Verbal Use Case Specifications for Informal Requirements Elicitation

Eliezer Kantorowitz

Verbal Use Case Specifications for Informal Requirements Elicitation

Eliezer Kantorowitz

Technion – Israel Institute of Technology
Technion City
3200 Haifa, Israel

kantor@cs.technion.ac.il

Abstract. Constructing a software system from poor specifications may necessitate costly repairs. We introduce the notion "satisfactory specifications" for quality specifications that do not require costly repairs. Satisfactory specifications may be produced by a Computer Supported Collaborative Work (CSCW) team incorporating all the relevant experts and the stakeholders. It is suggested that the CSCW team develops use case specifications, where its expertise is especially useful. Specifying in a natural language understood by all team members facilitates needed intensive cooperation between different team members. Compared to specifications formulated in formal terminology, verbal formulations in domain language represent a textual visualization. Translating the verbal specifications into formal UML diagrams provides a further graphical visualization. It is suggested that each specification is provided with a separate example for each kind of the possible situations. These examples may clarify meaning of poorly formulated specifications, facilitate identifying faults in formal specifications and employed for software debugging.

Keywords: Satisfactory specifications, Computer Supported Cooperative Work, CSCW, UML, Use Case, Software Specifications, Software Requirements, Requirements elicitation, Human Factors, Verification, non-ambiguous, natural language, correct specifications, verbal specifications, formal specifications, behavioral study

Introduction

A common model of software system development begins with eliciting the requirements of the system, i.e. what the system is required to be able to do. Thereafter a system that meets the elicited requirements is specified. The specification of the system includes its structure and behavior as well as nonfunctional requirements such as the maximal permitted system size in bytes (called "foot print"). The specification process unearths sometimes further requirements. The requirement elicitation and

system specification are therefore to some extend done simultaneously. The specifications are the basis for a later detailed design and construction of the system. If the requirements elicited in the beginning of the process are faulty, the constructed system will have the corresponding faults. The repair of such a faulty system typically involves high costs and unfortunate delivery delays. Getting the requirements right from the beginning is therefore an important goal. As a measure for achieving this goal, we introduce the concept of satisfactory specifications. Specifications are said to be satisfactory when no major modifications are needed for producing a system that satisfies the intended users. Getting the specification right is, however, difficult to achieve. A study of the faults found in requirements of critical software systems [1] found 9.5 errors per each 100 requirements. This is a surprising high rate, as critical software is subject to an intensive verification and validation. It seems therefore that we have to work hard to produce satisfactory specifications. We discuss a number of known methods for requirement elicitation and specification development that we consider especially useful from a cognitive ergonomically point of view. Developing satisfactory specifications requires different kinds of expertise as well as knowledge of the needs of the users and customers of the specified system. It may therefore be a good idea to employ a Computer Supported Cooperative Work (CSCW) [2] team, including all those who may contribute to the creation, modification or removal of requirements (the stakeholders). Such a team may consider the system from the different points of view of the different stakeholders. Dan Beery [3] suggest including in the team persons that are not familiar with the problem domain. Such a person may ask questions that are out of the entrenched train of thoughts of the experts. In addition, the requirements elicited or omitted represent the priorities of the designers of the system. Such priorities can be negotiated in a CSCW team which includes the relevant stakeholders of the system.

A CSCW team may include software engineers, domain experts, potential users and mangers. Such a team is thus composed of persons of different professional backgrounds and of different human natures. A productive collaboration between such team members involves both behavioral and social challenges [4][5]. These challenges have been the subject of many studies [2]. There are also difficulties in managing the large amount of data found in the requirements and specifications of real life systems. Validating the many different details of the system specification can be very important. Consider for example the case of the aircraft that failed to brake on landing in Warsaw airport in 1993 [6]. The essence of the cause of the accident, where two human were killed, is that the software for controlling the brake system was specified to be activated on landing, i.e. when the weight on each one of the two landing gears exceeded 12 tons. However due to side winds this happened on only one of the two gears. The paper [6] did not report whether the specification team included an experienced pilot (a domain expert) who may have suggested the inclusion of the side wind case in the specifications. Such a possible pilot member of the team should have carefully read the very large number of the aircraft specifications in order to detect this deficiency. The experience of this pilot may have been from the era, when the human pilot activated the brakes and they worked in any kind of wind. Our experienced pilot may therefore not have been aware of the effect off possible insufficient weight on one of the two landing gears. And not detected the side wind problem of

