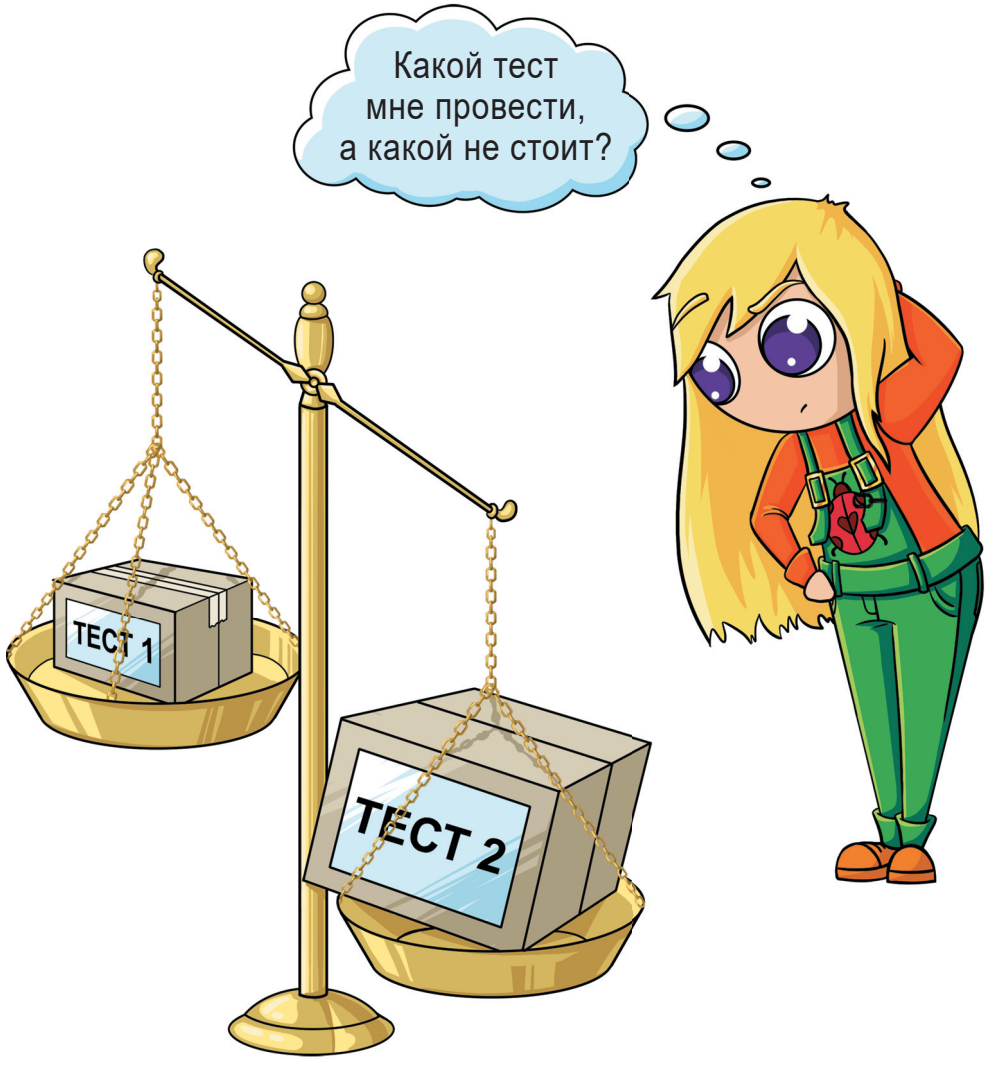
ЧТО ТАКОЕ ТЕСТ-ДИЗАЙН И ЗАЧЕМ ОН НУЖЕН?



Тест-дизайн — это набор техник, которые позволяют нам приложить мало усилий и получить много результата. В терминах тестирования это означает: провели минимум тестов, нашли максимум багов.

Ведь если в арсенале есть только молоток, вам придется им и гвозди забивать, и винтики закручивать. Или наоборот, отверткой гвозди забивать, что можно, но очень неудобно.

Поэтому приятно иметь доступ к самым разным техникам — когда есть из чего выбирать, задачи решаются быстрее и приятнее.

