<epam>

What is Requirement and Why Requirements are Important

Software Testing Introduction

TRAINING

CENTER

<epam>

Read and remember!

Requirement – a condition or capability needed by a user to solve a problem or achieve an objective that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.

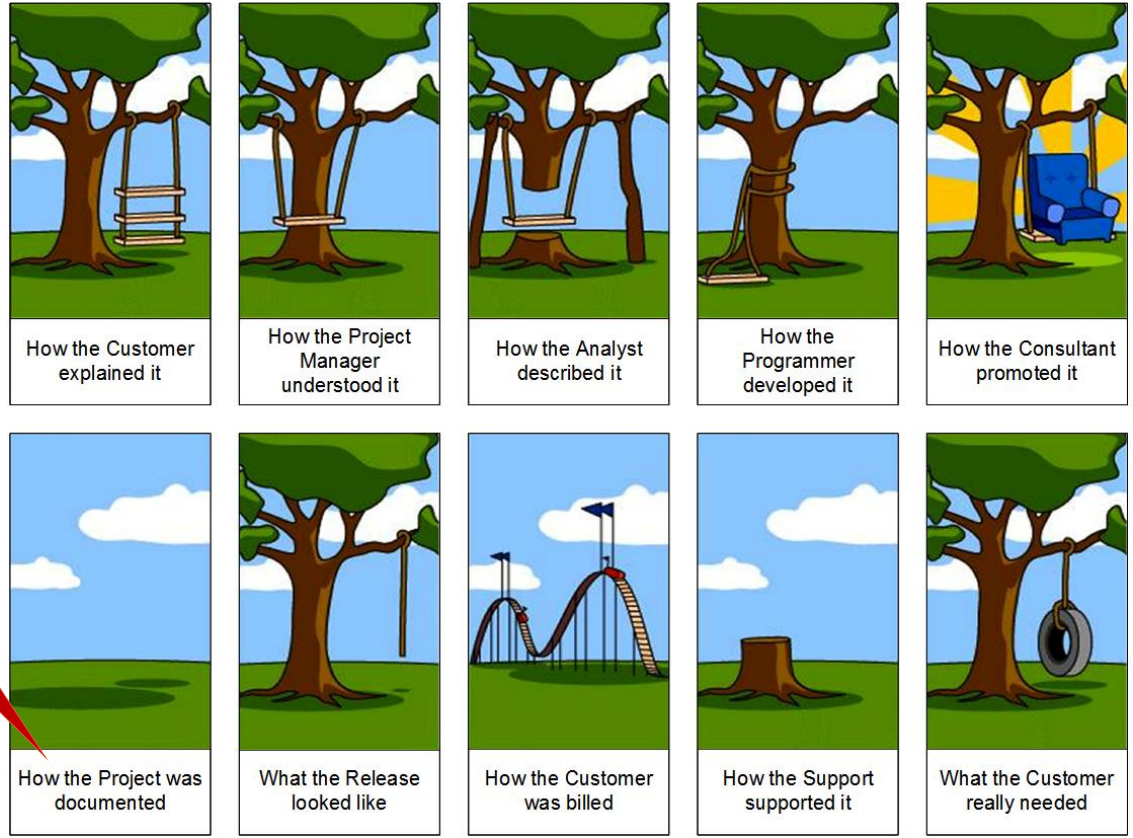
General Documentation Overview

Product documentation
Project management plan
Test plan
Requirements
Architecture and design
Test cases, test suites
Technical specifications

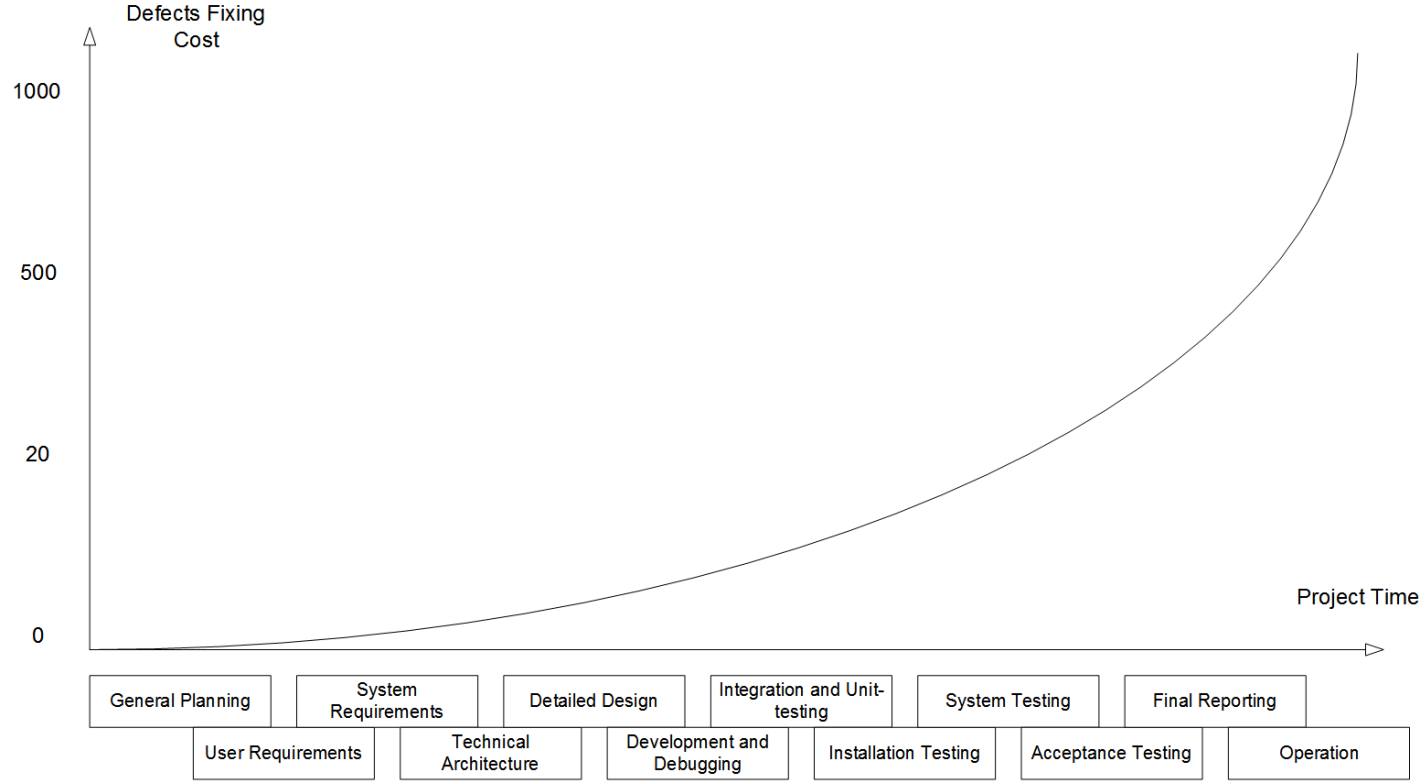
Project documentation
User and accompanying documentation
Market requirements