the requirements. Could this problem have been detected by such a visualization means as storytelling? Possible not, as our story teller, i.e. the experienced pilot, was not looking for solutions for the side wind problem. Our pilot may have detected the problem by asking "is it possible that the brakes will not work?" To arrive at the side wind possibility requires out of the box thinking, where the non-expert member of the team may help. To determine whether the side wind is a real threat, the experienced pilot has to ask an aeronautical engineer to investigate the expected winds at landing situations and do the non-trivial computation of their effects on the landing gear weights. Such an investigation is time consuming and costly. The manger may not welcome these costs and delays. In addition our experienced pilot must admit that her/his air craft expertise is limited. The last problem was addressed in Weinberg's egoless programming approach [7], which is widely employed in current day's cooperative efforts, such as CSCW teams. Specifications developments by a CSCW team are, for example, are done in a friendly collegiate way. A team member may thus not be afraid of admitting lack of some expertise, as the overriding concern of the team is to come close to satisfactory specifications. The egoless honest approach is also needed in learned discussions in the CSCW team when attempting to clarify difficult problems, such as the above discussed example of whether the braking may not take place in some situations.

Use case specifications.

Applications that are based on a well elaborated mathematical model may advantageously exploit this model. An example is the successful SQL database management systems, which employ the relational model. Today, some forty years after their introduction, the SQL model is widely employed, which is remarkable in fast evolving software world. In this paper we consider systems where some of the CSCW members may not be familiar with the use of formal methods. In these cases this paper suggests that after the validation of the elicited requirements, the CSCW team develops a verbal use case specification [8][9][10] of the system. Such specifications are widely employed in the industry. We explain why this kind of specification is especially useful in striving to satisfactory specifications. A use case specification of a system is the set of all its use cases [8]. Consider for example a library information system. The use case specification of this system may be composed of such use cases as "Lending a Library Book" (shown in Fig. 1), "Register a Library Book" and "Register a new Lender". A use case is a specification of the interactions between an external actor and the system that are required for accomplishing one particular application of the system, e.g. "Lending a book". An actor is either a human or different system. The interactions specified in the use case are implemented by software that may be called the use case software. Consider for example an actor that asks for data that are stored in a database. The use case software receives this request from the actor and conveys it to the underlying system software that manages the database. The use case software gets the requested database data from the underlying system software and conveys it to the actor. The use case specification does not specify the underlying system software. The underlying system software is specified advantageously after the completion of the use case specification, such that it interfaces to the

already completed use case specifications. The specification of the underlying system software requires only software engineering knowledge and may therefore be produced by a software engineering team. The insights of domain expert members of the CSCW team are, on the other hand, essential for the specification of the use cases. Consider for example the design of a library information system. An experienced librarian (a domain expert) can provide needed information on the activities and problems of a library. This labor division between the CSCW team doing the use case specification and the engineering team specifying the underlying system software enables each team to work in an area where it is most useful.

Identify the book by bar code reading.
(Elapsed computer processing time should be less than 0.1 sec.**)**
Identify the lender by bar code reading.
(Elapsed computer processing time should be less than 0.1 sec.**)**

Check if the lender is entitled to lend the book –
(Lending permission rules come here.)
(Elapsed computer processing time should be less than 0.1 sec.)
Permit or reject the lending,
EXCEPTION HANDLING (what to do if :)
(**The book is not registered in the database)**
(The book bar code is corrupted)
(**The book is not bar coded**
(**The lender is not registered**)
Etc
REQUIREMENTS IMPLEMENTED BY THIS USE CASE
...

Fig. 1. Simplified specification of the use case called "Lending a Library Book", where some details are omitted. It includes nonfunctional requirements, such as "Duration of bar code processing < 0.1 sec." Handling of exceptional cases and list of the elicited requirements that are implemented by this use case are also included.

