Requirements Eng (2006) 11: 42–57
DOI 10.1007/s00766-005-0018-1

ORIGINAL ARTICLE

Artem Katasonov · Markku Sakkinen

Requirements quality control: a unifying framework

Received: 15 November 2004 / Accepted: 9 September 2005 / Published online: 7 October 2005

Abstract Literature tends to discuss software (and system) requirements quality control, which includes validation and verification, as a heterogeneous process using a great variety of relatively independent techniques. Also, process-oriented thinking prevails. In this paper, we attempt to promote the point that this important activity must be studied as a coherent entity. It cannot be seen as a rather mechanical process of checking documents either. Validation, especially, is more an issue of communicating requirements, as constructed by the analysts, back to the stakeholders whose goals those requirements are supposed to meet, and to all those other stakeholders, with whose goals those requirements may conflict. The main problem, therefore, is that of achieving a sufficient level of understanding of the stated requirements by a particular stakeholder, which may be hindered by, for example, lack of technical expertise. In this paper, we develop a unifying framework for requirements quality control. We reorganize the existing knowledge around the issue of communicating requirements to all the different stakeholders, instead of just focusing on some techniques and processes. We hope that this framework could clarify thinking in the area, and make future research a little more focused.

1 Introduction

It is commonplace that errors in a software or a system development project are costlier, the earlier they are committed and the later they are detected. Glass states as one of his 55 facts: Requirements errors are the most expensive to fix when found during production but the cheapest to fix early in development [1].

Boehm [2] claimed that savings up to 100:1 are possible by finding and fixing requirements problems early rather than late in the life-cycle, and that besides the savings there are also significant payoffs in improved reliability, maintainability, and human engineering of the resulting software product.

Rigorous *Requirements Engineering (RE)* [3] is therefore considered to be the cornerstone for efficient development of quality software and systems. It is acknowledged that the requirements should be explicitly elicited, negotiated and documented, and then followed through in design and implementation.

Obviously, the quality of the developed software relies on the quality of the requirements themselves. Therefore, requirements quality control is recognized as a necessary activity in RE. As quality control of any work product, requirements quality control includes two parts: *validation* and *verification*. Although these are conceptually different activities, in practice they are almost inseparable. For this reason, also for the sake of brevity, and also because requirements quality control as such mainly belongs to the validation side of the quality control process for the software system being developed, most RE literature sources use the word "validation" to denote both validation and verification, i.e., the whole requirements quality control process. In Sect. 2 we will clarify the distinction between validation and verification, and comment on the consequences of clubbing the two.

Requirements quality control is in the intersection of two established research areas: RE and Software V&V (verification and validation). In the latter, mainly software testing is studied, but quality control in earlier phases of the development is treated as well. Unfortunately, as the literature review reveals, requirements quality control seems to be somewhat marginal for both areas. Some books on RE contain a single chapter dedicated to it, but others only mention it occasionally. In books on software V&V, it is rare to find a whole chapter about requirements quality.

A. Katasonov (✉) · M. Sakkinen
University of Jyväskylä, Jyväskylä, Finland
E-mail: artem.katasonov@titu.jyu.fi
E-mail: sakkinen@cs.jyu.fi

Moreover, literature tends to discuss software (and system) requirements quality control as a heterogeneous process using a great variety of relatively independent techniques, rather than treating it as a coherent activity. Most books present some set of ''good practices'', whose application may contribute to the quality of requirements and to one's confidence in this quality. Good practices for requirements validation and analysis listed by Wiegers [4] include the following: inspect requirements documents, write test cases from the requirements, create user interface prototypes, model the requirements with various diagrams, and analyze requirements' feasibility. The first edition of that book [5] included also the practice of drafting a user manual. Hetzel [6] names three possible ways of testing software requirements: through understanding them, through test-case design, and through prototypes. Bashir and Goel [7] list the following ''typical activities in requirements testing'': comprehend requirements, mental test requirements, define use cases, create test cases, test completeness, test consistency, create or update prototype, and solicit feedback from users. Validation techniques discussed by Kotonya and Sommerville [8] are: requirements reviews, prototyping, validation of formal models, and designing test cases. Each of these books, explicitly or implicitly, recommends the application of all (or most) of its listed good practices in a development project. Also, none of these authors discusses the way that different good practices in their lists might be connected to each other.

It is also very rare to see a journal or conference paper devoted to requirements quality control and discussing it as a whole. Recent research articles as a rule discuss one of the known good practices, or even some details of it. Some older articles can be found, such as those by Boehm [2, 9], but they belong to the same tradition as the books above. Boehm lists many different practices for requirements validation classifying them into simple manual techniques (reading, interviews, checklists, simple scenarios, manual cross-referencing, manual models), simple automated techniques (automated cross-referencing, simple automated models), detailed manual techniques (detailed scenarios, mathematical proofs), and detailed automated techniques (detailed automated models, prototypes). It is notable that this list and an earlier paper by Boehm [10] show that 30 or 20 years ago people tended to be more optimistic about the possibilities of automation in requirements quality control than today.

We believe that both phenomena—the limited attention to requirements quality control and not treating it as a coherent entity—are consequences of the two key assumptions that frame traditional RE [11]. The first one is that requirements exist 'out there' in the minds of stakeholders (users, customers, clients), and that they only need to be elicited through various mechanisms. The second is that the key stakeholders operate in a state of goal congruence, in which there is widespread agreement on the general goals of the system under development. Under these assumptions, even validation of requirements is nothing more than checking whether the analysts have understood the stakeholders' intention correctly and have not introduced any errors when writing the specification. Accordingly, Kotonya and Sommerville [8] define requirements validation quite narrowly as checking the requirements documents for consistency, completeness and accuracy.

At present, it is increasingly acknowledged that the above assumptions seldom hold true (e.g., [11]). Requirements are not discovered but constructed, and there is usually at least some goal incongruence between stakeholders (see Sect. 2). If one agrees with that, one must also realize that requirements validation has a wider meaning than that explained above. It is not a mechanical process of checking documents. It is more an issue of communicating requirements, as constructed by the analysts, back to the stakeholders whose goals those requirements are supposed to meet, and to all those other stakeholders, with whose goals those requirements may conflict.

In this article, we develop a general framework for requirements quality control. Our main goal is to promote the point that it must be studied as a coherent entity. Also, the focus of attention must be moved from the process to human communication issues, i.e., communication between analysts and stakeholders. From a practical perspective, we do not develop much new prescriptive knowledge, but rather reorganize the existing body, in our opinion, in a more sensible way. Instead of focusing on some techniques and processes, we focus on the issue of achieving a sufficient level of understanding of the stated requirements by every stakeholder. We hope that this framework could clarify thinking in the area, and make future research a little more focused.

2 The role of requirements validation

The first of the two key assumptions that frame traditional RE is that requirements exist 'out there' in the minds of stakeholders and need only to be elicited from there [11]. However, this assumption is generally false.

Fig. 1 depicts the model of Michael Jackson [12, 13]. When developing a system (*machine*), one needs to answer three questions, one after another:

1. What is the effect on the environment that is desired?
2. What would the behavior of the machine on the boundary (interface) be that would achieve the desired effect on the environment?
3. What would the internal behavior and structure of the machine be that would lead to the needed behavior on the boundary?

When answering the first question, one defines the *outer requirements* (in terms of [13]) for the system being developed. When answering the second question, one defines the *inner requirements* (which are usually meant

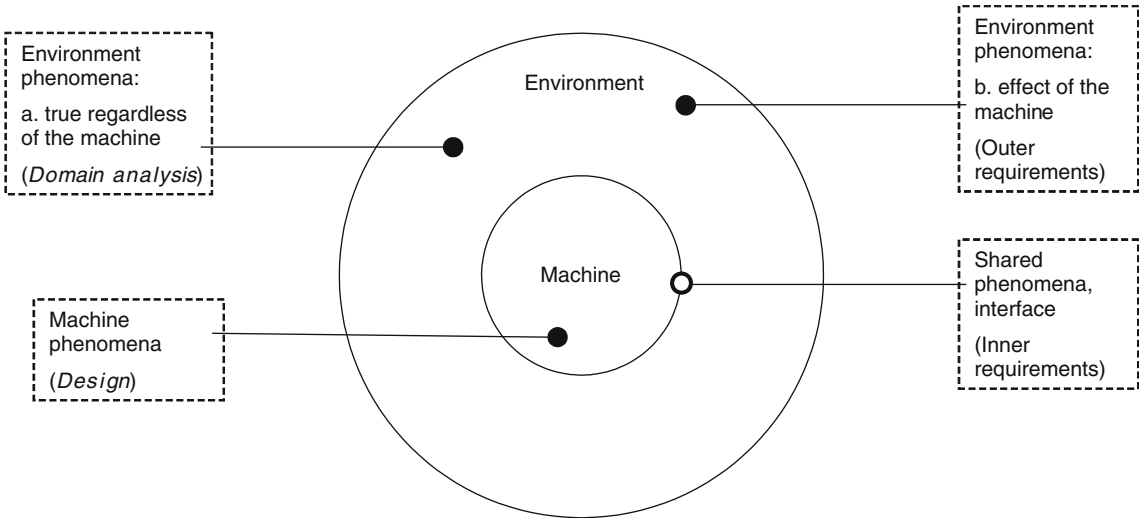

Fig. 1 Environment, machine, and boundary

when one speaks of "requirements"), and giving an answer to the third one means designing the machine.