Why Documentation Is Important

One of the greatest reasons for the Failure is here

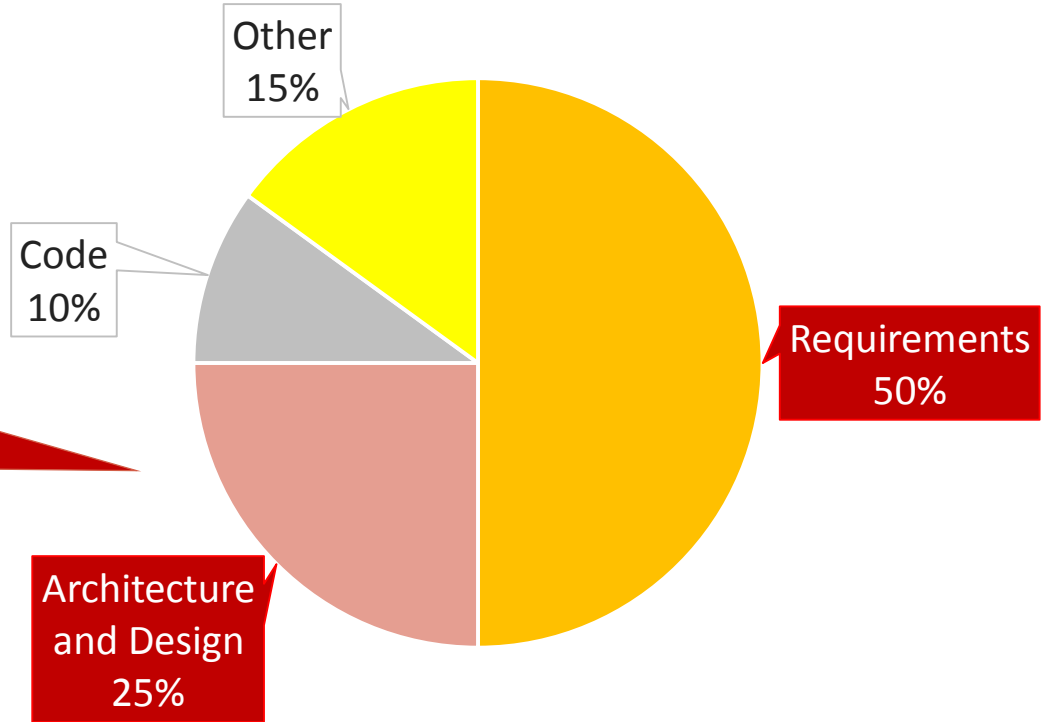


Why Documentation Is Important



Why Documentation Is Important

¾ of defects are originated from documentation



<epam>

What is Requirement and Why Requirements are Important

Software Testing Introduction

TRAINING

CENTER

<epam>

<epam>

Ways of Requirements Gathering – Part 1

Software Testing Introduction

TRAINING

CENTER

<epam>

General Overview

Interview	Meetings and Brainstorming	Questioning
Observation	Prototyping	Modeling
Documentation Analysis	Work with Focus Groups	Self-(re)writing

Read and remember!

Interview – the most common way of identifying requirements. Usually, two main roles involved: interviewer (project specialist, BA) and interviewee (customer representative or expert, user, etc.)

Fast, easy, no special sophisticated techniques required.
Everybody understands the concept, so everybody can participate.
Variations possible: in-person, via (video) call, e-mail, messenger, etc.

Often some visual aids are required.
It takes time to rework the results as a useable set of data.
Interviewer bears the most responsibility for the results.

Read and remember!

Meetings ~~and Brainstorming~~ – unlike with **Interview**, several people are playing active role here to share information, discuss issues and be on the same page immediately.

Several key people share information and receive feedback at once.
Meetings are faster than e-mail/messenger communication.
“Collective thinking” may create new good ideas.

Meetings take time. A lot of time. No, really, **A LOT OF TIME**.
Sometimes it’s difficult to gather all necessary people together.
With intense rhythm meetings become boring and inefficient.

Read and remember!

Questioning (and questionnaires) – is a fast and easy way of gathering and aggregation of information from thousands of respondents. When organized properly, questioning brings a lot of useful results.

Great audience coverage – thousands and millions of respondents.
Nice data processing and visualizing capabilities.
Possibility to easily compare several results (over time).

If held improperly (bad questions, wrong target audience, demotivated respondents and so on), the questioning can give us no data, mess data or (in worst case) wrong data that looks like a good one.

Read and remember!

Observation – is a method that shows things you never hear about. People are apt to lie (not only to others but to themselves), they forget things, do not know things… But observation shows the truth.

Experienced observer may pick some mission critical details.
This method is as objective, as possible.
Observation results may lead to the whole problem re-evaluation.

It’s hard to organize observation well most of the time.
Experienced observers are rare.
When improperly organized, observation gives a lot of “bad data”.

<epam>

Ways of Requirements Gathering – Part 1

Software Testing Introduction

TRAINING

CENTER

<epam>

<epam>

Ways of Requirements Gathering – Part 2

Software Testing Introduction

TRAINING

CENTER

<epam>

General Overview

Interview	Meetings and Brainstorming	Questioning
Observation	Prototyping	Modeling
Documentation Analysis	Work with Focus Groups	Self-(re)writing

Read and remember!

Prototyping allows to use “something real” (not imaginary) both as (and always these two ways): a) the source of the new information; b) real material thing to discuss.

Most people prefer to see the object in discussion.
No matter how detailed the description is, real object provides more data.

Prototyping takes time. Some work may be discarded after discussion.
It is crucial to do enough and **stop**, otherwise a lot of work may be wasted.

Read and remember!

Modeling is somehow similar to prototyping, but it (usually) does not require “physical” development. A model may be mathematical one, computational one and so on.

Model makes a lot of difficult things easy and understandable.
Model allows to see hidden properties of some object / process.
Model allows to see if something is even possible / good.

Modeling requires a lot of experience, knowledge, tools.
Improper model may give us bad data.
Models are harder to understand than prototypes.

Read and remember!

Documentation analysis allows us to find issues with existing project/product documentation and to get extra information (when reading standards, manuals, books, etc.)

This method requires no special actions, tools, and so on: you may just sit and read a document. Anytime you like. Anywhere you like. Reading documentation makes you a better professional.

Sometimes with new data new questions rise (you have to write them down for further discussion). Working alone increases the probability to miss a defect or to create one.

Work with Focus Groups

Read and remember!

Work with focus groups looks a lot like questioning except it involves in-person communication, discussions, end-user involvement in beta-testing, deep end-user participation in product development.

We can gather a lot of information from real end-users.
Unlike questioning this activity creates deep involvement.
We may not only gather information, but influence people opinion.

Sometimes it’s difficult to gather a proper focus group.
Experienced professional is required to organize the process.
Some NDA issues may arise.

Read and remember!