This paper further suggests specifying the use cases in a natural language that is familiar to all CSCW team members. This should enable efficient collaboration between the different CSCW team members, which may be needed in clarifying difficult situations, such as the side wind example discussed in the previous section. Verbal use case specifications are employed in the industry using for example the instructions of [10]. This paper suggests that a use case specification should include all information related to the use case. Having all the information in one place may help the CSCW team members in understanding of difficult situations. Fig 1, which shows

an example specification of the use case “Lending a Library Book”, where some details are omitted. A use case specifies both the normal and the exceptional cases, e.g. what to do if the book is not registered in the database of the library. A use case specification should also specify non-functional requirements. As an example, it is required in Fig. 1, that the duration of the processing of the bar code reading should be shorter than 0.1 sec. The purpose of this requirement is to provide the user of the system with the feeling of a fast responding system, i.e. a user experience purpose. This requirement is a “non-functional” one, because it is not about the functioning or non-functioning of a bar code reading, but about the duration of the reading. The use case specification should also provide a list the elicited requirements that the use case implements [11]. This enables a person that for some reason modifies the code of the use case software, to check that the modified code still implements the relevant elicited requirements.

It is suggested that after a thorough validation of the developed verbal use case specifications by the CSCW team, these specifications are manually translated to a formal UML specification [12][13][14]. UML was designed to support the software development process. It provides tools for visualization of the specifications and for some validations. There are UML tools for code generation and for test case generation [15]. UML tools regarding system dependability are discussed in [16]. UML tools for Model Driven Software Engineering are discussed in [17]. These powerful tools may be employed throughout the life cycle of the software for future extensions and modifications.

English Issues.

In order to simplify the formulations in the following part of this paper, we employ the term “English” as an abbreviation of “Natural language, for example English”. Summerville [18] lists a number of difficulties in writing natural language requirements specifications. One of the problems is that “Natural language understanding relies on the specification readers and writers using the same words for the same concept.” This problem is avoided by many writers of mathematical proofs, scientific papers and commercial contracts, who succeed in conveying their messages in a clear and correct way. Writers of software requirements specifications face the same linguistic problems as the above mentioned writers and may therefore adopt their methods and be equally successful. We consider first an example of a mathematician writing a proof. Her intended readers are mathematician, who like herself, have been trained during their university studies to employ the by and large globally accepted vocabulary of mathematics and the English idioms employed in mathematical explanations. Before her paper is published, it is typically reviewed by three mathematicians who check both the correctness of the proof and the clarity of English explanations. Their improvement suggestions may then be incorporated in the published revised paper. By employing the by and large globally accepted mathematical vocabulary it is ensured that the author and her readers understand the concepts in the same way.

Consider now, for example, the writer of the requirements specifications of a library information system. Following the mathematician example, the requirement

specification writer will employ the vocabulary of library science in the requirements that regard library issues and computer science vocabulary in the requirements that regard software issues. Similarly to the mathematical proof case, the requirements specifications should be thoroughly reviewed for their correctness as well as for their understandability.

The handbook [19] distinguishes between linguistic and software engineering ambiguities in requirements specifications. Software engineering ambiguity is when some specifications needed for the implementation of the requirements are missing. An example of such a software engineering ambiguity is found in Fig. 1 in the instruction "Identify the book by bar code reading". There exist however four different possible bar code standards and it is not specified which to employ. Such ambiguities may be normal at an early stage of the specification process, when some of the design decisions have not yet been made.

As regard the linguistic ambiguities, the manual [19] provides detailed instructions for producing precise formulation in English and avoiding unintended ambiguities, i.e. ambiguities not intended by the specification writer. The manual provides linguistic equivalents to mathematical formulations. Linguistic formulations may thus be as precise as mathematical formulations. This enables employing argumentations in English in mathematical proofs. In the following we compare some usability issues of formal and verbal specifications. The comparison will be illustrated by an example specification of a small restaurant (Fig.2, 3 and 4).

1. The restaurant has two tables named table A and table B.
2. Each table can accommodate up to and including four diners.

Examples:

Two dine on table A. Three dine on table B.
No one dines on table A. No one dines on table B.
Four dine on table A. Four dine on table B.

Clarifying statement:

The restaurant can accommodate up to and including eight diners.

Fig. 2. Specification of a small restaurant formulated by the instructions of [19]

In order to avoid misunderstanding due to poor English formulations [19] suggest adding clarifying examples and explanations as illustrated in Fig. 2. We suggest an engineering praxis, where a separate example is provided for each kind of situation that may occur. The examples of Fig. 2 represent thus a "normal" situation and two "extreme" situations. These examples clarify for the reader the intents of the specification writers in each of the three possible situation kinds. These examples may also be employed by proof readers and specification validators for checking purposes. The specification of Fig. 2 could have employed the domain knowledge, that a table may be free (no diners), but we preferred to clarify this point by the second example.