As can be seen, there are two problem-solution iterations. The outer requirements pose a problem and the inner requirements are proposed as satisfying them. In turn, the inner requirements pose the next-level problem to be addressed by the design. Requirements engineering is mainly concerned with specifying the behavior of the machine, as visible to the stakeholders, i.e., with defining the inner requirements, and therefore should be seen rather as a problem-solving process, not just a problem-stating one.

The focus of attention of software end-users and customers is on the outer requirements. One practical implication of this fact is that when a customer says "I want a software system that does that and that ...", he means actually "I want a system that would have such an effect on me and my organization. And I think that such an effect would be achieved with a system doing that and that ...". In other words, when a customer is "stating his requirements", he is actually just *proposing* an answer to question 2 in the list given above.

To be able to answer that question in an effective and feasible way, one obviously needs to possess not only domain knowledge but also technical expertise. Software end-users and customers are seldom experts in computers and software. Therefore, the customer's proposed answer is seldom complete, and the requirements analysts need to elaborate it further. Also, the customer's original answer is seldom the best, and the analysts should be critical about it (the customer is not always right) and expect it to change as the customer's understanding evolves.

Even while it may also not be absolutely true, one can assume that a project's customers will be able to state properly their outer requirements, i.e., the effects they expect. However, it is not wise to assume that they will be able to resolve those outer requirements themselves and specify what exactly the machine needs to do in order to cause the desired effects. Therefore, the inner requirements cannot be really discovered from the stakeholders, but need to be actively constructed by the requirements analysts.

For the present discussion, the most important consequence of this fact is that there is an obvious need for an information flow about requirements being constructed from the analysts to the stakeholders. Maintaining this information flow is the core of the requirements validation activity, the same way as maintaining the flow from the stakeholders to the analysts is the core of the elicitation activity. The goal is to give the stakeholders a chance to check early whether the solution proposed will really solve their problem. In fact, this is easier than to develop a solution oneself and requires less technical expertise. However, some technical expertise is still required just to understand the requirements as stated. This makes requirements validation far from being a trivial task.

With the discussion above, we attempt to justify requirements validation as a very important part of RE. We also attempt to justify validation as a separable part of RE, even while practical means used in the validation may overlap, to a large extent, with the means used in the other RE activities. While elicitation and analysis are an attempt to cross the boundary from the domain world to the machine world, validation is an attempt to cross the same boundary but in the opposite direction, in order to create a feedback cycle.

Actually, some technical expertise is required already for knowing what environmental effects can, in principle, be caused by computers and software. Therefore, customers are not always able to state even their outer requirements satisfactorily right from the beginning of a project. Existence of the validation feedback link

stimulates the evolution of customers' understanding and therefore works also as a catalyst of the elicitation process.

All the above holds true even in the simplest project with a single customer. In more complex projects, with many stakeholders, the situation is more complicated. The second of the two key assumptions that frame traditional RE is that the key stakeholders operate in a state of goal congruence, in which there is a widespread agreement on the general goals of the system under development [11]. However, in practice there is usually at least some goal incongruence between stakeholders. This increases the importance of requirements validation even more, because requirements, as constructed, need to be communicated not only to the stakeholders whose goals those requirements are supposed to meet, but also to all other stakeholders with whose goals those requirements may conflict. This makes validation more difficult. Since different stakeholders are likely to have different backgrounds, in addition to lacking technical expertise, validation needs to overcome any potential lack of some special domain knowledge. As a result, validation becomes a communication issue even more, rather than a process-to-follow issue.

We need to clarify also the difference between validation and verification of requirements. The concept of validation is often seen to answer the question "Am I building the right product?". It is therefore checking a work product (the final software or the output of a given phase of the development cycle) against higher-level work products or authorities that frame *this particular* product. In contrast, verification is seen to answer the question "Am I building the product right?". It is therefore checking a work product against some standards and conditions imposed on this *type* of product and the *process* of its development.

When considering a requirements specification as a work product, we can say that only the system stakeholders can validate the requirements. However, requirements verification can be performed more or less by the requirements analysts themselves. Validation requires the domain and the problem knowledge, but for verification one needs knowledge of RE, logic, language, etc. The quality of requirements is usually measured with a set of quality attributes (see Sect 4.1.2). Whether one speaks of verification or validation depends actually on which of the attributes are in question, as is shown in Fig. 2.

Completeness, consistency and unambiguousness are somewhere in the middle. This does not mean that it is impossible to say whether we are validating or verifying those attributes. It means that, generally, those attributes have to be validated by the stakeholders; however, also some verification is possible. Some omissions, inconsistencies, and ambiguous statements can often be identified without any knowledge of the specific domain or the problem. It can be done ad hoc or with the help of some simple verification techniques. An example is CRUDL (Create-Read-Update-Delete-List) [4] for identifying missing requirements. That technique mechanically checks whether each data entity has at least one use case in which it is created, at least one in which it is used, and at least one in which it is deleted.

While requirements validation and verification are conceptually different activities, in practice they are almost inseparable. When a person, for example, reviews a requirements document, he uses all his knowledge simultaneously, both the domain and the problem knowledge needed for validation, and the knowledge needed for verification, i.e., knowledge of RE, logic, language, etc.

This inseparability seems to be one more reason for limited attention to validation as the issue of communicating the requirements back to stakeholders. It is because it is shaded by the more mechanical and process-oriented nature of verification.

3 The unifying framework

As we discussed in Sect 2, the central problem in requirements validation, and therefore in requirements quality control in general, in our opinion is communicating requirements, as constructed, back to the system stakeholders. It is a problem because the customers and end-users are likely to lack technical expertise. Also, stakeholders are likely to have different backgrounds and as a result lack each other's special domain knowledge. While elicitation and analysis are attempts to cross the boundary from the domain world to the machine world, validation is an attempt to cross the same boundary in the opposite direction. The difficulty of the former is well acknowledged, and the latter can be almost as difficult.

The goal of the research reported in this article is not to come up with an ultimate solution to this problem. We attempt rather to lay the foundation for that by reorganizing the existing body of prescriptive knowledge in a way more appropriate for the task. As we already discussed, the present RE literature treats requirements quality control neither as an issue of communication, nor as a coherent entity. Also, validation is somewhat shaded by the more mechanical and process-oriented nature of verification.

Most RE literature sources propose, for requirements quality control, some unstructured sets of "good practices" of various kinds. By generalizing about the nature of those practices, we can, however, distil several relevant basic ideas. Given that the goal is to facilitate a human's understanding of some stated requirements for a system (i.e., understanding of a proposed machine's behavior and effects in the environment that the behavior will cause) and so give him a chance to validate (and verify) them, one could:

- Present the requirements in a form that can be better understood by this specific person.
- Give him an explicit task to accomplish. Based on the "good practices" available we can identify two

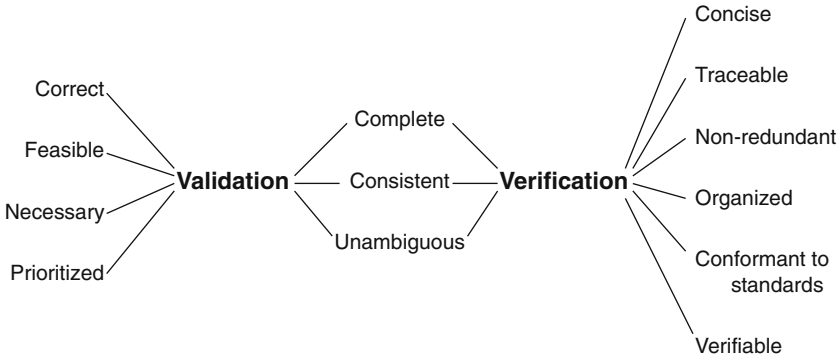

Fig. 2 Requirements validation and verification

possible tasks: looking for defects, i.e., review, and translating requirements from one representation form to another.
- Split the review problem into several smaller subproblems, by defining a set of attributes of the requirements quality to consider.
- Split the review problem even more, by describing some smaller steps to accomplish. Here, different approaches are possible leading to what is often called *reading techniques*. The present literature does not seem to propose much beyond that. Based on these basic ideas, we developed a general framework for requirements quality control. This framework covers all the good practices we ever encountered in literature. It also covers both validation and verification.