Self-(re)writing is mostly a way not for gathering requirements, but for formalizing them (because it's extremely difficult to think for the customer and somehow telepathically know his ideas ☺).

Each and every just mentioned way of requirements gathering provides us with information that has to be re-worked, re-formulated, and included into Requirements document. Self-(re)writing gives us that.

Sometimes it's difficult to avoid adding your own ideas into the Requirements. You can propose ideas but absolutely have to get the approval from the Customer side.

<epam>

Ways of Requirements Gathering – Part 2

Software Testing Introduction

TRAINING

CENTER

<epam>

<epam>

Requirements Levels and Types

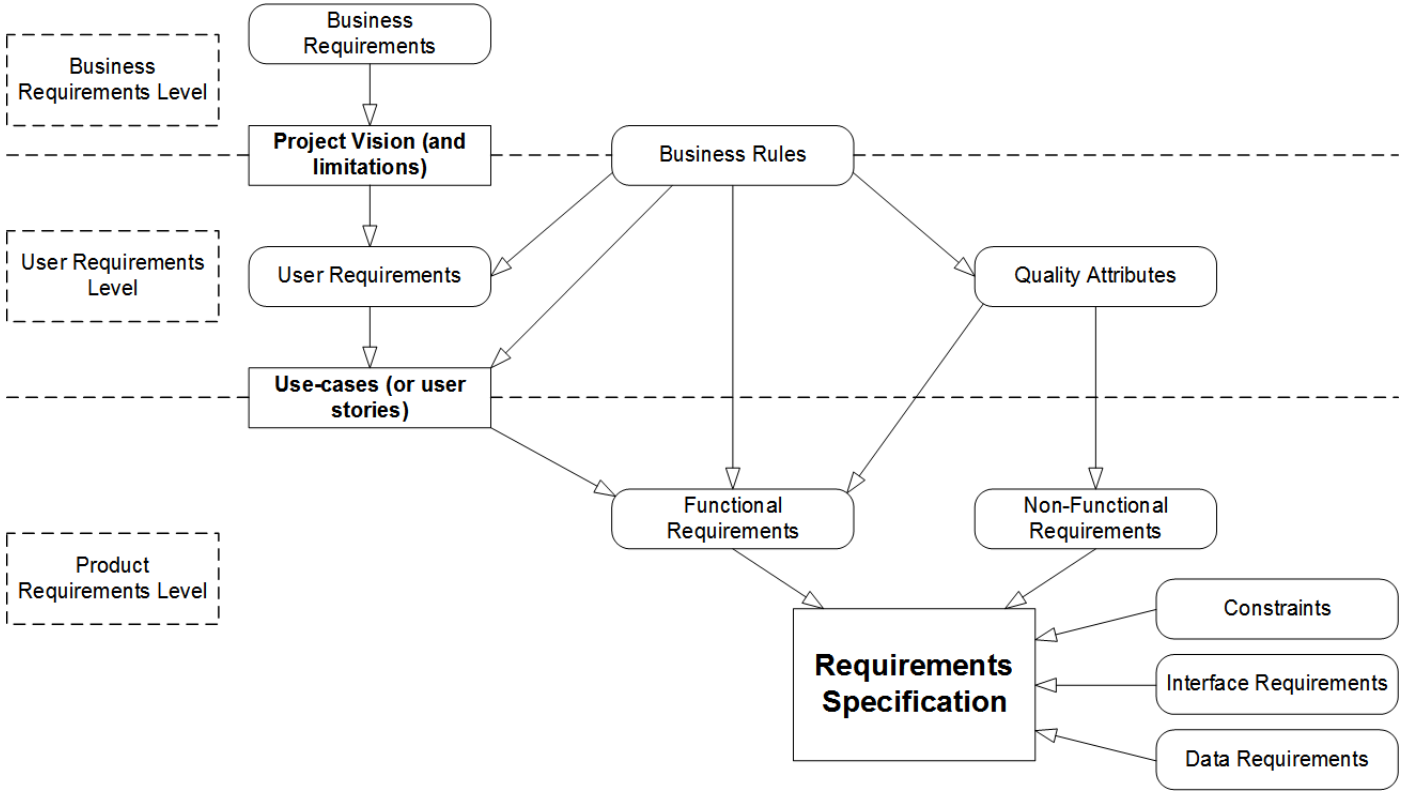
Software Testing Introduction

TRAINING

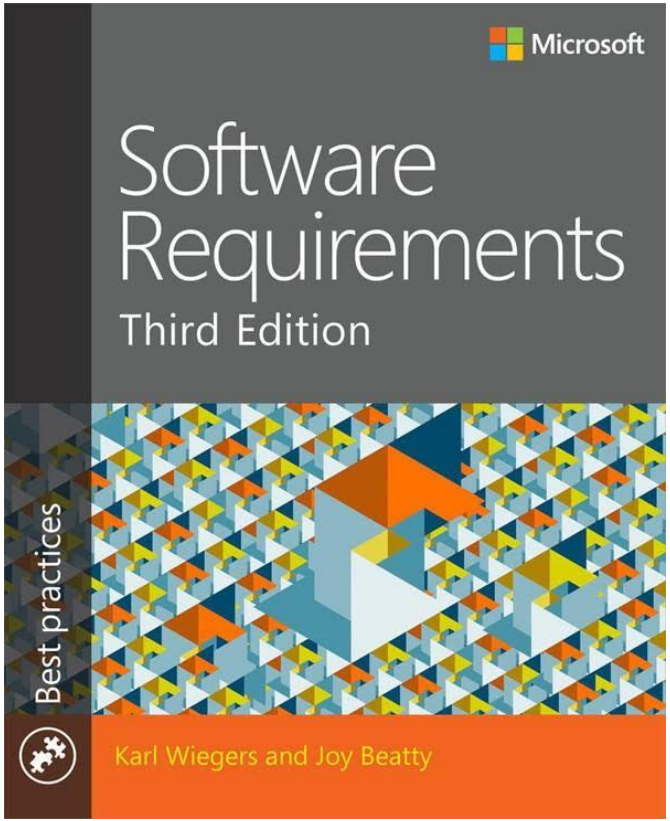
CENTER

<epam>

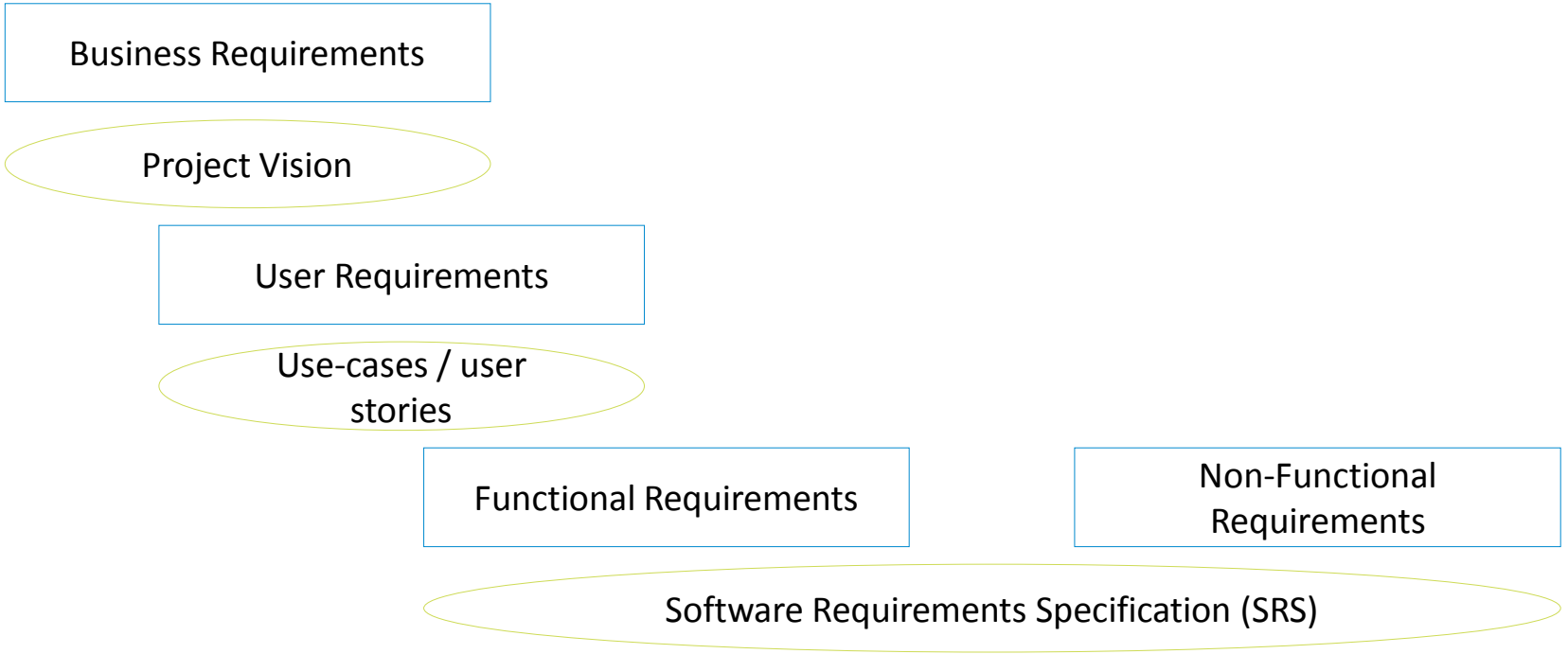
General Overview



Where to read MORE

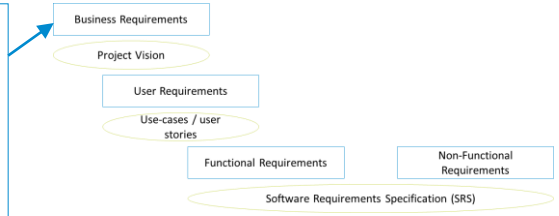


Simplified Model



Read and remember!

Business requirements express the purpose for which the product is developed (why is it needed at all, what benefits are expected from it).

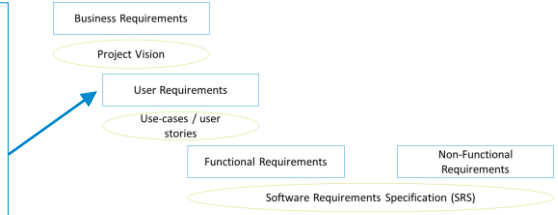


E.g.:

- *We need a tool to display the most profitable currency exchange rate in real time.*
- *It is necessary to increase by 2-3 times the number of tickets processed by one operator per shift.*
- *It is necessary to automate the process of issuing invoices based on contracts.*

Read and remember!

User requirements describe the tasks that a user can perform with the system (system response to user actions, user scenarios).



E.g.:

- *The license agreement should be displayed on the user first login.*
- *The administrator should be able to view a list of all users currently working in the system.*
- *When you first save a new article, the system should produce a request to either save it as a draft or to publish it immediately.*

Read and remember!

Functional requirement – a requirement that specifies a function that a component or system must be able to perform.

Non-functional requirement – a requirement that describes how the component or system will do what it is intended to do.

