<epam>

Checklists

Software Testing Introduction

TRAINING

CENTER

<epam>

Read and remember!

Checklist – a set of ideas.

In testing checklists come first: this is a kind of draft for further activities. Writing and re-writing checklists is (relatively) fast and simple. When checklist is ready (and reviewed by colleagues) each item may become a source for one or several test cases.

Example (famous “Triangle Task” by Glenford J. Myers)

A console C++ program (32-bit, Win32) reads three integers from the command line. These integers represent the three lengths of the sides of a triangle. Then the program displays a message which states whether the triangle is scalene (i.e., no two sides are equal), isosceles (two sides equal) or equilateral (all sides equal).

Take a pause and write down your own ideas on how to test such program.

Checklist example

1. Valid scalene triangle.
2. Valid isosceles triangle: (a = b) or (b = c) or (a = c).
3. Valid equilateral triangle.
4. The length of one side = 0: (side a) or (side b) or (side c).
5. The length of one side <0: (side a) or (side b) or (side c).
6. The sum of two sides is equal to the third: (a + b = c) or (a + c = b) or (b + c = a).
7. The sum of two sides is less than the third: (a + b < c) or (a + c < b) or (b + c < a).
9. Non-integers (i.e. real values).
10. Not-numbers (i.e. letters, special symbols, etc.).
11. Invalid number of entered values.

General ideas

Write down your checklists!

This prevents you from forgetting any ideas and allows you to easily share and re-use a lot of checklists.

Start with simple ideas!

It is recommended to start with ideas of simple tests to check basic functionality and/or user scenarios. Don’t try to check rare extreme cases until you’re done with simple and obvious ones.

Checklists writing approaches

- By application parts, sub-parts, sub-sub-parts
- By user scenarios and actions
- By subject matter scenarios and cases
- By testing objectives and quality criteria
- By requirements
- By interface

<epam>

Checklists

Software Testing Introduction

TRAINING

CENTER

<epam>

<epam>

Basic Testing Techniques

Software Testing Introduction

TRAINING

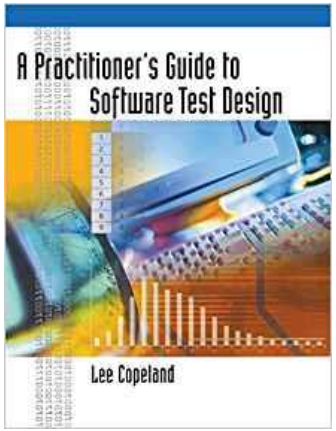
CENTER

<epam>

General overview

It is much more simple to produce good checklists when you follow special techniques. We'll overview some.

For more details see this outstanding book:



Read and remember!

An **equivalence class** consists of a set of data that is treated the same by a module or that should produce the same result.

The **boundaries** — the “edges” of each equivalence class.

So we have to find out which input data is treated the same way and use boundary values technique to select values to test.

Equivalence classes and boundary values: example

The function determines if a given string value represents correct (return **true**) or incorrect (return **false**) user name.
Username requirements:

- Length: 3-20 symbols.
- Allowed symbols: numbers, underscores, English letters (in upper and lower case).

Take a pause and write down your own ideas on how to test such function.

Equivalence classes and boundary values: example

For those who is too optimistic: don't be ☺!

Username requirements:

- Length: 3-20 symbols.
- Allowed symbols: numbers, underscores, English letters (in upper and lower case).

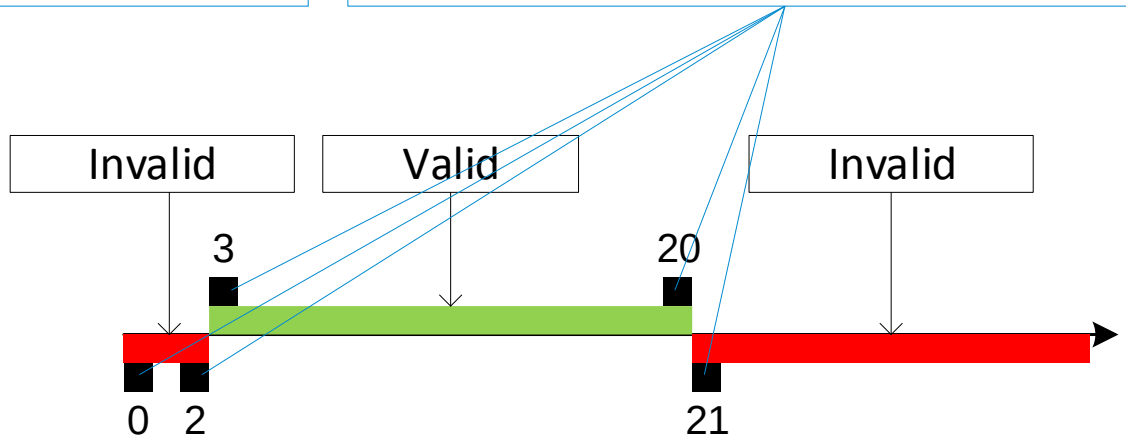
Valid names: 18-digits 63-demical number, i.e. 2.4441614509104E+32. And the infinity of invalid names.

Equivalence classes and boundary values: example

Equivalence classes by length:

- 3-20 (valid).
- 0-2 (invalid).
- 21+ (invalid).

Boundary values

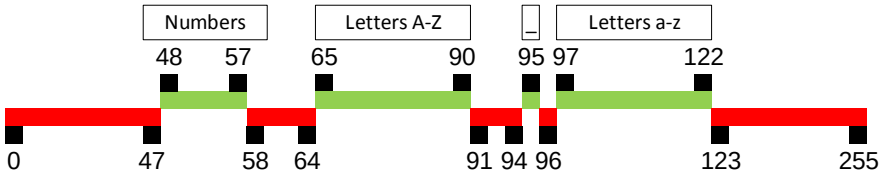


Equivalence classes and boundary values: example

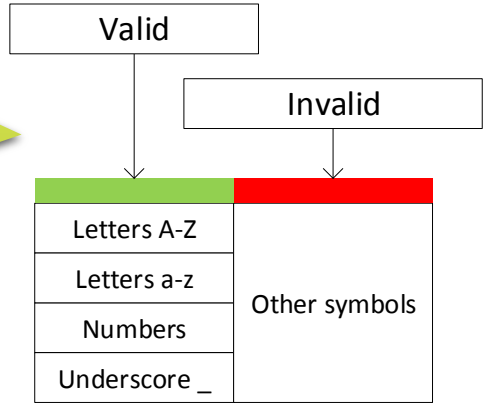
Equivalence classes by symbols:

- Letters (A-Z, a-z), numbers (0-9), underscore (_) – (valid).
- Other symbols (invalid).

Bad decision. Software doesn't work this way.



Good decision.



Equivalence classes and boundary values: example

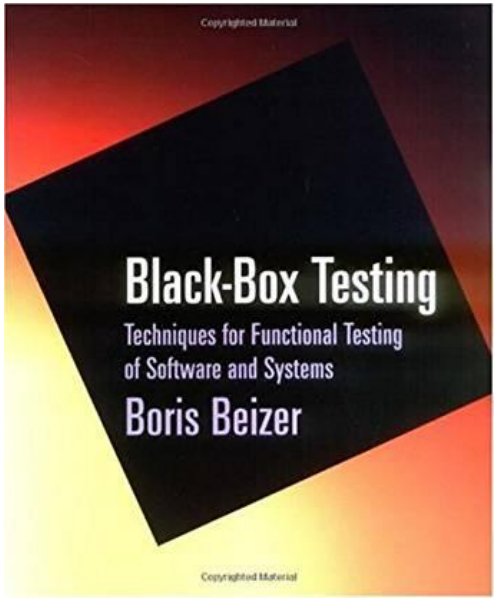
Finally we have the following values to test:

- Valid length, valid symbols: ABC, ABCDEFGHIJKLMNOPQRST, abc_12_def.
- Invalid length, valid symbols: AA, {empty string}.
- Invalid length, valid symbols: AAAAAAAAAAAAAAAAAAAAA (21 characters).
- Valid length, invalid symbols: Abcd#23456%@#&#%^#.

Seven tests instead of «2.4441614509104E+32 of positive» + «infinity of negative».

And some more ideas…

Are there more ways to pick values in such situations? Yes. See this book:



Read and remember!

Other powerful approach to generating test-ideas is thinking about software functions: what should it do (or shouldn't do) in some situation.

Example 1: e-mail field.