Итак, основные техники тест-дизайна:

Boundary Value Testing	Классы эквивалентности
Equivalence Class Testing	Граничные значения
State & Transition Diagram Testing	Схемы состояний и переходов
Decision Table Testing	Таблицы решений
User Case Testing	Варианты использования
Allpairs Algorithm testing	Попарное тестирование
...	...

Их мы и рассмотрим в следующих главах!

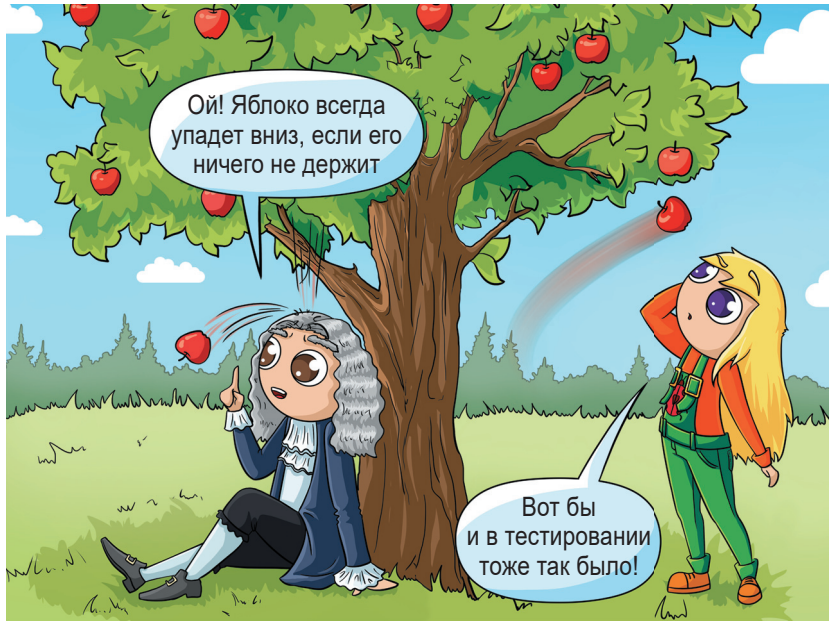
Тест-дизайн — это не наука

Но сначала стоит усвоить эту истину. Закон Ньютона — он и в Африке закон Ньютона, не меняется и всегда работает одинаково. В тестировании же «серебряной пули» нет. Нельзя один раз написать мегаклевый чек-лист и сказать, что он подойдет вообще подо всё.

В каждой программе свои баги, свои проблемы и особенности. Есть некоторые фишки, которые стоит проверять всегда, но всё равно даже универсальный чек-лист[1] вам придется подкручивать под особенности своей системы. Каждый

[1] Погуглите «чит-листы».

разработчик совершает какие-то свои ошибки, каждая функциональность хоть немного, но отличается от той, что вы видели раньше.



Поэтому техники тест-дизайна — это лишь набор эвристик, с помощью которых мы можем улучшить наши результаты[1]. А можем и не улучшить, если мы неправильно использовали эвристику. ;-)

Но в любом случае опираться на что-то нужно. Любая техника — это лучше бездумного прокликивания в надежде напороться на ошибку. Это как ориентир. Когда-то я была у Алексея Баранцева на тренинге, и он рассказал такую историю (я не помню ее точно, поэтому немного перевру. Главное, что останется смысл):

> Военная группа заблудилась где-то в Альпах. Генерал достает из-за пазухи карту и говорит: «Слушайте, вот у меня карта, я точно знаю, куда нам надо идти. Нам туда!»
>
> Группа пошла за генералом. Они шли-шли-шли и наконец вышли к людям. Те уже вызвали спасательную бригаду; она приехала, группу забрала, привезла обратно в штаб, и всё было хорошо.
>
> Но полковник подошел к генералу и попросил показать ему эту карту. Он взял карту, посмотрел на нее и говорит:

[1] Эвристика — совокупность исследовательских методов, способствующих открытию ранее неизвестного.

— Слушай, это же не Альпы, это Гималаи!