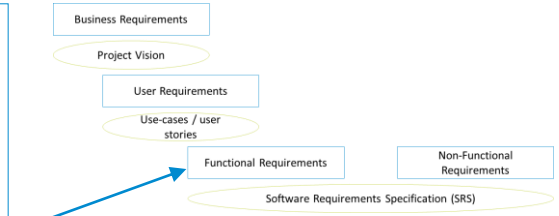
E.g.:

"A bird should fly high".

"A singer should sing nicely".

Read and remember!

Functional requirements describe the behavior of the system, i.e. its actions (calculations, transformations, checks, processing, etc.)

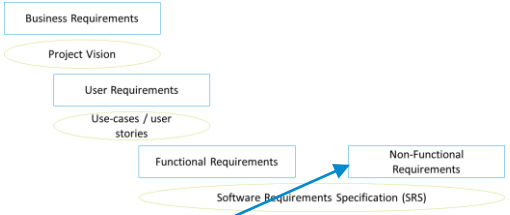


E.g.:

- *The application should check the remaining free space on the target media during installation.*
- *The system should automatically backup data daily at the specified time.*
- *The user's email address must be verified against RFC822.*

Read and remember!

Non-functional requirements describe the properties of the system (usability, security, reliability, extensibility, etc.) that it must possess when implementing its behavior.



E.g.:

- *The minimum "time to failure" must be >= 100 hours with simultaneous continuous work of 1000 users.*
- *Under no circumstances the total memory used by the application should exceed 2 GB.*
- *The font size for any label on the screen must support a range of 5-15 points.*

<epam>

Requirements Levels and Types

Software Testing Introduction

TRAINING

CENTER

<epam>

<epam>

Good Requirements Properties – Part 1

Software Testing Introduction

TRAINING
CENTER

<epam>

General Overview

Read and remember!

A requirement is **atomic** if it cannot be broken down into separate requirements without loss of completeness, and it describes one and only one situation.

Typical problems

- *One requirement actually contains several independent requirements.*
- *The requirement allows discrepancies due to the grammatical features of the language.*
- *Description of several independent situations combined into one requirement.*

Read and remember!

How to detect typical problems

Think, discuss with colleagues and use common sense. If we consider that a certain section of requirements is overloaded and requires decomposition, most likely it is so.

How to fix typical problems

Process, re-write and structure the requirements: split them into sections, subsections, paragraphs, etc.

Read and remember!

The requirement is **complete** if it provides all the necessary information, if nothing is missed or left out for reasons such as “this is already clear to everyone”.