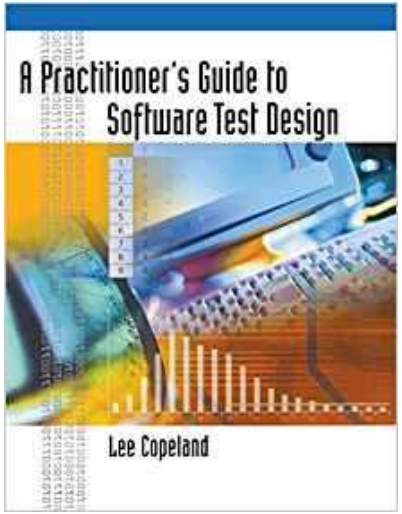
1. What is it used for?
2. What are special requirements (if any)?
3. What is the reaction to valid/invalid input?
4. What is the reaction to empty field (is it valid or not)?
5. What is the reaction to already existing e-mail?
6. Is the data filtering secure enough? Can we use SQL injection or code injection?
7. *And so on and so forth…*

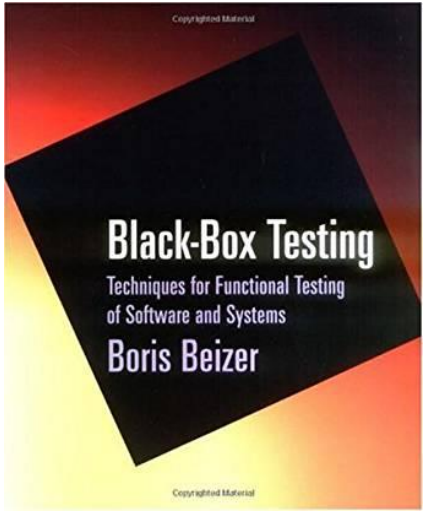
Example 2: two date fields (vacation start and finish).

1. What are special requirements (if any)?
2. What is the reaction to valid/invalid input?
3. Can both values be in the past? In the future?
4. What is the reaction to empty field (is it valid or not)?
5. Can start date and end date be the same?
6. What is the minimum/maximum vacation length?
7. Can start/end date be Saturday and/or Sunday? Or any holiday?
8. *And so on and so forth…*

Please read these books!

A lot (and lot, and lot!) of nice ideas on testing techniques you can find in these books:





<epam>

Basic Testing Techniques

Software Testing Introduction

TRAINING

CENTER

<epam>

<epam>

Test cases

Software Testing Introduction

TRAINING

CENTER

<epam>

Read and remember!

Test case – a set of preconditions, **inputs**, **actions** (where applicable), **expected results** and postconditions, developed based on test conditions. Test case always follows some **purpose**.

These four parts are crucial for a good test case.

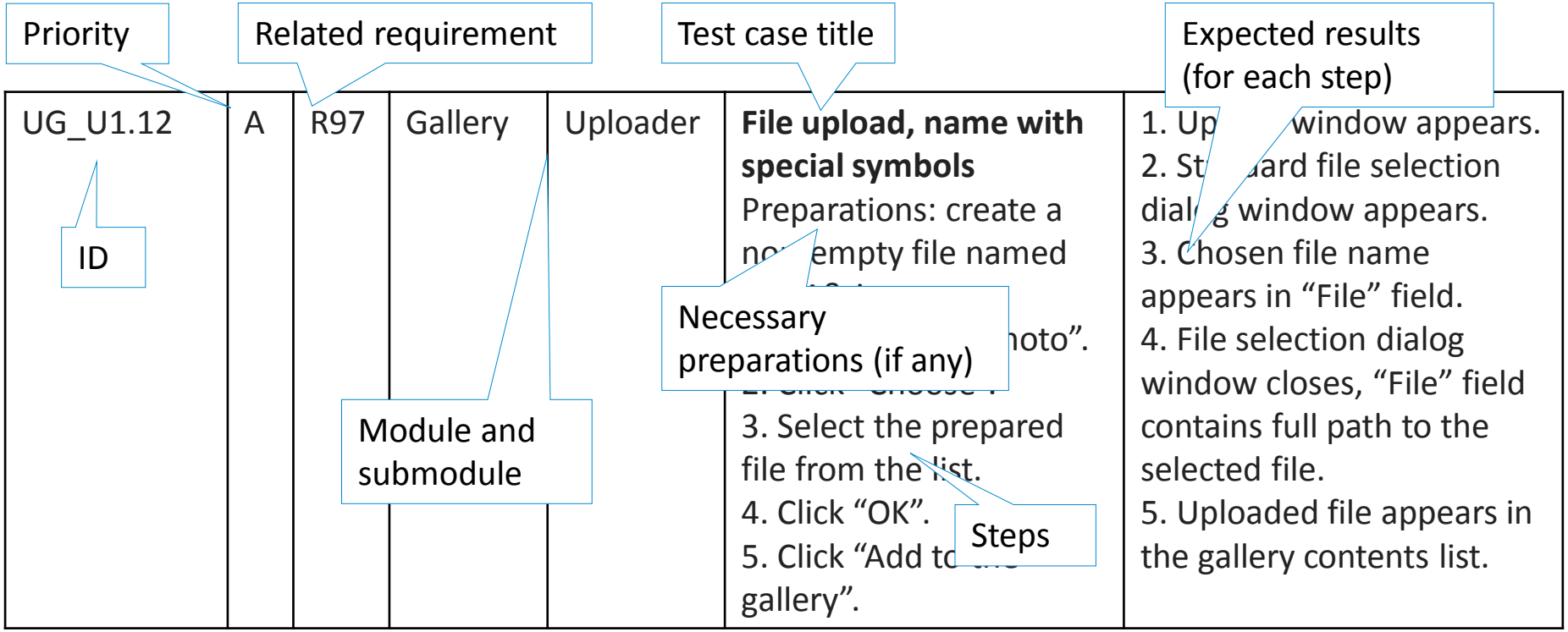
Key test case fields

Disclaimer: each and every test case management tool has much more fields to fill. There are different approaches in details, but now we are going to review only **KEY FIELDS** that remains more or less the same for past 3-4 decades.

Key test case fields: general overview

UG_U1.12	A	R97	Gallery	Uploader	**File upload, name with special symbols** Preparations: create a non-empty file named #$%^&.jpg. 1. Click “Upload photo”. 2. Click “Choose”. 3. Select the prepared file from the list. 4. Click “OK”. 5. Click “Add to the gallery”.	1. Upload window appears. 2. Standard file selection dialog window appears. 3. Chosen file name appears in “File” field. 4. File selection dialog window closes, “File” field contains full path to the selected file. 5. Uploaded file appears in the gallery contents list.

Key test case fields: general overview



UG_U1.12	A	R97	Gallery	Uploader	**File upload, name with special symbols** Preparations: create a no[illegible]empty file named [illegible]hoto”. [illegible] 3. Select the prepared file from the list. 4. Click “OK”. 5. Click “Add to [illegible] gallery”.	1. Up[illegible]window appears. 2. St[illegible]ard file selection dial[illegible]g window appears. 3. Chosen file name appears in “File” field. 4. File selection dialog window closes, “File” field contains full path to the selected file. 5. Uploaded file appears in the gallery contents list.

Key test case fields: ID

UG_U1.12 ID	A	R97	Gallery	Uploader	**File upload, name with special symbols** 1. Click “Upload photo”. 2. Click “Choose”. 3. Select the prepared file from the list. 4. Click “OK”. 5. Click “Add to the gallery”.	1. Upload window appears. 2. Standard file selection dialog window appears. 4. File selection dialog window closes, “File” field contains full path to the selected file. 5. Uploaded file appears in the gallery contents list.

- Unique.
- Meaningful (if test management tool allows).

Key test case fields: priority

Priority

UG_U1.12	A	R

- Shows how important this test case is.
- Represented by letters (A, B, C, D, E), numbers (1, 2, 3, 4, 5), words («extremely high», «high», «medium», «low», «extremely low») or any other convenient way.
- Correlated with:
  - the requirement importance;
  - potential severity of a defect that this test case may detect.

Key test case fields: related requirement

Related requirement

UG_U1.12	A	R97	Gallery	Uploader	**File upload, name with special symbols** Preparations: create a non-empty file named gallery”.	1. Upload window appears. 2. Standard file selection dialog window appears. 3. Chosen file name

- Contains the ID of the requirement this test case is intended to test.
- Facilitates traceability between requirements and test cases.

Key test case fields: module and submodule

Module and submodule

UG_U1.12	A	R97	Gallery	Uploader	**File upload, name with special symbols**	1. Upload window appears. 2. Standard file selection

- Shows the application (sub)parts covered by the test case.
- Simplifies understanding of test case purpose.
- Module and submodule are **parts** of the application, **NOT actions**! See the difference (using human as example): «respiratory system, lungs» – module and submodule, but «breathing», «snuffling», «sneezing» – are NOT; «head, brain» – module and submodule, but «nodding», «thinking» – are NOT.

Key test case fields: title

Test case title

UG_U1.12	A	R97	Gallery	Uploader	**File upload, name with special symbols**	1. Upload window appears. 2. Standard file selection
					5. Click “Add to the gallery”.	the gallery contents list.

