

## Лекция 14 Примеры использования различных техник

### тестирования 2.7.1. Позитивные и негативные тест-кейсы

Ранее мы уже рассматривали<sup>(148)</sup> алгоритм продумывания идей тест-кейсов, в котором предлагается ответить себе на следующие вопросы относительно тестируемого объекта:

- Что перед вами?
- Кому и зачем оно нужно (и насколько это важно)?
- Как оно обычно используется?
- Как оно может сломаться, т.е. начать работать неверно?

Сейчас мы применим этот алгоритм, сконцентрировавшись на двух последних вопросах, т.к. именно ответы на них позволяют нам придумать много позитивных<sup>(77)</sup> и негативных<sup>(77)</sup> тест-кейсов. Продолжим тестировать наш «Конвертер файлов<sup>(55)</sup>», выбрав для исследования первый параметр командной строки — SOURCE\_DIR — имя каталога, в котором приложение ищет файлы, подлежащие конвертации.

**Что перед нами?** Путь к каталогу. Казалось бы, просто, но стоит вспомнить, что наше приложение должно работать<sup>(56)</sup> как минимум под управлением Windows и Linux, что приводит к необходимости освежить в памяти принципы работы файловых систем в этих ОС. А ещё может понадобится работа с сетью.

**Кому и зачем оно нужно (и насколько это важно)?** Конечные пользователи не занимаются конфигурированием приложения, т.е. этот параметр нужен администратору (предположительно, это человек квалифицированный и не делает явных глупостей, но из его же квалификации вытекает возможность придумать такие варианты использования, до которых не додумается рядовой пользователь). Важность параметра критическая, т.к. при каких-то проблемах с ним есть риск полной потери работоспособности приложения.

**Как оно обычно используется?** Здесь нам понадобится понимание принципов работы файловых систем.

- Корректное имя существующего каталога:
  - Windows:
    - X:\dir
    - "X:\dir with spaces"
    - .\dir
    - ..\dir
    - \\host\dir
    - Всё вышеперечисленное с "\" в конце пути.
    - X:\
  - Linux:
    - /dir
    - "/dir with spaces"
    - host:/dir
    - smb://host/dir
    - ./dir
    - ../dir
    - Всё вышеперечисленное с "/" в конце пути.
    - /

И всё, т.е. в данном конкретном случае существует единственный вариант верного использования первого параметра — указать корректное имя существующего каталога (пусть вариантов таких корректных имён и много). Фактически мы получили чек-лист для позитивного тестирования. Все ли варианты допустимых имён мы учли? Может быть, и не все. Но эту проблему мы рассмотрим в следующей главе, посвящённой классам эквивалентности и граничным условиям<sup>(232)</sup>.

В настоящий момент важно ещё раз подчеркнуть мысль о том, что сначала мы проверяем работу приложения на позитивных тест-кейсах, т.е. в корректных условиях. Если эти проверки не пройдут, в некоторых совершенно допустимых и типичных ситуациях приложение будет неработоспособным, т.е. ущерб качеству будет весьма ощутимым.

**Как оно может сломаться, т.е. начать работать неверно?** Негативных тест-кейсов (за редчайшим исключением) оказывается намного больше, чем позитивных. Итак, какие проблемы с именем каталога-источника (и самим каталогом-источником) могут помешать нашему приложению?

- Указанный путь не является корректным именем каталога:
  - Пустое значение (“”).
  - Слишком длинное имя:
    - Для Windows: более 256 символов. (Важно! Путь к каталогу длиной в 256 символов допустим, но надо учесть ограничение на полное имя файла, т.к. его превышение может быть достигнуто естественным образом, что приведёт к возникновению сбоя.)
    - Для Linux: более 4096 байт.
  - Недопустимые символы, например: ? < > \ \* | " \0.
  - Недопустимые комбинации допустимых символов, например: “....\dir”.
- Каталог не существует:
  - На локальном диске.
  - В сети.
- Каталог существует, но к нему нет прав доступа.
- Доступ к каталогу утерян после запуска приложения:
  - Каталог удалён или переименован.
  - Изменены права доступа.
  - Потеря соединения с удалённым компьютером.
- Использование зарезервированного имени:
  - Для Windows: com1-com9, lpt1-lpt9, con, nul, prn.
  - Для Linux: “..”.
- Проблемы с кодировками, например: имя указано верно, но не в той кодировке.

Если погружаться в детали поведения отдельных операционных систем и файловых систем, данный список можно значительно расширить. И тем не менее открытыми будут оставаться два вопроса:

- Все ли эти варианты надо проверить?
- Не упустили ли мы что-то важное?

На первый вопрос ответ можно найти, опираясь на рассуждения, описанные в главе «Логика создания эффективных проверок»<sup>(147)</sup>. Ответ на второй вопрос помогут найти рассуждения, описанные в двух следующих главах, т.к. классы эквивалентности, граничные условия и доменное тестирование значительно упрощают решение подобных задач.



**Задание 2.7.а:** как вы думаете, почему в вышеприведённых чек-листах мы не учли требование о том, что SOURCE\_DIR не может содержать внутри себя DESTINATION\_DIR?

## 2.7.2. Классы эквивалентности и граничные условия

В данной главе мы рассмотрим примеры упомянутых ранее техник тестирования на основе классов эквивалентности<sup>(89)</sup> и граничных условий<sup>(90)</sup>. Если уточнить определения, получается:

!!!	<b>Класс эквивалентности</b> (equivalence class <sup>349</sup> ) — набор данных, обрабатываемых одинаковым образом и приводящих к одинаковому результату.
!!!	<b>Граничное условие</b> (border condition, boundary condition <sup>350</sup> ) — значение, находящееся на границе классов эквивалентности.
.....	Иногда под классом эквивалентности понимают набор тест-кейсов, полное выполнение которого является избыточным. Это определение не противоречит предыдущему, т.к. показывает ту же ситуацию, но с другой точки зрения.

В качестве пояснения идеи рассмотрим тривиальный пример. Допустим, нам нужно протестировать функцию, которая определяет, корректное или некорректное имя ввёл пользователь при регистрации.

Требования к имени пользователя таковы:

- От трёх до двадцати символов включительно.
- Допускаются цифры, знак подчёркивания, буквы английского алфавита в верхнем и нижнем регистрах.

Если попытаться решить задачу «в лоб», нам для позитивного тестирования придётся перебрать все комбинации допустимых символов длиной [3, 20] (это 18-разрядное 63-ричное число, т.е. 2.4441614509104E+32). А негативных тест-кейсов здесь и вовсе будет бесконечное количество, ведь мы можем проверить строку длиной в 21 символ, 100, 10000, миллион, миллиард и т.д.

Представим графически классы эквивалентности относительно требований к длине (см. рисунок 2.7.a).

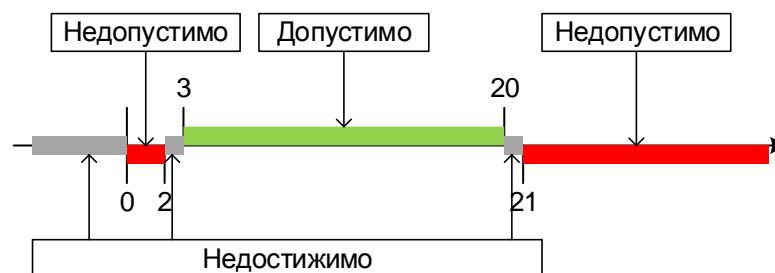


Рисунок 2.7.a — Классы эквивалентности для значений длины имени пользователя

Поскольку для длины строки невозможны дробные и отрицательные значения, мы видим три недостижимых области, которые можно исключить, и получаем окончательный вариант (см. рисунок 2.7.b).

Мы получили три класса эквивалентности:

- [0, 2] — недопустимая длина;
- [3, 20] — допустимая длина;
- [21, ∞] — недопустимая длина.