Typical problems

- *There is no non-functional part of the requirement or link to the corresponding non-functional requirement (e.g.: “passwords must be stored in encrypted form” - what is the encryption algorithm?).*
- *Only a part of some enumeration is mentioned (e.g.: “export is carried out in PDF, PNG, etc.” formats - what should we understand by “etc.”?).*
- *The references cited are ambiguous (e.g.: “see above” instead of “see section 123.45.b”).*

Read and remember!

How to detect typical problems

Most requirements testing techniques are applicable, but the best are: asking questions and using graphical representations of the system being developed.

How to fix typical problems

Once we see that something is missing, it is necessary to obtain the missing information and add it to the requirements. A little re-writing may be needed.

Read and remember!

A **consistent** requirement should not contain internal contradictions and/or contradictions with other requirements and documents.

Typical problems

- *Contradictions within one requirement.*
- *Contradictions between two or more requirements, between a table and text, a picture and text, a requirement and a prototype, etc.*
- *Incorrect terminology or different terms to refer to the same phenomenon.*

Read and remember!

How to detect typical problems

Good memory helps best ☺, but tool for graphical representation of the system being developed are also extremely useful.

How to fix typical problems

Once we detect an inconsistency, it is necessary to clarify the situation with the Customer and make the necessary changes to requirements.

Read and remember!

A requirement is **unambiguous** (clear) if it is described without the use of jargon, non-obvious abbreviations and vague wording, and allows only one objective understanding.

Typical problems

- *Use of terms or phrases that allow subjective interpretation.*
- *Use of non-obvious or ambiguous abbreviations without decoding.*
- *Writing the requirement assuming that some parts are obvious to everybody by default and therefore needs no explanation.*

Read and remember!

Indicators of problems with unambiguousness

Adequate, be able to, easy, provide for, as a minimum, be capable of, effectively, timely, as applicable, if possible, to be determined (TBD), as appropriate, if practical, but not limited to, capability of, capability to, normal, minimize, maximize, optimize, rapid, user-friendly, simple, often, usual, large, flexible, robust, state-of-the-art, improved, efficient.

E.g. (real quotation)

"In case of necessity to optimize large files transfer, the app should effectively use the minimum of RAM (if possible)."

Read and remember!

How to detect typical problems

Look for specific words and phrases. Writing check-lists is also effective: it is hard to come up with a good check-list for a requirement that is ambiguous.

How to fix typical problems

Use numbers and formulas instead of a verbal description. If this is not possible, at least use the most accurate technical terms, references to standards, etc.

<epam>

Good Requirements Properties – Part 1

Software Testing Introduction

TRAINING
CENTER

<epam>

<epam>

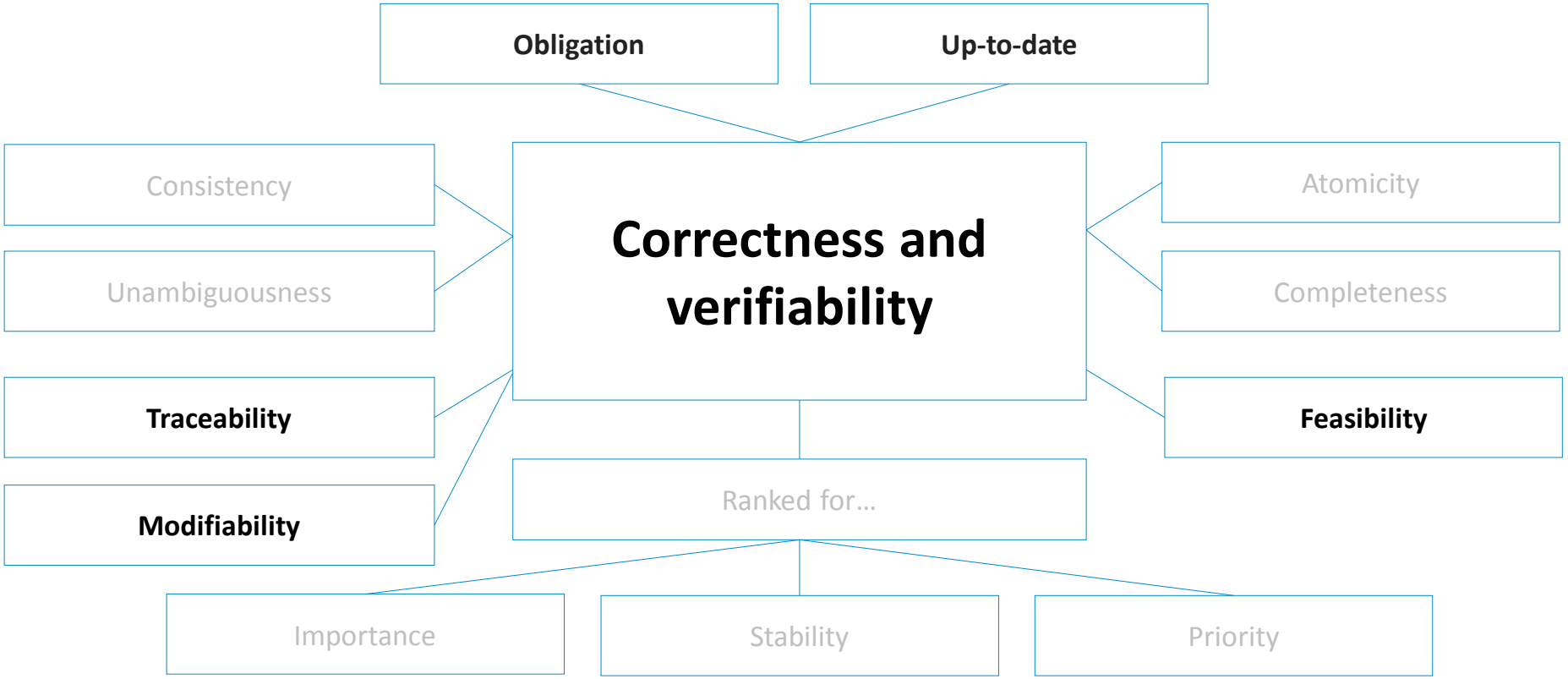
Good Requirements Properties – Part 2

Software Testing Introduction

TRAINING
CENTER

<epam>

General Overview



Obligation and up-to-date state

Read and remember!

Any requirement should be **obligatory and up-to-date**, otherwise (if the requirement is not mandatory for implementation or is outdated) it should be removed.

Typical problems

- *The requirement was added for "why not" reason although there is no real need for it.*
- *The requirement has wrong values for priority/importance properties.*
- *The requirement is outdated, but it has not been removed.*

Read and remember!

How to detect typical problems

Periodic review of requirements (preferably with the participation of the Customer) allows you to notice fragments that have lost their obligation or have become outdated ones.

How to fix typical problems

Re-write requirements to eliminate fragments that have lost obligation or have become outdated ones.

Read and remember!

A **feasible** requirement is the one that technologically achievable and may be implemented within the project budget and schedule.

Typical problems

- *So called "Gold plating" – requirements that are extremely expensive to implement and at the same time almost useless for end users.*
- *Requirements that are technically unrealistic at the present level of technology.*
- *Unrealizable requirements (usually – requirements not to the product, but to the user, environment, laws of physics and so on).*

Read and remember!