- Shows the main idea (purpose) of the test case.
- Populates the test cases list.
- Is one of the main data sources when searching for a particular test case.
- **A test case should ALWAYS have a title!**

Key test case fields: preparations

Necessary preparations (if any)

UG_U1.12	A	R97	Gallery	Uploader	**File ad, name with special ymbols** Preparations: create a non-empty file named #$%^&.jpg. 1. Click “Upload photo”.	1. Upload window appears. 2. Standard file selection dialog window appears. 3. Chosen file name appears in “File” field. 4. File selection dialog

- Is optional.
- Describes actions and conditions to be done or met before the test case itself begins.
- Often does not interfere with the application under test (unlike “Steps” data).

Key test case fields: steps and expected results

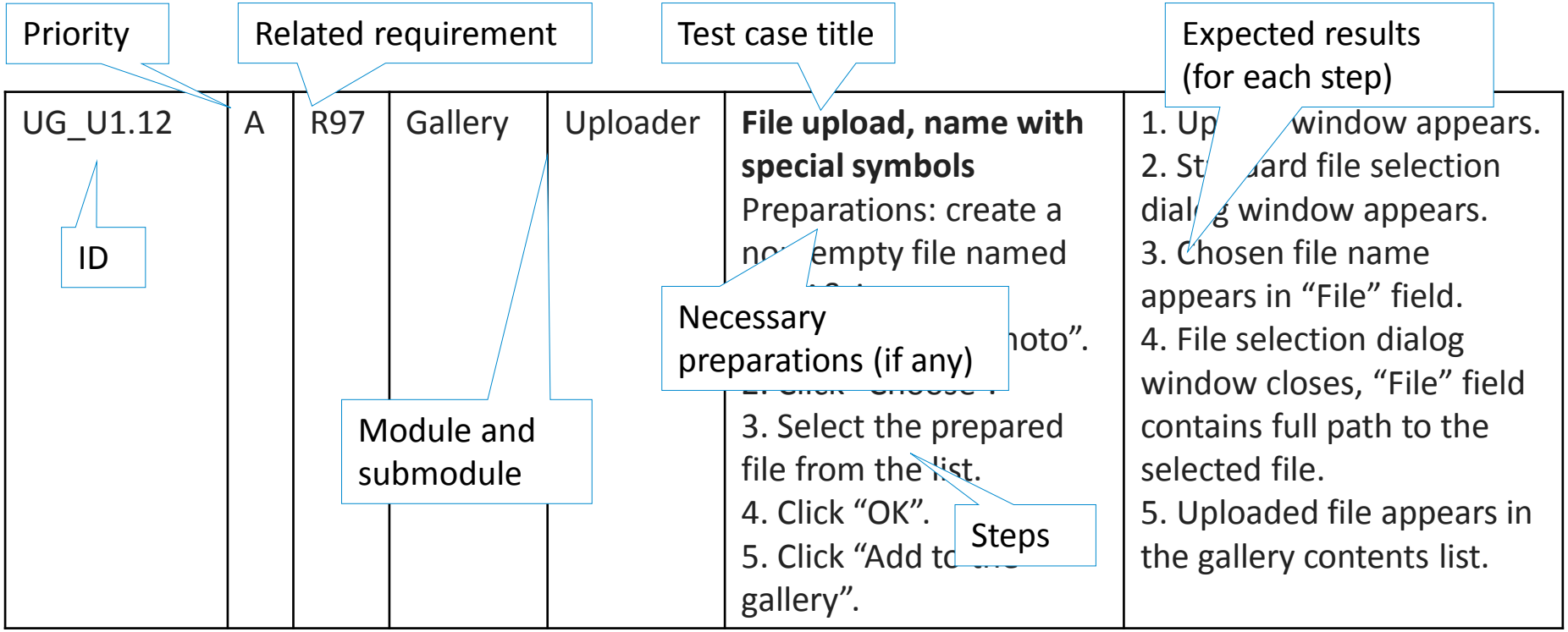
Steps

Expected results (for each step)

UG_U1.12	A	R97	Gallery	Uploader	**File up[illegible]d, name with special s[illegible]bols** Preparation: create a [illegible] 4. Click “OK”. 5. Click “Add to the gallery”.	1. Upload window appears. 2. Standard file selection dialog window appears. 5. Uploaded file appears in the gallery contents list.

- Steps describe action sequence to perform during the test case execution.
- Expected results describe the application reaction to the specific input or action.
- Both steps and expected results have the same numeric ids.

Key test case fields: general overview



UG_U1.12	A	R97	Gallery	Uploader	**File upload, name with special symbols** Preparations: create a no[illegible]empty file named [illegible]hoto”. [illegible] 3. Select the prepared file from the list. 4. Click “OK”. 5. Click “Add to [illegible] gallery”.	1. Up[illegible]window appears. 2. St[illegible]ard file selection dial[illegible]g window appears. 3. Chosen file name appears in “File” field. 4. File selection dialog window closes, “File” field contains full path to the selected file. 5. Uploaded file appears in the gallery contents list.

Useful ideas

- Use active voice and simple phrases in steps section
- Use objective description in expected results section
- Write simple, this is not a fiction novel
- Use exact names for interface elements
- Don’t explain basics

Useful ideas

Use active voice and simple phrases in steps section

Bad:
“When a user seems to try to click ‘Open’ button…”

Good:
“Click ‘Open’ button.”

Useful ideas

Use active voice and simple phrases in steps section

Use objective description in expected results section

Bad:
"The application works as it has to", "Operation successful", "OK"…

Good:
"The 'New document' label appears", "Calculation result equals to A*(B+C)", "Error message 'Bad email appears'"…

