# 17 “Good Quality” Requirements in Unified Process

*Nur Yilmaztürk*

**Abstract:** As supported by many empirical evidences since early 1970s, “good quality” requirements are the leading factor for a successful software development project that delivers a “good quality” product with originally specified features and functionalities, on time, and within the originally estimated budget. The challenge gets tougher and more critical when the competition in the market is severe, the number of customers on the world is rather limited and static, and the customer demands are high. As functioning in such a market, with the main goal to maintain the leading position of the previous versions of its Stressometer®, ABB has adopted a RUP®[1]-based software development process in the new generation Stressometer systems development projects. Stressometer Unified Process (SUP) integrates the RUP essentials with some features of agile processes such as heavy involvement of various stakeholders, preparation of test cases before coding, and continuous testing during development. This chapter describes the essential quality characteristics of requirements –both individual and aggregates such as embodied in a use-case model or in a specification, analyses the relations among them, evaluates RUP regarding the means it provides or lacks for developing “good quality” requirements, and discusses how ABB Stressometer projects have tackled these shortcomings via SUP.

**Keywords**: Quality attributes of requirements, Requirements quality metrics, Requirements engineering and management via Rational Unified Process, Use-cases, Use-case model, Measuring quality of requirements.

## 17.1 Introduction

“Good quality” software requirements are prerequisite for “good quality” software products. Results of the research by Standish Group [23] verify our theory. The Standish Group’s CHAOS report that covers the findings from study of 8380 IT projects illustrates that 31.1% of projects are cancelled before they are completed. The results indicate 52.7% of projects cost 189% of their original estimates, and still deliver fewer features and functionalities than originally specified. Only 16.2% of software projects are completed on time and on budget. Among the projects completed by the large companies, only 42% of them comprise the originally proposed features and functions. The top three factors on challenged projects are lack of user input (12.8%), incomplete requirements and specifications (12.3%), and changing requirements and specifications (11.8%). Finally, the major reason for projects cancellation is reported as incomplete requirements (13.1%).

[1] Rational Unified Process®

Cost of "bad quality" requirements have been studied since early 1970s. Boehm's study of 63 software projects from three companies, namely GTE, TRW, and IBM, illustrated that the cost of change grows exponentially as the project progresses [2]. [4] reiterates this result by stating that the relative cost of repair is two hundred times greater in the maintenance phase than if it is detected in the requirements phase. Further, it bases the escalation in cost on two factors: (i) the delay from when the defect was introduced until it was detected, (ii) the amount of rework needed to correct both the original defect as well as the consequent defects in the later stages. As referred to by [4], DeMarco states that 56% of the bugs detected during testing can be traced to the requirements errors.

Iterative nature of RUP assists in eliminating above mentioned risks by integrating a software product progressively throughout its development life cycle, by managing requirements change and "creep" in a controlled manner, by learning early and improving incrementally, and by detecting flaws early thus, building higher quality over several iterations. Yet, RUP is a generic process and it is inevitable to tailor it according to the needs of a particular project or the projects of a specific department for better efficiency and effectiveness. In an attempt to establish a balance between delivering good quality software products and delivering them on time, ABB's Stressometer product line adapted RUP in an agile fashion while adhering to the RUP essentials.

The main aim of this chapter is to evaluate a use-case driven, iterative software development process during which modeling is done via UML[2], within the context of requirements development and management, against the quality of the requirements established during such a process. To this end, Sect. 17.2 provides background information about ABB and the Stressometer product line. Section 17.3 presents the requirements management and engineering activities involved in ABB's RUP-based software development process, SUP. Section 17.4 describes the characteristics of "good quality" requirements, elaborates on the relations among the characteristics, and further discusses how ABB Stressometer projects managed to achieve "good quality" requirements, supplying the discussions with experiences from the three major projects at ABB. Finally, Sect. 17.5 concludes the chapter.

## 17.2 Background

ABB (Asea Brown Boveri Ltd.) began operations in 1988 following a merger of two parent companies namely, ASEA AB and BBC Brown Boveri Ltd, each of which has been in business for more than a century (*www.abb.com*). Today, with about 105000 employees in around 100 countries, the ABB Group of companies functions in two core business areas, automation and power technologies that enable utility and industry customers to improve performance while lowering environmental impact.

[2] Unified Modelling Language

ABB Power Technologies serves industrial and commercial customers, as well as electric, gas and water utilities, with a broad range of products, services and solutions for power transmission and distribution. The portfolio includes transformers, switchgear, breakers, capacitors and cables, as well as high- and medium-voltage applications, many of which are also sold through external channel partners like distributors, system integrators, contractors and original equipment manufacturers. ABB Automation Technologies serves the automotive, building, chemicals, consumer, electronics, life sciences, manufacturing, marine, metals, minerals, paper, petroleum, transportation, turbo-charging and utility industries. Key technologies include control, drives, enterprise software, instrumentation, low-voltage products, motors, robots and turbochargers. These offerings are supported by field maintenance and asset management services, and are sold both directly and through channel partners.

As a part of the ABB Automation Technologies, Force Measurement unit supplies products, systems, and services for measurement and control in a broad range of application from steel making to paper conversion. Stressometer is a Force Measurement product line that involves software intensive systems, which have been providing rolling mills with accurate online control of the flatness of cold rolled strips for more than 30 years. Stressometer system measures flatness, analyzes and stores flatness data, generates output for automatic flatness controls, and presents data in informative displays. Stressometer systems are designed for minimum maintenance and maximum uptime to ensure undisturbed continuous production and minimized scrap levels. Over the years, ABB has been continuously improving the Stressometer product line parallel to the technological progress in software engineering in an attempt to keep its number one position in the market [26].

## 17.3 Practice

New generation Stressometer systems are implemented by using SUP that is RUP tailored to fit the needs of the Stressometer department’s development projects. The major issue considered during such tailoring is being agile by involving stakeholders with different profiles –external customers as well as the internal ones –actively and heavily throughout the development life cycle, preparing the test cases before coding, and having continuous testing during development. SUP facilitates agile, use-case driven, iterative development during which modeling is done via UML [26]. This section presents the requirements management and engineering activities that are involved in the SUP. For a comprehensive discussion on agile methods, and particularly, requirement engineering via agile methods readers should refer to Chap. 14 in this book.

The first step in the requirements engineering process via SUP is to elicit information from the stakeholders in order to understand their needs. SUP imposes the involvement of external customers with business knowledge and internal customers with technical domain knowledge, in this activity. It recommends interviews and requirements workshop as the techniques to elicit the needs. The findings are used as primary inputs to defining the features of the prospective product hence, the high-level requirements that are described in a Vision document. A Vision may include features that do not fit in the project scope or the existing business plans yet, should be kept for future references. Accordingly, the stakeholders prioritize the features based on pre-agreed attributes in order to identify the final set to be attended by the particular iteration of the project. Before moving to the lower level requirements identification, the complete Vision document and the prioritization results are reviewed and approved by all the stakeholders who took part in the elicitation. Eventually, approved Vision together with the prioritization matrix is checked into the configuration management database, and is labeled as "Approved–IterX". As the next activity, the same group of stakeholders gathers at a use-case workshop to define the functional requirements of the system. Initial group of actors and use-cases derived from appropriate features are compiled in a use-case model and illustrated in use-case diagram(s) during the meeting by using a tool. Brief descriptions for each actor and use-case are also entered. The results are further documented in a Use-Case Model Survey. A few review meetings with the same attendees follow in order to finalize an approved version. Features that could not be traced to functional requirements in use-cases, for example those that imply non-functional requirements such as performance requirements, are revisited in order to compile a Supplementary Specifications document. As any other formal artifact in the process, Supplementary Specifications document is also reviewed, approved by the stakeholders, and eventually, version controlled.

The identified use-cases are prioritized according to a set of pre-agreed attributes in a separate session by the same requirements team. Those use-cases assessed as high priority to attend are assigned to the requirements specifier for detailing.