How to detect typical problems

Use every bit of your subject matter experience to understand that a certain requirement “costs” too much or is infeasible at all within current Projects limitations.

How to fix typical problems

There is nothing else to do, but to discuss the situation with the Customer and either change the requirement, or reconsider the terms of the Project (making this requirement feasible).

Read and remember!

Vertical traceability shows the connection between levels of requirements, **horizontal traceability** shows the connection between a requirement and a test plan paragraph, test cases, architectural solutions, etc.

Typical problems

- *Requirements have no ids, not structured, don't have a table of contents, don't have cross-references.*
- *No tools and/or techniques of requirements management were used during the development of requirements.*
- *The set of requirements is incomplete, it is fragmented and has obvious "blind spots".*

Read and remember!

How to detect typical problems

Traceability problems mean no answers to questions like “where did this requirement came from?”, “where are the related requirements?”, “what does it affect and what is it affected by?”

How to fix typical problems

Re-work requirements: change the structure, arrange cross-references, add tags, bookmarks, etc.

Read and remember!

Modifiability characterizes the ease of making changes to individual requirements or to a set of requirements. With good modifiability changes go well without unexpected side effects.

Typical problems

- *Requirements are non-atomic and/or untraceable, therefore changes lead to inconsistency.*
- *Requirements are initially contradictory, so changes (not related to eliminating inconsistencies) only worsen the situation.*
- *Requirements are in an inconvenient form (e.g., requirements management tools are not used, so the team has to work with dozens of huge text documents).*

Read and remember!

How to detect typical problems

No traceability – no modifiability. Also, the modifiability decreases with the violation of almost any “good requirement” property.

How to fix typical problems

Re-write requirements with a primary goal is to increase the traceability. While doing this, you can fix a lot of other problems with requirements.

<epam>

Good Requirements Properties – Part 2

Software Testing Introduction

TRAINING
CENTER

<epam>

<epam>

Good Requirements Properties – Part 3

Software Testing Introduction

TRAINING
CENTER

<epam>

General Overview

Rank for importance, stability, priority

Read and remember!

Importance shows how the success of the Project depends on the success of the implementation of the requirement.
Stability shows the likelihood that in the foreseeable future the requirement will not be changed.
Priority shows what requirements should be implemented earlier or later.

Typical problems

No rank (or incorrect rank, or outdated rank) for importance, stability, priority

Read and remember!

How to detect typical problems

Review requirements (with the participation of the Customer) to detect incorrect values of importance, stability, and priority. Repeat such review periodically.

How to fix typical problems

Make corrections during the review process.

Read and remember!

Correctness and verifiability state that it is possible to create an objective test case (test cases) that clearly shows that the requirement is implemented correctly and the behavior of the application exactly corresponds to the requirement.

Typical problems

As these properties follow from compliance with all of the above mentioned ones, if any other "good property" is violated the requirement most likely becomes incorrect and unverifiable.

Read and remember!

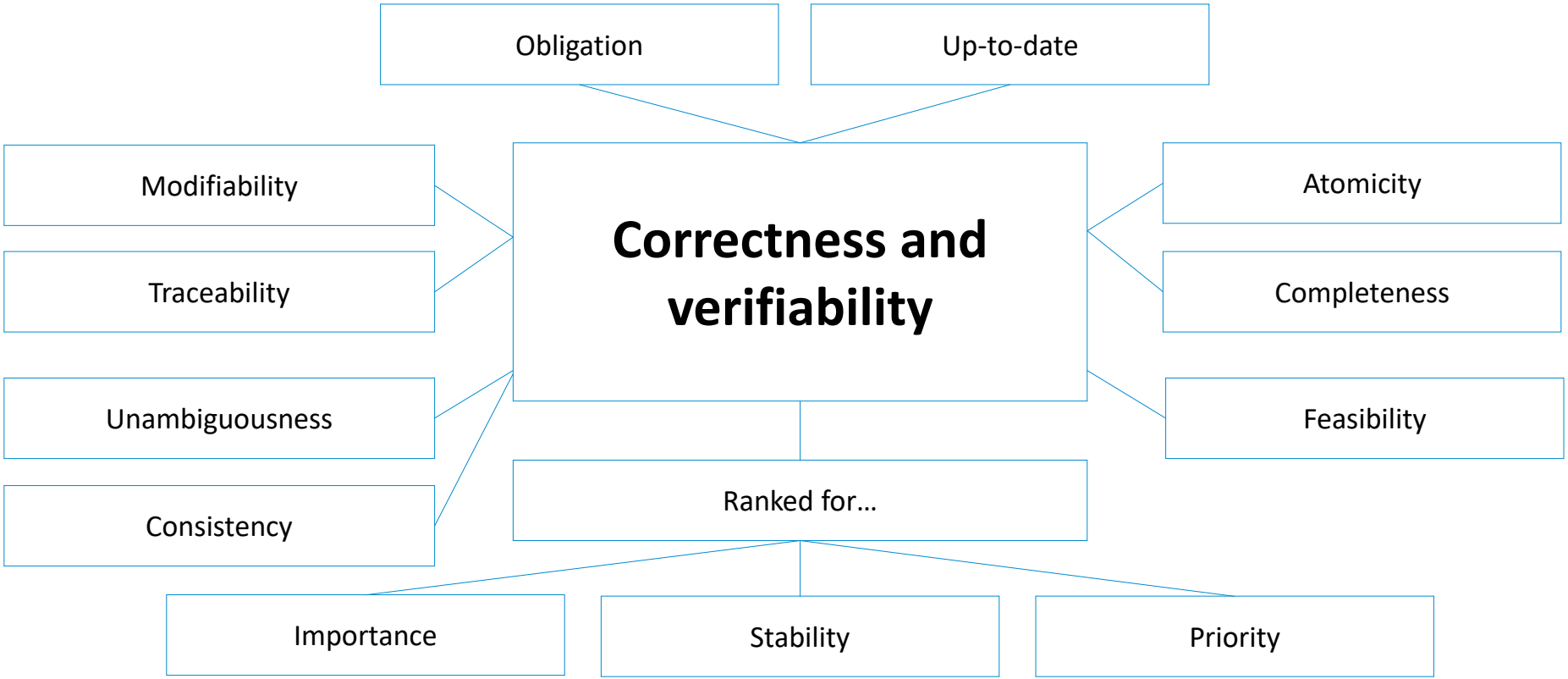
How to detect typical problems

Since we are dealing with an “integral problem”, we can use methods described earlier. There are no unique techniques here.

How to fix typical problems

Make necessary changes to the requirements. Starting from simple typo correction, and up to global re-writing of the entire requirement set.

General Overview



The main idea!

If we can not verify the requirement, we have a **problem**.

Testers are responsible for ensuring that the requirement is verified, so we have to have a way to verify it, so the requirement has to be a good one with no “good properties” violated.