Useful ideas

Use active voice and simple phrases in steps section

Use objective description in expected results section

Write simple, this is not a fiction novel

Use exact names for interface elements

Don’t explain basics

<epam>

Test cases

Software Testing Introduction

TRAINING

CENTER

<epam>

<epam>

Good Test Cases Properties – Part 1

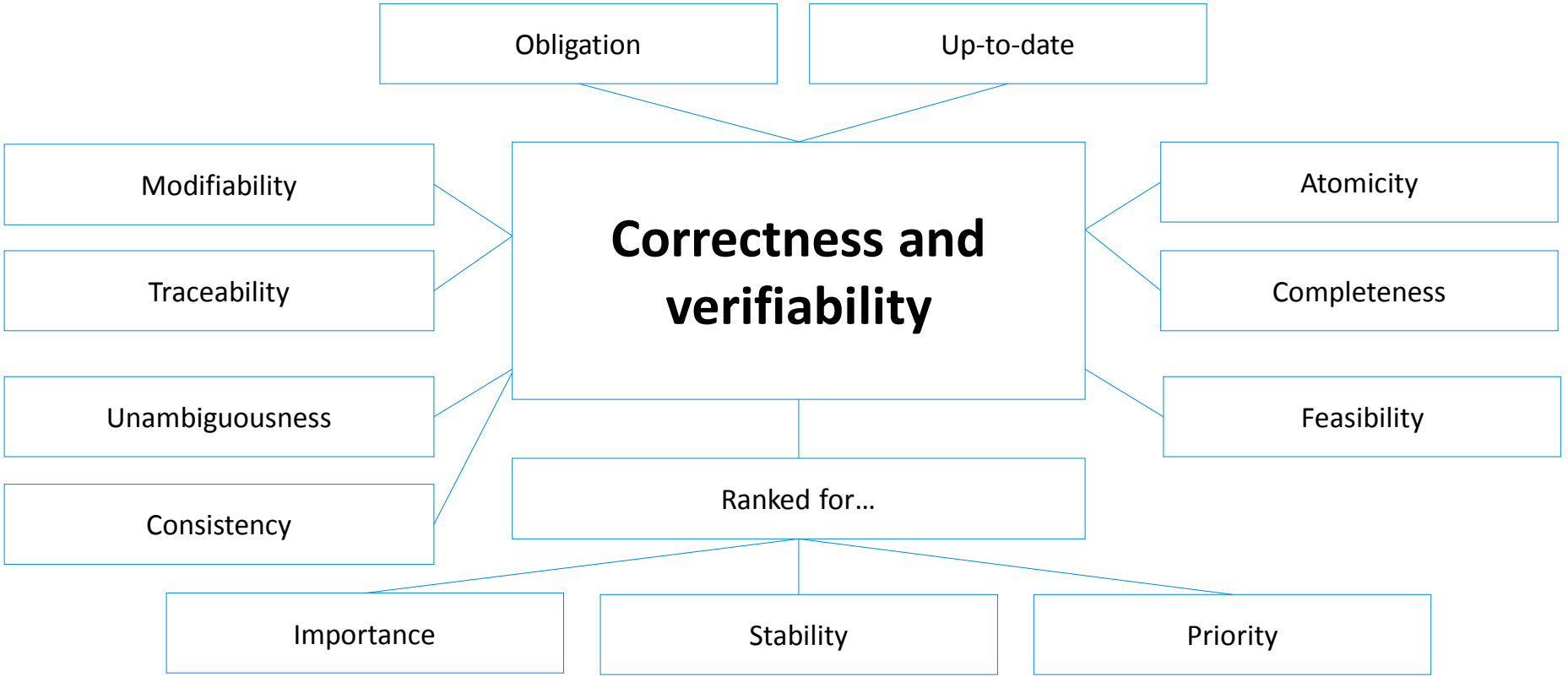
Software Testing Introduction

TRAINING

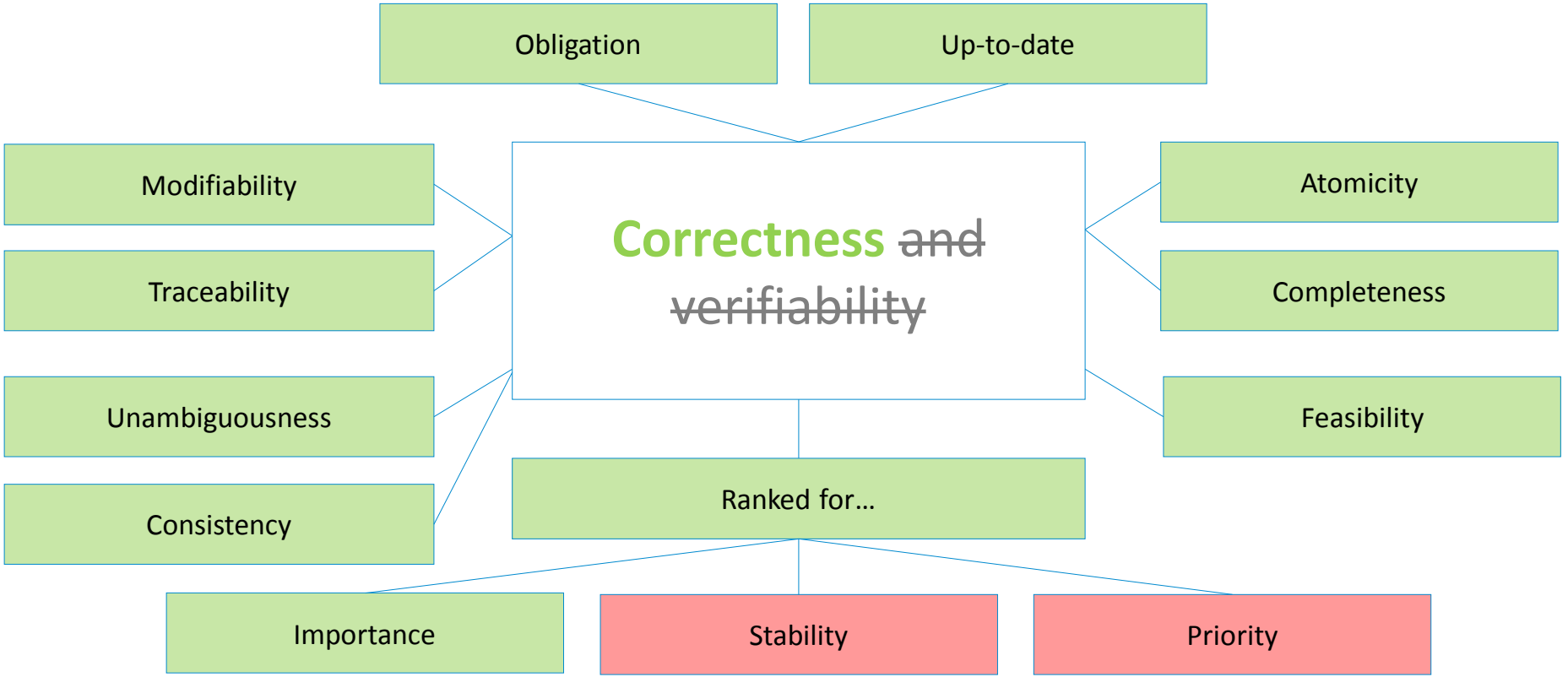
CENTER

<epam>

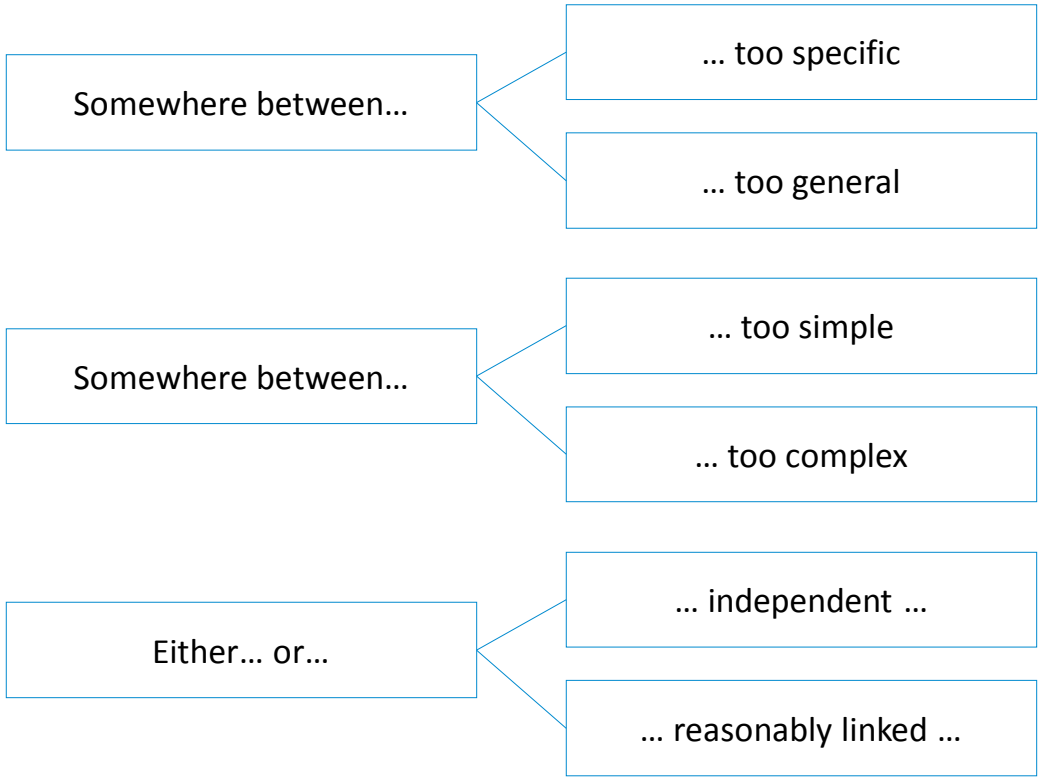
A little step aside: let's consider good requirements properties



A little step aside: let's consider good requirements properties



And there are more specific properties



Somewhere between being too specific or too general

Which test case is good and which test case is bad?

1. Enter 10 to the “A” field. 2. Enter 15 to the “B” filed. 3. Click the “Add” button.	3. The “C” field contains 25.

1. Check that the app adds two numbers properly.	1. The app adds two numbers properly.

Both test cases are **BAD**!

Somewhere between being too specific or too general

1. Enter 10 to the “A” field. 2. Enter 15 to the “B” filed. 3. Click the “Add” button.	3. The “C” field contains 25.

1. Check that the app adds two numbers properly.	4. The app adds two numbers properly.

Too specific:

- the probability of detecting an error is reduced due to re-executing exactly the same actions;
- the time of creation and support of the test case increases.

Too general:

- it’s difficult for beginners (or new people on the project) to understand such test cases;
- even huge effort to understand the initial idea may leave some gaps.

Somewhere between being too specific or too general

Summation of two numbers 1. Enter a valid integer into the “A” field. 2. Enter a valid integer into the “B” field. 3. Click the “Add” button. 4. Repeat steps 1-3 for the following values: 0, max valid values, min valid values.	1. The value appears in the field. 2. The value appears in the field. 3. The “C” field contains the sum of values from “A” and “B” fields.

Balanced:

- we are not tied to specific values (so each time we use different ones);
- we know exactly how to check the result (no guessing needed);
- we reduce the time to write and support the test case by referring its own steps;
- we have the list of values we particularly interested in.

Between being too simple or too complex

Read and remember!

A **simple test case** operates with single object (or the main object is obvious), it contains a small number of trivial actions.

A **complex test case** operates with several equal objects and/or contains many nontrivial actions.

Somewhere between being too simple or too complex

The simplicity or the complexity themselves are OK. Problems begin with extreme simplicity or extreme complexity.

Filling the “A” field 1. Enter the value into the “A” filed.	1. The value appears in the “A” field.