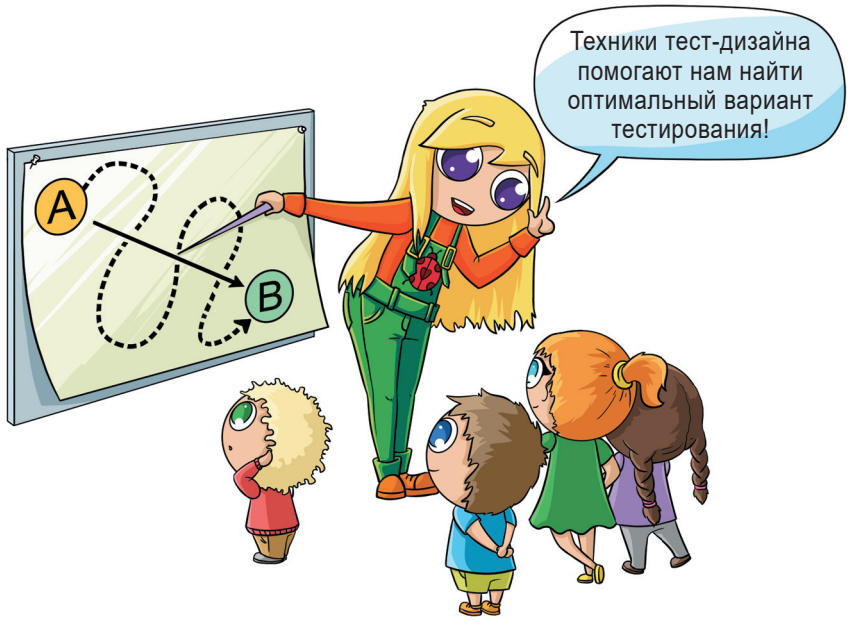
— Я знаю.

— Это не та карта. Как ты смог их вывести??

Генерал пожал плечами:

— Нам нужен был ориентир. Мы им воспользовались и смогли выйти. Без паники. Никто же не знал, что реальной карты нету. А если бы они остались на месте или стали плутать туда-сюда, то поднялась бы паника, в итоге ситуация бы усугубилась, и они бы просто замерзли.

Вот видите? Карта же не та была! Но они всё равно выжили. И даже не волновались, так как были уверены в том, что идут в правильном направлении. Точно так же в техниках тест-дизайна. Если вообще ничего не делать, вы никуда не придете и багов не найдете. А если вы будете хоть куда-то идти, пусть даже вы определили неправильные эвристики или неверно выделили классы эквивалентности, всё равно вы куда-нибудь да придете. Со временем вы научитесь корректировать свои ориентиры и находить более быструю дорогу к тому же самому результату.



Часть I

О ТОМ, КАК ПРОЕКТИРОВАТЬ ТЕСТЫ...

Когда нужно накидать чек-лист проверок на любую функциональность, мы проходим через несколько этапов:

1. Сгенерировать идеи «что тут можно проверить?»
2. Пополнить список «опасными точками».
3. Удалить всё лишнее (времени-то в обрез на проверку!)

Помогает нам в этом тест-дизайн. В главах этой части мы рассмотрим самые важные техники, которые помогают за минимум усилий получить максимум результата.

Если проектировать тесты плохо, то можно попасть впросак:

- не подумали о важной проверке — пропустили ошибку;
- накидали кучу идей, а потом выкинули «лишнее», среди которого были нужные тесты, — и снова пропустили баг.

Но как понять, что лишнее, а что нет? Ведь проводить исследование системы бесконечно просто нет времени. И в выделении «умных» тестов, покрывающих самое важное, нам помогут классы эквивалентности и граничные значения.

А потом при помощи анализа тестов и техники Pairwise мы научимся уменьшать их количество!

Глава 1
КЛАССЫ ЭКВИВАЛЕНТНОСТИ

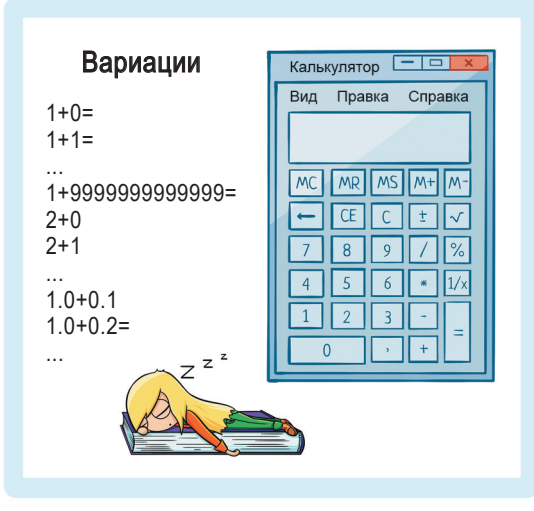
Классы эквивалентности — самая важная тема тест-дизайна. Именно они помогают нам уменьшить количество тестов, улучшив при этом результат. Давайте разбираться, как!