<epam>

Good Requirements Properties – Part 3

Software Testing Introduction

TRAINING
CENTER

<epam>

<epam>

Requirements Testing Techniques

Software Testing Introduction

TRAINING
CENTER

<epam>

General Overview

- Peer review
- Asking questions
- Writing check-lists
- Visualization
- Modeling and prototyping

Read and remember!

Peer review – a form of review of work products performed by others qualified to do the same work.

Peer review types

Walkthrough

Quick feedback from colleagues

Technical review

Review by a group of specialists (preferable with different skillsets)

Formal inspection

Formal and systematical approach with a lot of specialists following special procedures

Asking questions

If something bothers you, if you don’t understand something, if any kind of clarification is needed – **ASK A QUESTION**!

Asking questions

Bad requirement	Bad questions	Good questions
“The app should start fast”.	• “How fast?” (What do you expect to hear? “Extremely fast”? “As fast as possible”? “Fast enough”?) • “What if it wouldn't start fast?” (Do you really want to upset the Customer or even make him angry?) • “Always?” (“Yes, always!” Did you expect some other answer?)	• “What is the maximum time to start and what hardware / OS are we talking about? How busy is the system with other tasks?” • “Why is the start time so important? What is depending on it?” • “May we start/load some components of the app in background?” • “What state/conditions should we see as the mark that the app is started?”

Writing check-lists

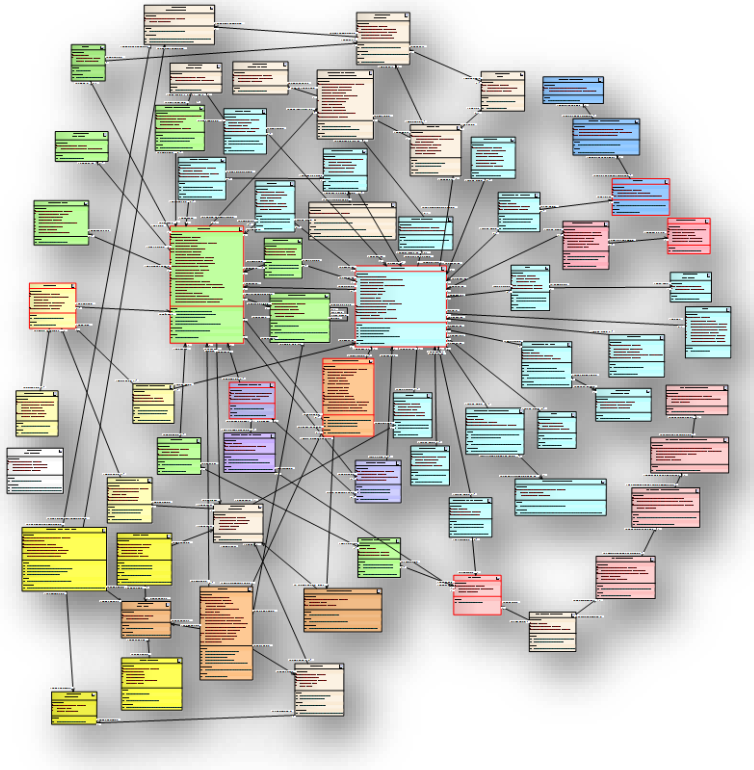
Ask yourself: “How shall I test this requirement? Are there tests that show objectively that this requirement is implemented correctly?”

If you have a lot of ideas in mind – that’s good, but not enough: this requirement may be outdated, inconsistent and so on.

If you have no ideas – that’s a reason to investigate this situation more thorough, gather more info, ask questions.

Visualization

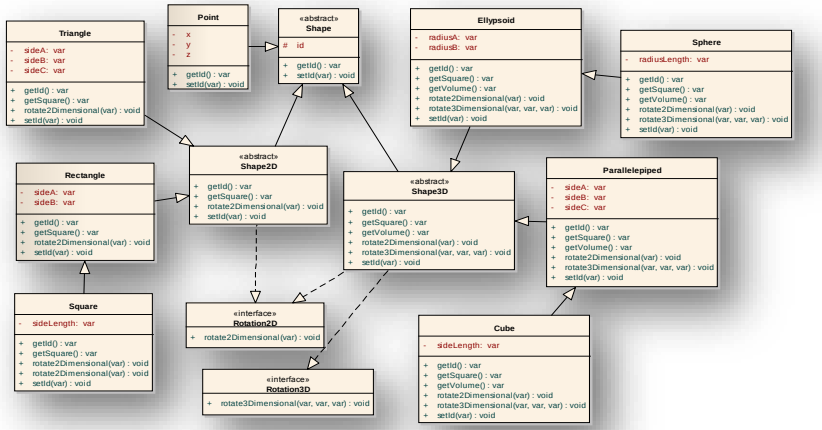
Either to see the overall big picture or to drill into details you can use drawings, diagrams, schemas, mind maps, concept maps, etc.



Modeling and prototyping

Use mental simulation of the user experience, think about ambiguous or completely undescribed options for the behavior of the system.

Prototyping and modeling may require special tools and knowledge, but it can save a lot of time and effort.



<epam>

Requirements Testing Techniques

Software Testing Introduction

TRAINING
CENTER

<epam>

«File Converter» Project

Requirements

SAMPLE

Project Documentation

Background	Full set of requirements specification.
Purpose	To organize both development and testing process.
Scope	Business requirements, user requirements, detailed specification, limitations.
Audience	Management staff, project team.
File	03 06 - Requirements Sample.docx

Contents

1. Project scope

Development of a tool to eliminate encoding multiplicity in text documents stored locally.

2. Main goals

- Eliminate the necessity for manual detection and conversion of encoding in text documents.
- Decrease document-processing time by the amount needed for manual encoding detection and conversion.

3. Criteria for main goals achievement

- Full automation of encoding detection and conversion.
- Document-processing time reduction by 1-2 minutes (average) due to elimination the necessity for manual encoding detection and conversion.

4. Risks

- High complexity of accurate detection of text document initial encoding.