Repeated conversion Preparations: * Create three separate folders for input files, output files, and log file inside the root of any drive. * Prepare a set of several files of the maximum supported size and supported formats with supported encodings, as well as several files of the supported size, but in invalid format. 1. Start the application specifying the corresponding paths from the preparation for the test case in the parameters (the name of the log file is mandatory). 2. Copy several files with the valid format into the input folder. 3. Move the converted files from the output folder into the input folder. 4. Move the converted files from the output folder into the folder containing initial set of files prepared for the test case. 5. Move all the files from the initial folder with the set of files prepared for the test into the input folder. 6. Move all the converted files from the output folder into the input folder.	1. The app starts and is ready for file conversion. 2. Files are moved from the input folder to the output folder, messages about successful file conversion appear in the console and in the log file. 3. Files are moved from the input folder to the output folder, messages about successful file conversion appear in the console and in the log file. 5. Files are moved from the input folder to the output folder, messages about successful conversion of files of the valid format and messages about ignoring files of the invalid format appear in the console and in the log file. 6. Files are moved from the input folder to the output folder, messages about successful conversion of files of the valid format and messages about ignoring files of the invalid format appear in the console and in the log file.

Simplicity: pros and cons

- Easy to understand
- Easy to execute
- Easy to see the defects
- Found defects are obvious

Too simple test cases are nothing more than a step (or several steps) of more complex test cases. That's why such extremely simple test cases are useless.

Somewhere between being too simple or too complex

Complexity: pros and cons

More chances to break something

More similar to real user actions

Developers rarely test such things

Too complex test cases take a lot of time to write and maintain. And they often need huge maintenance with each application change.

It’s recommended to move from simple test cases to complex ones during the testing process.

<epam>

Good Test Cases Properties – Part 1

Software Testing Introduction

TRAINING

CENTER

<epam>

<epam>

Good Test Cases Properties – Part 2

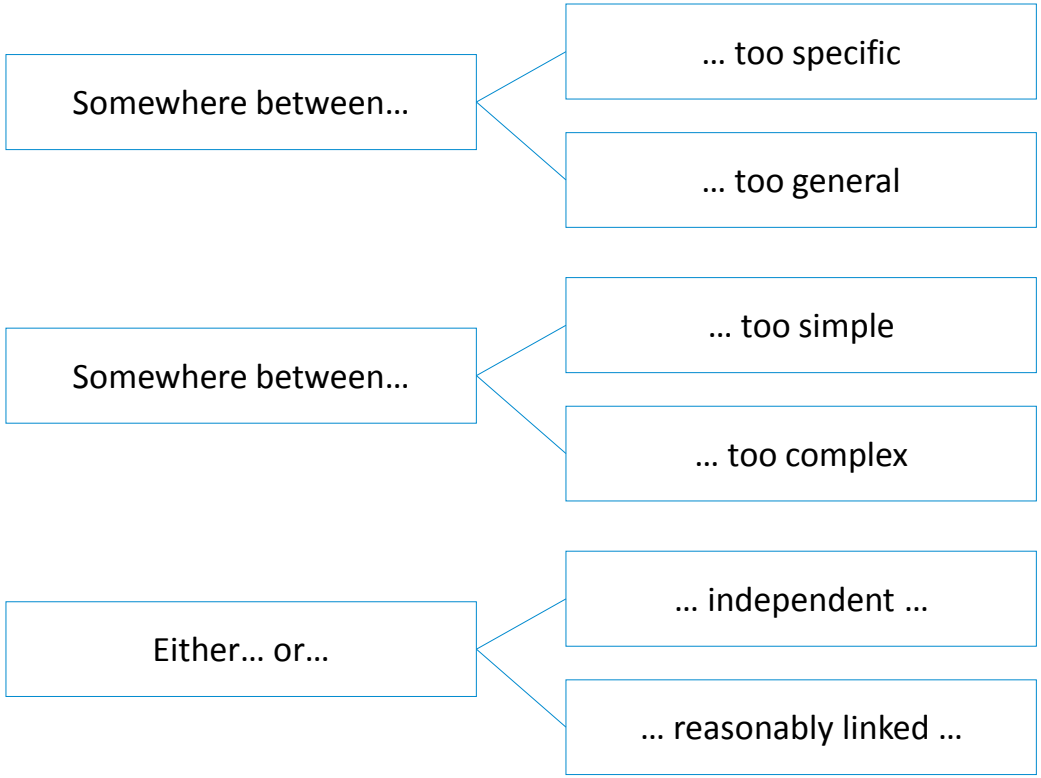
Software Testing Introduction

TRAINING

CENTER

<epam>

And there are more specific properties



Either independent or reasonably linked

Read and remember!

An **independent test case** does not reference to any other test case and is not referenced by any other test case.

A **linked test ca** ither references to some other test case(es), or ha d place in a chain of test cases.

Independent test cases are industrial standard

Independency: pros and cons

Easy to execute in any order or set	Usually need more preparations
Easy to combine into sets/scenarios	Usually need more steps
Work if some other test cases fail	May produce some redundancy

Either independent or reasonably linked

Being linked: pros and cons

Usually need less preparations	More difficult to maintain
Usually need less steps	The structure is fixed and rigid
Start work from the point where previous test case ends	**If previous test case fails, all next-in-chain test cases fail too**

Once again: independent test cases are industrial standard

Some more useful ideas about good test cases…

A good test case exposes some defect with high probability

This one is better…

1/1 = 1

-5/0 = err!

A good test case follows its goals without retreat

Really? What for?!

File saving

1. Open “File -> Print”.
2. …

Some more useful ideas about good test cases…

A good test case does not contain unnecessary steps

…
3. Open “File -> Print” dialog.
4. Click several times in random places of the windows.
…

What for?!

Some more useful ideas about good test cases…

A good test case is not redundant with other test cases

17+98

If we have this…

… we don't need this (at the same iteration)

1+1	3+4	1+98
1+2	7+5	20+21
2+1	8+1	30+12

Some more useful ideas about good test cases…

A good test case makes found defects obvious

237465689645896589236892365 * 2375641647647647816478 = 5.64133389910162755434020 79018219e+47

Is there any defect?!

100 + 50 = 212

Yes, this is a defect!

Some more useful ideas about good test cases…

A good test case performs some non-trivial actions

…
3. Upload the file with the following name “%^##76 / // \ ^^ [ ] : .jpg” as user avatar.
…

<epam>

Good Test Cases Properties – Part 2

Software Testing Introduction

TRAINING

CENTER

<epam>

<epam>

Test Suites

Software Testing Introduction

TRAINING

CENTER

<epam>

Read and remember!

Test Suite – a set of test cases or test procedures to be executed in a specific test cycle.

Usually test suites are sets of test cases selected with some common goal or on some common basis.

The main idea here is to manage testing process, to organize huge number of test cases into small convenient sets.

Test Suites creation logic

When composing test suites, we may use any common sense. I.e. we may compose a test suite with test cases that…

Cover the same requirements section

Have the same priority, module/submodule/etc…

Work with the same functionality (e.g. FS, DBMS)

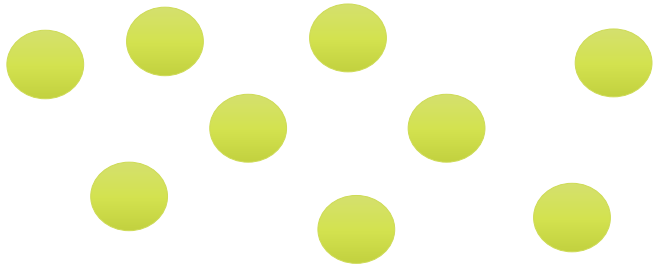
Cover some user scenario or typical set of actions

Read and remember!

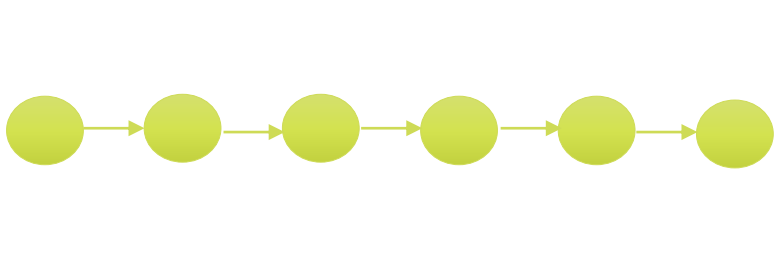
Free set of test cases doesn't imply any specific execution order, it only composes several test cases together.

Linked set of test cases implies specific execution order.

Free



Linked



Test Suites: free sets and linked sets

Free	Linked
Easy to compose	Need less preparations and steps
Easy to change execution order	Continues work from the point where previous test case ends
Still work if some test cases fail	Easy to emulate user scenarios
May produce some redundancy	**If previous test case fails, all next-in-chain test cases fail too**
Hard to emulate user scenarios	

<epam>

Test Suites

Software Testing Introduction

TRAINING

CENTER

<epam>

<epam>

Focusing on Important Things

Software Testing Introduction

TRAINING
CENTER

<epam>

Closer look to some familiar things…

Defect – an imperfection or deficiency in a work product where it does not meet its requirements or specifications.