The requirements specifier with assistance of the end-users from both external and internal customers describes the flows of each use-case under concern in detail in separate specification documents. She/he also writes the supplementary requirements to the level of detail needed to hand off to the next stages in the development. If required, she/he can prepare sub-supplementary specifications. For example, user-interface descriptions, control algorithms, digital and analogue signal descriptions are detailed in separate sub-supplementary specification documents. As soon as the first version of a specification is ready, it is passed to the test designer(s) for test case preparations. Each specification is reviewed by a group that includes the external customers with business knowledge, internal customers with technical domain knowledge, requirements specifier, end-users that assisted during detailing the requirements, software architect, designer, and test designer. Upon approval, each document is checked into the configuration management database and labeled as "Approved–IterX". Subsequently, the design team starts working on the architectural and detailed design of the requirements. The test cases are updated according to the final changes in the related requirements specifications, reviewed and approved by the requirements specifying team and the test team before they are version controlled and passed to the attention of the test team. Parallel to the above activities, the project team also continuously gathers terminology in a project Glossary.

## 17.4 Evaluation

Quality of requirements can be characterized by a number of attributes. We collect those that are commonly discussed by the academia and the industry, and merge them into a set of 26 quality attributes in Table 17.1. During our study of these attributes, we encountered the following inconsistencies: (i) Different references may use different terms for the same attribute. For example, the first attribute in the table is termed “Attainable” in [11], “Feasible” in [24], and “Achievable” in [11]. In such cases, we either include all different terms found in the literature, or refer to all of them by using the most common one; (ii) Content of an attribute may differ from reference to reference. For example, [12] and [5] define “Correct” as what is termed “Necessary” in [24], which also presents “Correct” as a separate requirements quality attribute but with a definition that differs from the one found in [12] and in [5]. In such cases, we keep both attributes and assume a positive relation between the two attributes; (iii) No clear distinction between quality attributes that are applicable only to individual requirements and quality attributes that are applicable only to the aggregate requirements. In most of the cases, the definition of an attribute presented as an attribute of an aggregate implies dependency on the individual requirements of the aggregate constituting the same quality. Moreover, one can hardly find a consensus between different references on whether an attribute is applicable to an individual requirement or to an aggregate. For example, “Complete”-ness is claimed to be an attribute of an aggregate by [4] and [5] whereas, it is suggested to be applicable to an individual requirement by [9], and to both an individual requirement and an aggregate by [24]. In our evaluation, we disregard such distinction and use the attribute to measure both individual requirements and aggregates, unless there is common consensus on the applicability of an attribute for example as in the case of “Achievable/Feasible/Attainable”, “Clear/Precise/Meaningful” etc.

These attributes are not independent: (i) It is not possible to achieve a certain quality unless another one exists. For example, if a requirement is not “Unambiguous” it cannot be “Verifiable”. Naturally, there is no means to verify a requirement if multiple interpretations exist for it [4, 5, 24, 12]. (ii) An attribute may affect achievement of another attribute depending on the way the affecting attribute is achieved. For example, if we try to make a requirement more “Unambiguous”, more “Verifiable”, “Complete”, and “Consistent” by using extremely formal notations, we definitely decrease the level of “Understandability” by especially the non-computer specialist stakeholders [4]. Whereas, on the other hand, by no means “Unambiguous”, “Verifiable”, “Complete”, and “Consistent” requirements are un”Understandable”. On the contrary, “Unambiguous”ness, “Complete”ness, and “Consisten(t)”cy enhance “Understandabl(e)”ity when achieved via less formal means such as by using Natural Language augmented with more formal models [5]. (iii) Existence of an attribute jeopardizes achievement of another attribute. For example, if all use-cases included in a use-case model were “Necessary” then why would we need to “Rank”ing one or more of them as *optional* “by relative importance”? We have summarized our findings from experiences with relations between various quality attributes in Tables 17.2(a) and 17.2(b).

Finally, most of the requirements attributes are subjective. In such cases, it can be difficult to measure a quality objectively via metrics; it may require performing expert reviews for the ultimate assessment. Still, it is possible to associate those characteristics with indicators that point at existence or absence of the quality under concern.

Our experiences at ABB have proven that the level of quality achieved in requirements produced during a software development project highly depends on the process adopted. A feature of a process can influence a specific quality by leading to an improvement in the quality, by detracting from the quality, or by doing both hence, a trade-off situation; as well as a process might not address the quality at all. An individual requirement or an aggregate of requirements created via RUP would score very well across most of the quality attributes, whereas fare rather insufficiently on others. Tailoring the standard RUP practices to fit a specific software development project’s needs helps enhancing the poor quality but mainly those attributes that matter most to the project. In the following sub-sections, we describe those quality characteristics that were deemed important by the Stressometer projects at ABB, elaborate on their relations with other characteristics, discuss the indicators of strengths and weaknesses, evaluate how the projects attempted to achieve the quality, and specify the metrics for measuring the quality where applicable.

**Table 17.1** Quality attributes of requirements

| Quality Attributes | [4] | [5] | [9] | [11] | [12] | [14] | [19] | [24] | [25] |
|---|---|---|---|---|---|---|---|---|---|
| Achievable/Feasible/Attainable | | I | | I | | I | | I | |
| At the Right Level of Detail | | I, A | | | | I | | | |
| Clear/ Precise/Meaningful | | I | I | I | | | | | |
| Complete | A | A | I | | A | I, A | A | I, A | A |
| Concise | A | A | | I | | I | | | |
| Correct | I, A | I, A | | | I, A | | | I | A |
| Cross-Referenced | | A | | | | | | | |
| Design Independent | I, A | I, A | | | | I | | | |
| Electronically Stored | | A | | | | | | | |
| Executable/Interpretable | | A | | | | | | | |
| Externally Consistent | A | A | | | | | | A | |
| Forward Traceable | I, A | I, A | | | I, A | | I | I, A | I |
| Implementation Independent | | | | | | I | | | |
| Internally Consistent | A | A | | | A | A | | A | A |
| Modifiable | A | A | | | A | | | A | A |
| Necessary | | | | I | | I | | I | |
| Not Redundant | | I, A | | | | | | | |
| Organized | A | A | | | | | | | |
| Prioritized/Ranked/Annotated by Relative Importance | I | I, A | | | I, A | | | I | I, A |
| Prioritized/Ranked/Annotated by Relative Stability | I | I, A | | | I, A | | | | I, A |
| Prioritized/Ranked/Annotated by Version | | I, A | | | | | | | |
| Reusable | | A | | | | | | | |
| Traced/Backward Traceable | I, A | I, A | | | I, A | I | I | | I |
| Unambiguous | I, A | I, A | I | I | I, A | I | I | I | I |
| Understandable | A | I, A | | | | | A | | |
| Verifiable | I, A | I, A | | I | I, A | I | | I | I, A |

**I**= Applies to an **i**ndividual requirement; **A**=Applies to **a**ggregate requirements such as a complete SRS, a use-case model, a use-case specification etc.

**Table 17.2(a)** Relations between quality attributes of requirements

| | Achievable/ Feasible/ Attainable | At the Right Level of Detail | Clear/ Precise/ Meaningful | Complete | Concise | Correct | Cross-Referenced | Design Independent | Electronically Stored | Executable/ Interpretable | Externally Consistent | Forward Traceable | Implementation Independent | Internally Consistent |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Achievable/Feasible/Attainable | ▨ | | | | | | | | | | | | | |
| At the Right Level of Detail | | ▨ | | | | | | | | | | | | |
| Clear/ Precise/Meaningful | | | ▨ | +? | | | | | | + | | | | |
| Complete | | | | ▨ | | | | | | + | | | | |
| Concise | | + | | -? | ▨ | | | + | | | | | + | |
| Correct | | | | | | ▨ | | | | + | + | | | + |
| Cross-Referenced | | | | | | | ▨ | | + | | | | | |
| Design Independent | | | | | | | | ▨ | | | | | | |
| Electronically Stored | | | | | | | | | ▨ | +? | | | | |
| Executable/Interpretable | | | | | | | | | | ▨ | | | | |
| Externally Consistent | | | | | | | | | | | ▨ | + | | |
| Forward Traceable | | | | | | | | | + | | | ▨ | | |
| Implementation Independent | | | | | | | | | | | | | ▨ | |
| Internally Consistent | | | | | | | + | | | +? | | | | ▨ |