The framework is depicted in Figs. 3 through 5. Fig. 3 presents the structural view of the framework, while Fig. 5 shows the procedural view of it, i.e., the set of steps that need to be performed. Fig. 4 presents the taxonomy of the framework, listing the set of possibilities that the present literature provides for each of the structural elements of the framework.

As can be seen, visualizations, test cases, prototypes, etc., are interpreted as different representation forms, to which requirements are translated and then, possibly, reviewed. Scenario-based validation, which is popular in research literature, is interpreted as a review based on using scenario-based reading, as contrasted to ad hoc, checklist, etc. reading.

In the center of the framework, there is the *review* task. This is not a single event. A set of reviews is to be conducted, some consecutively and some in parallel. For each single review, some subsets from the four pools need to be selected:

1. *Stakeholders* A stakeholder is a person, group, or organization that is involved in the system development, affected by the system, or can affect the system [4]. This covers the development team, including the requirements analysts. This pool will be discussed in Sect 4.1.3.
2. *Representation forms* Requirements can be represented in a variety of different forms. However, besides a natural language description, each form can only provide a partial view of the requirements for a system. This pool will be studied in Sect. 5.
3. *Requirements quality criteria* These criteria specify what is to be considered as requirements defects. This pool will be discussed in Sect 4.1
4. *Reading technique* There are different techniques to be used during a review, mainly related to the type of guidance the reviewers receive for accomplishing their task. This pool will be studied in Sect 4.3.

Thus, in our framework, a requirements review is a process in which a subset of the system stakeholders investigates the requirements represented in one or more of the available forms, using one or more of the available reading techniques, based on a subset of the quality

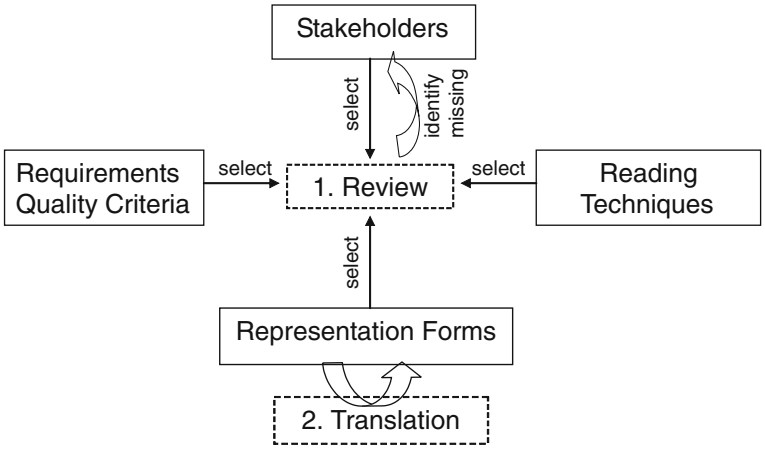

Fig. 3 The structural view of the framework

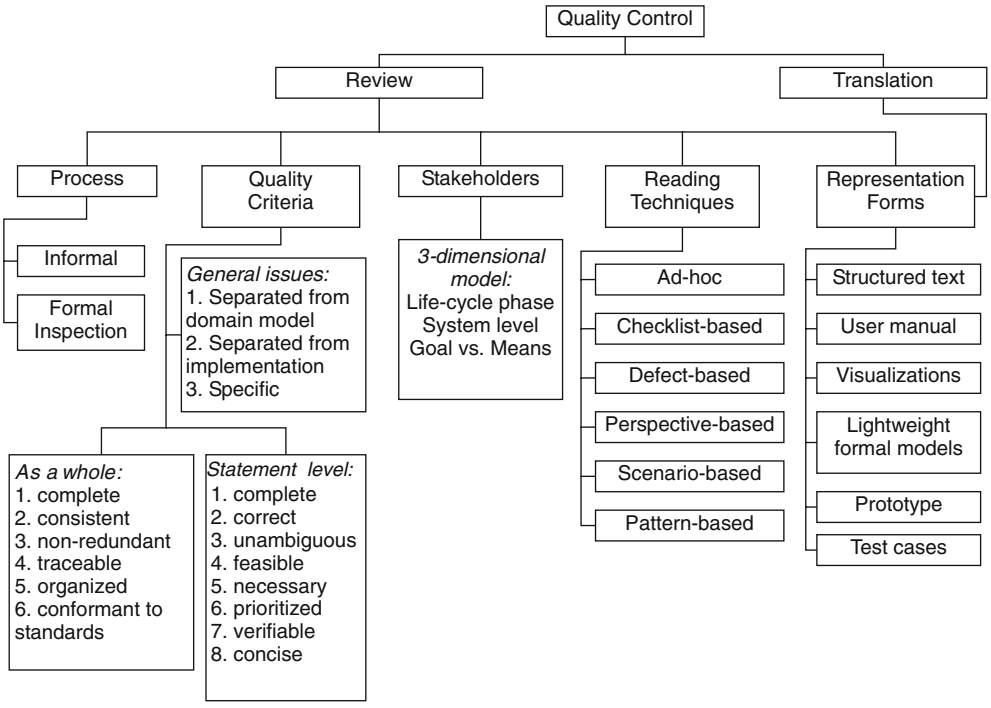

Fig. 4 The taxonomy of the framework

criteria defined. These four determinants of the review process will be studied in the rest of the paper, in sections referred to above. Also, the process itself will be briefly discussed in Sect. 4.2.

Stakeholders are the ones whose goals the requirements are supposed to meet, and therefore, the ones who should conduct the reviews. In practice, however, surrogates for some stakeholders often need to be used, and can be either representatives or experts. For example, a project may need to hire an attorney to be a surrogate of the stakeholder ''government'', i.e., to validate that the system's behavior will comply with existing laws and regulations.

Also, if the requirements document or some part of it is in a sufficiently formal language, some quality properties can be automatically verified by software. In this paper, we will not discuss much such automated checking. One reason for this is that only a small part of requirements quality control can be automated, even in theory, and only in areas belonging to the verification side. Checking for inconsistencies between requirements expressed in suitable formalism is the main candidate for automation. Some further reasons for not elaborating about automated checking are given in the beginning of Sect 5.

Practice shows that it is quite easy to overlook a specific type of stakeholder if they are not actively in-

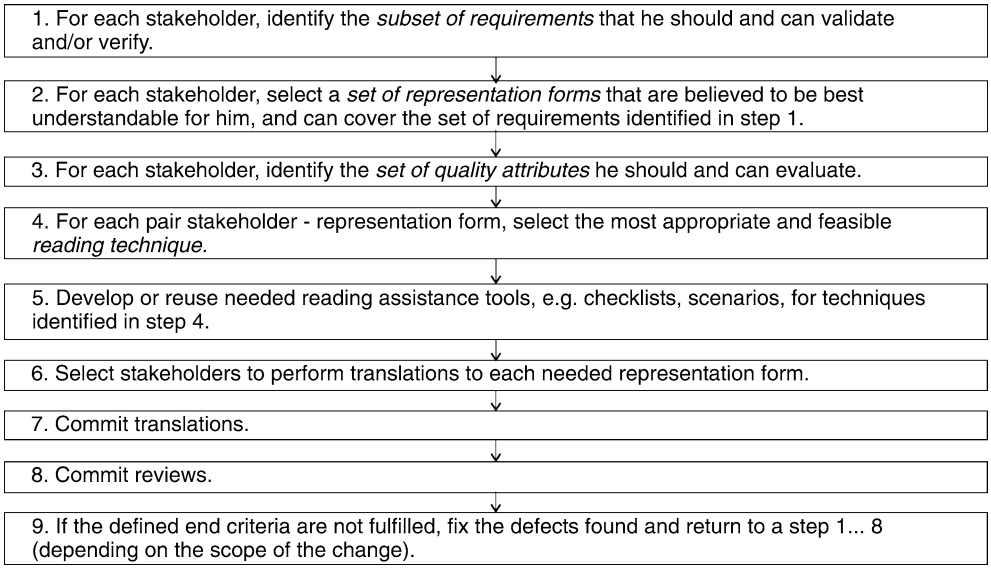

Fig. 5 The procedural view of the framework

volved in the project, and thus the corresponding requirements. For example, Wiegers [4] warns against overlooking indirect or secondary users of a software system that might not use the application directly and instead access its data or services through other applications or through reports. Therefore, one important function of the reviewing activity is validating whether all the relevant stakeholders are taken into account. A review may thus identify some missing stakeholders, which then will enrich the stakeholders pool.