<sup>349</sup> An **equivalence class** consists of a set of data that is treated the same by a module or that should produce the same result. [Lee Copeland, «A practitioner's guide to software test design»]

<sup>350</sup> The **boundaries** — the «edges» of each equivalence class. [Lee Copeland, «A practitioner's guide to software test design»]

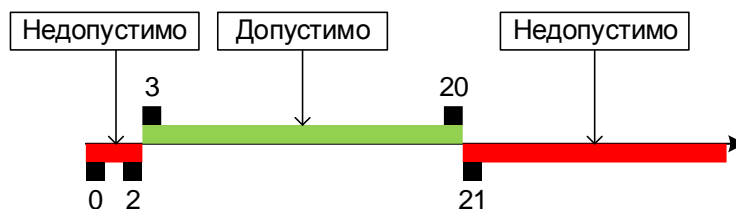


Рисунок 2.7.b — Итоговое разбиение на классы эквивалентности значений длины имени пользователя

Обратите внимание, что области значений  $[0, 2]$  и  $[21, \infty]$  относятся к разным классам эквивалентности, т.к. принадлежность длины строки к этим диапазонам проверяется отдельными условиями на уровне кода программы.

Граничные условия уже отмечены на рисунке 2.7.b — это 2, 3, 20 и 21. Значение 0 тоже стоит включить в этот набор на всякий случай, т.к. в программировании ноль, NULL, нулевой байт и т.п. исторически являются «опасными значениями».

В итоге мы получаем следующий набор входных данных для тест-кейсов (сами символы для составления строк можно выбирать из набора допустимых символов случайным образом, но желательно учесть все типы символов, т.е. буквы в обоих регистрах, цифры, знак подчёркивания).

Таблица 2.7.а — Значения входных данных для тест-кейсов (реакция на длину имени пользователя)

	<b>Позитивные тест-кейсы</b>		<b>Негативные тест-кейсы</b>		
<b>Значение</b>	AAA	123_zzzzzzzzzzzzzzzz	AA	Пустая строка	1234_zzzzzzzzzzzzzzzz
<b>Пояснение</b>	Строка минимальной допустимой длины	Строка максимальной допустимой длины	Строка недопустимой длины по нижней границе	Строка недопустимой длины, учтена для надёжности	Строка недопустимой длины по верхней границе

Осталось решить вопрос с недопустимыми символами. К сожалению, столь же наглядно, как с длиной, здесь не получится. Даже если подойти строго научно, т.е. выбрать кодировку и по её кодовой таблице определить диапазоны кодов символов (на рисунке 2.7.с приведён пример такого разделения для ASCII-таблицы), у нас нет никакой гарантии, что символы с кодами из каждого диапазона трактуются единообразно.

Здесь мы видим ярчайший пример случая, в котором тестирование по методу белого ящика сильно облегчило бы нам жизнь. Если бы мы видели, как в коде приложения реализована проверка на допустимые и недопустимые символы, мы могли бы подобрать очень показательные значения входных данных.

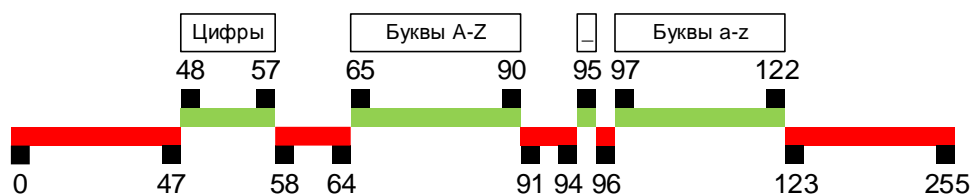


Рисунок 2.7.с — Неудачный способ поиска классов эквивалентности для наборов допустимых и недопустимых символов (коды символов приведены по ASCII-таблице)

Раз оказалось, что по кодам символов подбирать классы эквивалентности в нашем случае нерационально, посмотрим на ситуацию по-другому (и намного проще). Поделим символы на недопустимые и допустимые, а последние, в свою очередь, — на группы (см. рисунок 2.7.d).

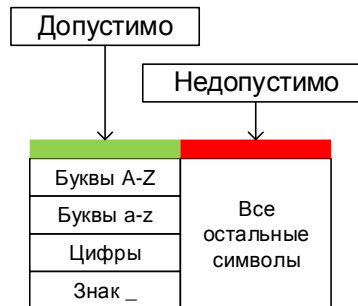


Рисунок 2.7.d — Классы эквивалентных допустимых и недопустимых символов

Интересующие нас комбинации допустимых символов (с представителями всех групп) мы уже учли при проверке реакции приложения на имена пользователя допустимых и недопустимых длин, потому остаётся учесть только вариант с допустимой длиной строки, но недопустимыми символами (которые можно выбирать случайным образом из соответствующего набора). В таблицу 2.7.a добавим одну колонку и получим таблицу 2.7.b.

Таблица 2.7.b — Значения всех входных данных для тест-кейсов

Значение	Позитивные тест-кейсы		Негативные тест-кейсы			
	AAA	123_ zzzzzzzzzzzzzzzzzzz	AA	Пустая строка	1234_ zzzzzzzzzzzzzzzzzzz	#\$%
Пояснение	Строка минимальной допустимой длины	Строка максимальной допустимой длины	Строка недопустимой длины по нижней границе	Строка недопустимой длины, учтена для надёжности	Строка недопустимой длины по верхней границе	Строка допустимой длины, недопустимые символы



Конечно, в случае критически важных приложений (например, системы управления ядерным реактором) мы бы проверили с помощью средств автоматизации реакцию приложения на каждый недопустимый символ. Но предположив, что перед нами некое тривиальное приложение, мы можем считать, что одной проверки на недопустимые символы будет достаточно.

Теперь мы возвращаемся к «Конвертеру файлов»<sup>(55)</sup> и ищем ответ на вопрос<sup>(230)</sup> о том, не упустили ли мы какие-то важные проверки в главе «Позитивные и негативные тест-кейсы»<sup>(229)</sup>.

Начнём с того, что выделим группы свойств SOURCE\_DIR, от которых зависит работа приложения (такие группы называются «измерениями»):

- Существование каталога (изначальное и во время работы приложения).
- Длина имени.
- Наборы символов в имени.
- Комбинации символов в имени.
- Расположение каталога (локальный или сетевой).
- Права доступа к каталогу (изначальные и во время работы приложения).
- Зарезервированные имена.

- Поведение, зависящее от операционной системы.
- Поведение, зависящее от работы сети.



**Задание 2.7.b:** какие ещё группы свойств вы бы добавили в этот список и как бы вы выделили подгруппы у уже имеющих в списке свойств?

Очевидно, что отмеченные группы свойств оказывают взаимное влияние. Графически его можно отобразить в виде концепт-карты<sup>351</sup> (рисунок 2.7.е).

