Practically Relevant Quality Criteria for Requirements Documents

T. Simon, J. Streit, and M. Pizka
itestra GmbH, Ludwigstr. 35, 86916 Kaufering, Germany

Abstract. *This paper presents common weaknesses of requirements documents from commercial software projects that frequently cause problems in practice. Many documents contain extensive, unstructured or even superfluous descriptions of details. At the same time, they lack a thorough description of the aim of the software system or one of its features. Such documents are difficult to read and to work with and they unnecessarily restrict possible solutions.*

We argue that two quality criteria must be considered in addition to common criteria. First, requirements documents should be based on a sound refinement of the main goals to concrete requirements. Secondly, they should follow the principle of minimality, i.e., no more details than necessary.

In order to gain improvements in projects in practice, quality criteria must be communicated to authors. For this purpose, we propose two visualizations, which describe the information structure of the document in a technology-free and domain-independent manner.

Keywords: Software Requirements, Requirements Documentation, Quality Assurance, Refinement, Minimality.

1 Lost in details

1.1 Quality assurance for requirements documents

It is widely recognized that requirements engineering plays a crucial role in a software project and that it is—at the same time—a very difficult task [1, 2]. The quality of requirements documents and their adequacy for further use in the project must therefore be assured and—if necessary—be increased.

Characteristics of a good requirements specification according to IEEE [3] are: correct, unambiguous, complete, consistent, ranked for importance and/or stability, verifiable, modifiable and traceable.

Much effort has been spent in the scientific community on the aspects correctness, consistency and unambiguousness. Since the natural language is inherently ambiguous, formal languages and notations such as SCR ("Software Cost Reduction") for requirements specification have been introduced [4]. The intention behind the formalization is the translation of the informal requirements specification into a form that can be validated and checked for inconsistencies and collisions of requirements. Tools for these tasks are presented for example in [5, 6]. Other research works developed methods for generating tests or even the code implementing the requirements [7, 8].

1.2 Requirements documentation in practice

In commercial software projects, text documents—usually enriched with diagrams or GUI designs—in natural language are still the predominant form of requirements documentation [9]. Authors of such documents often focus on the third aspect, completeness, fearing to forget a requirement or to describe it insufficiently.

We believe that neither correctness nor completeness pose the most urgent problems in practice. We have experienced that engineers who implement a software system frequently struggle with large requirements documents containing extensive, unstructured descriptions of details while it remains unclear how the software is going to be used and why a particular feature is needed. Frequently, the document dictates a solution that is costly to implement and/or less useful than another solution that appears in the course of the project and would fulfill the same requirement.

One of the finalized requirements documents we analyzed was even abandoned at the beginning of the realization, because it was not considered readable any more. The software engineers responsible for the implementation then started again to write down module per module what the software was expected to do.

In this paper, we argue that writing a requirements document that can be used efficiently in the subsequent project phases is one of the most important and most often neglected goals in practice. We present an analysis of weaknesses that hinder efficient usage of requirements documents and define additional criteria for better documents.

Table 1. Use Case describing trivialities

Use Case: Editing a data item		
1.	The actor chooses one data item from the list.	The system selects the item.
2.	The actor chooses "Edit item".	The system opens a dialog for editing the item.
2a.	The actor enters values and confirms.	The system stores the entered values.
2b.	The actor enters values but does not confirm.	The system asks back whether the changes shall be abandoned.
2ba.	The actor confirms and thus approves storage of the changes.	The system stores the entered values.
2bb.	The actor negates and thus disapproves storage of the changes.	The system discards the changes and does not store the entered values.

Finally we introduce two visualizations of the information structure in a requirements document that make it easier to explain the quality criteria and possible weaknesses to authors.

2 Typical weaknesses of requirements documents

In the following we present weaknesses of requirements documents we have frequently encountered in commercial projects. The examples are taken from an in-depth analysis of information system requirements documents from six projects, involving four companies from three different domains, and altogether comprising approximately 1.500 pages. For confidentiality reasons the examples were slightly modified; their core however remains the same. Note that some of the described weaknesses are related, so a sharp distinction is not in all cases possible.

Trivialities. Almost all documents contain lengthy descriptions of commonly known issues.

In table 1, a Use Case scenario description taken from one of the analyzed documents can be seen that mainly describes how editing a data item in the system can be aborted. This is an example of a common feature which in most cases does not have to be specified further in detail. Therefore the lengthy description could be easily summarized with one sentence "The scenario can be aborted by the user at any time. The system asks for confirmation".