In the framework, there is also the *translation* task. It means that requirements are translated from one representation form to another. This has two goals.

First, it is not considered feasible to maintain several concurrent representations of requirements because of modifiability problems. Therefore, usually only one form is maintained, for example, a natural-language structured requirements specification, while others are created on demand for validation purposes. The idea is that a specific form is believed to be better understood by a specific group of reviewers. In addition, the new form may be amenable to automated checking.

Second, some requirements defects can be found during the translating process itself. The translation process may fail (translation turns out to be partially impossible), thereby pointing to some ambiguity, completeness, consistency defects etc. Also, if two or more persons translate independently and the results are significantly different, this also points to problems (mainly ambiguity) in the document.

An issue here is that the result of translation could deviate from the original, i.e., the translation process by itself could introduce new defects or correct existing ones. The former case is actually less problematic. An error introduced can get spotted and as a result can lead to identification of some ambiguity in the original document. Otherwise, if it goes unnoticed, this can create a problem only if the translated form is to be kept along with the primary form and is to be used later in the development process. Even so, the inconsistency between the two forms will probably be spotted later. A more interesting situation is created when a person performing translation inadvertently fixes a defect present in the original. The new form gets validated, but the defect stays in the primary representation form and may finally appear in the implemented system. The literature does not discuss this issue much. At this stage, we cannot provide specific recommendations either.

As can be seen in Fig. 5 presenting the procedural view of the framework, there is a set of steps where some choices are to be made. A list of some basic criteria that could direct the selection process follows:

- The stakeholder's *domain knowledge*. This obviously affects the set of requirements that can be validated by him/her, and also the set of quality attributes he/she can evaluate.
- The stakeholder's *technical expertise*. This is the main criterion for choosing between requirements representation forms. We could arrange them in the ascending order of required technical expertise: prototypes, user manual, test cases, structured text, diagrams, and formal models. Prototypes are probably the best form for non-technical stakeholders such as end users and customers, while diagrams and formal models can be appropriate for technical stakeholders such as developers or maintenance staff. The availability of technical expertise also defines whether the stakeholder can evaluate some quality attributes such as feasibility.
- The stakeholder's *knowledge of RE*. This is relevant for selecting reading techniques. An experienced person could be left more or less on his own, i.e., to perform an ad hoc review, while less experienced stakeholders require some assistance to be provided, i.e., checklists, procedures, scenarios, etc.
- The *resources* available. This is obviously a limiting factor, since development of both reading assistance tools and alternative representation forms requires time and effort. Based on the schedule and/or budget on the one hand, and on the project priorities (whether quality is a project's driver) on the other, the choice may be limited to cheaper alternatives only. In Sects. 4 and 5, we will tackle review and translation tasks, correspondingly. As already said, our goal was not to develop new prescriptive knowledge but to reorganize the existing body. Therefore, we only identify the set of possibilities that the present literature provides for each of the structural elements of our framework (as depicted in Fig. 4) and discuss them, but do not attempt to propose any novel ones.

4 Reviewing requirements

Every time someone other than the author of a software work product examines the product for problems, a *technical review* takes place [4]. Requirements review is the basic activity in which requirements are validated, or in which requirements defects are identified. At best, all the stakeholders of the software system should participate in this activity. This is not always possible however, and identifying all of them can be a difficult task in itself (see Sect. 4.1.3). Requirements presented in different forms can be reviewed.

The basic issues about requirements reviews are (1) what to check for, (2) what process to follow, and (3) what reading technique to use. These questions will be discussed in the following subsections.

4.1 Requirements Quality

The straightforward answer to the question about what to check for in the requirements documents during review is simple: that the requirements are of high quality. However, this does not explain much.

As mentioned in Sect 2, the quality of any software engineering work product includes ''being right'', i.e., compliance with higher-level work products or authorities that frame this particular product, and ''being built right'', i.e., compliance with the standards and conditions imposed on these types of products and the process of their development.

''Being right'' for a system design could be defined as compliance with the stated requirements. In fact, the question is about how well the requirements are fulfilled if the implementation is made strictly according to the design. Obviously, this is not easy to assess before the working software has been built. ''Being right'' for the requirements themselves can only be defined with respect to the ''highest authority'', namely the stakeholders' opinion. In other words, this part of the quality of the stated requirements is about how well the stakeholders are satisfied if the requirements are fully met by the implementation. This is clearly even more difficult to assess in advance than the quality of artifacts at the following levels in the development hierarchy. The common situation when different stakeholders have different, even contradictory, views on the system, makes the situation even more unclear.

The above problems are a reason why the talk is usually only about some aspects, factors, or attributes of requirements quality, not about requirements quality as a whole. Another reason is that ''being built right'' necessarily means compliance with different, usually independent, standards and conditions.

As we already mentioned, defining a set of attributes for the requirements quality also facilitates a reviewer's task. The review problem gets split into several smaller sub-problems that are easier to handle as well as focus upon.

4.1.1 General Problems

The first thing that must be kept in mind, and checked in a review, is that requirements should not be mixed with other types of information. Mixing would actually not be a defect by itself; however, it easily creates ambiguity and opens doors for many kinds of defects to sneak in.

First, specifications should be explicit about causality, which is often forgotten. This means that statements in *indicative mood* and *optative mood* should be strictly separated [‘12]. The former ones specify things that are true in the environment regardless of the software (e.g., laws of physics). The latter ones specify things that the software should do or cause. This distinction between statements in indicative and optative mood is usually not made very explicitly. Both are called ‘requirements’, although the former ones could perhaps better be called ‘facts’, ‘assertions’, or ‘statements’. In other words, requirements and the domain analysis model are mixed together (see Fig. 1). Even many formal specification methods and languages, such as Z and SCR (see Sect. 5.3), fail to make a clear separation and have insufficient capabilities for describing the environment [14].

Second, a well-known common problem (also noted in [12]) is that many things which should belong to the design and implementation of the software tend to be presented as requirements. On the other hand, things that affect the environment and thus should be defined in the requirements are decided only at the design or implementation stage. One reason for these problems may be a wrong belief in that requirements are just more abstract and less detailed than design representation of the system. Therefore, one may wish to state something about the internal architecture of the software in the requirements specification on the one hand, and suppress exact definitions of interactions on the other hand. However, if you rely on too much abstraction in the wrong places your specification will be about an abstract problem, not about the real problem that your customer expects you to solve [12].

Requirements are not much more abstract than design in the sense that they also describe concrete phenomena; the actual difference is that requirements describe the machine's boundary while design describes its internal structure and behavior (see Sect. 2).

We need to make a clarification. When saying that requirements and design documents must be strictly separated, we do not imply that their development must be separated in time. Unfortunately, it is impossible or unwise in most practical situations to first explore the problem space completely and write the requirements document, and only then go over to the solution space and think about implementation. One reason for this is that some desired requirements may be unfeasible to implement (cf. Sect. 4.1.2). Another factor that makes it necessary to take the implementation into account is the fact that most software development projects today do not start from a clean slate, but augment or modify some existing software. The paper [14] briefly suggests that in such situations the existing ''legacy machine'' should be regarded as part of the environment in the new project. This looks sensible to us only when one or more existing pieces of software are used without modifications, with well-defined interfaces to the new software. Recently, Nuseibeh et al. have proposed a model called Twin Peaks, in which the refinement of requirements (problem) and software architecture (solution) are intertwined or alternated [15, 16]. These two short papers do not convey much more than the main ideas, but they seem to make sense. The basic underlying observation that specification and implementation affect each other both ways is much older [17].