**Table 17.2(a)** Relations between quality attributes of requirements (cont.)

| | Achievable/ Fea-sible/ Attainable | At the Right Level of Detail | Clear/ Precise/ Meaningful | Complete | Concise | Correct | Cross-Referenced | Design Independent | Electronically Stored | Executable/ Inter-pretable | Externally Consistent | Forward Trace-able | Implementation Independent | Internally Consis-tent |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Modifiable | | | | | | | + | | + | | | + | | |
| Necessary | | | | | | | | | | | | | | |
| Not Redundant | | | | | | | | | | | | | | |
| Organized | | | | | | | | | | | | | | |
| Prioritized/Ranked/Annotated by Relative Importance | | | | | | | | | | | | | | |
| Prioritized/Ranked/Annotated by Relative Stability | | | | | | | | | | | | | | |
| Prioritized/Ranked/Annotated by Version | | | | | | | | | | | | | | |
| Reusable | | | | | | | | | | | | | | |
| Traced/Backward Traceable | | | | | | | | | + | | | | | |
| Unambiguous | | | | | | | | | +? | +? | | | | |
| Understandable | | | + | | | | | | | + | | | | |
| Verifiable | + | | + | | | | | | | | | | | |

**+** = Strengthens the related attribute; **-** = Weakens the related attribute;  = No relation;
**-?** = May strengthen the related attribute;     **+?** = May weaken the related attribute

**Table 17.2(b)** Relations between quality attributes of requirements

| | Necessary | Not Redundant | Organized | Prioritized/Ranked/Annotated by Relative Importance | Prioritized/Ranked/Annotated by Relative Stability | Prioritized/Ranked/Annotated by Version | Reusable | Traced/Backward Traceable | Unambiguous | Understandable | Verifiable |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Achievable/Feasible/Attainable | | | | | | | | | | | |
| At the Right Level of Detail | | | | | | | | | | | |
| Clear/ Precise/Meaningful | | | | | | | | | + | | |
| Complete | + | | + | | | | | + | | | |
| Concise | | + | | | | | | | | | |
| Correct | + | | | | | | | + | | | |
| Cross-Referenced | | - | | | | | | | | | |
| Design Independent | | | | | | | | | | | |
| Electronically Stored | | | | | | | | | | | |
| Executable/Interpretable | | | | | | | | | | | |
| Externally Consistent | | | | | | | | + | | | |
| Forward Traceable | | | | | | | | | | | |
| Implementation Independent | | | | | | | | | | | |
| Internally Consistent | | + | +? | | | | | | | | |

**Table 17.2(b)** Relations between quality attributes of requirement (cont.)

| | Necessary | Not Redundant | Organized | Prioritized/Ranked/Annotated by Relative Importance | Prioritized/Ranked/Annotated by Relative Stability | Prioritized/Ranked/Annotated by Version | Reusable | Traced/Backward Traceable | Unambiguous | Understandable | Verifiable |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Modifiable | | | + | -? | -? | -? | | + | | | |
| Necessary | ▨ | | | -? | | | | + | | | |
| Not Redundant | | ▨ | | | | | | | | | |
| Organized | | | ▨ | | | | | | | | |
| Prioritized/Ranked/Annotated by Relative Importance | - | | +?/ -? | ▨ | -? | -? | | | | | |
| Prioritized/Ranked/Annotated by Relative Stability | | | +?/ -? | -? | ▨ | -? | | | | | |
| Prioritized/Ranked/Annotated by Version | | | +?/ -? | -? | -? | ▨ | | | | | |
| Reusable | | | | | | | ▨ | | | | |
| Traced/Backward Traceable | | | | | | | | ▨ | | | |
| Unambiguous | | | | | | | | | ▨ | | |
| Understandable | | -? | + | | | | | | +?/ -? | ▨ | |
| Verifiable | | | | | | | | | + | | ▨ |

**+** = Strengthens the related attribute; **-** = Weakens the related attribute;  = No relation;
**-?** = May strengthen the related attribute;     **+?** = May weaken the related attribute

### 17.4.1 Achievable/Feasible/Attainable

A requirement or an aggregate is achievable/feasible/attainable if and only if there exists at least one system design and implementation that correctly implements the requirement or all the requirements stated in the aggregate [5] at a definable cost [14].

There are no particular means utilized or recommended by RUP to ensure or to measure the achievability of all kinds of requirements involved in a development project at an early stage of a software development project. Only standard RUP activity that have relevance to ensure feasibility is *constructing architectural-proof-of-concept*, which helps with determining whether there exists, or is likely to exist, a solution that satisfies the architecturally-significant requirements, i.e. the activity does not cover all the requirements.

Yet, for an industrial company that launches a software project with considerable amount of investment, tight time-to-market constraints, and severe competition, it is vital to know: (i) whether it is technically possible to achieve the identified requirements; (ii) whether it is possible to achieve the requirements within the limitations imposed by time and budget. At ABB, we ensure the first concern by including the developers in the reviews of the requirements artifacts. In the Vision document, which comprises the high-level requirements, feasibility is not a high priority quality to achieve; yet if a feature or a need is determined to be infeasible with today's technical knowledge, it is noted during the review meeting to be negotiated with the stakeholders. If the stakeholders insist keeping the requirement in the Vision, the requirement is annotated with "not to be included in an immediate release". Accordingly, infeasible requirements may stay in the Vision but they are not traced forward to any use-cases or any lower level supplementary requirements, at least not until the next iteration or until a new technological improvement in the area. It is higher importance to achieve feasibility nature in the lower level requirements, i.e. in the use-case model, in the supplementary specifications documents, and in the use-case specifications, because the actual work is defined based on these artifacts. The first concern, i.e. technical feasibility, is achieved via reviews and including not only the stakeholders but also the software architect(s), and designer(s) in the reviews. The second concern, i.e. financial feasibility, is ensured by preparing a number of scenarios, and computing the project length and cost in the case of each scenario (see Table 17.3). The calculations are performed to view the worst possible case, the best possible case and three optimal cases that demonstrate the probable, very probable, and most probable proceeding of the project. These states differ from each other based on the number of weeks per iteration, number of developers that can be involved throughout the development process, number of use cases identified for the whole system, number of weeks to be spent on the development of each use case, and the characteristics of each developer during the development process. Our method is adapted from "Use-Case Points" of Gustav Karner [15], [20]. We ignore the weight of actors in the calculations. We consider our "supplementary requirements" as the technical factors, and include their effect in the calculations indirectly via the complexity of use-cases. Finally, we decide on complexity of use-cases by ranking them on a 5-point scale, 5 illustrating the highest complexity.

**Table 17.3** Financial feasibility scenarios

| No. weeks /iteration (3..8) | **L** | 8 | 7 | 6 | 5 | 5 |
|---|---|---|---|---|---|---|
| No. of developers | **N** | 2 | 3 | 4 | 4 | 4 |
| No of use cases | **K** | 15 | 15 | 15 | 15 | 15 |
| No. weeks/use case | **T** | 6 | 5 | 4 | 4 | 5 |
| Efficiency per user (0..1) | **U** | 0.5 | 0.6 | 0.5 | 0.5 | 0.7 |
| | | **Worst** | **Best** | **Optimal 1** | **Optimal 2** | **Optimal 3** |
| Developer effort (dev/iteration) | **E** | 8 | 13 | 12 | 10 | 14 |
| No of iterations | **M** | 11.25 | 6 | 5 | 6 | 5 |
| Project length (weeks) | **S** | 90 | 42 | 30 | 30 | 27 |
| Project costs (men* week) | **P** | 90 | 75 | 60 | 60 | 75 |

The computations are done by using the following formulas:
$E = U*L*N$
$M = T*K / (U*L*N) = T*K/E$
$S = T*K / (U*N) = M*L$
$P = S*N*U$

Upon completion of computations, we compare the existing situation in the project with the results of different scenarios, and determine whether the project is too optimistic about the number and content of the requirements to be fulfilled by the final product. Measurement of requirements attainability is done at least once by the beginning of a project. Depending on the volatility of the requirements and changes in the environmental factors for the team, it may be repeated by the beginning of each iteration.

## 17.4.2 Clear/ Precise/Meaningful

A requirement or an aggregate is clear/precise/meaningful if and only if (a) numeric quantities are used whenever possible, and (b) the appropriate levels of precision are used for all numeric quantities [5]. Keeping a proper scope in the sense of providing a definite amount of information, avoiding "motherhood" statements like "shall provide a continuous service", "shall ensure the highest system security" is vital for clarity [9].