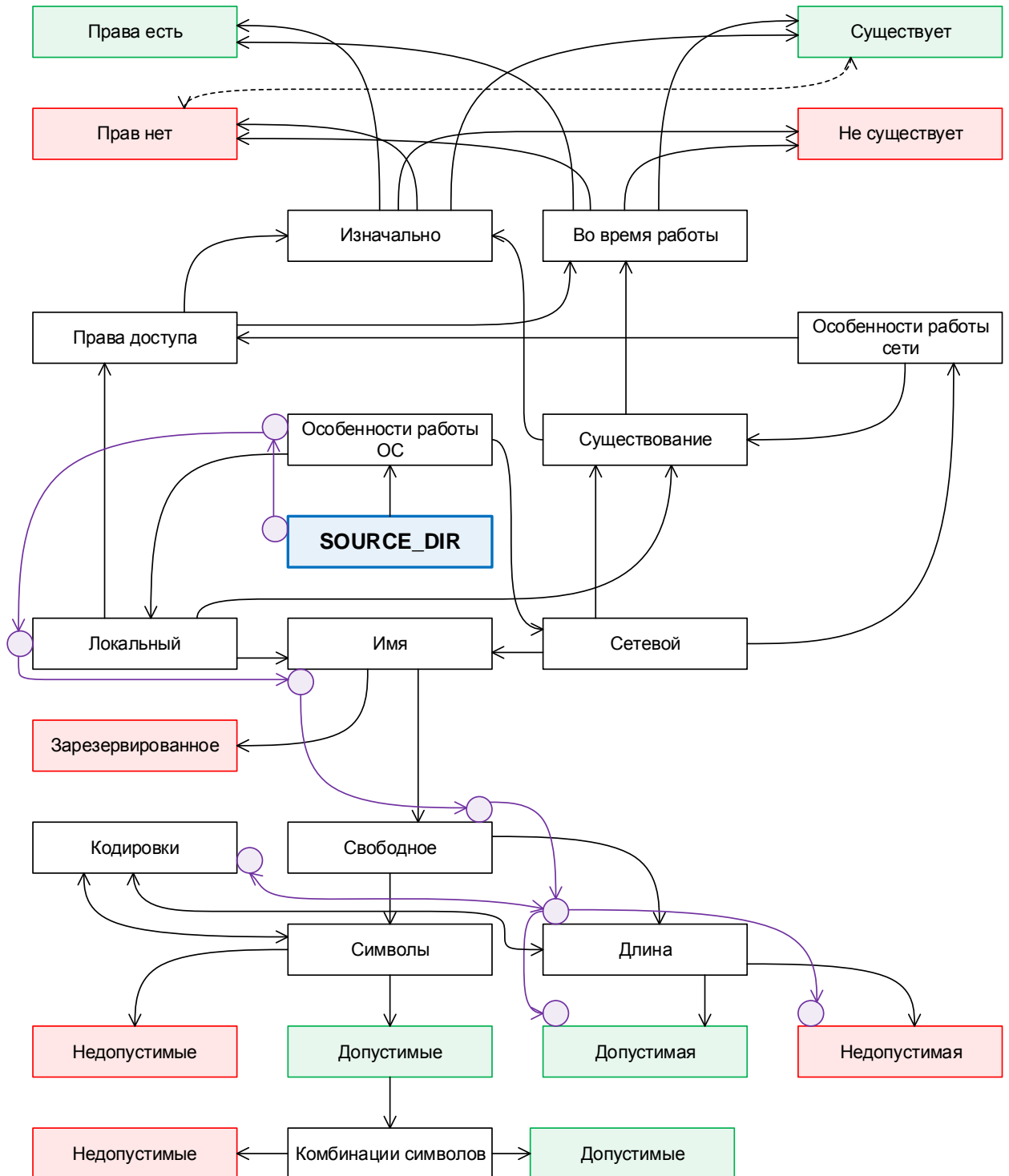


Рисунок 2.7.е — Концепт-карта взаимовлияния групп свойств каталога

<sup>351</sup> «Concept map», Wikipedia [[http://en.wikipedia.org/wiki/Concept\\_map](http://en.wikipedia.org/wiki/Concept_map)]

Чтобы иметь возможность применить стандартную технику классов эквивалентности и граничных условий, нам нужно по рисунку 2.7.е дойти от центрального элемента («SOURCE\_DIR») до любого конечного, однозначно относящегося к позитивному или негативному тестированию.

Один из таких путей на рисунке 2.7.е отмечен кружками. Словесно его можно выразить так: SOURCE\_DIR → Windows → Локальный каталог → Имя → Свободное → Длина → В кодировке UTF16 → Допустимая или недопустимая.

Максимальная длина пути для Windows в общем случае равна 256 символам: [диск][:][\][путь][null] = 1 + 2 + 256 + 1 = 260. Минимальная длина равна 1 символу (точка обозначает «текущий каталог»). Кажется бы, всё очевидно и может быть представлено рисунком 2.7.f.

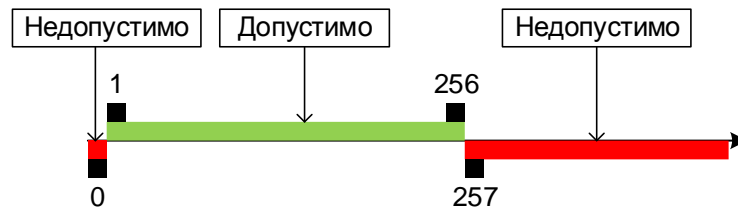


Рисунок 2.7.f — Классы эквивалентности и граничные условия для длины пути

Но если почитать внимательно спецификацию<sup>352</sup>, выясняется, что «физически» путь может быть длиной до 32'767 символов, а ограничение в 260 символов распространяется лишь на т.н. «полное имя». Потому возможна, например, ситуация, когда в каталог с полным именем длиной 200 символов помещается файл с именем длиной 200 символов, и длина полного имени файла получается равной 400 символам (что очевидно больше 260).

Так мы подошли к ситуации, в которой для проведения тестирования нужно либо знать внутреннюю реализацию поведения приложения, либо вносить правки в требования, вводя искусственные ограничения (например, длина имени SOURCE\_DIR не может быть более 100 символов, а длина имени любого файла в SOURCE\_DIR не может быть более 160 символов, что в сумме может дать максимальную длину в 260 символов).

Ввод искусственных ограничений — плохая идея, потому с точки зрения качества мы вполне вправе считать представленное на рисунке 2.7.f разбиение корректным, а сбои в работе приложения (если таковые будут), вызванные описанной выше ситуацией вида «200 символов + 200 символов», — дефектом.

Таблица 2.7.с — Значения всех входных данных для тест-кейсов по проверке выбранного на рисунке 2.7.е пути

	Позитивные тест-кейсы		Негативные тест-кейсы	
Значение	. (точка)	C:\256символов	Пустая строка	C:\257символов
Пояснение	Имя с минимальной допустимой длиной	Имя с максимальной допустимой длиной	Имя с недопустимой длиной, учтено для надёжности	Имя с недопустимой длиной

Итак, с одним путём на рисунке 2.7.е мы разобрались. Но их там значительно больше, и потому в следующей главе мы рассмотрим, как быть в ситуации, когда приходится учитывать влияние на работу приложения большого количества параметров.

<sup>352</sup> «Naming Files, Paths, and Namespaces», MSDN [<https://msdn.microsoft.com/en-us/library/aa365247.aspx#maxpath>]



### 2.7.3. Доменное тестирование и комбинации параметров

Уточним данное ранее<sup>[90]</sup> определение:

!!!	<b>Доменное тестирование</b> (domain testing, domain analysis <sup>353</sup> ) — техника создания эффективных и результативных тест-кейсов в случае, когда несколько переменных могут или должны быть протестированы одновременно.
-----	--

В качестве инструментов доменного тестирования активно используются техники определения классов эквивалентности и граничных условий, которые были рассмотрены в соответствующей<sup>[232]</sup> главе. Потому мы сразу перейдём к практическому примеру.

На рисунке 2.7.е кружками отмечен путь, один из вариантов которого мы рассмотрели в предыдущей главе, но вариантов может быть много:

- Семейство ОС
  - Windows
  - Linux
- Расположение каталога
  - Локальный
  - Сетевой
- Доступность имени
  - Зарезервированное
  - Свободное
- Длина
  - Допустимая
  - Недопустимая

Чтобы не усложнять пример, остановимся на этом наборе. Графически комбинации вариантов можно представить в виде иерархии (см. рисунок 2.7.g). Исключив совсем нетипичную для нашего приложения экзотику (всё же мы не разрабатываем сетевую утилиту), вычеркнем из списка случаи зарезервированных сетевых имён (отмечены на рисунке 2.7.g серым).

Легко заметить, что при всей своей наглядности графическое представление не всегда удобно в обработке (к тому же мы пока ограничились только общими идеями, не отметив конкретные классы эквивалентности и интересующие нас значения граничных условий).

Альтернативным подходом является представление комбинаций в виде таблицы, которое можно получать последовательно за несколько шагов.

Сначала учтём комбинации значений первых двух параметров — семейства ОС и расположения каталога. Получается таблица 2.7.d.

Таблица 2.7.d — Комбинации значений первых двух параметров

	Windows	Linux
Локальный путь	+	+
Сетевой путь	+	+

На пересечении строк и столбцов можно отмечать необходимость выполнения проверки (в нашем случае таковая есть, потому там стоит «+») или её отсутствие, приоритет проверки, отдельные значения параметров, ссылки и т.д.