1.1. Что такое «классы эквивалентности»?

Классы эквивалентности в калькуляторе

Начнем сразу с примера, чтобы понятнее было. Допустим, мы разрабатываем новый модный калькулятор. Возможно, вы уже знаете этот пример, так как его используют во многих книгах по тестированию, показывают на конференциях... Не будем нарушать традиций!

Итак, мы разрабатываем калькулятор. На текущий момент готова только функция сложения, и нам надо ее проверить. Как будем проверять? Если у нас нет доступа к коду, мы не можем сказать уверенно, как разработчик эту функцию реализовал. И чтобы точно сказать, что «всё работает, багов нет», нам придется проверить сумму каждого с каждым:



- 1 + 0
- 1 + 1
- 1 + 2
- ...
- 1 + 999999999999999
- 2 + 0
- 2 + 1
- ....
- 2 + 999999999999999
- ...
- 9999999999999999 + 1
- ...

Уже получится очень много тестов, а это мы еще до дробных значений не дошли. Что, если один знак после запятой? А если два? А если три-четыре-десять? Тестов получаются триллионы. На одну простую функциональность...

Мы просто не можем позволить себе полный перебор всех значений, иначе на проверку одного только сложения уйдет несколько лет работы полной командой. А еще вычитание, умножение, деление, комбинации, у-у-у-у-у...

Ну и потом, если нам надо делать полный перебор — это значит, что и разработчик у нас слегка туповатый, и в коде тоже делает перебор вместо функции суммирования. Но в таком случае он бы тоже несколько лет писал простую функциональность, а раз он ее сделал за пару дней — значит, там всё как-то пооптимальнее работает.

Вот и нам надо пооптимальнее работать. Какие месяцы и годы? Нам и двух дней, да что уж там, порой двух часов не дадут на проверку функции. Что логично — функциональность простая, зачем ее долго тестировать?

Поэтому мы начинаем делать предположения. Наверное, если «2+2» работает, то и «2+3» будет работать, так как сама функция сложения работает. Проверки получаются эквивалентными друг другу, если вывести их на уровень абстракции «сложить два простых числа».

Потом мы начинаем думать, что еще можно складывать, кроме двух простых чисел? И сразу получаем следующее разделение.

- Что складывать?
- Как складывать?

Что можно складывать? Для этого надо подумать — а какие вообще бывают числа?

- Простые — которые делятся только на 1 и сами на себя.
- Обычные: 1, 2, 3... Те, что чаще всего используются.
- Большие значения — мы чуть позже поговорим о граничных значениях и обсудим, почему большие значения надо выносить в отдельный класс. Но фишка в том, что, если «1+1» работает, то «максимум + максимум» может и упасть. Ну и просто, мало ли, число не влезет на экран, что делать будем? Глазки потупив, говорить, что тестировали?

Мы предполагаем, что если проверено «2+2», то и «2+3» работать будет, эти значения эквивалентны

- Дробные — стоп, а какие возможны варианты их записи? Дробная часть через точку или запятую — уже два разных класса!

А как мы можем складывать?

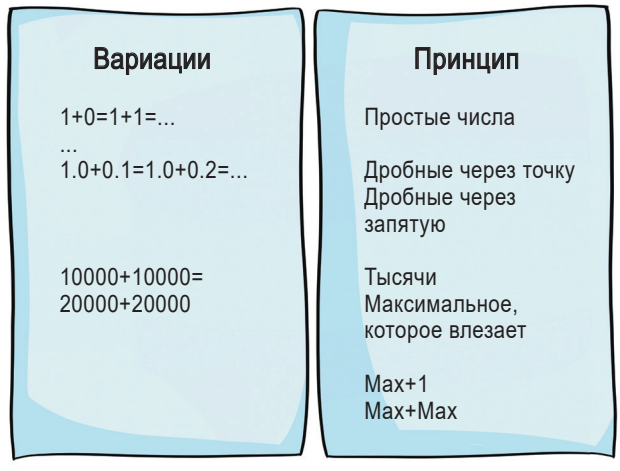
- Два числа — один знак сложения.
- Три и более — несколько знаков сложения.
- Одинаковые числа.
- Разные числа.