Jackson [12] notes as another common problem in specifications that there can be confusion between things in the software system and things in the environment. One reason for such confusion is that many things in the environment are also modeled (have their counterparts) within the software. This mistake is thus particularly easy to make in object-oriented specifications.

4.1.2 Requirements Quality Attributes

The quality attributes or aspects of a requirements document (not to be confused with those quality attributes that are nonfunctional requirements on the software) can be classified in many ways.

Based on the books [4] and [8] as well as on the papers [18] and [19] we can list the following requirements quality attributes. Each individual statement in a requirements specification should be *complete*, *correct*, *unambiguous*, *feasible*, *necessary*, *prioritized*, *verifiable*, and *concise*. Additionally, the requirements specification document as a whole should be *complete*, *consistent*, *traceable*, *non-redundant*, *organized*, and *conformant to standards*. The list in [18] includes also 'consistency among parts of individual requirements', but that seems superfluous.

Some books, such as [7] and [20], mention completeness, correctness and consistency only; those are probably the most important quality attributes.

The most risky quality factor of all is probably *completeness*. According to Glass, missing requirements are the hardest requirements errors to correct [1].

It might be even more to the point to say that they are the hardest *to detect*. Glass adds a corollary to this fact: The most persistent software errors—those that escape the testing process and persist into the production version of the software—are errors of omitted logic. Missing requirements result in omitted logic.

We discuss this important factor separately in Sect. 4.1.3.

4.1.3 Validating completeness

Completeness is probably the most risky requirements quality factor. At the same time, missing requirements are obviously the hardest to detect during reviews. The scenario-based approach (cf. Sect. 4.3) provides means for detecting some omissions in requirements; however, it is rather unsystematic. This subsection presents some general considerations on requirements completeness issues.

Roughly put, a requirements specification describes the interface between the software system and its environment. Carson [21] considers this interface to be actually a combination of the interfaces between the system and all the system stakeholders. Based on this, he states that requirements are complete if and only if (1) *all stakeholder interfaces* have been identified and quantified for *all applicable life-cycle phases and related operating modes* and then (2) *all interface conditions over all parameter spaces* have been analyzed and captured in the system requirements.

Carson's first point presents the problem of identifying different stakeholders of the software system. It is quite easy in practice to overlook a specific type of stakeholder and thus the corresponding requirements. Wiegers [4] emphasizes the importance of considering different user classes for a product, including indirect users who are affected by the product in any way, for example, through reports or other applications. This already presents a demanding task; however, the list of stakeholders is much broader.

Pohl [22] divides the context of RE for an information system into four worlds. *The subject world* is the domain about which the system is intended to maintain information. *The usage world* comprises stakeholders who are owners, or direct or indirect users of the system. *The system world* comprises both the software/hardware system and the people involved in its operation and maintenance. *The development world* is where the information system development process itself takes place. Stakeholders arise from all four worlds. These four worlds are not necessarily disjoint; for example, users are quite often also subjects. However, even then it is useful to differentiate between the worlds, and thus not to overlook requirements that come from those stakeholders as users (e.g., functionality, usability) and requirements that come from them as subjects (e.g., privacy).

Another classification of stakeholders is presented by Preiss and Wegmann [23]. It is an orthogonal classification with the following dimensions:

- *Domain of inquiry* This means the life-cycle phase of the system. The authors restrict consideration to the development and operation domains.
- *System or suprasystem* For the operation domain, 'system' means the actual software system and 'suprasystem' means its environment (e.g., an organization). For the development domain, 'system' means the development project and 'suprasystem' means the development company and other external factors affecting the project.
- *Goal or means* Goal stakeholders are interested in the perceived behavior of the system or suprasystem. Means stakeholders are those responsible for how the system or suprasystem achieves that behavior.

In comparison to Pohl's classification, this model refines the development and usage worlds; however, it does not distinguish the subject world. The problem actually is that only one level of 'system-suprasystem' relation is considered. It would be more appropriate to allow not only two but multiple values for this dimension (e.g., 'application', 'company', 'society')—whatever relevant levels may be defined. Then, subjects of a software system will likely be classified as "goal stakeholders for the society system". Also, requirements coming from "means society stakeholders", for example, government regulations, will naturally be taken into account. With this modification, this classification seems to be a useful means for systematic discovery of stakeholders or for verifying whether all the stakeholders are taken into account.

Carson's second point (see above) presents the problem of covering all the possible combinations of conditions on all the stakeholder interfaces. It is quite

difficult to provide any useful recommendations here. It should be mentioned that one of the easiest ways to miss a condition is to use universal quantifier words such as 'all', 'always', 'none', and 'never' in requirements statements. Berry and Kamsties [24] discuss this topic. They conclude that universally quantified *indicative* statements are usually not true and thus must be avoided or at least very carefully investigated for exceptions. On the other hand, they claim that universally quantified *optative* statements are not dangerous and that they often are even desired. However, a statement like 'the system must always do this' will very often need exceptions. If there are many such statements coming from different origins, there may very probably be logical conflicts between them. Our general recommendation is that any appearance of universally quantified statements should be carefully reviewed and validated.

Another tricky issue is that of *exclusive* or *negative* requirements, i.e., what the system must *not* do. Those are not very often stated explicitly. Even the literature does not seem to say much about them. This seems to be an important open problem in requirements quality and validation. When some exclusive requirements *have* been stated, their verification in system or acceptance testing can be very difficult as well—especially if they are universally quantified (cf. Sect. 5.5).

4.2 Review process

Reviews can be informal or formal. Examples of informal approaches are distribution of the work product to several peers to look through, and a walk-through in which the author describes the product in front of an audience and solicits comments [4]. When the work product is examined by inexperienced reviewers, such as software end users, rather than asking them to review the product on their own and provide feedback, it may be better to watch them examining it and ''looking for the furrowed brow that indicates a puzzled user'' [4]. The above quote is actually about evaluating prototypes (cf. Sect. 5.4); but, in our opinion, it is also relevant to the reviewing of any other work products by end users.

The best-known type of formal review is called *software inspection*. It was developed by Fagan at IBM [25]. A description of the technique and a survey of recent developments to it may be found in [26]. Wiegers [4] describes it as well. Probably the best known full book about inspections is that of Gilb and Graham [27].

The technique involves creating a team of inspectors that should represent different perspectives. In general, the team should include the author of the inspected document, the author of any predecessor document or specification, and people who have to do work based on the document [4]. Therefore, a requirements inspection team should include requirements engineers (authors), design engineers, and various other stakeholders (sources of requirements), at least representatives of customers and end users.

The inspection process then proceeds in a sequence of well-defined stages. In the planning stage, the inspection team is created and the number of inspection meetings is defined. In the overview meeting, the authors inform the team about the background of the material they will be inspecting, any assumptions made, and specific inspection objectives. In the preparation stage, each inspector (except the authors) examines the document on his own to identify possible defects and issues to be raised; this is the stage where most the defects are discovered. In the inspection meeting, the team meets to discuss discoveries made during the preparation stage, decides on required actions, and goes once again through the document to reveal additional defects. In the rework stage, the authors fix the defects identified. And finally, in the follow-up, a designated individual follows up on the rework that the authors performed.

Reviews involve a lot of time and expense so it makes sense to minimize the work of the reviewers. Errors which are avoidable and which can be detected without a full review should be removed from the requirements document before it is circulated to the review team [8]. Such a *pre-review checking* should be the responsibility of one person and include straightforward activities such as checking the document for obviously missing information, checking it against standards, and running automatic checkers if available (e.g., for spelling mistakes or cross-reference errors). In other words, the reviews should ideally be only concerned with validation of requirements, while all the possible verification should be done prior to that.

One suggested way to reduce the large amount of time and labor that inspections tend to require is to omit most or all of the meetings. Several authors have claimed that this can often be done and good results achieved [28].