<sup>353</sup> **Domain analysis** is a technique that can be used to identify efficient and effective test cases when multiple variables can or should be tested together. [Lee Copeland, «A practitioner's guide to software test design»]

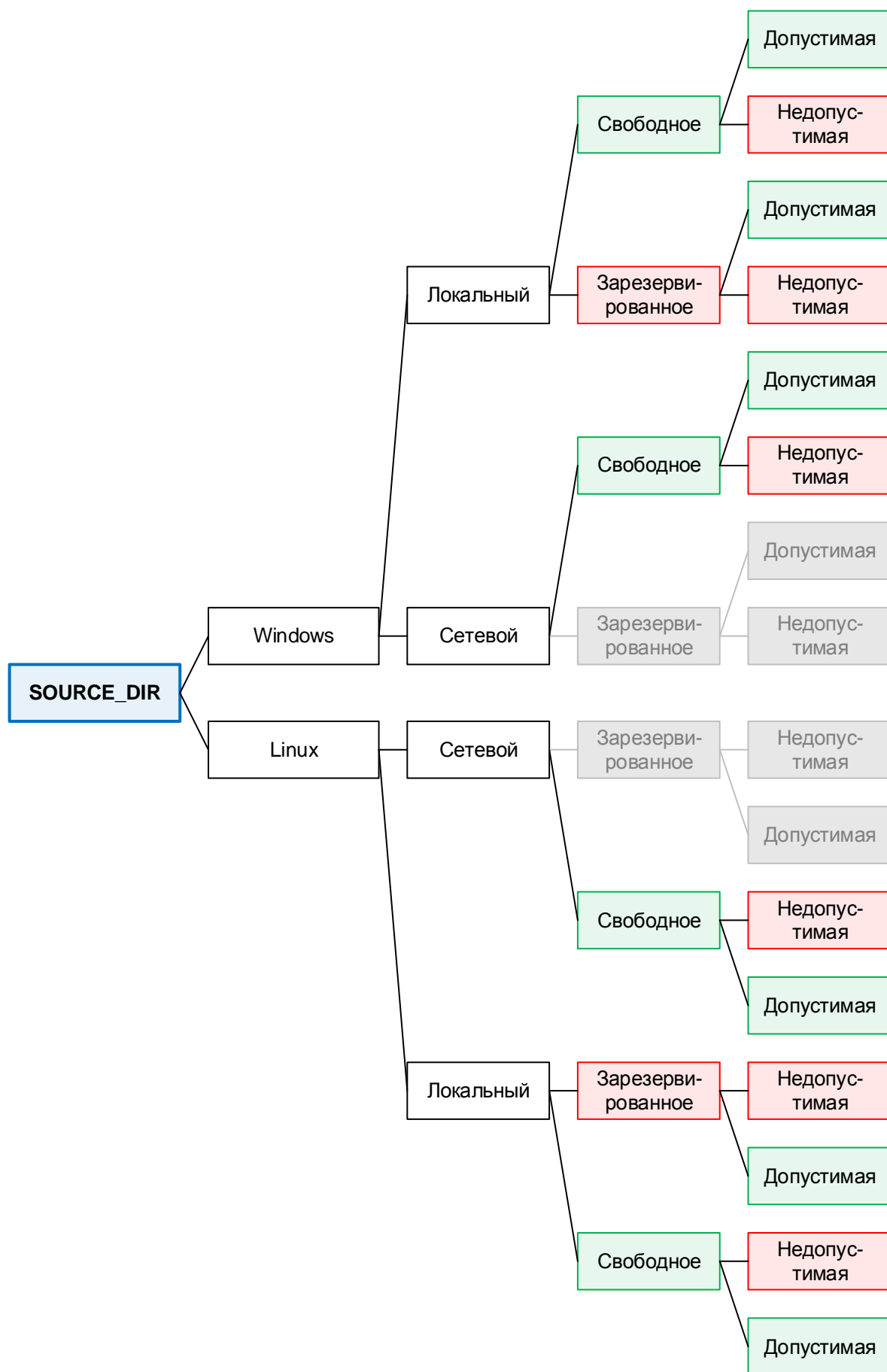


Рисунок 2.7.g — Графическое представление комбинаций параметров

Добавим третий параметр (признак зарезервированного имени) и получим таблицу 2.7.e.

Таблица 2.7.e — Комбинации значений трёх параметров

		Windows	Linux
Зарезервированное имя	Локальный путь	+	+
	Сетевой путь	-	-
Свободное имя	Локальный путь	+	+
	Сетевой путь	+	+

Добавим четвёртый параметр (признак допустимости длины) и получим таблицу 2.7.f. Чтобы таблица равномерно увеличивалась по высоте и ширине, удобно добавлять каждый последующий параметр попеременно — то как столбец, то как строку. Третий параметр мы добавили как столбец, четвёртый добавим как строку.

Таблица 2.7.f — Комбинации значений четырёх параметров


		Недопустимая длина		Допустимая длина	
		Windows	Linux	Windows	Linux
Зарезервированное имя	Локальный путь	-	-	+	+
	Сетевой путь	-	-	-	-
Свободное имя	Локальный путь	+	+	+	+
	Сетевой путь	+	+	+	+

Такое представление по сравнению с графическим оказывается более компактным и позволяет очень легко увидеть комбинации значений параметров, которые необходимо подвергнуть тестированию. Вместо знаков «+» в ячейки можно поставить ссылки на другие таблицы (хотя иногда все данные совмещают в одной таблице), в которых будут представлены классы эквивалентности и граничные условия для каждого выбранного случая.

Как несложно догадаться, при наличии большого количества параметров, каждый из которых может принимать много значений, таблица наподобие 2.7.f будет состоять из сотен строк и столбцов. Даже её построение займёт много времени, а выполнение всех соответствующих проверок и вовсе может оказаться невозможным в силу нехватки времени. В следующей главе мы рассмотрим ещё одну технику тестирования, призванную решить проблему чрезмерно большого количества комбинаций.


## 2.7.4. Попарное тестирование и поиск комбинаций

Уточним данное ранее<sup>[90]</sup> определение:

	<p><b>Попарное тестирование</b> (pairwise testing<sup>354</sup>) — техника тестирования, в которой вместо проверки всех возможных комбинаций значений всех параметров проверяются только комбинации значений каждой пары параметров.</p>
---	--

Выбрать и проверить пары значений — звучит вроде бы просто. Но как выбирать такие пары? Существует несколько тесно взаимосвязанных математических методов создания комбинаций всех пар:

- на основе ортогональных массивов<sup>355, 359</sup>;
- на основе латинских квадратов<sup>356</sup>;
- IPO (in parameter order) метод<sup>357</sup>;
- на основе генетических алгоритмов<sup>358</sup>;
- на основе рекурсивных алгоритмов<sup>359</sup>.

	<p>Глубоко в основе этих методов лежит серьёзная математическая теория<sup>359</sup>. В упрощённом виде на примерах суть и преимущества этого подхода показаны в книге Ли Коупленда<sup>360</sup> и статье Майкла Болтона<sup>355</sup>, а справедливая критика — в статье Джеймса Баха<sup>361</sup>.</p>
---	--

Итак, суть проблемы: если мы попытаемся проверить все сочетания всех значений всех параметров для более-менее сложного случая тестирования, мы получим количество тест-кейсов, превышающее все разумные пределы.

Если представить изображённую на рисунке 2.7.е схему в виде набора параметров и количества их значений, получается ситуация, представленная таблицей 2.7.g. Минимальное количество значений получено по принципу «расположение: локально или в сети», «существование: да или нет», «семейство ОС: Windows или Linux» и т.д. Вероятное количество значений оценено исходя из необходимости учитывать несколько классов эквивалентности. Количество значений с учётом полного перебора получено исходя из технических спецификаций операционных систем, файловых систем и т.д. Значение нижней строки получено перемножением значений в соответствующей колонке.