Executable requirements are Clear requirements. A requirement that is written in a formally defined computer executable, rather than a natural language, provides a more precise description. For example, the MATLAB simulation of the automatic mode of our cluster type control system operation provided more precise and validated requirements input into the design phase of the development. Moreover, Unambiguousness enhances Clarity of requirements. If we take an example to ambiguous requirements from one of our Stressometer projects at ABB, initially what the marketing department desired was “The system shall have a fast computation time”. Such a requirement was rather vague and too general to work with for the development team. There were questions as “How fast is good enough?”, “We can have various configurations of the system, which configuration are we talking about? The speed of computation time differs depending if it is a monolithic system or a distributed one; if it is a measurement only or a full control system; etc.” Eventually, the requirement had to take a clearer format as “A full computation of the main functions, from the time the Base Measurement System TCP/IP signal is received until an output is issued (external communication not included), for a reversible mill single node flatness measurement system with 64 measurement zones, shall not be greater than 6.0 ms”. The problem with this requirement was not only that it was ambiguously stated but also that there was quite a lot vital information missing. Accordingly, we can infer that incompleteness may lead to unclear requirements; or in other words, completeness may increase the possibility of having clear requirements.

RUP supplies templates and examples, which provide structure and guidance for content of different types of requirements thus assists in preparing clear/precise/meaningful requirements. Further, it recommends review of these artifacts against checkpoints, which include criteria for fulfilling the attribute. Some examples to the checkpoints for requirements clarity are “It is clear how and when the use case's flow of events starts and ends” “It is clear who wishes to perform a use case” “The purpose of the use case is also clear.” “The actor interactions and exchanged information are clear.” “The use case model clearly presents the behavior of the system.” “The Introduction section of the use-case model provides a clear overview of the purpose and functionality of the system.”

SUP did not add any new means to what is already suggested by the general RUP. In our Stressometer projects, we did not measure requirements clarity directly but rather ensured a common agreement on existence of it through reviews by the stakeholders that constituted the domain experts, representatives of the external customers who bought the system, and representatives of the internal customers who used the requirements in the subsequent steps of the development lifecycle.

### 17.4.3 Complete

A requirement is complete if it is capable of standing alone when separated from other requirements and does not need further amplification [14]. An aggregate of requirements is complete if and only if (a) It includes all significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces. In particular, any external requirements imposed by a system specification should be acknowledged and treated. (b) It involves all responses of the software to all realizable classes of input data in all realizable classes of situations –including responses to both valid and invalid input values. (c) All figures, tables, and diagrams in the aggregate are fully labeled and referenced; all terms are defined; units of measure are provided [12]. (d) No sections are marked “To Be Determined (TBD)” [4]. (e) It covers all allocations from higher level [14]. (f) It must not include situations that will not be encountered or unnecessary capability features [25].

Organizing the requirements in a logical way, for example by following a template recommended by a specific process or by a standard, helps readers understand the structure of a functionality described in a use-case or in a standard requirements specification document, and makes it easier for them to identify if something is missing; hence, complete requirements. In similar sense, executing requirements via prototyping or via simulation during requirements analysis gives the stakeholders opportunity to validate the requirements as well as reflect on the missing ones, leading to a more complete set of requirements and more complete definition of requirements. Further, considering the condition (f) in the above definition, we can conclude that for requirements to be complete they have to be necessary. In other words, preparing an immense use-case model with “golden plating” use-cases omitting the necessary functionalities does not make the use-case model more complete. In fact, if we refer to the condition (e) in our definition, we determine that it is essential to establish backward traceability from the use-case model to the higher-level requirements specification, for example in ABB’s case, to the vision document that includes all the features and user needs of the prospective software system.

Focusing on user tasks instead of system functions during requirements elicitation avoids overlooking the requirements as well as including requirements that are not necessary [24]. To this end, using use-cases for capturing requirements are the ideal means. In addition, semi-formal nature of use-cases makes it easy for the stakeholders to read and understand a requirements document, and eventually, provide a feedback on the missing parts. Further, using a standard specification format, a template, can reveal omissions and prevent loss of requirements [10]. Moreover, iterative development of RUP brings about assessment of and maturing accordingly the quality of artifacts throughout the development life cycle. Every iteration results in an executable release, which facilitates identification of missing requirements that can be dealt with in the subsequent iterations.

During our projects at ABB, we considered completeness of requirements as one of the primary quality characteristics. SUP mainly utilized the strategies and tools provided by the RUP. Further, we ensured that the release produced by the end of an iteration was executed and continuously tested in an environment that simulated a typical final customer environment. Watching real life scenarios increased the interest level, the concentration, and the comprehension of the stakeholders thus opened new discussions, which led to identification of new, insufficiently described, or missing requirements. Even though, we highly depended on qualitative means as stakeholders' judgment, compliance with templates and guidelines, we also used the metrics listed in Table 17.4 in an attempt to quantify the maturity of completeness of different requirements artifacts by the end of each iteration:

**Table 17.4** Completeness metrics

| Metric | Related Requirement Artifact and Implications |
|---|---|
| Number of Use-Cases Traced Back to Features/Total Number of Use-Cases | Completeness of Use-Case Model. Low value indicates existence of use-cases without any origin. |
| Number of Supplementary Requirements Traced Back to Features/Total Number of Supplementary Requirements | Completeness of Supplementary Specifications. Low value indicates existence of non-functional requirements without any origin. |
| Number of Incompletes in a Use-Case Specification | Completeness of a Use-Case Specification. SUP recognizes incompletes such as TBD, TBS, Not defined, Not determined etc. as risk indicators for requirements completeness. SUP imposes minimizing the usage of incompletes, allows usage of such terms if and only if they are followed by information regarding when and by whom the incomplete portion will be attended, and considers it as high risk for the project if the number of incompletes were not decreased after two consequent iterations. |
| Number of Incompletes in a Supplementary Specification | Completeness of a Supplementary Specification document. Implications apply as in the case of use-case specifications. |

### 17.4.4 Concise

A requirement or an aggregate is concise if it is as short as possible without adversely affecting any other quality [5].

Generally, conciseness is measured in terms of size. Going overboard with completeness may easily increase the size, and consequently, jeopardize the conciseness of the requirement or the aggregate. A requirement, no matter in which format it is, must only state what is required and not how it shall be met in terms of design or implementation. Obviously, including such unnecessary information will bring about unnecessary increase in size hence, less concise. Besides, requirements can be stated at different levels of abstraction highly depending on the preferences of different projects. For example, [3] has defined two different use-case specification formats, namely casual and fully dressed, both of which are valid but may differ in size and thus, in conciseness. Finally, in order to increase understandability, requirements specifiers often use redundancy, which is not an error itself [12, 4, 5], yet can easily lead to problems in achieving other qualities one of which is conciseness.

RUP does not provide any particular assistance for conciseness. During our projects at ABB, we were mainly concerned about the size of the use-case models increased with the number of use-cases, number of included use-cases, number of extending use-cases, and number of each type of relations. Besides, writing extensive use-cases by keeping a low level of abstraction was a topic discussed at almost every review meeting. Yet, the first two projects proved that conciseness of the use-case model or the conciseness of use-case specifications did not constitute a high risk for the project or for the quality of the final product. Accordingly, it was not addressed in the subsequent projects by the SUP.

### 17.4.5 Correct

A requirement is correct if it accurately describes a functionality to be delivered [24]. An aggregate is correct if and only if every requirement stated therein is one that the software shall meet [12].

As mentioned earlier in Sect. 17.4.3, executing requirements enables the stakeholders to validate the specified requirements, thus, to ensure the correctness of the requirements. Externally and/or internally inconsistent requirements hinder establishing correctness for it can be difficult to know which one of the conflicting requirements is correct if there is any. Further, regarding our definition of correctness it is explicit that a requirement or an aggregate of requirements is always correct if it is necessary. Finally, based on the relations both with external consistency and with necessity, we can infer that a requirement is correct if it can be traced back to its source at a higher-level –naturally, on the condition that the higher-level requirement itself is correct.

RUP suggests involvement of end users in the requirements review meetings only *if possible*. It provides guidelines for test case generation from the requirements, but leaves the preparation of the test cases until the implementation work is scheduled for them. It does not require review of test cases either. By recommending usage of UML, and tools that do not provide any facilities for internal or external consistency checks of requirements, RUP hinders achievement of correctness. Yet, the iterative nature of the process enables continuous learning and improving throughout the development life cycle, and accommodating corrective changes in requirements as a result of such learning, any time during the project. On the other hand, we believe it is only the end users who can determine the correctness of user requirements. Accordingly, SUP process imposes involvement of representatives of both external users that work at the customer site and the internal users that customize, install, and maintain the system, in the review of use-case model, use-case specifications, and supplementary specifications. Further, according to the SUP, test cases should be derived from the requirements and parallel to the specification of the requirements so that any errors in the requirements can be revealed and corrected before the design activities start. The test cases should be reviewed by the requirements reviewers. Finally, continuous execution and testing of incremental releases in an environment that simulated a typical final customer site provides continuous and realistic feedback to the development team about the requirements that conflict with customer expectations.

