское тестирование не всегда относят к техникам тест-дизайна. Но по его результатам могут составлять тест-кейсы, поэтому рассмотрим его как технику. Напомним, что исследовательское тестирование — это подход, когда тестирующий не использует тест-кейсы, а тестирует приложение по определённому сценарию, который часто составляется прямо во время проверки.

Тестирующий проводит исследовательское тестирование приложения, в результате которого выявляются дефекты. Тот сценарий (тест), который выявил дефект, нужно задокументировать (создать тест-кейс), чтобы в дальнейшем проверять, что дефект исправлен и не появился вновь. Кроме того, стоит создать тест-кейсы (если их нет) и для проверки похожих сценариев, чтобы обнаружить другие подобные дефекты. В некоторых случаях проверки, проведённые при исследовательском тестировании, следует документировать (создавать тест-кейс), даже если они не обнаружили дефект. Это нужно, чтобы повторять проверки в будущем, в том числе при регрессионном тестировании.

Таким образом, исследовательское тестирование как техника тест-дизайна позволяет дополнять наборы тест-кейсов новыми тестами, а также создавать актуальные тест-кейсы, которые выявляют дефекты.

Исследовательское тестирование также используют как вспомогательный подход к тестированию по тест-кейсам. Оно помогает исключить эффект пестицида (когда



тест-кейсы перестают выявлять дефекты) при частом использовании одних и тех же тест-кейсов.

Ещё случаи, когда исследовательское тестирование может быть эффективным:

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

Плюсы исследовательского тестирования:

- не нужно тратить время на предварительное описание всех сценариев;
- не нужна поддержка тестовых сценариев;
- нет привыкания к тестовым сценариям, их прохождение не происходит «не глядя»;
- не теряется цельное видение продукта;
- критические дефекты находятся быстрее;
- повышается скорость тестирования;
- можно сразу начинать тестировать продукт, даже если требований нет вообще;
- исследовательское тестирование интереснее и креативнее (тесты ограничиваются только фантазией и глубиной знаний о продукте).

Минусы:



- сложно планировать время на проведение тестирования без задокументированных заранее сценариев;
- вероятность пропустить ключевые проверки, так как отсутствует ранжирование сценариев по степени важности;
- сложность оценки полноты покрытия требований тестами;
- требуется высокая квалификация тестировщиков и хорошее знание тестируемого приложения;
- сложно использовать для регрессионного тестирования;
- невозможно автоматизировать такое тестирование.

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

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

Тест-туры — неформальный подход, который позволяет сделать исследовательское тестирование ещё более осмысленным и интересным. Они содержат инструкции по исследованию приложения — по аналогии с тем, как турист изучает город. Подробнее с этим способом работы с приложением можно познакомиться в статьях:

- [Исследовательское тестирование и исследовательские туры Виттакера](#)
- [Как искать баги — исследовательские туры Уиттакера](#)

## Заключение

В этом уроке мы рассмотрели три техники тест-дизайна: тестирование переходов и состояний, таблицы принятия решений и исследовательское тестирование. Они



помогают сделать тестирование более эффективным. Конечно, для применения этих техник на практике нужны навыки, они нарабатываются постепенно. Главное — помнить о существовании техник и пробовать внедрять их, учитывая особенности проекта.

## Контрольные вопросы

1. Для чего нужны техники тест-дизайна?
2. В чем заключается техника эквивалентного разбиения?
3. В чем заключается техника граничных значений?
4. В чем заключается техника попарного тестирования?
5. В чем заключается техника тестирования состояний и переходов?
6. Для чего нужны таблицы принятия решений?
7. Почему исследовательское тестирование относится к техникам тест-дизайна?

## Дополнительные материалы

1. [Тест-дизайн. Что это такое? Тест дизайн в тестировании ПО. Test design](#)
2. [Процесс тестирования. Часть 2: Анализ тестирования и тест дизайн](#)
3. [Кто такие тест-дизайнеры и зачем они нужны](#)