<sup>354</sup> The answer is not to attempt to test all the combinations for all the values for all the variables but to test **all pairs** of variables. [Lee Copeland, «A practitioner's guide to software test design»]

<sup>355</sup> «Pairwise Testing», Michael Bolton [<http://www.developsense.com/pairwiseTesting.html>]

<sup>356</sup> «An Improved Test Generation Algorithm for Pair-Wise Testing», Soumen Maity and oth. [<http://www.iiser-pune.ac.in/~soumen/115-FA-2003.pdf>]

<sup>357</sup> «A Test Generation Strategy for Pairwise Testing», Kuo-Chung Tai, Yu Lei [<http://www.cs.umd.edu/class/spring2003/cmcs838p/VandV/pairwise.pdf>]

<sup>358</sup> «Evolutionary Algorithm for Prioritized Pairwise Test Data Generation», Javier Ferrer and oth. [<http://neo.lcc.uma.es/staff/javi/files/gecco12.pdf>]

<sup>359</sup> «On the Construction of Orthogonal Arrays and Covering Arrays Using Permutation Groups», George Sherwood [<http://testcover.com/pub/background/cover.htm>]

<sup>360</sup> «A Practitioner's Guide to Software Test Design», Lee Copeland.

<sup>361</sup> «Pairwise Testing: A Best Practice That Isn't», James Bach [<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.3811&rep=rep1&type=pdf>].

Таблица 2.7.g — Список параметров, влияющих на работу приложения

Параметр	Минимальное количество значений	Вероятное количество значений	Количество значений с учётом полного перебора
Расположение	2	25	32
Существование	2	2	2
Наличие прав доступа	2	3	155
Семейство ОС	2	4	28
Зарезервированное или свободное имя	2	7	23
Кодировки	2	3	16
Длина	2	4	4096
Комбинации символов	2	4	82
<b>ИТОГО тест-кейсов</b>	<b>256</b>	<b>201'600</b>	<b>34'331'384'872'960</b>

Конечно, мы не будем перебирать все возможные значения (для того нам и нужны классы эквивалентности), но даже 256 тест-кейсов для проверки всего лишь одного параметра командной строки — это много. И куда вероятнее, что придётся выполнять около 200 тысяч тест-кейсов. Если делать это вручную и выполнять по одному тесту в пять секунд круглосуточно, понадобится около 11 суток.

Но мы можем применить технику попарного тестирования для генерации оптимального набора тест-кейсов, учитывающего сочетание пар каждого значения каждого параметра. Опишем сами значения. Обратите внимание, что уже на этой стадии мы провели оптимизацию, собрав в один набор информацию о расположении, длине, значении, комбинации символов и признаке зарезервированного имени. Это сделано потому, что сочетания вида «длина 0, зарезервированное имя com1» не имеют смысла. Также мы усилили часть проверок, добавив русскоязычные названия каталогов.

Таблица 2.7.h — Список параметров и их значений

Параметр	Значения
Расположение / длина / значение / комбинация символов / зарезервированное или свободное	<ol style="list-style-type: none"> <li>1. X:\</li> <li>2. X:\dir</li> <li>3. "X:\пробелы и русский"</li> <li>4. .\dir</li> <li>5. ..\dir</li> <li>6. \\host\dir</li> <li>7. [256 символов только для Windows] + Пункты 2-6 с "\" в конце пути.</li> <li>8. /</li> <li>9. /dir</li> <li>10. "/пробелы и русский"</li> <li>11. host:/dir</li> <li>12. smb://host/dir</li> <li>13. ./dir</li> <li>14. ../dir</li> <li>15. [4096 символов только для Linux] + Пункты 9-14 с "/" в конце пути.</li> <li>Недопустимое имя.</li> <li>16. [0 символов]</li> <li>17. [4097 символов только для Linux]</li> <li>18. [257 символов только для Windows]</li> <li>19. "</li> <li>20. //</li> <li>21. \\\</li> </ol>

	22. ... 23. com1-com9 24. lpt1-lpt9 25. con 26. nul 27. prn
Существование	1. Да 2. Нет
Наличие прав доступа	1. К каталогу и его содержимому 2. Только к каталогу 3. Ни к каталогу, ни к его содержимому
Семейство ОС	1. Windows 32 bit 2. Windows 64 bit 3. Linux 32 bit 4. Linux 64 bit
Кодировки	1. UTF8 2. UTF16 3. OEM

Количество потенциальных тест-кейсов уменьшилось до 2736 ( $38 \cdot 2 \cdot 3 \cdot 4 \cdot 3$ ), что уже много меньше 200 тысяч, но всё равно нерационально.

Теперь воспользуемся любым из представленных в огромном количестве инструментальных средств<sup>362</sup> (например, PICT) и сгенерируем набор комбинаций на основе попарного сочетания всех значений всех параметров. Пример первых десяти строк результата представлен в таблице 2.7.i. Всего получилось 152 комбинации, т.е. в 1326 раз меньше ( $201'600 / 152$ ) исходной оценки или в 18 раз меньше ( $2736 / 152$ ) оптимизированного варианта.

Таблица 2.7.i — Наборы значений, полученные методом попарных комбинаций

№	Расположение / длина / значение / комбинация символов / зарезервированное или свободное	Существование	Наличие прав доступа	Семейство ОС	Кодировки
1	X:\	Да	К каталогу и его содержимому	Windows 64 bit	UTF8
2	smb://host/dir/	Нет	Ни к каталогу, ни к его содержимому	Windows 64 bit	UTF16
3	/	Нет	Только к каталогу	Windows 32 bit	OEM
4	[0 символов]	Да	Только к каталогу	Linux 32 bit	UTF8
5	smb://host/dir	Нет	К каталогу и его содержимому	Linux 32 bit	UTF16
6	../dir	Да	Ни к каталогу, ни к его содержимому	Linux 64 bit	OEM
7	[257 символов только для Windows]	Да	Только к каталогу	Windows 64 bit	OEM
8	[4096 символов только для Linux]	Нет	Ни к каталогу, ни к его содержимому	Windows 32 bit	UTF8
9	[256 символов только для Windows]	Нет	Ни к каталогу, ни к его содержимому	Linux 32 bit	OEM
10	/dir/	Да	Только к каталогу	Windows 32 bit	UTF16

Если исследовать набор полученных комбинаций, можно исключить из них те, которые не имеют смысла (например, существование каталога с именем нулевой длины или проверку под Windows характерных только для Linux случаев — см.

<sup>362</sup> «Pairwise Testing, Available Tools» [<http://www.pairwise.org/tools.asp>]

строки 4 и 8). Завершив такую операцию, мы получаем 124 комбинации. По соображениям экономии места эта таблица не будет приведена, но в приложении «Пример данных для попарного тестирования»<sup>(288)</sup> представлен конечный итог оптимизации (из таблицы убраны ещё некоторые комбинации, например, проверка под Linux имён, являющихся зарезервированными для Windows). Получилось 85 тест-кейсов, что даже немного меньше минимальной оценки в 256 тест-кейсов, и при этом мы учли куда больше опасных для приложения сочетаний значений параметров.



**Задание 2.7.с:** в представленной в приложении «Пример данных для попарного тестирования»<sup>(288)</sup> в колонке «Наличие прав доступа» иногда отсутствуют значения. Как вы думаете, почему? Также в этой таблице всё ещё есть «лишние» тесты, выполнение которых не имеет смысла или представляет собой крайне маловероятный вариант стечения событий. Найдите их.

Итак, на протяжении последних четырёх глав мы рассмотрели несколько техник тестирования, позволяющих определить наборы данных и идей для написания эффективных тест-кейсов. Следующая глава будет посвящена ситуации, когда времени на столь вдумчивое тестирование нет.

### 2.7.5. Исследовательское тестирование

Исследовательское<sup>(80)</sup> и свободное<sup>(80)</sup> тестирование уже было упомянуто ранее на уровне определения. Для начала ещё раз подчеркнём, что это разные виды тестирования, пусть в каждом из них степень формализации процесса значительно меньше, чем в тестировании на основе тест-кейсов<sup>(79)</sup>. Сейчас мы будем рассматривать применение именно исследовательского тестирования.

Сэм Канер определяет<sup>363</sup> исследовательское тестирование как стиль, основанный на свободе и ответственности тестировщика в непрерывной оптимизации своей работы за счёт выполняемых параллельно на протяжении всего проекта и взаимодополняющих изучения, планирования, выполнения проверок и оценки их результатов. Если сказать короче, исследовательское тестирование — это одновременное изучение, планирование и тестирование.