Expected result – the predicted observable behavior of a component or system executing under specified conditions, based on its specification or another source.

Actual result – the behavior produced/observed when a component or system is tested.

Closer look to some familiar things…

Defect – an imperfection or deficiency in a work product where it does not meet its requirements or specifications.

Ex
co
ba

Ac
component or system is tested.

We absolutely have to have answers:
WHAT is expected?
By WHOM is it expected?
How IMPORTANT is this?

Quality – is some **value** for the customer

This is not necessarily (not only) functionality

How to deliver the value

Imagine a colleague asks you to develop an application (see the requirements below). He may wait for only 30 minutes and agrees to pay you a chocolate bar for this work ☺.

The console application searches for all text files in the specified directory and subdirectories.
Found files are displayed in two ways:

- [mandatory] As an HTML table (columns: file name, file size, full path to file).
- [optional] As a text file: each found file is listed in a separate line: file name, file size, full path to file.

What to do and what not to do? Applying common sense.

1. Search for text files in a given directory:
   - Without subdirectories.
   - With subdirectories.
   - Directory without txt-files.
2. File Names:
   - Latin letters only.
   - Russian letters, spaces, special characters.
   - Unicode.
   - Unicode special characters (including “unprintable”).
   - Names of different (including invalid) lengths.
   - Text files with the extension written in different ways (including upper case only).

What to do and what not to do? Applying common sense.

3. Output (in txt-file and HTML-file):
- Output data correctness:
  + Encodings.
  + The names of different files in different encodings.
- Output data validity:
  + HTML compliance to the standards.

What to do and what not to do? Applying common sense.

4. Work with critical parameters:
- The directory nesting depth.
- The path length.
- File size:
  + Zero.
  + 1 byte - 2 GB.
  + 2-4 GB.
  + 4+ GB.
- Number of files. (??? See in the documentation how many items in the catalog are supported by different FS.)

What to do and what not to do? Applying common sense.

5. “Looped” symbolic links.
6. Access rights missing to the analyzed FS object.
7. Output file unavailable for writing:
   - Opened by other application.
   - Read-only.
   - No more disk space.

What to do and what not to do? Applying common sense.

8. File system:
   - FAT16, FAT32, NTFS4, NTFS5, ext3fs, ext4fs;
   - raiserfs (?), FS for MacOS (???);
   - Check each of the FSs under different OSs.
9. Location:
   - On a local drive.
   - On a network:
     + Network folder.
     + Mounted network drive.

What to do and what not to do? Applying common sense.

O! M! G!

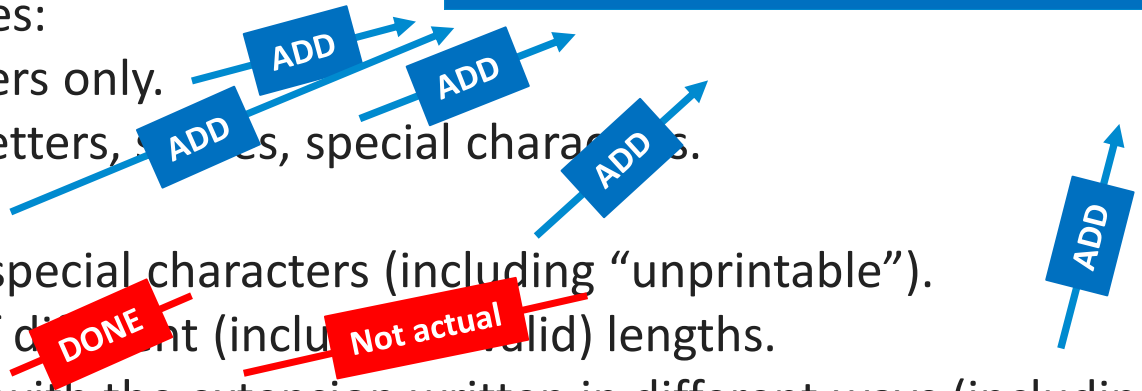
According to the most pessimistic estimations this testing process may take several months. But we only have 30 minutes!

What to do and what not to do? Applying common sense.

1. Search for text files in a given
   - Without subdirectories.
   - With subdirectories.
   - Directory without txt-files.

Lets create a directory tree with 5-7 nesting levels. There we'll have empty, non-empty, etc. directories with and without txt-files and other files. (2-3 minutes using FAR manager.)

2. File Names:
   - Latin letters only.
   - Russian letters, s…es, special chara…s.
   - Unicode.
   - Unicode special characters (including "unprintable").
   - Names of di…nt (inclu… …alid) lengths.
   - Text files with the extension written in different ways (including upper case only).



What to do and what not to do? Applying common sense.

3. Output (in txt-file and HTML-file):
   - Output data correctness:
     + En[DONE]gs.
     + The names of different [Not actual] different encodings.
   - Output data validity:
     + HTML complian[Not actual] standards.

For such a “huge” project it is enough to just look at the output results.

What to do and what not to do? Applying common sense.

4. Work with critical parameters:
   - The directory nesting depth. DONE
   - The path length. DONE
   - File size:
     + Zero. ADD to the existing set
     + 1 byte – 2 GB. DONE
     + 2-4 GB. ADD to the existing set (3GB)
     + 4 Not actual
   - Number of files Not actual (??? See in the documentation how many items in the catalog are supported by different FS.)

Another 1-2 minutes using FAR manager.

What to do and what not to do? Applying common sense.

5. “Loope[illegible]ymbolic links.
6. Access rights missin[illegible] analyzed FS object.
7. Output file unavailable for writing:
   - Opened by other application.
   - Read-only.
   - No more disk space.

ADD to the existing set

ADD to the existing set

Not actual (sorry, but…)

What to do and what not to do? Applying common sense.

8. File system:
   - FAT16, FAT32, NTFS4, NTFS5, ext3fs, ext4fs;
   - raiserfs (?), FS f… OS (???);
   - Check each of the FSs under different OSs.

Not actual

9. Location:
   - On a loc… ive.
   - On a network:
     + Network folder.
     + Mounted network drive.

DONE

Run the application from a virtual machine or from another PC in the LAN.

So, 5-10 minutes instead of several months.

Yes, IF we have much-much-much more resources, we may add… some percent to the quality.

Quality – is some **value** for the customer

This is what the customer pays for. So, provide affordable results, not “a tea cup with a complexity and a price of a space ship”.

<epam>

Focusing on Important Things

Software Testing Introduction

TRAINING
CENTER

<epam>

<epam>

Good Test Cases with Four Questions

Software Testing Introduction

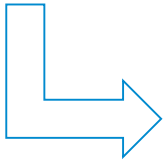
TRAINING

CENTER

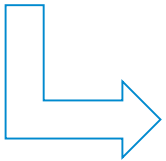
<epam>

First, some obvious ideas…

Test Cases help us to find defects

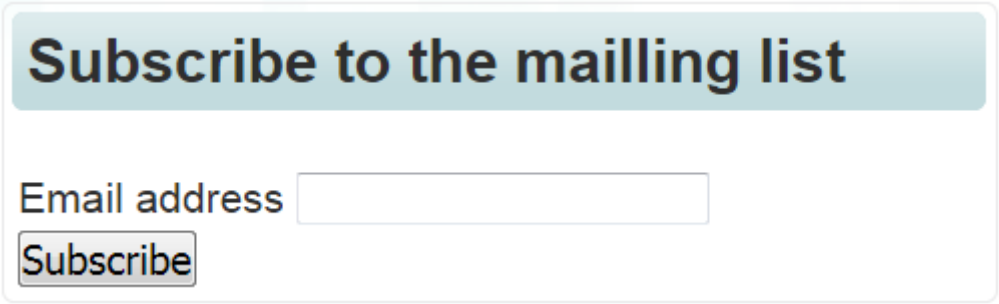


But it’s impossible to find ALL defects



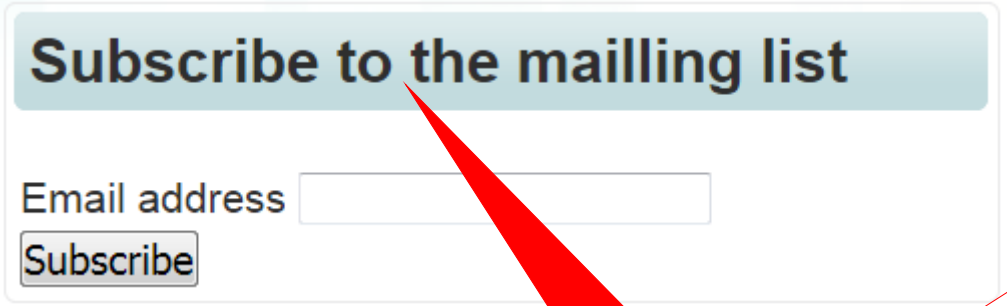
So, we have to find as many IMPORTANT defects as we may with the time available

BAD! Don’t do such things!