Все эти идеи — и есть набор тестов, которые нам стоит провести, чтобы проверить функцию сложения. Мы не можем позволить себе полный перебор всех комбинаций, поэтому стараемся уменьшить их количество.

Для этого нам приходится предполагать, что комбинации из одной группы равны по смыслу. То есть эквивалентны друг другу. Это и называется *классами эквивалентности* — группировка проверок по какому-то принципу. И вся соль тестирования — правильно определить принципы одинаковости:

Если класс будет слишком общий («сложение чисел») — мы пропустим баги, так как проверим только «2+2», а на больших или дробных значениях всё упадет.

Если класс будет слишком узкий (числа до 10, до 20…) — получится слишком много тестов. Поэтому вся наша задача сводится к наиболее правильному предположению о том, какие проверки дадут одинаковый результат, то есть какие из них можно выкинуть.

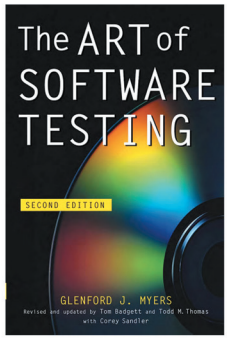


Главное — понять принцип разбиения.
Не слишком детализированный,
но и не слишком общий

Выводим определение

Обратимся к классикам.

Гленфорд Майерс в своей книжке «The ART of software testing» пишет следующее:



Glenford Myers

By identifying an equivalence class, we are stating that if no error is found by a test of one element of the set, it is unlikely that an error would be found by a test of another element of another element of the set.

Суть его позиции состоит в том, что, выделяя классы эквивалентности, мы формируем какую-то группу или подмножество данных и утверждаем, что если баг не находится на каком-то представителе из этой группы (мы его попробовали и убедились, что бага нет), то он, скорее всего, не найдется и на остальных представителях этой группы. И поэтому пробовать остальные нет смысла. Достаточно протестировать только одного представителя из этой группы. То есть, чтобы уменьшить количество тестов и количество усилий на тестирование, нужно данные разбить на классы эквивалентности.

Более точную формулировку предлагает Кем Канер:

Cem Kaner

If you expect the same result from two tests, you consider them equivalent. A group of tests forms an equivalence class if you belive that:

- *They all test the same thing.*
- *If one test catches a bug, the others probably will too.*
- *If one test doesn't catch a bug, the others probably won't either.*

Naturally, you should have reason to believe that test cases are equivalent.

Если мы от выполнения двух тестов ожидаем одинаковый результат, то можем считать их эквивалентными.

Группа тестов образует класс эквивалентности, если вы верите, что:

- Они все тестируют одно и то же.
- Если какой-то один представитель класса эквивалентности ловит баг, тот же самый баг, скорее всего, был бы пойман и любым другим представителем этой группы тоже.
- И наоборот, если какой-то один тест из группы баг не ловит и завершается успешно, то и остальные должны завершаться успешно.

Но! Это хоть и субъективно, но у вас должны быть причины верить в то, что какие-то тесты эквивалентны.

Тут много слов «верите, что», «скорее всего»... А всё почему? Потому что тест-дизайн — это не точная наука! Ну нету здесь четкого и формального разбиения на классы эквивалентности. Приходится полагаться на свое чутье. Или на опыт.

Но есть в этом и хорошая новость. Тестировщик имеет право *верить*, что тесты проверяют одно и то же. Что значит «одно и то же» непонятно, но он может верить. ;-)

Может быть, потом окажется, что тесты проверяют разное — и это будет опыт в копилку тестировщика. Он запомнит, что «дробные числа» и «простые числа» — разные вещи, разные классы. Или что свежесозданный пользователь не равен давно существующему. Или что-то еще...

И именно поэтому тестировщик с опытом находит больше ошибок. Потому что «прошелся по граблям» и запомнил, в каких местах ожидать подставы.