### 17.4.6 Design Independent

A requirement or an aggregate is design independent if and only if there exists more than one system design and implementation that correctly implements the individual requirement or the requirements in the aggregate [5].

RUP provides only assistance for design independence via brief information about how to distinguish “what” from “how” in the use-case model guidelines. Templates and examples provided are also useful but not sufficient. In order to ensure design independence of the requirements, SUP imposes including the software architect and designers in the review of requirements artifacts so that they can point out those details that may limit their ability to consider alternative design possibilities in order to synthesize the most optimal one.

### 17.4.7 Externally Consistent

An aggregate is externally consistent if and only if no requirement stated therein conflicts with any already baselined project documentation [5].

Traditionally, external consistency is defined in terms of compliance with the preceding documents [4] and in most of the cases, those that include higher-level requirements [24]. Yet, considering the importance of configuration and change management during the whole lifecycle of software development, especially when following an iterative and incremental approach, at ABB we preferred to adopt a definition that emphasizes the importance of promoted baselines. In this way, we aimed to: (i) handle inconsistencies as a part of our formal change management, (ii) extend the context of external inconsistency to include project artifacts other than the high-level requirements documents such as project plan, a baselined release from the previous iteration, etc. Traceability is the only characteristic that we have experienced to affect the external consistency. If there is a link from every low-level software requirement, for example a use-case in the use-case model, a supplementary requirement in a supplementary specifications document, to a higher-level requirement, for example a feature or a need in the vision document, i.e., backward traceable –then the aggregate including these requirements is externally consistent with the high-level requirements. In the same manner, if there is a link from each requirement to at least one lower-level requirement or to a further development artifact such as a sequence diagram, a class diagram, a test case, i.e. forward traceable –then the aggregate including these requirements is in agreement with the lower-level documentation thus externally consistent with the particular documentation.

RUP provides well-defined requirements management activities, which includes detailed guidance for establishing and maintaining implicit and explicit traceabilities to and from requirements at different levels, and for managing changing requirements, and change management activities. Further, it presents Requisite Pro to facilitate its requirements management practice. Yet, as being a UML-based software development process, both RUP and SUP suffer inter- and intra-model inconsistencies. For example, during our projects at ABB we experienced difficulties in keeping the use-case models of different sub-systems consistent with each other. Eventually, we decided to use one common use-case model, which was in the end too large to manage. Besides, without any support for automatic consistency checks from Rational Rose, it required considerable amount of manual effort to ensure consistency even among the elements of the same model. Similar situation applied in preserving the existing consistencies between different models during model transformations, for example while reflecting changes in the implementation model to the design model and eventually to the relevant use-case, actor, or portion of the specification of a use-case in the use-case model.

### 17.4.8 Forward Traceable

A requirement or an aggregate is forward traceable if and only if it is written in a manner that facilitates the referencing of the requirement or each individual requirement of the aggregate in future development or enhancement documentation [5, 12].

Common methods used for explicit traceability includes numbering every paragraph hierarchically, numbering every requirement with a unique number, using a convention to indicate a requirement and using a tool to extract and uniquely number all sentences that comply with the particular convention [4]. To this end, it will be much easier to achieve forward traceability if the requirements are electronically stored by using a tool that facilitates numbering and/or extracting sentences according to a defined convention. Besides explicit traceability, there is certain amount of traceability implicit in every development process [21]. For example in the case of projects that follow RUP, such traceabilities are achieved via: (i) Naming Conventions, (ii) The construction of mappings between the models, (iii) Relationships between the model items themselves, (iv) The creation of different perspectives illustrating how the elements of one model satisfy the demands implicit in the elements of another model. Some of these are easier to fulfill by electronically storing the requirements in a tool that has the UML meta-model defined in it, such as Rational Rose. A detailed discussion about forward traceability can also be found in Chap. 5.

One of the best practices with RUP is managing requirements [18]. As a major part of the requirements management, RUP puts specific emphasis on establishing traceabilities among different levels of requirements and from the requirements to the rest of the software development artifacts. It provides information about and guidance for various possible traceability strategies, most common of which are No Use-Case Model; Use-Case Model Only; Features Drive the Use Case Model; The Use-Case Model is an interpretation of the Software Requirements Specification; The Use Case Model reconciles multiple sets of traditional software requirements [21]. Further, RUP facilitates building and utilizing these strategies via tool support. For example, it recommends Rational RequisitePro as a tool for defining, capturing, and tracking the traceability links. Whereas, on the other hand, as being a UML-based software development process, RUP employs a "use-case driven approach", meaning use cases that can only describe the functional requirements are the basis for the entire development process [18]. It describes the activities to move from specifications of use-cases to the realization of use-cases subsequently to the implementation and testing of use-cases, in detail. It provides no similar assistance for the non-functional requirements, which must also be provided to the customer in the final product together with the functionality thus, must be designed and tested together with the functionality.

**Table 17.5** Forward traceability metrics (1 of 2)

| Metric | Related Requirement Artifact and Implications |
|---|---|
| (Number of Features Traced to Use-Cases) + (Number of Features Traced to Supplementary Specifications )/Total Number of Features | Forward Traceability of Vision. This metric is mainly used before lower level requirements specifications are prepared. Low value may suggest unsatisfactory quality in various areas. It directly illustrates poor forward traceability from the high-level requirements to the lower level ones. In addition, it may imply inconsistency between the high-level requirements and the lower level requirements. It may indicate incorrect requirements at the lower level. It may signal incompleteness unless the Vision includes requirements to be fulfilled in the long-term, as it was the case in our projects. |
| (Number of Features Traced to Use-Case Specification Sections) + (Number of Features Traced to Supplementary Requirements)/Total Number of Features | Forward Traceability of Vision. This metric can be used after starting to prepare the lower level requirements specifications. The implications are of the same nature as described regarding the previous metric; yet it provides results that are more accurate thus, facilitates identifying the root causes. |
| Number of Use-Case Specification Sections Traced to Sequence Diagrams / Total Number of Use-Case Specification Sections to be Traced to Sequence Diagrams<br><br>(Previously: Number of Use-Case Specification Flows Traced to Sequence Diagrams/Total Number of Use-Case Specification Flows) | Forward Traceability of Use-Cases to the Design Model. Low value indicates low traceability to the sequence diagrams. All development cases prepared according to the SUP principles imposes one-to-one relation between the flows of a use-case specification and of a use-case realization specification. Yet, as it was observed in some projects, it might be easier, less redundant, more concise, and more understandable to describe the design of more than one flow in the same sequence diagram. Besides, due to the iterative nature of the projects, not all flows might be considered for a design in a particular iteration. Further, occasionally, we encountered the need to design use-case specification sections other than the flows via sequence diagrams. Accordingly, we adjusted our initial metric. |

**Table 17.5** ***(cont.)*** Forward traceability metrics (2 of 2)

| Metric | Related Requirement Artifact and Implications |
|---|---|
| Number of Use-Case Specifications Traced to Class Diagrams/Total Number of Use-Case Specifications to be Traced to Class Diagrams<br><br>(Previously: Number of Use-Case Specification Flows Traced to Class Diagrams/Total Number of Use-Case Specification Flows) | Forward Traceability of Use-Cases to the Design Model. Low value indicates low traceability to the class diagrams thus, eventually quality problems in the code. In the very first project, it was decided to illustrate each use-case flow with one class diagram in the design model. By doing so, we experienced difficulties in keeping the diagrams consistent, and the design model and the use-case realization documents concise. Accordingly, we adjusted the development case and our initial metric. |
| Number of Use-Case Scenarios Traced to Functional Test Cases/Total Number of Use-Case Scenarios to be Traced to Test Cases | Forward Traceability of Use-Cases to the Test Model. Low value indicates low traceability to the test cases thus, insufficient testing. |

RUP does not recognize any explicit link between the use-cases and the supplementary, i.e. the non-functional, requirements, either. In brief, even though some of the traceability strategies include links from the Supplementary Specifications to the subsequent artifacts, there exists no particular RUP guidance for how to establish such traceabilities.