Кроме очевидной проблемы с тестированием на основе тест-кейсов, состоящей в высоких затратах времени, существует ещё одна — существующие техники оптимизации направлены на то, чтобы максимально исследовать приложение во всех учтённых ситуациях, которые мы можем контролировать — но невозможно учесть и проконтролировать всё. Эта идея визуально представлена на рисунке 2.7.h.

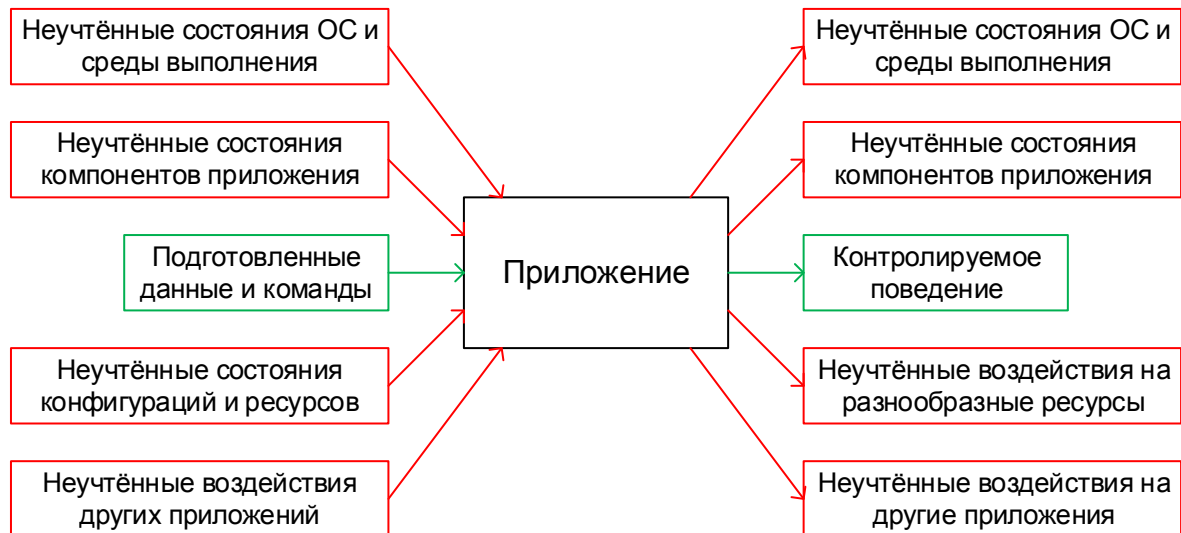


Рисунок 2.7.h — Факторы, которые могут быть пропущены тестированием на основе тест-кейсов<sup>363</sup>

Исследовательское же тестирование часто позволяет обнаружить дефекты, вызванные этими неучтёнными факторами. К тому же оно прекрасно показывает себя в следующих ситуациях:

- Отсутствие или низкое качество необходимой документации.
- Необходимость быстрой оценки качества при нехватке времени.
- Подозрение на неэффективность имеющихся тест-кейсов.
- Необходимость проверить компоненты, разработанные «третьими сторонами».
- Верификация устранения дефекта (для проверки, что он не проявляется при незначительном отступлении от шагов воспроизведения).

<sup>363</sup> «A Tutorial in Exploratory Testing», Cem Kaner [<http://www.kaner.com/pdfs/QAExploring.pdf>]



В своей работе<sup>363</sup> Сэм Канер подробно показывает способы проведения исследовательского тестирования с использованием базовых методов, моделей, примеров, частичных изменений сценариев, вмешательства в работу приложения, проверки обработки ошибок, командного тестирования, сравнения продукта с требованиями, дополнительного исследования проблемных областей и т.д.

Вернёмся к нашему «Конвертеру файлов»<sup>(55)</sup>. Представим следующую ситуацию: разработчики очень уж быстро выпустили первый билд, тест-кейсов (и всех тех наработок, что были рассмотрены ранее в этой книге) у нас пока нет, а проверить билд нужно. Допустим, в уведомлении о выходе билда сказано: «Реализованы и готовы к тестированию требования: [СХ-1](#), [СХ-2](#), [СХ-3](#), [ПТ-1.1](#), [ПТ-1.2](#), [ПТ-2.1](#), [ПТ-3.1](#), [ПТ-3.2](#), [БП-1.1](#), [БП-1.2](#), [ДС-1.1](#), [ДС-2.1](#), [ДС-2.2](#), [ДС-2.3](#), [ДС-2.4](#), [ДС-3.1](#), [ДС-3.2](#) (текст сообщений приведён к информативному виду), [ДС-4.1](#), [ДС-4.2](#), [ДС-4.3](#)».

Ранее мы отметили, что исследовательское тестирование — это тесно взаимосвязанные изучение, планирование и тестирование. Применим эту идею.

## Изучение

Представим полученную от разработчиков информацию в виде таблицы 2.7.j и проанализируем соответствующие требования, чтобы понять, что нам нужно будет сделать.

Таблица 2.7.j — Подготовка к исследовательскому тестированию

Требование	Что и как будем делать
<a href="#">СХ-1</a>	Не требует отдельной проверки, т.к. вся работа с приложением будет выполняться в консоли.
<a href="#">СХ-2</a>	Не требует отдельной проверки, видно по коду.
<a href="#">СХ-3</a>	Провести тестирование под Windows и Linux.
<a href="#">ПТ-1.1</a>	Стандартная проверка реакции консольного приложения на различные варианты указания параметров. Учесть, что обязательными являются первые два параметра из трёх (третий принимает значение по умолчанию, если не задан). См. «Идеи», пункт 1.
<a href="#">ДС-2.1</a>	
<a href="#">ДС-2.2</a>	
<a href="#">ДС-2.3</a>	
<a href="#">ДС-2.4</a>	См. «Идеи», пункт 2.
<a href="#">ПТ-1.2</a>	
<a href="#">ПТ-2.1</a>	
<a href="#">ПТ-3.1</a>	
<a href="#">ПТ-3.2</a>	На текущий момент можно только проверить факт ведения лога и формат записей, т.к. основная функциональность ещё не реализована. См. «Идеи», пункт 4.
<a href="#">ДС-4.1</a>	
<a href="#">ДС-4.2</a>	
<a href="#">ДС-4.3</a>	
<a href="#">БП-1.1</a>	См. «Идеи», пункт 3.
<a href="#">БП-1.2</a>	
<a href="#">ДС-1.1</a>	Тестирование проводить на PHP 5.5.
<a href="#">ДС-3.1</a>	Проверять выводимые сообщения в процессе выполнения пунктов 1-2 (см. «Идеи».
<a href="#">ДС-3.2</a>	

## Планирование

Частично планированием можно считать колонку «Что и как будем делать» таблицы 2.7.j, но для большей ясности представим эту информацию в виде обобщённого списка, который для простоты назовём «идеи» (да, это — вполне классический чек-лист).

Идеи:

1. Проверить сообщения в ситуациях запуска:
  - a. Без параметров.
  - b. С верно указанными одним, двумя, тремя параметрами.
  - c. С неверно указанными первым, вторым, третьим, одним, двумя, тремя параметрами.
2. Остановить приложение по Ctrl+C.
3. Проверить сообщения в ситуациях запуска:
  - a. Каталог-приёмник и каталог-источник в разных ветках ФС.
  - b. Каталог-приёмник внутри каталога-источника.
  - c. Каталог-приёмник, совпадающий с каталогом-источником.
4. Проверить содержимое лога.
5. Посмотреть в код классов, отвечающих за анализ параметров командной строки и ведение лога.



**Задание 2.7.d:** сравните представленный набор идей с ранее рассмотренными подходами<sup>(147), (229), (232), (237), (240)</sup> — какой вариант вам кажется более простым в разработке, а какой в выполнении и почему?

Итак, список идей есть. Фактически, это почти готовый сценарий, если пункт 2 (про остановку приложения) повторять в конце проверок из пунктов 1 и 3).

## Тестирование

Можно приступить к тестированию, но стоит отметить, что для его проведения нужно привлекать специалиста, имеющего богатый опыт работы с консольными приложениями, иначе тестирование будет проведено крайне формально и окажется неэффективным.