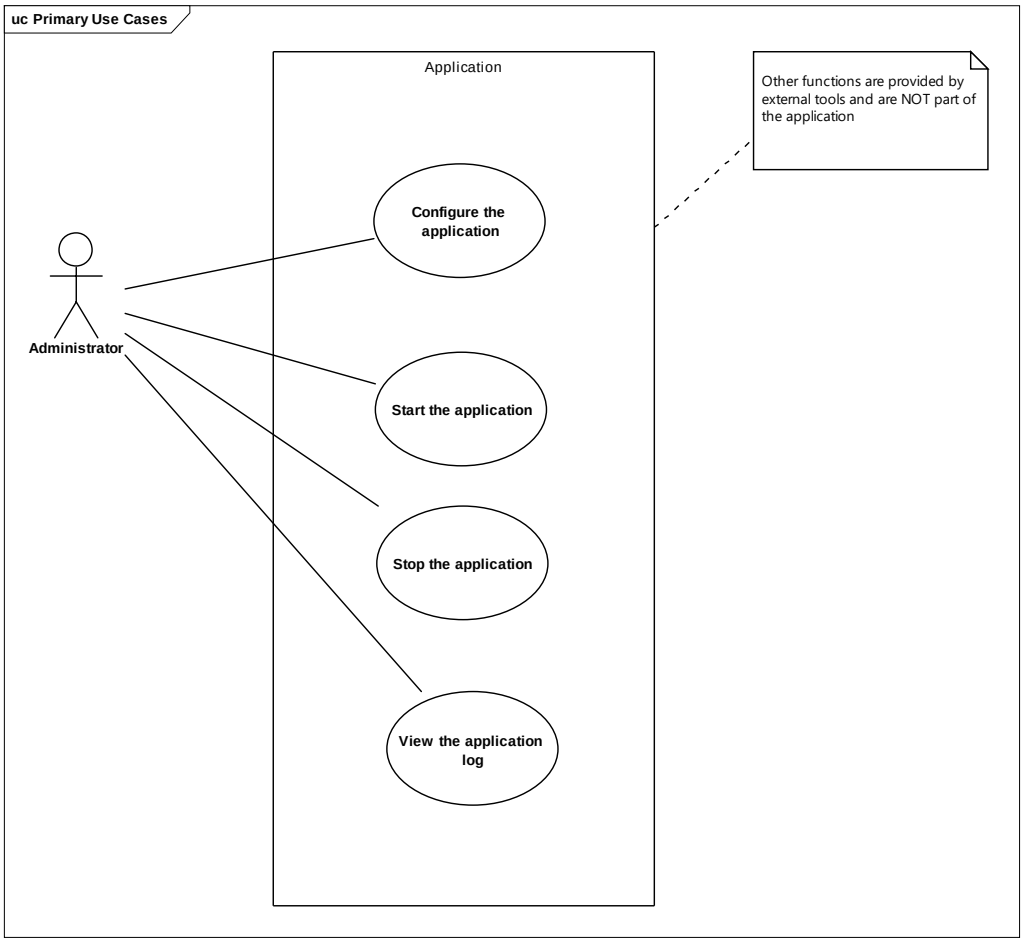
5. System characteristics

- SC-1: The application should be a console one.
- SC-2: The application should be developed using PHP (see L-1 for the explanation; PHP-related details are described in DS-1).
- SC-3: The application should be a multi-platform one (taking into account L-4).

6. User requirements

- See also the use cases diagram below for details.
- UR-1: Start and stop of the application.
    - UR-1.1: The application start should be performed by the following console command: “php converter.phar SOURCE_DIR DESTINATION_DIR [LOG_FILE_NAME]” (see DS-2.1 for parameters description, see DS-2.2, DS-2.3, and DS-2.4 for error messages on any misconfiguration situation).
    - UR-1.2: The application stop (shutdown) should be performed by applying Ctrl+C to the console window, which holds the running application.
- UR-2: Configuration of the application.
    - UR-2.1: The only configuration available is through command line parameters (see DS-2).
    - UR-2.2: Target encoding for text file conversion is UTF8 (see also L-5).
- UR-3: Application log.
    - UR-3.1: The application should output its log both to the console and to a log-file (see DS-4). Log file name should comply with the rules described in DS-2.1.

- UR-3.2: Log contents and format are described in DS-4.1, the application reaction to log file presence/absence is described in DS-4.2 and DS-4.3 accordingly.



7. Business rules

- BR-1: The source directory and the destination directory
  - BR-1.1: The source directory and the destination directory may NOT be the same directory (see also DS-2.1 and DS-3.2).
  - BR-1.2: The destination directory may NOT be inside the source directory or any of its subdirectories (see also DS-2.1 and DS-3.2).

8. Quality attributes

- QA-1: Performance
  - QA-1.1: The application should provide the processing speed of at least 5 MB/sec with the following (or equivalent) hardware: CPU i7, RAM 4 GB, average disc read/write speed 30 MB/sec. See also L-6.
- QA-2: Resilience to input data
  - QA-2.1: See DS-5.1 for the requirements to input file formats.
  - QA-2.2: See DS-5.2 for the requirements to input file size.
  - QA-2.3: See DS-5.3 for the details on application reaction on incorrect input

file format.

9. Limitations

- L-1: The application should be developed using PHP as the Customer is going to support the application with their own IT-department.
- L-2: See DS-1 for PHP version and configuration details.
- L-3: PHP setup and configuration process are out of this project scope and therefore are NOT described in any product/project documentation.
- L-4: Multi-platform capabilities of the application are the next: it should work with Windows and Linux assuming that proper PHP version (see DS-1.1) works there.
- L-5: The target encoding (UTF8) is fixed. There is no option to change it.
- L-6: The QA-1.1 may be violated in case of objective reasons (e.g., system overload, low-performing hardware and so on).

10. Detailed specifications

DS-1: PHP

DS-1.1: Minimal version – 5.5.

DS-1.2: The mbstring extension should be installed and enabled.

DS-2: Command line parameters

DS-2.1: The application receives three command line parameters during the start process:

- SOURCE_DIR – mandatory parameter, points to the directory with files to be processed;
- DESTINATION_DIR – mandatory parameter, points to the directory to store converted files (see also BR-1.1 and BR-1.2);
- LOG_FILE_NAME – optional parameter, points to the log file (if omitted, “converter.log” file should be created in the same directory where “converter.phar” is located);

DS-2.2: If some mandatory command line parameter is omitted, the application should shut down displaying standard usage-message (see DS-3.1).

DS-2.3: If more than three command line parameters are passed to the application, it should ignore any parameter except listed in DS-2.1.

DS-2.4: If the value of any command line parameter is incorrect, the application should shut down displaying standard usage-message (see DS-3.1) and incorrect parameter name, value, and proper error message (see DS-3.2).

DS-3: Messages

DS-3.1: Usage message: “USAGE converter.phar SOURCE_DIR DESTINATION_DIR LOG_FILE_NAME”.

DS-3.2: Error messages:

- Directory not exists or inaccessible.
- Destination dir may not reside within source dir tree.
- Wrong file name or inaccessible path.

DS-4: log

DS-4.1: The log format is the same for the console and the log file: YYYY-MM-DD HH:II:SS operation_name operation_parameters operation_result.

DS-4.2: If the log-file is missing, a new empty one should be created

DS-4.3: If the log file exists, the application should append new records to its tail.

DS-5: File format and size

DS-5.1: The application should process input files in Russian and English languages with the following encodings: WIN1251, CP866, and KOI8R.

Supported file formats (defined by extension) are:

- Plain Text (TXT).
- Hyper Text Markup Language Document (HTML).
- Mark Down Document (MD).

DS-5.2: The application should process files up to 50 MB (inclusive), the application should ignore any file with the size larger than 50 MB.

DS-5.3: If a file with supported format (see DS-5.1) contains format-incompatible data, such data may be damaged during file processing, and this situation should be treated as correct application work.