Trivialities may appear just as a waste of effort but not really as harmful. However, they artificially inflate the size of requirements documents. Lengthy documents are not only harder to read and to maintain, but are also susceptible to inconsistencies and errors, as it is more difficult to retain the overview. It thus becomes more likely that information added to the document conflicts with already existing information.

Information out of scope. We repeatedly identified sections of requirements documents containing information that does not add any value to the description of the system to be build.

In one case, the document contained the following statement:
Department X is responsible for entering the data into the system.
This information is superfluous. The business process had already been described based on roles. The assignment of responsibilities to departments should not be part of a software requirements document. Even worse, the sentence triggered a discussion between the departments about the responsibilities, as they feared that such a statement in an approved document could be used to shift responsibilities from one department to the other.

Information out of scope also unnecessarily increases the size of the document, and we have seen in the previous section that this is to be considered problematic. Additionally, such information may lead to irritations and unnecessary struggles.

Thinking in solutions. Another frequent defect is the description of solutions or products where a description of the problem to be solved would be appropriate.

The following excerpt represents this flaw:
The user shall be notified when another user attempts to edit the same data. The server must inform the client software using a push-mechanism.
The second sentence clearly describes an implementation decision. What the author actually wanted to express is that no user interaction shall be needed for getting the notification. Using a push-mechanism in the client-server communication is one but not the only solution, and the choice should normally be made during software design (unless there is another reason to prefer one architecture, for example an existing software landscape).

Describing a solution obscures the actual problem to be solved to the reader. This is problematic if the solution turns out to be difficult, costly or impossible to implement, or if there are better solutions the author had not thought of. Provided with a description of the problem, a software engineer who reads the requirements document can propose alternatives. Without such a description, he or she can only guess what the intentions of the author were. Needlessly complex, clumsy and expensive software is the result of this weakness.

Pinpointing details. Often, authors pinpoint details that unnecessarily complicate the implementation, i.e., an equally well or better suited solution would be possible at equal or lower cost. This weakness is related to "Thinking in solutions", however, its root is not the missing problem description but just a needless detailing.

An example that represents this common flaw is hidden in the following description of a data field taken from on of the analyzed documents:
There are 5 different values to choose from.
A closer investigation revealed that only two values were currently known and in use. The other three were planned as placeholders for later extensions. The restriction to five values is however completely unnecessary, as this field would probably be implemented as an enumeration of n values anyway, and adding three new values entails the same cost as adding ten of them. Furthermore, additional code is needed to limit the number of entries to five. Instead, it would have been better to state that two values were currently known and that an extension should be possible. This way, a solution that increases the extensibility of the system and decreases its development cost at the same time could have been implemented.

Lacking rationale. Many requirements documents fail to describe what shall be achieved with the software or one of its particular features. Note that in contrast to the "Thinking in solutions" defect, the concrete requirements in these documents may be described at an appropriate level of detail.

The following example extracted from one of the analyzed requirements documents represents this frequent weakness:
The content of the database table SHARES_AMOUNT has to be exported to a Microsoft Excel spread sheet.
The context and the rationale explaining why spread sheets at all or why spread sheets of this specific vendor should be used was lacking. The developers later faced the problem that Microsoft Excel spread sheets can only contain up to 65536 rows and the number of rows in the database table exceeded this number. Because the document also did not mention what the resulting spread sheet is used for, the developers were not able to implement an appropriate solution based on the available information. Due to a scheduled meeting with the stakeholders it turned out that the spread sheet was determined only to be printed and handed to the analysts as a report. If this information had been contained in the requirements document, they would have been able to develop an alternative solution, such as a PDF export, providing the same or an even better result.

Lacking rationale in the requirements documentation is problematic for several reasons. First, it usually makes the document more difficult to understand for persons not familiar with the subject. Secondly, it makes it much harder for a software engineer to find alternative solutions when a particular requirement turns out to be costly, difficult or impossible to implement.

3 Practically relevant quality criteria for requirements documents

The weaknesses presented in the last section are not captured sufficiently by common quality criteria for requirements documents and they are usually not in the authors' focus, consequently causing severe problems in later project phases.

We therefore argue that additional criteria have to be considered for judging and improving the quality of requirements documents, namely adherence to a refinement relation between requirements and minimality.