Fig. 3 illustrates how an example disambiguates an unintended ambiguity, which is due to a poor formulation of statement 2 of the specification. This statement 2 has the false interpretation marked as II, which was not intended by the specification writer.

Examples one and three as well as the clarifying statement of Fig. 3 show that interpretation II is false. The unintended ambiguity of statement 2 in Fig. 3 contrasts with the efficient specification employed in Fig. 2, where the statement marked as 2 is intentionally ambiguous. It has 25 different correct interpretations, three of which are the three examples. In other words the single statement 2 specifies 25 possible situations. This is an example of the possible usefulness of ambiguities in natural languages.

1. The restaurant has two tables named table A and table B.
2. All the tables can accommodate up to and including four diners.

Examples:

Two dine on table A. Three dine on table B.
No one dines on table A. No one dines on table B.
Four dine on table A. Four dine on table B.

Clarifying statement:

The restaurant can accommodate up to and including eight diners.

Statement 1 of the specification can be understood in two ways:

I. Each table can accommodate up to and including four diners.
II. The two tables can together accommodate up to and including four diners.

The clarification examples show that I is the intended specification

Fig. 3. The restaurant of Figure 2 specified with an unintended ambiguity in statement 2. The clarifying examples enable a disambiguation. The proof reader of this poorly written specification may detect the problem and improve formulation.

We shall now employ our restaurant specification for a brief illustration of the differences between verbal and formal specifications. Fig. 4 is a formal specification of the same restaurant.

n –number of tables
a – number of diners on table 1
b – number of dinners on table 2
n=2
$\{a \in Z: a \geq 0\hat{}a \leq 4\}$
$\{b \in Z: b \geq 0\hat{}b \leq 4\}$

Examples

a = 2, b = 3.
a = 0, b = 0.
a = 4, b = 4.

Clarifying statement

$a + b \leq 8$

Fig. 4 Formal specification of the restaurant specified in Fig. 2 .

We suggest employing examples and clarifying statements even in the very simple specification of Fig. 4, as they may facilitate fast and correct understanding. Examples and clarifying texts may be very useful in complicated specifications, where the specification writer may err. Examples may also have an explanatory value for the reader. It is noted that writing and reading of long logical expressions in formal specifications are error prone processes [20]. This indicates that the deciphering of mathematical expressions involves a non-negligible cognitive effort. It is therefore recommended to employ only short expressions when possible. A further problem is that understanding a formal specification involves cognitively both a deciphering the mathematical expression and formulating it in the terminology of the domain. This contrasts with a verbal specification which readily expresses in the terminology of the domain. Whether the possible differences in the cognitive effort involved in manipulating formal and verbal specifications influence significantly the efficiency of the system specification process is difficult to tell without an elaborate experimental comparison. The training and experience of the specifications developers must be considered. Furthermore, formal specifications may in some cases be advantageously manipulated mathematically by computer programs.

Discussion and Future Research

This paper introduces the Satisfactory Specifications concept, which is defined as specifications that without major modifications enables production of a system that satisfies the intended users. The goal is thus to produce a quality specification such that costly repairs of the system and unfortunate delays are avoided. A CSCW team having the relevant experts and stakeholders has the insights needed for producing satisfactory specifications. However, even for a CSCW team, achieving satisfactory system specifications is a difficult task, especially when the system includes novel components that have not yet been tried, e.g. using a computer instead of a human for controlling the brakes of a landing air-craft. Analyzing such an unknown situation may require an intensive cooperative analysis by team members having different training and expertise. These members must understand and agree on the specifications that they develop. Therefore, if some of the members involved in the discussion are not familiar with formal specifications, the use of verbal specification seems to be the right thing. Alternatively the team members familiar with formal methods write the formal specifications and translate them into English for the members who are not familiar with formal specifications. This involves the risk that the translations do not convey some details correctly. A possible future behavioral study may compare these two specifications development processes. The paper suggests translating validated verbal specifications into standardized formal UML diagrams, which visualize the design and enables using UML tools, e.g. for validation and code generation. This approach exploits the advantage of both verbal and formal specifications.