During our projects at ABB, we used "Features Drive the Use Case Model", which is the default strategy recommended by the Rational Unified Process. The Use-Case Model and Supplementary Specifications form a complete software requirements specification. Features are documented in the Vision Document and are traced to use cases. If they are not reflected in the Use Case Model then they are traced to supplementary requirements in the Supplementary Specifications [21]. Accordingly, we handled the tracing from features to the use-case sections and to the supplementary requirements, from use-case specifications to the use-case realizations, to the functional test cases and eventually to the test procedures whereas, we managed the linkage from the supplementary requirements to the use-case realizations and to the test procedures in an ad hoc manner. For example, we could easily point at which test case realized which part of which use-case in the test model; whereas, supplementary specifications were directly entered into the test procedures, and in most of the cases to a degree depending on the initiative of the test designer. Table 17.5 includes the forward traceability metrics we used in our projects run according to SUP:

### 17.4.9 Internally Consistent

An aggregate is internally consistent if and only if no subset of individual requirements stated in it conflict [12]. The same term is used for the same item in all requirements of the aggregate [14].

When an aggregate is not organized, it may be difficult to identify the inconsistencies [14]. Therefore, it should be preferred to organize the requirements according to a standard or by using a template recommended by the process used. In addition, we often use redundancies in documentation in order to increase the readability, while causing a risk for internal inconsistency. When altering one occurrence of a requirement we may forget to do so with other occurrences; hence, internal inconsistency; yet, we can decrease the risk by using cross-references. Finally, better consistency can be achieved with executable requirements depending on whether the tool used has a consistency check facility and how sophisticated the facility is. For example, [7] describes a consistency algorithm for the live sequence charts of the "play engine" [8] mentioned earlier. By adopting such an algorithm in the "play engine", it is aimed to automatically detect inconsistencies in a specification, enable a user to track the reason for inconsistencies via play out, suggest a consistent scenario with "good" order of events whenever there is one, and avoid abnormal abortion of play outs due to inconsistencies [8]. [4] identifies four types of inconsistencies: (i) Conflicting behavior; (ii) Conflicting terms; (iii) Conflicting characteristics; (iv) Temporal inconsistency.

RUP recommends developing a Glossary during the early phases of a project, in order to ensure consistent usage of the terms throughout the whole development life cycle hence, assistance to avoid conflicting terms. Even though, as our experiences showed, it might occasionally be difficult to keep the Glossary itself consistent, it is helpful to have one Glossary. On the other hand, both RUP and SUP rely highly on the reviews for detecting the conflicting behavior, conflicting characteristics, and temporal inconsistencies. Tracing such inconsistencies manually in a large, evolving use-case model or supplementary specifications can be hard and error prone.

### 17.4.10 Modifiable

An aggregate is modifiable if and only if its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style [12].

Our experiences from software development projects at ABB have illustrated high importance of requirements modifiability for: (i) requirements change; (ii) concerns other than but affecting software requirements change; (iii) requirements evolve; (iv) requirements can be wrongly stated due to various inadvertent reasons. In such cases, it is easier to identify and subsequently, apply the modifications if (i) the requirements are organized in a coherent and easy-to-use way; (ii) redundancy is kept to minimum; (iii) cross-references are used where necessary; (iv) the requirements are uniquely labeled to ease both forward and backward traceabilities; and (iv) the requirements are electronically stored. On the other hand, ranking requirements by importance, stability, or version may inhibit modifiability if the aggregate is organized according to the ranking instead of according to some logical grouping recommended by a standard, or by a template provided by the process followed, or chosen by the project in order to keep the related concerns together and unrelated ones separate.

RUP iterative life cycle allows changes to the requirements at almost any point in the development. Besides, since development is done incrementally, it is easier to detect the effects, estimate the cost of, and eventually carry out a suggested modification. RUP distinguishes between different types of requirements, and provides templates for organizing each type of requirements. It recommends using Rational Rose to electronically store the use-case models and diagrams, and supplies specification templates ready to be used in Microsoft Word, Adobe FrameMaker, and HTML formats. SUP inherits the advantages of the generic RUP as described above.

### 17.4.11 Necessary

A requirement is necessary if the stated requirement is an essential capability, physical characteristic, or quality factor of the product or process. If it is removed or deleted, a deficiency will exist, which cannot be fulfilled by other capabilities of the product or process [14].

One common way suggested by the literature to decide on the necessity of a requirement is to trace the requirement back to its origin, for example in the case of our projects, which use RUP as the software development process, to trace a use-case back to a feature or a need in the vision. If it cannot be traced it may not be necessary. All definitions of necessity introduce the characteristic as a primary condition for a requirement to qualify for being included in the final product [11], [14], and [24]. Yet, depending on the scheme we use, ranking a requirement for importance may conflict with the necessary nature of the requirement. For example, [12] suggests ranking requirements based on a degree of necessity that distinguishes classes of requirements as essential, conditional, and optional. According to the scheme, essential requirements are those that must be provided for the final product to be accepted, hence necessary requirements. Whereas, conditional requirements are those that would enhance the final product but would not make it unacceptable if they are absent, and optional requirements are those that may or may not be worthwhile, hence not necessary requirements.

**Table 17.6** Necessity metrics

| Metric | Related Requirement Artifact and Implications |
|---|---|
| Number of Use-Case Sections Traced Back to Features/Total Number of Use-Case Sections | Necessity of Use-Cases. A value other than 1 indicates existence of not required use-case flows, special requirements, post, or pre-conditions. |
| Number of Supplementary Requirements Traced Back to Features/Total Number of Supplementary Requirements | Necessity of Supplementary Requirements. A value other than 1 indicates existence of unnecessary non-functional requirements. |

RUP describes specific and detailed activities for requirements elicitation. It suggests methods to follow for identifying what the stakeholders require. It enhances the assistance with guidelines where appropriate. It also provides related checkpoints to be adopted at the review meetings. As a part of its requirements management practice, RUP suggests various traceability strategies, which provide guidance on keeping links between requirements at different levels. Finally, the iterative nature of RUP allows continuous learning and improving the requirements throughout the development life cycle. SUP requires involvement of representatives of all types of stakeholders in the requirements elicitation and identification process. Besides, it uses well-defined traceability procedures between high level and lower level requirements. Accordingly, the risk with identifying requirements that do not contribute to the satisfaction of some customer needs is minimized. SUP also suggests collecting the metrics identified in Table 17.6 and discussing the results in the relevant review meetings.

### 17.4.12 Organized

An aggregate is organized if and only if its contents are arranged so that readers can easily locate information and logical relationships among adjacent sections are apparent [5].

RUP recommends organizing the functional requirements using use-cases. Instead of a traditional bulleted list of requirements, RUP suggests organizing them in a way that tells a story of how someone may use the final product [18]. Further, it provides templates complemented with guidelines and examples to assist in documenting the needs and features in Vision document, and lower level requirements in Use-Case Model survey, Use-Case Specifications, and Supplementary Specifications, in an organized manner. SUP adopts generic RUP means, with minor adaptations according to the ABB instructions. The "organized" nature of requirements is ensured via the checkpoints at the review meetings.

### 17.4.13 Prioritized/Ranked/ Annotated

A requirement is prioritized/ranked/annotated by relative importance if the requirement is assigned an implementation priority to indicate how essential it is to include it in a particular product [24]. An aggregate is prioritized/ranked/annotated by relative importance if each requirement in it has an identifier to indicate the importance of that particular requirement [12].

A requirement is prioritized/ranked/annotated by relative stability if the requirement is assigned an identifier to indicate the stability of the particular requirement [5]. An aggregate is prioritized/ranked/annotated by relative stability if each requirement in it has an identifier to indicate the stability of that particular requirement [12]. A requirement or an aggregate is prioritized/ranked/annotated by version if a reader can easily determine whether the particular requirement or which requirements of the aggregate will be satisfied in which version of the prospective product [5].