Что делать с обнаруженными дефектами? Для начала — фиксировать в таком же формате, т.е. как список идей: переключение между прохождением некоего сценария и написанием отчёта о дефекте сильно отвлекает. Если вы опасаетесь что-то забыть, включите запись происходящего на экране (отличный трюк — записывать весь экран так, чтобы были видны часы, а в списках идей отмечать время, когда вы обнаружили дефект, чтобы потом в записи его было проще найти).

Список «идей дефектов» можно для удобства оформлять в виде таблицы (см. таблицу 2.7.k).

Таблица 2.7.k — Список «идей дефектов»

№	Что делали	Что получили	Что ожидали / Что не так
0	а) Во всех случаях сообщения приложения вполне корректны с точки зрения происхождения и информативны, но противоречат требованиям (обсудить с заказчиком изменения в требованиях). б) Лог ведётся, формат даты-времени верный, но нужно уточнить, что в требованиях имеется в виду под «имя_операции параметры_операции результат_операции», т.к. для разных операций единый формат не очень удобен — нужно ли приводить всё к одному формату или нет?		
1	php converter.php	Error: Too few command line parameters. USAGE: php converter.php SOURCE_DIR DESTINATION_DIR [LOG_FILE_NAME] Please note that DESTINATION_DIR may NOT be inside SOURCE_DIR.	Сообщение совершенно не соответствует требованиям.
2	php converter.php zzz:/ c:/	Error: SOURCE_DIR name [zzz:] is not a valid directory.	Странно, что от «zzz:/» осталось только «zzz:».

3	php converter.php "c:/non/existing/directory/" c:/	Error: SOURCE_DIR name [c:/non/existing/directory] is not a valid directory.	Слешы заменены на бэк-слешы, конечный бэк-слеш удалён: так и надо? Глянуть в коде, пока не ясно, дефект это или так и задумано.
4	php converter.php c:/ d:/	2015.06.12 13:37:56 Started with parameters: SOURCE_DIR=[C:], DESTINATION_DIR=[D:], LOG_FILE_NAME=[.\converter.log]	Буквы дисков приведены к верхнему регистру, слешы заменены на бэк-слешы. Почему имя лог-файла относительное?
5	php converter.php c:/ c:/	Error: DESTINATION_DIR [C:] and SOURCE_DIR [C:] <b>mat</b> NOT be the same dir.	Опечатка в сообщении. Явно должно быть must или may.
6	php converter.php "c:/каталог с русским именем/" c:/	Error: SOURCE_DIR name [c:/ърСрыку ё Ёёёёшь шьхэхь] is not a valid directory.	Дефект: проблема с кодировками.
7	php converter.php / c:/Windows/Temp	Error: SOURCE_DIR name [] is not a valid directory.	Проверить под Linux: маловероятно, конечно, что кто-то прямо в / будет что-то рабочее хранить, но имя «/» урезано до пустой строки, что допустимо для Windows, но не для Linux.
8	Примечание: «e:» -- DVD-привод. php converter.php c:/ e:/	file_put_contents(e:f41c7142310c5910e2cfb57993b4d004620aa3b8): failed to open stream: Permission denied in \classes\CLPAnalyser.class.php at line 70 Error: DESTINATION_DIR [e] is not writeable.	Дефект: сообщение от PHP не перехвачено.
9	php converter.php /var/www /var/www/1	Error: SOURCE_DIR name [/var/www] is not a valid directory.	Дефект: в Linux обрезается начальный «/» в имени каталога, т.е. можно смело считать, что под Linux приложение неработоспособно (можно задавать только относительные пути, начинающиеся с «.» или «..»).

Выводы по итогам тестирования (которое, к слову, заняло около получаса):

- Нужно подробно обсудить с заказчиком форматы и содержание сообщений об использовании приложения и об ошибках, а также формат записей лог-файла. Разработчики предложили идеи, выглядящие куда более адекватно, чем изначально описано в требованиях, но всё равно нужно согласование.
- Под Windows серьёзных дефектов не обнаружено, приложение вполне работоспособно.
- Под Linux есть критическая проблема с исчезновением «/» в начале пути, что не позволяет указывать абсолютные пути к каталогам.
- Если обобщить вышенаписанное, то можно констатировать, что дымовой тест успешно пройден под Windows и провален под Linux.

Цикл «изучение, планирование, тестирование» можно многократно повторить, дополняя и рассматривая список обнаруженных проблем (таблица 2.7.k) как новую информацию для изучения, ведь каждая такая проблема даёт пищу для размышлений и придумывания дополнительных тест-кейсов.



**Задание 2.7.е:** опишите дефекты, представленные в таблице 2.7.k в виде полноценных отчётов о дефектах.

В данной главе в таблице 2.7.k некоторые пункты представляют собой очевидные дефекты. Но что их вызывает? Почему они возникают, как могут проявляться и на что влиять? Как описать их максимально подробно и правильно в отчётах о дефектах? Ответам на эти вопросы посвящена следующая глава, в которой мы поговорим о поиске и исследовании причин возникновения дефектов.

### 2.7.6. Поиск причин возникновения дефектов

Ранее мы отмечали<sup>(163)</sup>, что используем слово «дефект» для обозначения проблемы потому, что описание конечного симптома несёт мало пользы, а выяснение исходной причины может быть достаточно сложным. И всё же наибольший эффект приносит как раз определение и устранение первопричины, что позволяет снизить риск появления новых дефектов, обусловленных той же самой (ненайденной и неустранённой) недоработкой.



**Анализ первопричин** (root cause analysis<sup>364</sup>) — процесс исследования и классификации первопричин возникновения событий, негативно влияющих на безопасность, здоровье, окружающую среду, качество, надёжность и производственный процесс.

Как видно из определения, анализ первопричин не ограничивается разработкой программ, но нас он будет интересовать всё же в ИТ-контексте. Часто ситуация, в которой тестировщик пишет отчёт о дефекте, может быть отражена рисунком 2.7.i.

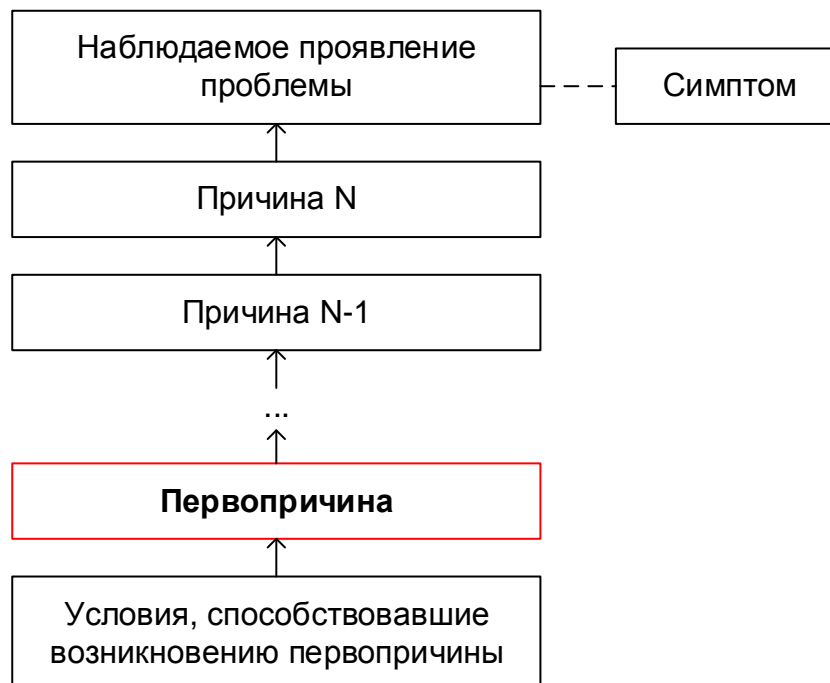


Рисунок 2.7.i — Проявление и причины дефекта

В самом худшем случае проблема вообще будет пропущена (не замечена), и отчёт о дефекте не будет написан. Чуть лучше выглядит ситуация, когда отчёт описывает исключительно внешние проявления проблемы. Приемлемым может считаться описание лежащих на поверхности причин. Но в идеале нужно стре-

<sup>364</sup> **Root cause analysis** (RCA) is a process designed for use in investigating and categorizing the root causes of events with safety, health, environmental, quality, reliability and production impacts. [James Rooney and Lee Vanden Heuvel, «Root Cause Analysis for Beginners», [https://www.env.nm.gov/aqb/Proposed\\_Regs/Part\\_7\\_Excess\\_Emissions/NMED\\_Exhibit\\_18-Root\\_Cause\\_Analysis\\_for\\_Beginners.pdf](https://www.env.nm.gov/aqb/Proposed_Regs/Part_7_Excess_Emissions/NMED_Exhibit_18-Root_Cause_Analysis_for_Beginners.pdf)]

миться добраться до двух самых нижних уровней — первопричины и условий, способствовавших её возникновению (хоть последнее часто и лежит в области управления проектом, а не тестирования как такового).