3.1 Root-based Refinement

The "Lacking rationale", "Thinking in solutions", and also the "Pinpointing details" weaknesses have in common that the requirements document fails to explain *why* a certain requirement is demanded. This *why*—the rationale—helps readers to get an understanding on what the software is meant for, as explained for the weakness "Lacking rationale", and to maintain the overview more easily. Also, it allows software engineers to propose alternative solutions in line with the main goals when obstacles appear during further analysis and implementation.

We therefore demand to include the rationale for all requirements in the document. The most abstract rationale is the goal for which the software system is being developed. Any concrete requirement to the software system must in the end serve this overall goal, and hence should be an iterative refinement of it, i.e., adhere to a refinement relation of requirements. Refinements restrict the set of possible implementations: for example, while the requirement "The system shall allow users of company X to compose emails." is fulfilled by any email software, the refinement "Users want to compose emails while traveling." rules out a web-based solution.

The refinement relation should be included entirely (i.e., with the main goal as root) in the document, and it should be traceable, e.g., by cross references, links or an adequate document structure. Besides improving the understandability of the document, the goal description and its refinements also help the authors to stay focused: sections that describe solutions instead of problems can be identified, as their refinement relation will not be sound from the abstract goal to a concrete requirement.

3.2 Minimality

A second issue that several of the above weaknesses have in common is that the requirements document contains too much information. Requirements documents need to specify what is required for developing the software as requested by the stakeholders[1]. In particular, they should not leave for implementations that do not fulfill the stakeholders' demands. However, any additional information possibly impedes the development of the software, as it makes the document longer and may rule out possible (and potentially more efficient) implementations, as explained for the "Trivialities", "Information out of scope" and "Pinpointing details" weaknesses.

We therefore affirm that the minimality of a requirements document—in particular the avoidance of superfluous details—is an important quality criterion, which is, unfortunately, often neglected.

Demanding minimality and the inclusion of the requirements rationale simultaneously may seem contradictory. However, there are good reasons for including the rationale in the document, as we have explained above. Therefore this information is not to be considered superfluous, while needlessly detailed descriptions and information out of scope are. Furthermore, the refinement relation described above can be helpful for obtaining minimality: if the information on a certain level of detail describes an aspect accurately enough to allow no wrong systems to fulfill the specification, it is unnecessary to refine it further. In other words, if the question "Does the goal description already express the needs of the stakeholders sufficiently?" can be positively answered, there is no need for a more detailed description.

3.3 Relation to other quality criteria

The proposed quality criteria do not only lead to a requirements document that can be efficiently used in the development process but also have a positive impact on the characteristics of good requirements specification according to IEEE [3].

With the overall goal and its refined goals included in the document, it is more likely that incorrect requirements will be detected in an early phase of the project, as they might be in contradiction with one of the goals and thereby visible to an attentive reader. The analysis of the rationale using "why"-questions also helps to identify requirements that were missed out and thus to increase completeness, as explained in [10].

[1]In this paper, we do not distinguish different groups of stakeholders with potentially differing requirements.

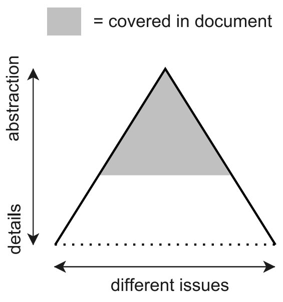

Figure 1. The Requirements Information Pyramid (RIP)

Furthermore, the modifiability of each requirement is improved, as the rationale leading to the requirement is contained in the same document and can be used to determine the impact of the change.

4 Information structure visualizations

The authors of requirements documents are usually unaware of the shortcomings described above and their consequences. It is thus crucial to not only define quality criteria for requirements documents, but to also develop ways of communicating them to authors in practice.

In the following, we introduce a graphical representation for this purpose. The criteria will be illustrated through visualizations of requirements documents and the systems they specify, the *Requirements Information Pyramid* (RIP) and the *Space of Solutions* (SoS).

4.1 RIP and SoS

The *Requirements Information Pyramid* shown in figure 1 visualizes the refinement structure of the information in a requirements document. Every piece of information in the document is represented by a gray dot. Hence, the gray area is a representation of the information that is actually contained in a document. If the dot is located between the black boundaries the information specifies a characteristic of the system that is demanded by the stakeholders. A gray area outside the pyramid represents unnecessary information that does not influence the implementation of the system and that therefore does not add any value to the description.