4.3 Reading techniques

In both formal and informal reviews, an important factor is the *reading techniques* the reviewers use, or, in other words, the level of guidance the reviewers receive for accomplishing their task. Aurum et al. [26] list the following relevant reading techniques:

- *Ad hoc*—no guidance is provided, reviewers use their own knowledge and experience to identify defects.
- *Checklist-based*—a list of questions is provided specifying what properties of the document must be checked and what specific problems should be searched for.
- *Defect-based*—not only a list of questions but also a collection of procedures to follow is provided; each procedure is aimed at finding a particular type of defect and different procedures are usually assigned to different reviewers. Aurum et al. prefer to call this approach 'scenario-based reading'. However, using the word 'scenario' meaning 'procedure' creates

confusion with 'scenario' as an example of the system use – see below. Aurum et al. seem to be confused themselves because one of the papers they refer to is about scenarios in the latter sense. Therefore, we use the term 'defect-based' as introduced in [29].
– *Perspective-based*—similar to defect-based reading in the sense that some procedures to follow are provided, but they are based on the viewpoints of different stakeholders.

Checklist-based reading is usually the only alternative that is discussed in requirements engineering books. Both Wiegers [4] and Kotonya and Sommerville [8] provide some checklists. Roughly speaking, every relevant requirements quality attribute (cf. Sect. 4.1.2) is rewritten in the form of a question, or refined into two or more questions. Therefore, a checklist works just as a reminder for reviewers of those quality attributes.

Providing procedures that describe the steps to be accomplished in order to answer the review questions requires additional work, but it seems plausible that such procedures may facilitate the reviewers' work. Porter et al. [29], who proposed defect-based reading for requirements inspections, empirically compared it to ad hoc and checklist-based approaches and found that it discovered 35% more defects than those two techniques, while the checklist-based approach performed no better than the ad hoc approach. However, a later replication of the same experiment [30] did not find empirical evidence of better performance of the defect-based approach.

Focusing procedures on the viewpoints of different stakeholders rather than on different defect types leads to perspective-based reading. A discussion of its advantages may be found in [31].

Business scenarios, i.e., examples of how the system may be used, are believed to be an appropriate means for both elicitation and validation of system requirements (e.g., [32]). Some kinds of scenario-based requirements reviews are described, for example, [33], [34], [20]. In [33], one of the two approaches (the second is described below) is that the validation team walks through the sequence of events in the normal and alternative courses of the scenario to detect incomplete and incorrect requirements. We interpret this as one more reading technique:

– *Scenario-based*—a set of concrete scenarios is derived from a predecessor of the reviewed document or from a general understanding of the problem, and then the document is examined for presence, correctness and other quality attributes of requirement statements that cover each scenario.

A "guided inspection" as described in [20] also takes a scenario-based approach. It is not really an inspection in Fagan's sense, because the principle is quite different. Defects are discovered only during inspection meetings. Inspectors prepare scenarios ("test cases" in [20]), and the authors of the reviewed document then explain how the system described by the document is assumed to handle these cases. The team follows the explanations of the authors and checks (1) whether the described system behavior is appropriate, and (2) whether everything was written down and was not left just in the heads of the authors. This process could rather be called "desk testing".

An enhanced version of the scenario-based reading approach is presented by Maiden et al. [33]. We would call it

– *Pattern-based*—a set of patterns is provided to reviewers that they can use when validating requirements against scenarios.

Similar to design patterns, Maiden et al. discuss requirements patterns as a novel technique to help in acquiring, modeling and validating system requirements. They describe some possible patterns. The simplest one, the 'MACHINE-FUNCTION' pattern, captures the essential fact that a good requirements document shall include at least one functional requirement statement for each action in which the software system is involved. Applying this pattern in validation means that for every action mentioned in a scenario, a corresponding requirement statement must be found. Other patterns are more complicated and capture not only facts essential to requirement documents themselves but also ideas essential to design of systems. The 'COLLECT-FIRST-OBJECTIVE-LAST' pattern suggests that a good mechanical device with which a person interacts using one or more personal items must ensure that the person will not go away leaving those personal items at the device. For example, the metro automatic gate machine should open the gate only after the passenger collects the ticket from the machine.

Maiden et al. discuss how such a requirements pattern can be represented by a formal *validation frame* that describes what a part of a scenario should look like to make the pattern applicable, and what should then be searched for in the requirements document. They also claim that such validation frames may be used even for automatic checking of requirements.

5 Translating requirements to other forms

Formal techniques for documenting software requirements have received considerable attention in the RE research community. Nuseibeh and Easterbrook [35] claim that since requirements engineering must span the gap between the informal world of stakeholder needs and the formal world of software behavior, the key question over the use of formal methods is not whether to formalize, but when to formalize. There is little doubt that formal specification methods such as *Z* and the *Vienna Development Method* (VDM) offer some advantages over informal methods [8]. They remove ambiguity and provide for formal (and even automated)

consistency checking as well as for partial completeness and correctness checking.

However, formal methods have not been widely accepted in RE practice so far, and practically oriented RE books (e.g., [4], [8]) usually say nothing or very little about them. There are two main reasons for this. First, formal methods are considered to be too heavy to use, and too difficult to learn for people without an exceptionally strong background in mathematics and logic. Second, only natural language is flexible enough to document any possible requirement at any needed level of detail for any kind of system. One could also counter the above-mentioned claim in [35] by saying that formalization is not absolutely necessary in the software life cycle before the coding phase.

Moreover, too early formalization may effectively hinder validation of requirements. The more technical the specifications, the less likely it is that those without technical expertise can understand them [11].

Therefore, natural language remains the most common way of documenting software requirements. Unfortunately, all natural languages are inherently ambiguous; this problem is exacerbated in many projects, because the requirements are written in English although many stakeholders are not fluent in it. In addition, a software requirements specification (SRS) document presenting the requirements as a collection of separate statements is probably not the best understandable form for either customers or developers. Wiegers [4] adds to this that actually no single view of requirements may provide a complete understanding of them. Therefore, the need for creating other representations must be obvious.

It is not considered feasible to maintain several concurrent representations of requirements due to modifiability problems. However, for validating them, rewriting natural language into other forms is considered very useful. This has a double advantage—requirements defects are found during the translation process itself and during the following review of the result. However, for a particular representation form, literature usually stresses on only one of these advantages.

In the following subsections, we will discuss different forms for expressing requirements, other than structured natural language. However, translating requirements from one natural language to another one is a relevant approach as well. At least it may lead to identification of some ambiguous statement, because, for example, some ambiguities possible in English are just impossible in Finnish, and vice versa. And, obviously, the level of understanding of requirements may be significantly better when they are presented to a reviewer in his native language.

5.1 User manual

Many authors propose drafting the user manual for the system early in the development process. A good manual describes all user-visible functionality in easily understandable language. It is still a natural language document, but with all 'shall' statements rewritten as if they were already implemented and presenting requirements in a more tangible form than an SRS. Berry et al. [36] suggest that in some cases the user manual may be used as a sole requirements specification document, i.e., instead of an SRS. However, a user manual is obviously only a partial view of the requirements since it presents only functionality (not performance or other characteristics) and only that functionality visible to end users of the software. Manuals for other stakeholders (e.g., people who will be responsible for maintenance of the system) could also be drafted, of course.

A quality SRS should provide all the information needed for writing the user manual. If the writer finds this (partially) impossible, this points to some problems with the SRS that must be investigated.

After the draft is finished, it is to be reviewed by targeted stakeholders (other stakeholders could do this as well, of course). Note that all the techniques discussed in Sect. 4 could be applied. The review may be informal or formal; we can leave the reader alone and then ask for comments or try to learn more by watching him read (or by interacting with him). We can just give him the manual to read (ad hoc) or provide a checklist with questions, a procedure to follow, or a set of scenarios of the type 'suppose you want to do...' (see Sect. 4.3).

5.2 Visualizations

Visualization is often seen as a way to help people gain insight from large and complex data sets. The SRS for a complex system is such a data set, and it is usually difficult to understand and validate it directly. One can check whether all the individual requirements statements make sense; however, completeness and consistency of the whole description is much harder to verify. Graphical presentations provide help—they link individual statements together and present a coherent picture of a slice of the system.

Visualization is usually based on a semi-formal model. [4] proposes the following graphical representations: *data flow diagram*, *entity-relationship diagram*, *state transition diagram*, *dialog map* (for presenting user-visible functionality), and *class diagram* (mainly for domain analysis), *decision tables* and *decision trees*. Among these diagram types, the class diagram and the state-transition diagram (statechart diagram) are included also in the UML [37], and the dialog map looks like a simplified version of UML's *activity diagram*. Wiegers just mentions that some other UML diagram types may be utilized as well, for example, *sequence diagram* and *collaboration diagram*.