1. Enter «abc».
2. Enter «%^#^#^%$&^#^^#&^^%$%#^#&^».
3. Enter «; delete from `users`; -- »
4. Leave the field empty.

BAD! Don’t do such things!



1. Enter «abc».
2. Enter «%^#^#^%$&^[illegible]&^^%$%#^#&^ ».
3. Enter «; delete from `u[illegible]-- »
4. Leave the field empty.

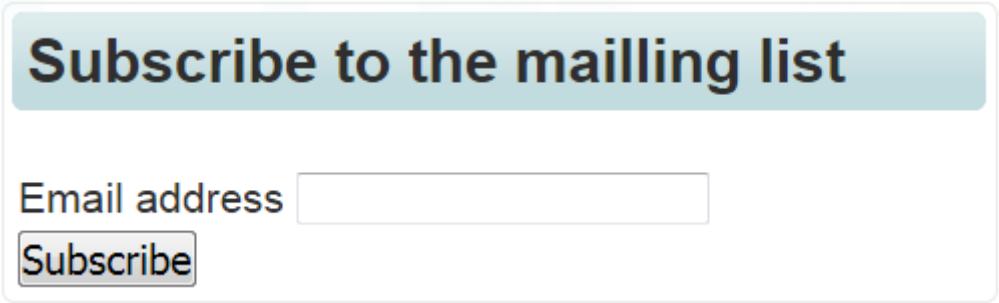
WHAT?! IS?! THIS?!

Positive tests? No…

Button click? No…

What is the purpose of this form?!

BAD! Don’t do such things!

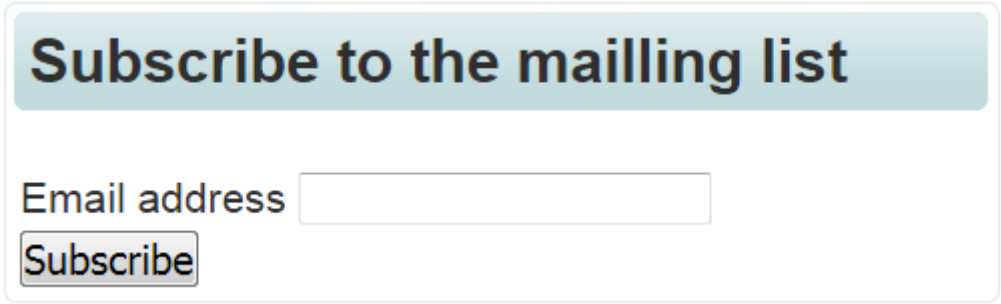


Test 1
open http://site.com/subscribe/

Test 2
verifyElementNotPresent //form

Test 3
type //input $%@%^#$%$%^^

BAD! Don’t do such things!



Test 1
open http://site.com/subscribe/

Test 2
verifyElementNotPresent //form

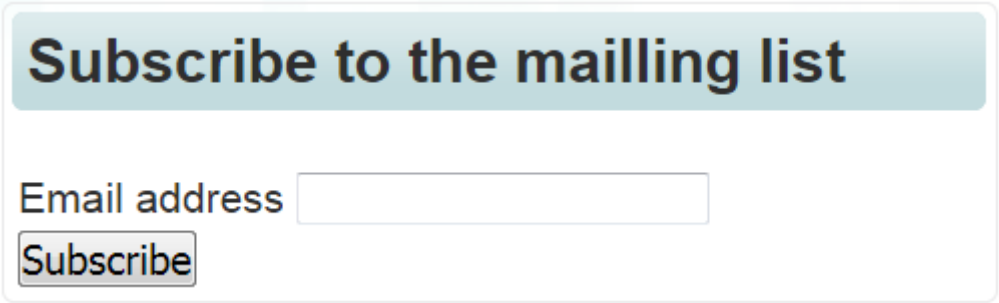
Test 3
type //input $%@%^#$%$%^^

This is insanity… ☹

Three mortal sins in testing

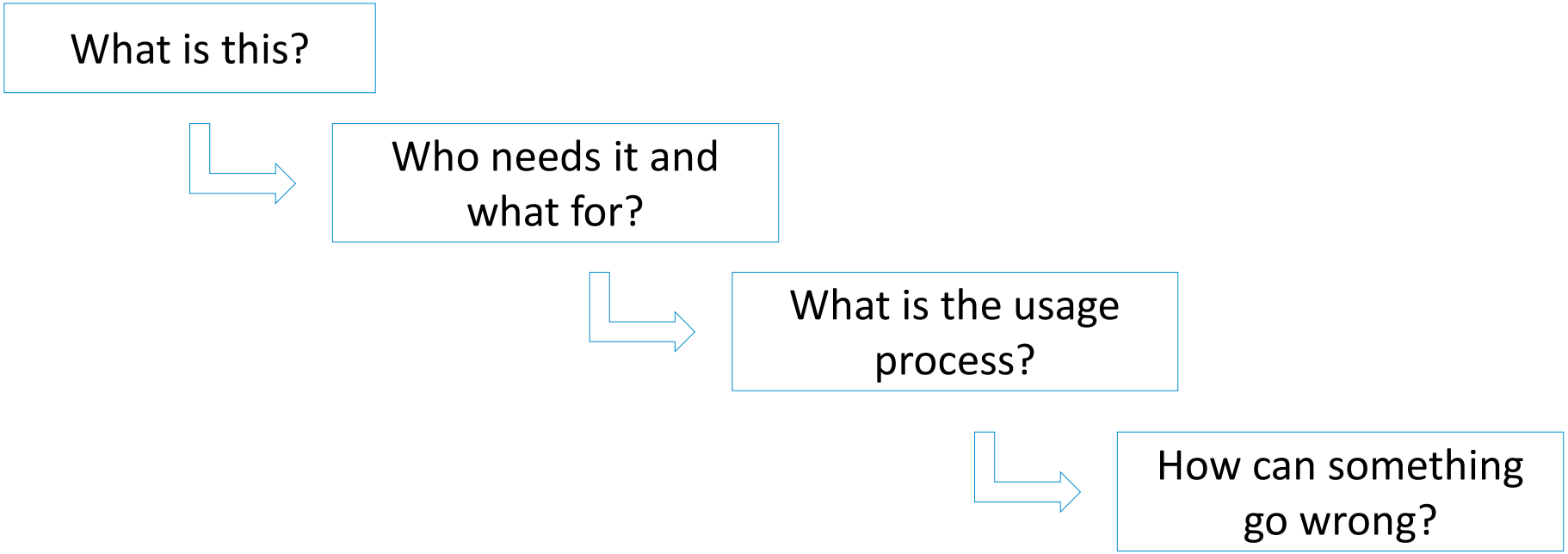
“False-positive tests”	Tests that pass successfully and does not find the existing defect.
“Blind tests”	Tests that follow some mysterious logic, do complex things and all this… for nothing.
“Garbage tests”	Tests that analyze insignificant behavior aspects (usually covered by other tests), e.g. tests like “fill this field, then this field, then this field… and that’s all”.

Good! Still may be improved, but really good!



1. Subscribe to the mailing list.
2. Check for the confirmation email.
3. Check for the mailing list emails.
4. Subscribe again with the same email.
5. Unsubscribe (how?).
6. What may go wrong during the subscription process? How can we simulate this situation?

So, the final idea:



Yes, it seems that simple. All the questions are simple for real, the difficulty is in the answers. The real answers. The answers that sometimes took days to find.

<epam>

Good Test Cases with Four Questions

Software Testing Introduction

TRAINING

CENTER

<epam>

«File Converter» Project

Checklists, Test Cases, Test Suites sample

SAMPLE

This is **NOT** a part of project documentation! This is just a way to demonstrate the whole process of checklists, test cases, and test suites creation.

Contents

1. Stage 1: read and improve the requirements

First, let us take a pause and once again read this document attentively:

Requirements Sample.docx

2. Stage 2: checklist creation logic and checklist sample

As we cannot “test the entire application at once” (this is a too huge task to solve), we have to follow some logic to build checklists – yes, there will be several of them (as a result, they can be combined into one, but this is optional action).

Typical approaches to create separate checklists are:

- by levels of functional testing;
- by individual parts (modules and submodules) of the application;
- by individual requirements, groups of requirements, levels and types of requirements;
- by typical user scenarios;
- by functions of the application that are mostly at risk.

Let’s use the logic of splitting application functions according to their degree of importance:

- Basic functions (i.e., the most important ones; failure with these functions make the application unusable).
- Functions used by most users in their daily work.
- Remaining functions (a variety of "little things"; failure with these function do not really affect the value of the application for the end user).

2.1. Basic functions

- Configuration and start.
- File processing:

		Input file format		
		TXT	HTML	MD
Input file encoding	WIN1251	+	+	+
	CP866	+	+	+
	KOI8R	+	+	+

- Stop.

Yes, that’s all. These are all basic functions of this small application.