The characteristics that may hinder from achieving prioritized/ranked/annotated by relative importance are those that are related to the organization of requirements in the aggregates. If it is preferred by the project to organize the requirements to be modifiable, or to rank by relative stability, or to rank by version, then prioritization by relative importance cannot be performed in the structure of the aggregate. Yet, by extracting the requirements into another means such as a workbook or a database, the project can still rank the requirements by relative stability and by version without adversely affecting the ranking by relative importance nature of the original aggregate. Besides, if an aggregate is organized by following a standard or a template provided by the process adopted, it cannot be organized according to the ranking of its requirements by relative importance. Similar situations also apply to achieving prioritization of requirements by relative stability and by version. Finally, if a requirement is necessary, it represents functionality, a capability, a physical characteristic, or a quality factor essential for the final product; therefore, it cannot be ranked to a level that degrades its necessity.

Traditionally, it is suggested to establish ranking according to relative importance, stability, or version in the organization and the structure of an aggregate. Accordingly, an aggregate organized to be modifiable would have a negative impact on this characteristic. However, in SUP, it is suggested to extract the requirements from the aggregate into a workbook, execute the rankings based on the attributes chosen beforehand, and eventually, sort and save the matrix in separate datasheets per each ranking. In this way, the project could keep the original organizations of the use-case model, supplementary specifications, and the vision while at the same time it could refer to the rankings when needed, for example for (re-)planning by the beginning of an iteration. In this way, it was also possible to generate different combinations of rankings in summary tables depending on the aim of the planning. For example, if it was decided that we should plan the iteration to develop the use-cases with the critical benefits, and to stabilize the architecture, then it would be necessary to view the matrix sorted first by relative importance and then by stability on one worksheet. Table 17.7 illustrates a portion of a use-case matrix resulted from such combined ranking during one of our projects at ABB.

As a part of its Requirements Management activities, the generic RUP recommends defining the attributes to be tracked for each type of requirement. Examples to such attributes are Stability, Effort to implement, Risk to the development effort, etc. It provides detailed guidelines how to identify, store, and review the attributes. Further, it supplies a tool mentor to facilitate these activities via RequisitePro, which enables defining attributes for different types of requirements, storing the requirements together with the attribute values, and retrieving and organizing the requirements by attribute values via filtering or sorting in views. In conclusion, RUP excels the “prioritization” related quality attributes by delivering the means for sophisticated groupings of requirements.

**Table 17.7** Example use-case attribute matrix

| Use-Case No | *Status* | *Benefit* | *Effort* | *Technical Risk* | *Architectural Impact* | *Stability* | *Priority* | | Scheduled for the Current Iteration | Responsible Party |
|---|---|---|---|---|---|---|---|---|---|---|
| *UC-20* | Proposed | Critical | High | Medium | Extends | High | High | | Yes | Christer |
| *UC-23* | Proposed | Critical | High | Medium | None | Medium | High | | Yes | Olle |
| *UC-51* | Proposed | Critical | Medium | Medium | None | Medium | High | | Yes | Olle |
| *UC-21* | Proposed | Critical | Medium | Medium | None | High | Medium | | Yes | Christer |
| *UC-49* | Proposed | Critical | Medium | Medium | None | High | Medium | | Yes | Christer |
| *UC-33* | Proposed | Critical | Medium | Low | None | High | Low | | Yes | LEM |
| *UC-55* | Proposed | Critical | Medium | Low | None | High | Low | | | |

For a more detailed survey on requirements prioritization and requirements prioritization techniques, readers should also refer to Chap. 4 in this book.

**Table 17.8** Backward traceability metrics

| Metric | Related Requirement Artifact and Implications |
|---|---|
| Number of Use-Cases Traced Back to Features/Total Number of Use-Cases | Backward Traceability of Use-Case Model. A value other than 1 indicates poorly traced use-cases. It also suggests existence of use-cases without any origin. |
| Number of Use-Case Specification Traced Back to Features/Total Number of Use-Case Specification Sections | Backward Traceability of a Use-Case. A value other than 1 indicates poorly traced use-cases. It also suggests existence of use-case sections, such as pre or post conditions, or special requirements, without any origin. |
| Number of Supplementary Requirements Traced Back to Features/Total Number of Supplementary Requirements | Backward Traceability of Supplementary Specification. A value other than 1 indicates poorly traced supplementary specification and supplementary requirement. It also suggests existence of supplementary requirements without any origin. |
| Number of Sequence Diagrams Traced Back to Use-Case Specification Sections/Total Number of Sequence Diagrams | Backward Traceability of Design Model to the Use-Cases. A value other than 1 indicates poorly traced sequence diagrams. It signifies existence of design elements without any origin. It may also suggest inconsistencies between what the end customer expects and what is being developed. |
| Number of Class Diagrams Traced Back to Use-Case Specification Sections/Total Number of Class Diagrams | Backward Traceability of Design Model to the Use-Cases. A value other than 1 indicates poorly traced class diagrams. It signifies existence of design elements without any origin. It may also suggest inconsistencies between what the end customer expects and what is being developed. |
| Number of Functional Test Cases Traced to Use-Case Scenarios/Total Number of Functional Test Cases | Backward Traceability of Test Model to the Use-Cases. A value other than 1 indicates poorly traced test cases. It suggests existence of test cases without origin. It also signifies that necessary functionalities were not tested and/or extra functionality was implemented without informing the requirements team first. |

### 17.4.14 Traced/Backward Traceable

A requirement or an aggregate is traced/backward traceable if the origin of the requirement or of each requirement of the aggregate is clear [5].

The discussion about the explicit and implicit traceability and the influence of electronically stored characteristics on the forward traceability (see Sect. 17.4.8) also applies to the backward traceability. Further, the discussion about the support by the generic RUP and SUP for establishing traceability in the same section should also be considered here. Yet, what differs is the metrics we used in our projects in order to measure the degree of backward traceability achieved thus, detect possible risks and flaws in the projects:

### 17.4.15 Unambiguous

A requirement or an aggregate is unambiguous if different readers with similar backgrounds would be able to draw only one interpretation of the requirement [9, 24] or of each requirement in the aggregate [12]. As discussed in detail earlier in Chap. 11, natural language is inherently ambiguous. In order to decrease the ambiguity thus increase the unambiguousness, one can use more deterministic methods and languages with well-defined semantics, such as state machines, predicate calculus, prepositional calculus, petri nets. Most of these methods and languages are supported by software tools that can automatically detect lexical, syntactic, and semantic errors. Accordingly, electronically stored and/or executable requirements may constitute less ambiguity.

RUP is a UML-based software development process. UML has limited notation to express different types of requirements. In fact, it only helps visualizing the actors and the use-cases that constitute the lower level functional requirements. UML does not provide support for detailing the use-cases. Even though RUP suggests using sequence diagrams to show how an actor interacts with a use-case, or using activity diagrams or state charts to describe a single use-case in order to formalize use-cases, the common means to describe use-cases is Natural Language. In addition, use-cases are not the only requirements of a software product. RUP uses Vision documents for specifying the high-level requirements, and Supplementary Specifications to describe the non-functional requirements. Both Vision and Supplementary Specifications are created also by using Natural Language. Natural Language has inherent ambiguity. Yet, RUP defines a common vocabulary in order to decrease ambiguity among team members. It recommends checkpoints to be used during the review of requirements specification documents. Such checkpoints are too general and insufficient to ensure a satisfying level of unambiguousness in the requirements artifacts.

Active participation of all types of stakeholders in the elicitation and review of the requirements and preparation of test cases parallel to the preparation of the use-cases are the main means that SUP recommends in order to decrease the ambiguity in the requirements. It recognizes a list of weak phrases that may cause uncertainty and lead to multiple interpretations, such as flexible, fault tolerant, adequate, as appropriate, maximize, minimize, at a given time, up to etc., and options that give the developers freedom to satisfy the related requirement by following more than one way such as can, may, optionally etc. During review meetings, checks are done in order to detect usage of these words. In addition, the following metrics in Table 17.9 are used to measure the ambiguity level in a specification.

**Table 17.9** Unambiguousness metrics

| Metric | Related Requirement Artifact and Implications |
|---|---|
| Number of Weak Phrases + Number of Options in a Use-Case Specification | Unambiguousness of a Use-Case Specification. Values other than 0 indicate ambiguity in the specification. |
| Number of Weak Phrases + Number of Options in a Supplementary Specification | Unambiguousness of a Supplementary Specification. Values other than 0 indicate ambiguity in the specification. |

### 17.4.16 Understandable

A requirement or an aggregate is understandable if all classes of readers can easily comprehend the meaning of the requirement or all requirements in the aggregate, with a minimum of explanation [5].