Also, dynamic representations, i.e., *animations*, are proposed [38, 39] and considered as facilitators in understanding the behavior of the system.

[40] lists the following characteristics of a good set of visualizations for system requirements: minimize semantic distance, match the task being performed, support the most difficult mental tasks, highlight hidden dependencies and provide context when needed, support top-down review, support alternative problem-solving strategies, provide redundant encoding, show roles being played by parts of the specification, and show side effects of changes in one part of the specification to others.

Again, problems with creating a graphical representation probably point to defects in requirements. Visualizations are to be reviewed by all the relevant stakeholders. Graphical models are undersood to some degree by all stakeholders, however, specialists understand them better than system end users.

5.3 Lightweight formal models

As discussed above, formal methods are considered difficult to use and not flexible enough, and, therefore, have not been widely accepted in RE practice. One formal method that seems to have found some industrial acceptance is *Software Cost Reduction* (SCR), which uses a tabular notation (see, e.g., [41]). However, this acceptance is limited to the "toy world" of nuclear reactors, space shuttles, and car cruise control systems. Those systems are inherently complex; but actual software is relatively simple—the sets of monitored and controlled variables are known and limited, and desired relationships among them are expressible either mathematically or in formal logic.

In response to this lack of acceptance, the research community has proposed so-called *lightweight formal methods*. These are actually the same VDM or SCR methods, but applied in a lightweight way. The term 'lightweight' is used to indicate that the methods can be used for partial analysis on partial specifications, without a commitment to developing and baselining a complete, consistent formal specification [42]. Therefore, the requirements specification is assumed to be informal; however, in order to find errors, some critical parts of it are rewritten into a formal notation and then formally verified.

Hörl and Aichernig [43] report on an experience of rewriting informal specifications into VDM, and show that just the attempt of rewriting discovered many defects that had been missed during the preceding inspection. These defects were also more serious that those discovered in inspection. Easterbrook and Callahan [44] experimented with rewriting informal specifications into Leveson-style AND/OR decision tables and into SCR. They also report that this allowed discovery of some requirements defects. Gervasi and Nuseibeh [45] present an approach and a case study of validating highly-structured natural language requirements specifications by building a set of formal models and then checking them for selected desirable properties, both phases automatically. Any violations of the desirable properties were reported to the specifiers. The authors report that this revealed a number of defects.

5.4 Prototypes

The primary reason for creating a prototype is to resolve uncertainties early in the development cycle. It is also an excellent way to reveal ambiguities in the requirements. We consider developing a prototype to be one more form for expressing requirements. Similar to user manuals, a prototype may present only a limited set of requirements. However, it is a useful tool for communication with stakeholders because it makes the requirements tangible. Users, managers, and other non-technical stakeholders usually find it very difficult to visualize how a written statement of requirements will translate into an executable software system. If a prototype is developed to demonstrate requirements, they find it easier to discover problems and suggest how the requirements may be improved [8]. Prototypes are easier to understand than the technical jargon developers sometimes use [4].

There are two main kinds of prototypes. An *evolutionary prototype* is an initial and partial version of the system, which is available early in the development process. It is intended to evolve into the final system through modification and extension. A *throwaway prototype* is developed solely for requirements analysis and validation and is discarded afterward. It can be implemented with completely different technology from that of the final system. Requirements that should be supported by an evolutionary prototype are those which are well-understood; in contrast, the requirements that should be prototyped with throwaway prototypes are those that cause most difficulties to customers and are the hardest to understand [8]. Therefore, particularly throwaway prototyping is useful for the purpose of requirements validation.

There are three approaches which allow the development of a throwaway prototype relatively quickly [8]. A *paper prototype* is a mock-up of the system that is developed with no executable software. Paper versions of the screens that may be presented to the end-user are drawn and various usage scenarios are planned. In a *'Wizard of Oz' prototype*, user interface software is required to be developed, but a person simulates the responses of the system in response to user inputs. An *automated prototype* is an executable prototype created with a fourth-generation language or another rapid prototyping environment.

Even a paper prototype is an effective tool for requirements validation. For example, [46] reports on an experiment in an embedded software project, where a technique was introduced for making early mock-ups of the user interface and testing their usability with real, potential users. The authors report that this led to a surprisingly large reduction in the number of usability problems – by about 70%.

Examples of automated prototyping approaches can be found in literature. State machines are relatively easy to handle and even animate; for example, experiences with rapid prototyping using the STATEMATE tool are presented in [47]. An approach for creating an executable prototype from a VDM formal specification (with the need to manually develop a user interface) is described in [48].

As for executable prototypes, [49] lists the following characteristics of a good prototype: it is simple, yet captures all system functions which are of interest to customers; demonstrates all user interactions with the system, has a user interface which looks and behaves similarly to the one intended for the final system, captures all exception conditions which are of interest to the users, is ''quick and dirty'', but robust enough to demonstrate some reliability and performance, is self-contained and easily portable for demonstration at geographically distant locations.

5.5 Test cases

As was noticed in sect. 4.1.2, a desirable attribute of every requirement statement is that it should be verifiable, i.e., it should be possible to define one or more test cases that can check whether the requirement has been met. Even though the tests will be applied to the system only after implementation, designing tests at an early stage is an effective way of revealing requirements defects. Creating test cases for the system requirements will make the expected system behaviors clearer to all the stakeholders and help validate the corresponding requirements early in the project.

Test cases can be based on the functional requirements or derived from use cases. They are basically detailed descriptions of anticipated user and system activities associated with each requirement statement. Naturally all activities implied by the requirements under scrutiny should be tested. If the system requirements are unambiguous and correct, the test cases should be fairly obvious and easy to write. In contrast, if the test cases are not easy to write, there is likely to be something wrong with the requirements.

Here, the objective of proposing tests is to validate requirements, not the system. Therefore, the writer of the test cases may ignore many issues the designer of a 'real' test would have to deal with, such as costs, redundancy, data definition, etc. Designing tests that may later be used as system tests does make sense, though, since it will clearly reduce the overall costs of test planning. Indeed, according to the so-called V model of software development [2], acceptance tests should be designed together with the requirements.

Not all types of requirements are by nature easy or even possible to test. For example, it is difficult to design tests for requirements for the system as a whole, exclusive requirements (the system must *not* do something; cf. Sect. 4.1.3, and some non-functional requirements [8].

There is obvious similarity between scenarios (Sect. 3) and test cases. However, we prefer to see test cases as a form for expressing requirements and scenarios as a review reading technique. A set of detailed test cases can still be reviewed, at least for completeness, with respect to a set of general business scenarios.

6 Conclusions

Requirements quality control seems to be somewhat marginal for RE. This situation is no different when compared to, for example, the role of testing in software engineering. Creative people seldom get excited about any form of quality control and often even see it as a necessary evil. We believe however that in RE this mindset is largely inappropriate, because requirements validation only to a very insignificant extent is checking that requirements analysts are doing their jobs properly. Validation is a necessary feedback link needed for stimulating the elicitation process, stimulating evolution of customers' understanding about which of their problems are solvable in principle, and for giving them a chance to check early whether the solutions being proposed will really solve their problems.

Literature also does not treat requirements quality control as a coherent activity. Most books present some set of ''good practices'', whose application may contribute to the quality of requirements and to our confidence in this quality. A tacit recommendation is that one should apply as many of those as possible in order to achieve better results.

In this paper, we developed a framework unifying requirements quality control. We attempted to promote the point that this important activity must be studied as a coherent entity. We tried to reorganize the existing prescriptive knowledge in a way that the focus would be on the task of achieving a sufficient level of understanding of the stated requirements by a particular stakeholder, instead of focusing on some processes and techniques, in which ''all the stakeholders should participate''.

The framework was depicted in Fig. 3, consisting of the two basic activities, review and translation, and the four pools: stakeholders, requirements quality criteria, reading techniques, and requirements representation forms. Requirements quality criteria include general issues related to what kind of information requirements should contain, quality attributes for the system requirements as a whole, and quality attributes for individual requirement statements. For stakeholders classification, we proposed to use the 3-dimensional model from [23] with a slight modification. We identified six different reading techniques, and six different forms for representing requirements. Of course, we do not imply that those lists are complete; however, they include everything we encountered in the literature.