In the discussion on whether to employ formal or verbal specifications, the value of the expressiveness of English is in our opinion not given sufficient weight. Providing this kind of expressiveness to an English like formal language [22] [23] may therefore be difficult. The problems of possible unintended ambiguities in English may be mitigated by using English correctly and by employing clarifying examples and explana-

tions [19]. This paper suggests employing a separate example for each kind of the possible situations, e.g. a use case for solving a real quadratic equation may have a separate example for no roots, one root and two roots. The examples provided for a specification may therefore also be employed for debugging the software. These examples are also useful illustrations for persons wishing to understand the specification.

REFERENCES

1. Ambrosio, A.M., Madeira, H., Silva, N., Véras, P.C. , Vieira, M., Villani, E.: Errors on Space Software Requirements: A Field Study and Application Scenarios, Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on Space Software Requirements Engineering
2. Grudin, J., Poltrock, S.: (2012) CSCW - Computer Supported Cooperative Work. In: Soegaard, Mads and Dam, Rikke Friis (eds.). "Encyclopedia of Human-Computer Interaction". Aarhus, Denmark: The Interaction-Design.org Foundation.
3. Berry, D. M.: The importance of ignorance in requirements engineering: An earlier sighting and a revisitation. Journal of Systems and Software 60(1): 83-85 (2002)
4. Grudin J.: Why CSCW applications fail: problems in the design and evaluation of organizational interfaces, CSCW '88 Proceedings of the 1988 ACM conference on Computer-supported cooperative work
5. Schmidt, K., Bannon, L.: Taking, CSCW seriously - Computer Supported Cooperative Work (CSCW), Springer 1992
6. Hawkins, R.D., Habli, I., Kelly, T.P. The Principles of Software Safety Assurance, 31st International System Safety Conference 2013
7. Weinberg, G.M.: The Psychology of Computer Programming, Dorset House 1971
8. Jacobson, I., Christenson, M. P., Jonsson, P. G.: Overgaard Object-Oriented Software Engineering: A Use Case Driven Approach Addison-Wesley, 1992
9. Cockburn, A.: (2008-1-9). "Why I still use use cases". alistair.cockburn.us.
10. Cockburn, A.: Writing Effective Use Cases, Addison-Wesley, 2001
11. Winkler, S., Pilgrim, J.: A survey of traceability in requirements engineering and model-driven development, Journal Software and Systems Modeling (SoSyM), Volume 9 Issue 4, September 2010
12. Object Management Group: Catalog of OMG Modeling and Metadata Specifications. http://www.omg.org/technology/documents/modeling_spec_catalog.htm
13. Rumbaugh, J., Jacobson, I., Booch, G.: Unified software development process, the complete guide to the unified process from the original designers. Addison Wesley 1999
14. Lange, C.F.J., Chaudron, M.R.V. , Muskens, J.: In practice: UML software architecture and design description. Software, IEEE (Volume:23 , Issue: 2
15. Abdurazik, A., Offutt, J.: Using UML collaboration diagrams for static checking and test generation. Lecture Notes in Computer Science Volume 1939, 2000, pp 383-395
16. Bernardi, S., Merseguer, J., Petriu, D.C.: Dependability modeling and analysis of software systems specified with UML. ACM Computing Surveys, Volume 45, Issue 1, November 2012

17. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering, September 2012.
18. Sommerville, I.: Software Engineering (9th Edition), Addison-Wesley 2010
19. Berry, D., M., Kamsties, E., Krieger, M.M.: From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity - A Handbook. 2003, available: https://cs.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf
20. Reisner, P., Boyce, R.., F., Chamberlin, D., D.: Human factors evaluation of two data base query languages: square and sequel, AFIPS '75 Proceedings of the May 19-22, 197
21. Arora, C., Sabetzadeh, M., Briand, L., Zimmer, F., Gnaga, R.: Automatic Checking of Conformance to Requirement Boilerplates via Text Chunking: An Industrial Case Study. ESEM'13, 2013
22. Umber, A., Bajwa, I.S.: Minimizing ambiguity in natural language software requirements specification, Sixth International Conference on Digital Information Management (ICDIM), 2011
23. Osborne, M., MacNish, C.K.: Processing natural language software requirement specifications, Requirements Engineering, 1996., Proceedings of the Second International Conference on