Naturally, an unambiguous requirement or aggregate is clearer/more precise and more meaningful thus more understandable. On the other hand, if the unambiguousness is achieved by using formal notations, understandability of the requirements by non-technical stakeholders will decrease. In addition, redundancy increases readability thus may increase understandability of requirements. Moreover, it is easier to comprehend behavior by seeing it in action than by reading about it in a document. Accordingly, executability/interpretability of requirements enhances the understandability of them. Further, organizing the requirements according to a standard or by using a template recommended by the process followed or according to another logical grouping accepted by the project will increase the understandability of the requirements. The iterative nature of the RUP process enables continuous learning and improving throughout the development life cycle. Every iteration results in an executable release, which improves effective understandability. Besides, our experiences have illustrated that organizing functional requirements by using use-cases leads to greater completeness and better understanding of the requirements hence, support by RUP for better understandability of requirements. In addition, RUP provides templates to organize the high-level requirements and non-functional requirements in logical groupings.

In the projects that follow SUP, since all types of stakeholders, i.e. representatives of end users, representatives of actual buyers of the system, architect, and designer of the system, those who do the installation and maintenance of the final product, and take part in the review of the requirements problems with understanding the requirements can easily be revealed and solved.

## 17.5 Conclusions

The Stressometer products have been providing rolling mills with accurate online control of the flatness of cold rolled strips for more than 30 years. As the early generation, PLC-based Stressometer systems have been migrated to a Java-based platform, ABB has faced a need for change in the way it works to continue providing value to its customers and ensuring customer satisfaction in a controlled manner. Accordingly, it adopted RUP in an agile fashion mainly by maintaining active and heavy involvement of stakeholders that include both external and internal customers, preparing the test cases before coding, and continuously testing during development. The resultant development process namely SUP has been applied in three projects and has presented satisfying results regarding the achievement of “good quality” requirements.

The Stressometer projects received major gains from the disciplined nature of RUP in *traceability*, *completeness*, and *necessity* attributes via well-defined traceability strategies that were provided as a part of thorough requirements management. Templates and examples together with the associated guidelines and checkpoints helped to achieve the *organized*, *modifiability*, and *clarity* qualities in the requirements. Further, iterative nature of RUP gave a considerable support in achieving *completeness*, *modifiability*, and *understandability*. On the other hand, standard RUP means was insufficient for ensuring *achievability*. Accordingly, in SUP we introduced financial feasibility scenarios and imposed active communication of the development team with the rest of the stakeholders including the external as well as the internal customers.

Tool support by RUP to “ease” achieving the *prioritization related attributes* was found not enough value providing to invest time and effort. Instead, in the SUP, simpler guidelines to follow were defined and usage of worksheets was suggested. Active involvement of the end users in the creation and review of the requirements artifacts, as imposed by SUP, proved to be an invaluable means to achieve *correctness*, and *unambiguousness*.

*Completeness* and *correctness* qualities were further excelled by producing an executable release by the end of every iteration, and allowing the external and the internal customers to interact with it in an environment, which simulates a typical final customer site, mainly as a part of continuous testing.

Including the architects and the designers in the review of requirements artifacts even though it is not required by the standard RUP procedures ensured *design independence* of the requirements. Preparing the test cases early in the development, parallel to the requirements specification, and having them reviewed by the stakeholders supported achievement of most of the quality attributes; however, the main benefits were perceived regarding the requirements *correctness*.

The projects had to put considerable amount of manual effort and had to maintain a close communication within the development team in order to ensure *internal* and *external consistency*. This was not a satisfactory practice and accordingly, was considered to be improved in the future.

## References

1. Basili V, Weiss D (1981) Evaluation of a software requirements document by analysis of change data. In: Proceedings of 5th IEEE International Software Engineering Conference, March 9-12, 1981, San Diego, California, United States, pp.314−323
2. Boehm B (1981) Software engineering economics. Prentice Hall: Englewood Cliffs, New Jersey
3. Cockburn A (2001) Writing effective use cases. Addison-Wesley: Boston, Massachusetts
4. Davis AM (1993) Software requirements: Objects, functions, and states. Revision. PTR Prentice Hall: Englewood Cliffs, New Jersey
5. Davis A, Overmyer S, Jordan K, Caruso J, Dandashi F, Dinh A, Kincaid G, Ledeboer G, Reynolds P, Sitaram P, Ta A, Theofanos M (1993) Identifying and measuring quality in a software requirements specification. In: Proceedings of 1st International Software Metrics Symposium, Baltimore, Maryland, United States, pp.141−152
6. Grieskamp W, Lepper M (2000) Using use cases in executable Z. In: Proceedings of IEEE Conference on Formal Engineering Methods, September 4-7, York, England, pp.111−120
7. Harel D, Kugler H (2002) Synthesizing state-based object systems from LSC specifications. International Journal of Foundations of Computer Science (IJFCS), 13(1): 5−51
8. Harel D, Marelly R (2002) Specifying and executing behavioral requirements: The play-in/play-out approach. Technical Report MCS01-15, The Weizmann Institute of Science
9. Harwell R, Aslaksen E, Mengot R, Hooks I, Ptack K (1993) What is a requirement? In: Proceedings of 3rd International Symposium of the NCOSE, July 26-28, Arlington, Virginia, United States, 1: 17−22
10. Hooks I, Farry K (2000) Customer-centered products: Creating successful products through smart requirements management. American Management Association: New York, New York
11. Hooks I (1993) Writing good requirements. In: Proceedings of 3rd International Symposium of the INCOSE, July 26-28, Arlington, Virginia, United States, 2: 197−203
12. IEEE (1998) IEEE Recommended practice for software requirements specifications, IEEE Std. 830-1998
13. (2004) http://www.ilogix.com/fs prod.htm. Last accessed: 2004-09-09
14. Kar P, Bailey M (1996) Characteristics of good requirements. In: Proceedings of 6th International Symposium of the NCOSE, 7-11 July, Boston, Massachusetts, USA 2: 284−291
15. Karner G (1993) Metrics for objectory. Diploma thesis, University of Linköping, Sweden, No LiTHIDA-Ex-9344:21
16. Kruchten P (1999) The rational unified process. Addison-Wesley: Reading, Massachusetts
17. Oberg R, Probasco L, Ericsson M (2000) Applying requirements management with use-cases. Rational Software White Paper, Technical Paper TP505 (Version 1.3), http://www.pureproject.com/reqs_mgmt_usecases.htm
18. Rational Sofware Corporation (2002) Rational unified process software. Version 2002.05.00

19. Rosenberg L, Hyatt L, Hammer T, Huffman L, Wilson W (1998) Testing metrics for requirements quality. In: Proceedings of 2nd International Software Quality Week, 9-13 November, Brussels, Belgium. Access on 13th December 2004. http://satc.gsfc.nasa.gov/support/
20. Schneider G, Winters J P (1998) Applying use-cases - A practical guide. Addison-Wesley: Reading, Massachusetts
21. Spence I, Probasco L (2000) Traceability strategies for managing requirements with use cases. Rational Software White Paper, Access on 13th December 2004. http://www.isk.kth.se/proj/2003/6b3403/sa3/www/RationalUnifiedProcess/papers/traceability.htm
22. (2004) http://www.mathworks.com/. Last accessed: 2004-11-12
23. (2004) http://www.standishgroup.com/. Last accessed: 2004-11-10
24. Wiegers KE (1999) Writing quality requirements. Software Development Magazine, May, http://www.sdmagazine.com/
25. Wilson WM (1997): Writing effective requirements specifications. In: Proceedings of 9th Annual Software Technology Conference, 27 April–2 May, Salt Lake City, Utah, USA
26. Yilmaztürk N (2003) RE in flatness measurement and control systems development at ABB. In: Proceedings of 11th IEEE International Requirements Engineering Conference, 8-12 September, Monterey Bay, California, USA, pp. 293

## Author Biography

Nur Yilmaztürk is a computer scientist in the Software Architecture and Processes (SWAP) Group of the Automation Technologies department at ABB Corporate Research. She worked in different roles including process and project management during development of new generation Stressometer systems at ABB. Her research interests involve software development processes mainly model-driven development, and iterative and incremental development, conceptual modeling, architectural visualization and analysis, and object oriented technologies. She received her bachelor’s degree in mathematical engineering from Istanbul Technical University (ITU) in Turkey, and her MSc and PhD in computer science from the University of Manchester Institute of Science and Technology (UMIST) in UK.