The pyramid shape results from the hierarchical refinement of the information in the requirements document. The peak of the pyramid represents the overall goal of the system to be built (level of detail 1). Every further gray area is a more refined and detailed description of the information above it: the middle region states general intentions

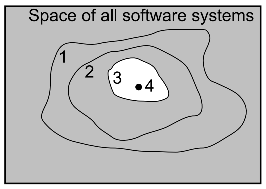

Figure 2. The Space of Solutions (SoS)

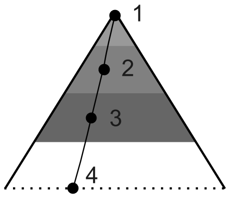

Figure 3. Different levels of detail

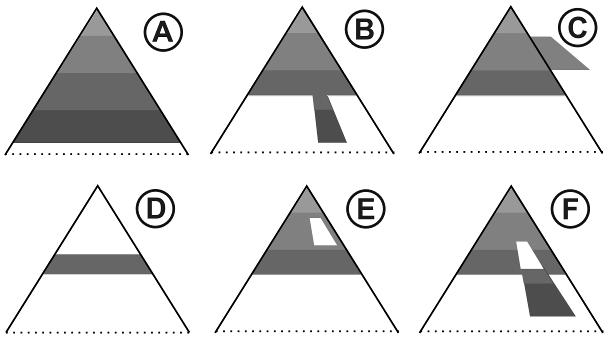

Figure 4. Weaknesses in documents

and requirements to the system, the lower regions stand for more detailed information on the same issues. We consider the implemented source code (together with additional information about the deployment and the infrastructure the system is running in) as the most detailed description of a system. This information can be found on the very bottom of the pyramid, and is usually not contained in the gray area (i.e., the document) any more.

Figure 2 shows the visualization *Space of Solutions*. The rectangle represents the space of all possible software systems. Every dot in the rectangle stands for a specific implementation. The areas surrounded by a line enclose sub sets that fulfill the requirements in a document at a certain level of detail. SoS and RIP provide two different perspectives on the same requirements document, as can be seen in figures 2 and 3 for four levels of detail of the requirements documentation (denoted with the numbers 1 to 4). In the first phase only the overall goal is specified and thus the set of possible solutions is only roughly delimited. In the further phases more detailed information about the requirements is added to the document. As a result, the gray area in the RIP becomes bigger and expands to the lower part of the pyramid. At the same time, the set of possible solutions becomes smaller and excludes those solutions that are in conflict with requirement details. When the requirements documentation is completed, only a few solutions remain in the set of solutions (level of detail 3) Finally, an implementation of the software would be represented as a completely filled RIP and a single dot in the SoS (level of detail 4).

4.2 Visualizing weaknesses

In the following, we are going to apply the visualizations to the weaknesses listed in section 2 and provide a RIP and/or SoS for each of the weaknesses. The visualizations represent the information structure of the requirements document in a technology-free and domain-independent manner. Thus, they can be used as the basis for discussions between the stakeholders, domain experts and software engineers about the quality of the document.

Trivialities / Pinpointing details. A document that contains trivialities is visualized in drawing (A) in figure 4. It refines the information contained in the document very deeply. A document in which a single requirement is described on a very fine-grained level of detail is shown in drawing (B). It is obvious that the larger the gray area is the more difficult it is to maintain the overview and to modify the document.

Still, these drawings may represent appropriate requirements documents if the most detailed information is specifically demanded by the stakeholders.

However, if the details are not actual requirements of the stakeholders, they unnecessarily restrict the possible implementations. Figure 5 shows an SoS where—due to overspecification—the set of implementations allowed by the document (white) is much smaller than the set of implementations that would satisfy the needs of the stakeholders (dotted border). Hence, the requirements document unnecessarily excludes a large number of implementations (dark stripes), some of which may be cheaper to implement or provide greater flexibility or usability than the solutions specified by the document.

Information out of scope. A representation of a document that contains information beyond the scope initially defined for the document can be seen in drawing (C). This information is a superfluous addition to the requirements document that only lengthens it without adding value for the implementation of the software.

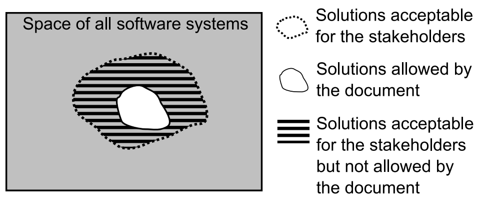
Space of all software systems
Solutions acceptable for the stakeholders
Solutions allowed by the document
Solutions acceptable for the stakeholders but not allowed by the document

Figure 5. Constraining the solution too much

Lacking rationale / Thinking in solutions. If the rationale for a requirement is partially or completely missing, the path from the top of the pyramid to the information item specifying the solution is broken. This situation is shown in drawing (D)—where the overall goal is missing—and drawing (E)—where an intermediate refinement step is missing. If authors write down solutions instead of requirements they usually skipped some levels of detail and finally ended on a very low level, as in (F).