Вкратце вся эта идея выражается тремя простыми пунктами. Нам нужно понять:

- **Что** произошло.
- **Почему** это произошло (найти первопричину).
- Как **снизить вероятность повторения** такой ситуации.

Сразу же рассмотрим практический пример. В таблице 2.7.k в строке с номером 9<sup>(247)</sup> есть упоминание крайне опасного поведения приложения под Linux — из путей, переданных приложению из командной строки, удаляется начальный символ «/», что для Linux приводит к некорректности любого абсолютного пути.

Пройдём по цепочке, представленной рисунком 2.7.i, и отразим этот путь таблицей 2.7.l:

Таблица 2.7.l — Пример поиска первопричины

Уровень анализа	Наблюдаемая ситуация	Рассуждения и выводы
Наблюдаемое проявление проблемы	Тестировщик выполнил команду «php converter.php <b>/var/www</b> /var/www/1» и получил такой ответ приложения: «Error: SOURCE_DIR name [ <b>var/www</b> ] is not a valid directory.» в ситуации, когда указанный каталог существует и доступен для чтения.	Сразу же бросается в глаза, что в сообщении об ошибке имя каталога отличается от заданного — отсутствует начальный «/». Несколько контрольных проверок подтверждают догадку — во всех параметрах командной строки начальный «/» удаляется из полного пути.



На этом этапе очень часто начинающие тестировщики описывают дефект как «неверно распознаётся имя каталога», «приложение не обнаруживает доступные каталоги» и тому подобными словами. Это плохо как минимум по двум причинам: а) описание дефекта некорректно; б) программисту придётся самому проводить всё исследование.

Таблица 2.7.l [продолжение]

Уровень анализа	Наблюдаемая ситуация	Рассуждения и выводы
Причина N	Факт: во всех параметрах командной строки начальный «/» удаляется из полного пути. Проверка с относительными путями («php converter.php . .») и проверка под Windows («php converter.php c:\ d:\») показывает, что в таких ситуациях приложение работает.	Дело явно в обработке введённых имён: в некоторых случаях имя обрабатывается корректно, в некоторых — нет. Гипотеза: убираются начальные и конечные «/» (может быть, ещё и «\»).
Причина N-1	Проверки «php converter.php \\\c:\\\\d:\\\\» и «php converter.php \\\c:\\\\d:\\\\» показывают, что приложение под Windows запускается, корректно распознав правильные пути: «Started with parameters: SOURCE_DIR=[C:], DESTINATION_DIR=[D:]»	Гипотеза подтвердилась: из имён каталогов приложение убирает все «/» и «\», в любом количестве присутствующие в начале или конце имени.

В принципе, на этой стадии уже можно писать отчёт о дефекте с кратким описанием в стиле «Удаление краевых “/” и “\” из параметров запуска повреждает абсолютные пути в Linux ФС». Но что нам мешает пойти ещё дальше?

Таблица 2.7.1 [продолжение]

Уровень анализа	Наблюдаемая ситуация	Рассуждения и выводы
Причина N-2	<p>Гипотеза: где-то в коде есть первичный фильтр полученных значений путей, который обрабатывает их до начала проверки каталога на существование. Этот фильтр работает некорректно. Откроем код класса, отвечающего за анализ параметров командной строки. Очень быстро мы обнаруживаем метод, который виновен в происходящем:</p> <pre>private function getCanonicalName(\$name) {     \$name = str_replace('\\', '/', \$name);     \$arr = explode('/', \$name);     \$name = trim(implode(DIRECTORY_SEPARATOR,     \$arr), DIRECTORY_SEPARATOR);     return \$name; }</pre>	Мы нашли конкретное место в коде приложения, которое является первопричиной обнаруженного дефекта. Информацию об имени файла, номере строки и выдержку самого кода с пояснениями, что в нём неверно, можно приложить в комментарии к отчёту о дефекте. Теперь программисту намного проще устранить проблему.



**Задание 2.7.f:** представьте, что программист исправил проблему сменой удаления краевых «/» и «\» на концевые (т.е. теперь они удаляются только в конце имени, но не в начале). Хорошее ли это решение?

Обобщённый алгоритм поиска первопричин можно сформулировать следующим образом (см. рисунок 2.7.j):

- Определить проявление проблемы
  - Что именно происходит?
  - Почему это плохо?
- Собрать необходимую информацию
  - Происходит ли то же самое в других ситуациях?
  - Всегда ли оно происходит одинаковым образом?
  - От чего зависит возникновение или исчезновение проблемы?
- Выдвинуть гипотезу о причине проблемы
  - Что может являться причиной?
  - Какие действия или условия могут приводить к проявлению проблемы?
  - Какие другие проблемы могут быть причинами наблюдаемой проблемы?
- Проверить гипотезу
  - Провести дополнительное исследование.
  - Если гипотеза не подтвердилась, проработать другие гипотезы.
- Убедиться, что обнаружена именно первопричина, а не очередная причина в длинной цепи событий
  - Если обнаружена первопричина — сформировать рекомендации по её устранению.
  - Если обнаружена промежуточная причина, повторить алгоритм для неё.



Здесь мы рассмотрели очень узкое применение поиска первопричин. Но представленный алгоритм универсален: он работает и в разных предметных областях, и в управлении проектами, и в работе программистов (как часть процесса отладки).

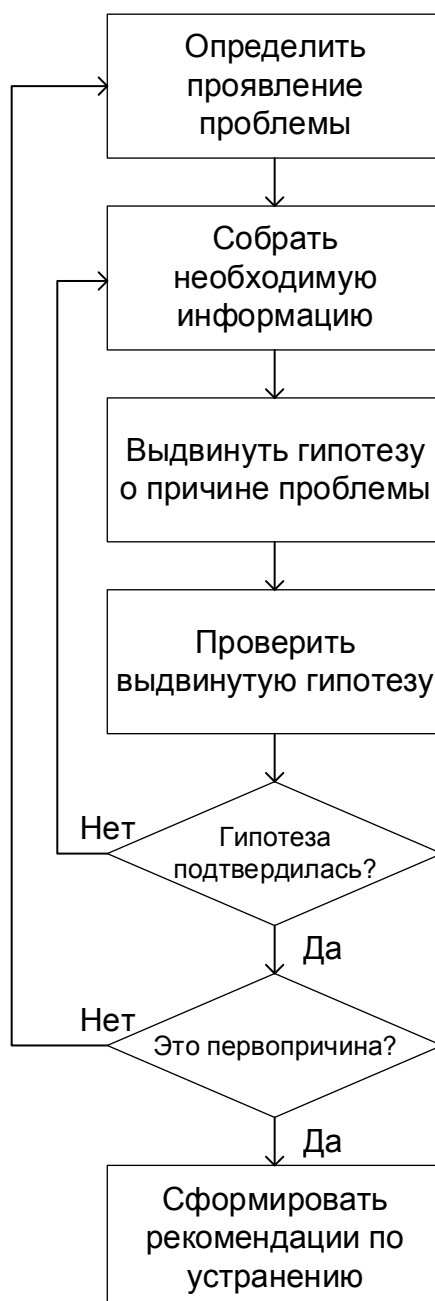


Рисунок 2.7.j — Алгоритм поиска первопричины дефекта

На этом мы завершаем основную часть данной книги, посвящённую «тестированию в принципе». Далее будет рассмотрена автоматизация тестирования как совокупность техник, повышающих эффективность работы тестировщика по многим показателям.