Our framework concentrates on informal approaches to RE, while formal approaches are present in it only in the form of lightweight models. Formal approaches are

seldom applied by practitioners, in part because they are considered to be too heavy to use, and too difficult to learn for people without an exceptionally strong background in mathematics and logic. Another problem is that too early formalization may effectively hinder validation of requirements, because the more technical the specifications, the less likely it is that those without technical expertise can understand them.

Throughout this paper, we assumed that a structured text document is used as the primary form for documenting requirements, as it is common in practice. However, in principle, one could have a formal model as the primary form, given that on-demand translation to other requirements representation forms needed for non-technical stakeholders is provided (which could possibly be automated). Elaborating such an approach so that formal techniques would appear more attractive to practitioners is a tough but important challenge for future work.

Acknowledgements The major part of this work was performed in the KOTEVA project, which was financially supported by the National Technology Agency of Finland (TEKES) (project number 676/31/01) and industrial partners TietoEnator, Elisa, Yomi, and Systeemiratkaisu. We would like to thank the referees for their insightful comments.

References

1. Glass RL (2003) Facts and fallacies of software engineering. Addison-Wesley
2. Boehm BW (1979) Guidelines for verifying and validating software requirements and design specifications, in Euro IFIP 79. London, UK: North-Holland, pp 711–719
3. Gause DC, Weinberg GM (2003) Exploring requirements: Quality before design. Dorset House
4. Wiegers KE (2003) Software Requirements, 2nd ed. Microsoft Press
5. Wiegers KE (1999) Software Requirements. Microsoft Press
6. Hetzel B (1988) The complete guide to software testing, 2nd ed. Wiley
7. Bashir I, Goel AL (2000) Testing object-oriented software: life cycle Solutions. Springer-Verlag, New York
8. Kotonya G, Sommerville I (1998) Requirements engineering : processes and techniques. Wiley
9. Boehm BW (1984) Verifying and validating software requirements and design specifications. IEEE Software 1(1):75–88
10. Boehm BW(1974) Some steps towards formal and automated aids to software requirements analysis and design, in Information Processing 74: Proceedings of IFIP Congress 74. Stockholm, Sweden: North-Holland, pp 192–197
11. Bergman M, King JL, Lyytinen K (2002) Largescale requirements analysis revisited: The need for understanding the political ecology of requirements engineering. Requirements Engineering 7(3):152–171
12. Jackson M (1995) Software requirements & specifications : a lexicon of practice, principles and prejudices. ACM Press, Addison-Wesley
13. Jackson M (2004) Seeing more of the world, IEEE Software 21(6):83–85
14. Zave P, Jackson M (1997) Four dark corners of requirements engineering. ACM transactions on Software Engineering and Methodology (TOSEM) 6(1):1–30
15. Nuseibeh B (2001) Weaving together requirements and architectures. IEEE Computer 34(3):115– 119
16. Hall JG, Jackson M, Laney RC, Nuseibeh B, Rapanotti L (2002) Relating software requirements and architectures using problem frame. In Proc IEEE joint international conference on requirements engineering (RE'02) 137–144
17. Swartout W, Balzer R (1982) On the inevitable intertwining of specification and implementation. Communications of the ACM 25(7):438–440
18. Schneider RE, Buede DM (2000) Properties of a high quality informal requirements document. In Proc 10th annual international symposium of the INCOSE
19. Kar P, Bailey M (1996) Characteristics of good requirements, requirements working group of the INCOSE, available as http://www.incose.org/rwg/goodreqs.html.
20. McGregor JD, Sykes DA (2001) A practical guide to testing object-oriented software. Addison-Wesley
21. Carson RS, Requirements completeness: A deterministic approach, in Proc 8th annual international symposium of the INCOSE, 1998, available as http://www.incose.org/rwg/98_carson/paper016.pdf
22. Pohl K (1996) Process-centered requirements engineering. Research studies Press / Wiley
23. Preiss O, Wegmann A (2001) Stakeholder discovery and classification based on systems science principles, In Proc 2nd APQSC Asia-Pacific Conference on Quality Software. IEEE 194–198
24. Berry DM, Kamsties E (2000) The dangerous ''all'' in specifications In Proc 10th international workshop on software specification and design. IEEE 191–193
25. Fagan ME (1999) Design and code inspections to reduce errors in program development, IBM Systems Journal, vol 15, no. 3, pp. 182–211, 1976, reprinted in IBM Systems Journal 38(2–3):258–288
26. Aurum A, Petersson H, Wohlin C (2002) State-of-the-art: software inspections after 25 years. Software Testing, Verification & Reliability 12(3):133–154
27. Gilb T, Graham D (1993) Software Inspection. Addison-Wesley
28. Votta LGJ (1993) Does every inspection need a meeting? in Proc. 1st ACM symposium on the foundations of software engineering, 1993, pp 107–114 (Software Engineering Notes, vol. 18, no. 5)
29. Porter AA, Votta LGJ, Basili VR (1995) Comparing detection methods for software requirements inspections: a replicated experiment. IEEE Transactions on Software Engineering 21(6):563– 575
30. Fusaro P, Lanubile F, and Visaggio G (1997) A replicated experiment to assess requirements inspection techniques. Empirical Software Engineering 2(1):39–57
31. Shull F, Rus I, Basili V (2000) How perspective-based reading can improve requirements inspections. IEEE Computer 33(7):73–79
32. Weidenhaupt K, Pohl K, Jarke M, Haumer P (1998) Scenarios in system development: current practice. IEEE Software 15(2):34–45
33. Maiden N, Cisse M, Perez H, Manuel D (1998) CREWS validation frames: Patterns for validating systems requirements, in Proc. 4th REFSQ international workshop on requirements engineering: foundation for software quality
34. Kösters G, Six H-W, Winter M (2001) Coupling use cases and class models as a means for validation and verification of requirements specifications. Requirements Engineering 6(1): 3–17
35. Nuseibeh B, Easterbrook S (2000) Requirements engineering: a roadmap. In Proc 22nd international conference on software engineering (ICSE-00). ACM pp 35–46
36. Berry DM, Daudjee K, Dong J, Fainchtein I, Nelson MA, Nelson T, Ou L (2004) User's manual as a requirements specification: case studies. Requirements Engineering 9(1):67–82
37. Booch G, Rumbaugh J, Jacobson I (1999) The Unified Modeling Language User Guide. Addison-Wesley
38. Lalioti V (1997) Animation for validation of business system specifications. In Proc 13th Hawaii International Conference on System Sciences vol. 2. IEEE, pp. 220–229.

39. Gemino A (2004) Empirical comparisons of animation and narration in requirements validation. Requirements Engineering 9(3):153–168
40. Dulac N, Viguier T, Leveson N, Storey M-A (2002) On the use of visualization in formal requirements specification. In Proc IEEE joint international conference on requirements engineering RE'02, pp 71–80
41. Heitmeyer C, Kirby J, Labaw B (1997) Tools for formal specification, verification, and validation of requirements, In Proc 12th annual COMPASS conference on computer assurance pp 35–47
42. Schneider F, Easterbrook SM, Callahan JR, Holzmann GJ (1998) Validating requirements for fault tolerant systems using model checking, in Proc 3rd IEEE international conference on requirements engineering, pp 4–13
43. Hörl J, Aichernig BK (2000) Validating voice communication requirements using lightweight formal methods. IEEE Software 17(3):21–27
44. Easterbrook S, Callahan J (1997) Formal methods for V&V of partial specifications: an experience report. In Proc 3rd IEEE international symposium on requirements engineering pp 160–168
45. Gervasi V, Nuseibeh B (2002) Lightweight validation of natural language requirements. Software - Practice and Experience 32(2):113–133
46. Lauesen S, Vinter O (2001) Preventing requirement defects: An experiment in process improvement. Requirements Engineering 6(1):37–50
47. Andrews BA, Goeddel WC (1994) Using rapid prototypes for early requirements validation. In Proc 4th annual international symposium of the INCOSE
48. Fenkam P, Gall H, Jazayeri M (2002) Visual requirements validation: Case study in a Corba-supported environment. In Proc IEEE joint international conference on requirements engineering RE'02 81–88
49. Ghajar J-Dowlatshahi, Vernekar A (1994) Rapid prototyping in requirements specification phase of software systems. In Proc 4th annual international symposium of the INCOSE