5 Related Work

The definition and refinement of goals has been identified as an essential part of requirements engineering [10]. While these goal-oriented approaches provide methods for requirements elicitation as well as formal and semi-formal models for requirements specification, we focus on the information structure within a (usually textual) requirements document. We highlight that the root of the refinement must be the overall business goal, and that all refinement steps must be included in the document.

The refinement of goals to detailed descriptions of requirements is related to the traceability of requirements [11, 12]. These articles investigate the ability to follow the lifecycle of a requirement, all the way from the requirements description to a specific code line. Our concept does not link a requirements descriptions with a code fragment but with more abstract information about the requirement within the requirements document.

Cockburn [13] distinguishes differently "colored" use cases for different levels of abstraction. White use cases describe overall goals, blue use cases describe user goals etc. Cockburn emphasizes that all levels are necessary in order to get a complete description. The RIP can thus be seen as a generalization of Cockburn's color system.

We pinpoint minimality as a crucial characteristic in requirements documentation and see it as a main success factor for the efficient usage of the requirements documents in the further development process. In literature, this issue is mentioned in [10, 14], but not at all addressed in the IEEE recommendations [3] or [15]. [16] describes the harmful effects of specifying superfluous constraints.

6 Conclusion

This paper demonstrates that a requirements document may, although providing a complete and consistent specification of a software, contain flaws that render it nearly unusable in subsequent project phases. Issues such as the sheer size of a document, the restrictions it imposes on the possible implementations and its readability are widely ignored in scientific research, although they pose serious threats to a project in practice. We have argued that following a *root-based refinement* approach and keeping a focus on *minimality* improves the quality of requirements documents, which is a prerequisite for their helpful use. As we have experienced in our daily work, the visualizations RIP and SoS with their technology-free nature serve as a simple and effective basis for discussions between stakeholders, domain experts and software engineers with the aim to identify, explain and resolve weaknesses in requirements documents.

References

[1] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on the Future of Software Engineering*, pages 35–46, New York, NY, USA, 2000. ACM Press.

[2] Betty H. C. Cheng and Joanne M. Atlee. Research directions in requirements engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 285–303, Washington, DC, USA, 2007. IEEE Computer Society.

[3] IEEE. Recommended practice for software requirements specifications. Standard IEEE Std 830-1993, Institute of Electrical and Electronic Engineers, 1993.

[4] Constance L. Heitmeyer. Software cost reduction. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, 2002.

[5] Vincenzo Gervasi and Didar Zowghi. Reasoning about inconsistencies in natural language requirements. In *ACM Transactions on Software Engineering and Methodology*, volume 14, pages 277–330, July 2005.

[6] C.L. Heitmeyer and R.D. Jeffords. Applying a formal requirements method to three NASA systems: Lessons learned. In *Aerospace Conference, IEEE*, pages 1–10, March 2007.

[7] Michael G. Hinchey, James L. Rash, and Christopher A. Rouff. Requirements to design to code: Towards a fully formal approach to automatic code generation. Technical Report TM-2005-212774, NASA Goddard Space Flight Center, USA, January 2005.

[8] Tom Rothamel, Yanhong A. Liu, Constance L. Heitmeyer, and Elizabeth I. Leonard. Generating optimized code from SCR specifications. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on language, compilers, and tool support for embedded systems*, pages 135–144, New York, NY, USA, 2006. ACM.

[9] Luisa Mich, Mariangela Franch, and Pierluigi Inverardi. Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9(1):40–56, 2004.

[10] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of 5th IEEE International Symposium on Requirements Engineering, Toronto*, pages 249–263, August 2001.

[11] Orlena C.Z. Gotel and Anthony C.W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First IEEE International Conference on Requirements Engineering, Colorado springs*, pages 94–101, April 1994.

[12] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, 2001.

[13] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[14] Donald G. Firesmith. Specifying good requirements. *Journal of Object Technology*, 2:77–87, July 2003.

[15] Stuart R. Faulk. *Software Requirements Engineering*, chapter Software Requirements: A Tutorial. IEEE Computer Society press, 2nd edition, 1997.

[16] Donald G. Firesmith. Common requirements problems, their negative consequences, and industry best practices to help solve them. *Journal of Object Technology*, 6(1):17–33, 2007.