2.2. Functions used by most users in their daily work

A very common question here is: can we copy here some ideas from the previous level? The answer is “yes and no at the same time”. “No” because it doesn’t make sense to repeat the same checks. “Yes” because any idea can be expanded and provided with additional details.

- Configuration and start:
  - With correct parameters:
    - Values for SOURCE_DIR, DESTINATION_DIR, LOG_FILE_NAME are passed to the application. These values (all of them) should contain spaces and Cyrillic symbols. Repeat this test for Windows and *nix file systems (pay attention to directory separators (“/” и “\”) and logical drives).
    - LOG_FILE_NAME value is omitted.
  - Without any parameter.
  - With some mandatory parameters omitted.
  - With incorrect parameters:
    - Wrong path for SOURCE_DIR.
    - Wrong path for DESTINATION_DIR.
    - Wrong file name for LOG_FILE_NAME.
    - DESTINATION_DIR is inside SOURCE_DIR.
    - DESTINATION_DIR and SOURCE_DIR are the same.
- File processing:
  - Miscellaneous formats, encodings and sizes:

		Input file format		
		TXT	HTML	MD
Input file encoding	WIN1251	100 KB	50 MB	10 MB
	CP866	10 MB	100 KB	50 MB
	KOI8R	50 MB	10 MB	100 KB
	Any	0 bytes		
	Any	50 MB + 1 B	50 MB + 1 B	50 MB + 1 B
	-	Any non-supported format		
	Any	Damaged file of any supported format		

  - Inaccessible input files:
    - No access rights.
    - File is opened and locked.
    - File with “read-only” attribute.
- Stop:
  - Close the console window (do not use Ctrl-C).
- Log:
  - Creation (if no log was created yet).
  - Appending (for the existing log) with application restart.
- Performance:
  - Execute some elementary rough test.

Please note that the checklist may contain not only “just ideas”, but also comments, details, and examples, if necessary.

2.3. Remaining functions

The time has come to pay attention to various potential problems that may hardly trouble the user, but formally such problems are still application defects.

- Configuration and start:
  - Values for SOURCE_DIR, DESTINATION_DIR, LOG_FILE_NAME:
    - In different styles (Windows-like + *nix-like): one value in one style and the other value in another style at the same time.
    - UNC-names.
    - LOG_FILE_NAME inside SOURCE_DIR.
    - LOG_FILE_NAME inside DESTINATION_DIR.
  - Log file size at the application start:
    - 2–4 GB.
    - 4+ GB.
  - Start of two (and more) copies of the application:
    - With the same SOURCE_DIR, DESTINATION_DIR, LOG_FILE_NAME.
    - With the same SOURCE_DIR and LOG_FILE_NAME, but different DESTINATION_DIR.
    - With the same DESTINATION_DIR and LOG_FILE_NAME, but different SOURCE_DIR.
- File processing:
  - File of valid supported format containing text in two or more encodings.
  - Input file size:
    - 2–4 GB.
    - 4+ GB.

3. Stage 3: test cases sample

Here we have only some test-cases samples (in real life there will be hundreds of test cases).

Test case sample 1:

Id	Priority	Requirement	Module	Submodule	Steps	Expected Results
ST.7	A	DS-5.1	File processor	Encoding converter	**Conversion from all supported encodings** Preparations: • Create four separate directories in the root folder of any logical drive (or in the user home directory for *nix systems): one for INPUT files, one for OUTPUT files, one for LOG file, one for TEMPORARY test files storage. • Unpack files from the attached archive into TEMPORARY directory. 1. Start the app with corresponding paths from preparations as parameters (use any log file name). 2. Copy all files from TEMPORARY directory into INPUT directory. 3. Stop the app.	1. The app starts and writes the startup message to the console and to the log file. 2. Files from INPUT directory move to OUTPUT directory. Messages about each file conversion (with information about source encoding) are displayed both in the console and in the log file. 3. The app writes shutdown message into the log file and stops.

Test case sample 2:

Id	Priority	Requirement	Module	Submodule	Steps	Expected Results
CPT.17	B	-	Scanner	Traverser	**Multiple copies of the app working simultaneously, file operations conflict** Preparations: • Create three separate directories in the root folder of any logical drive (or in the user home directory for *nix systems): one for INPUT files, one for OUTPUT files, one for LOG file. • Create several files of the maximum supported size, supported formats, and supported encodings. 1. Start the first copy of the app with corresponding paths from preparations as parameters (use any log file name). 2. Start the second copy of the app with the same parameters (see step 1). 3. Start the third copy of the app with the same parameters (see step 1). 4. Change the process priority for the second copy to “high” and for the third copy to “low”. 5. Copy the prepared set of files (see preparations section) into the INPUT directory.	3. All three copies of the app are started, the log file contains three sequential startup records. 5. Files gradually move from INPUT to OUTPUT directory. The console and the log file contains messages about successful file conversion for each file. The console and the log file may also contain the following error messages: a. “source file inaccessible, retrying”. b. “destination file inaccessible, retrying”. c. “log file inaccessible, retrying”. The key success indicator for this test is the successful conversion of all files while no copy of the app should crash. The log file should contain at least one record (thee – maximum) for each converted file. Mentioned above error messages are OK too. They may appear, or may not – it depends on too many conditions and too hard to predict.

Test case sample 3:

Id	Priority	Requirement	Module	Submodule	Steps	Expected Results
ST.5	A	DS-2.4	App loader	Error handler	**Incorrect start parameters, nonexisting paths** 1. Start the app with all three parameters (SOURCE_DIR, DESTINATION_DIR, LOG_FILE_NAME) pointing to nonexisting in current file system paths (e.g.: z:\src\, z:\dst\, z:\log.txt in case there is no z: drive in the system).	1. The console contains the following messages, the app stops. Messages: a. Usage message. b. SOURCE_DIR: directory not exists or inaccessible. c. DESTINATION_DIR: directory not exists or inaccessible. d. LOG_FILE_NAME: wrong file name or inaccessible path.

4. Stage 4: test suites sample

Imagine we have the set of the following tests:

Id	Priority	Requirement	Module	Submodule	Title
CPT.1	B	DS-2.1	App loader	Parameters analyzer	Default log file name
CPT.10	B	-	Scanner	Traverser	No access rights to the output directory
CPT.11	C	-	Scanner	Traverser	UNC names in parameters
CPT.12	C	-	App loader	Parameters analyzer	Windows-like and *nix-like paths in parameters
CPT.13	C	-	Logger	-	Log file size is 4+ GB
CPT.14	C	DS-5.3	File processor	Encoding converter	Text file contains binary data
CPT.15	C	-	Scanner	Traverser	Locked file conversion
CPT.16	C	-	Scanner	Traverser	“Read-only” file conversion
CPT.17	B	-	Scanner	Traverser	Multiple copies of the app working simultaneously, file operations conflict
CPT.2	B	BR-1.1	App loader	Error handler	Destination and input directory are the same
CPT.3	B	BR-1.2	App loader	Error handler	Destination directory is inside input directory
CPT.4	B	DS-5.2	File processor	Encoding converter	Conversion of file with maximum allowed size
CPT.5	B	DS-5.2	Scanner	Traverser	Too big files conversion
CPT.6	B	-	Scanner	Traverser	Empty files conversion
CPT.7	C	-	Scanner	Traverser	Directory tree inside the input directory
CPT.8	C	-	File processor	Encoding converter	Multiple encodings inside one file
CPT.9	B	DS-2.4	Scanner	Traverser	No access rights to the input directory
ST.1	A	DS-2.2	App loader	Error handler	Start with no parameters
ST.2	A	DS-2.2	App loader	Error handler	Start with some mandatory parameter omitted
ST.3	B	DS-2.4	App loader	Error handler	Invalid path to the log file
ST.4	A	DS-2.4	App loader	Error handler	Invalid paths to the input/output directory
ST.5	A	DS-2.4	App loader	Error handler	Incorrect start parameters, nonexisting paths
ST.6	B	-	App loader	Parameters analyzer	Valid parameters with spaces and Cyrillic symbols
ST.7	A	DS-5.1	File processor	Encoding converter	Conversion from all supported encodings

Now we can compose the following (or any other) sets:

- If we want tests for “Error handler” submodule, we get the following test cases: CPT.2, CPT.3, ST.1, ST.2, ST.3, ST.4, ST.5.
- If we want tests with the priority “A”, we get the following test cases: ST.1, ST.2, ST.4, ST.5, ST.7.
- If we want tests with the priority “B” for the “App loader” module, we get the following test cases: CPT.1, CPT.2, CPT.3, ST.3, ST.6.
- If we want tests for the “BR” section of Requirements, we get the following test cases: CPT.2, CPT.3.
- If we want tests with the priority “C” for the “File processor” module and the “DS” section of Requirements, we get the following test case: CPT.14.
- And so on: we choose the set of criteria and get the resulting set of test cases.