Computers, Materials & Continua
DOI: 10.32604/cmc.2023.035846
Article

A Defect Detection Method for the Primary Stage of Software Development

Qiang Zhi[1], Wanxu Pu[1], Jianguo Ren[1] and Zhengshu Zhou[2,*]

[1]School of Computer Science and Technology, Jiangsu Normal University, Xuzhou, 221116, China
[2]Graduate School of Informatics, Nagoya University, Nagoya, 4648601, Japan
*Corresponding Author: Zhengshu Zhou. Email: shu@ertl.jp
Received: 06 September 2022; Accepted: 09 November 2022

Abstract: In the early stage of software development, a software requirements specification (SRS) is essential, and whether the requirements are clear and explicit is the key. However, due to various reasons, there may be a large number of misunderstandings. To generate high-quality software requirements specifications, numerous researchers have developed a variety of ways to improve the quality of SRS. In this paper, we propose a questions extraction method based on SRS elements decomposition, which evaluates the quality of SRS in the form of numerical indicators. The proposed method not only evaluates the quality of SRSs but also helps in the detection of defects, especially the description problem and omission defects in SRSs. To verify the effectiveness of the proposed method, we conducted a controlled experiment to compare the ability of checklist-based review (CBR) and the proposed method in the SRS review. The CBR is a classic method of reviewing SRS defects. After a lot of practice and improvement for a long time, CBR has excellent review ability in improving the quality of software requirements specifications. The experimental results with 40 graduate students majoring in software engineering confirmed the effectiveness and advantages of the proposed method. However, the shortcomings and deficiencies of the proposed method are also observed through the experiment. Furthermore, the proposed method has been tried out by engineers with practical work experience in software development industry and received good feedback.

Keywords: Computer science; software engineering; requirement engineering; software quality; defect detection

1 Introduction

In the field of IT, especially in software development, it is crucial to clarify the needs of users. Only when the real needs are clarified can the developed software products be put into practice. Requirements engineering is the process of how to develop software that meets user requirements [1], and its life cycle basically consists of five stages: requirements acquisition, modeling, specification, verification, and management [2]. Among them, the requirements specification is the central stage of software development activities, in which the requirements analysts define the components and behaviors of the software to be developed, and the result is a software requirements specification

(SRS), which is used to describe the needs of software stakeholders, and trying to define what the target application will look like.

An SRS generally includes five modules: overview, requirements description, data description, operating environment regulations and restrictions, and is regarded as a contract or an agreement between customers and developers. Usually, an SRS is basically written in natural language (NL) format, and most NL expressions can be ambiguous [3], with problems such as lexical ambiguity, syntactic ambiguity, and semantic ambiguity. The SRS becomes explicit only if the software requirements specified in it have only one meaning [4]. After the stage of requirements specification, it will go to the requirements verification stage. The requirements verification can be understood as the SRS review, which purpose is to ensure the accuracy, consistency and integrity of the SRS, which can help stakeholders such as requirements analysts and software engineers to identify and correct defects such as inconsistencies, omissions, and errors, ensuring that all system requirements are free of ambiguity [5].

The Software Engineering Body of Knowledge (SWEBOK) makes general standardization regulations for the engineering process around SRS. As shown in Fig. 1, the engineering process of SRS includes basic steps such as acquisition, analysis, writing, and verification of SRS. The review of SRS is an important part of SRS verification. The review process is usually divided into three steps: first, let the reviewer independently read and analyze part or all of an SRS and identify as many defects as possible, then these defects would be collected at a meeting of the reviewer and the SRS author, and finally be revised and supplemented by the SRS author and perfect. After requirements verification (review), a high-quality SRS version is obtained to reduce the chance of defects in the final product delivered to the end user.

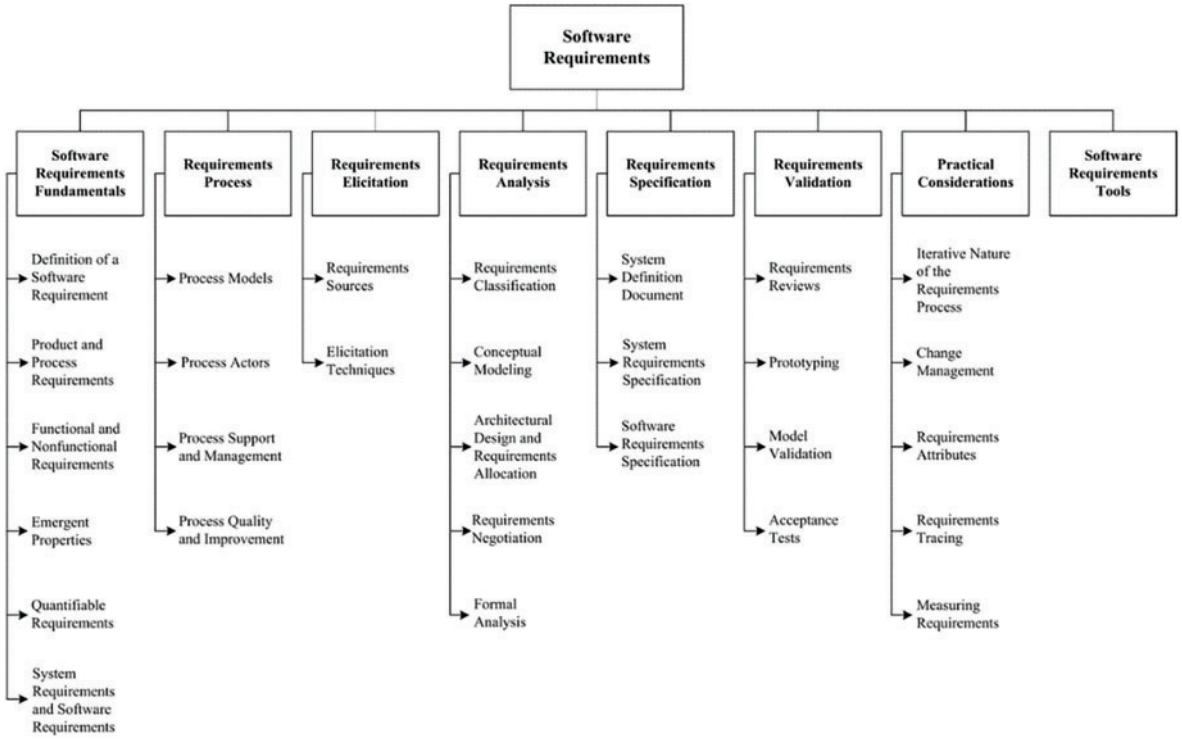

Figure 1: Software requirements engineering flowchart

However, there are many imperfections in the existing review methods (for example, it takes a considerable amount of time to conduct a detailed and comprehensive review with little effect, and is too easily interfered with by external factors, etc.). In this study, we combine the actual needs of the IT industry and propose a new SRS review method to improve the SRS quality. The remainder of this paper is structured as follows. Section 2 briefly reviews previous studies related to SRS review. Section 3 introduces the proposed SRS quality assessment and defect detection method. The empirical evaluation and the discussions are described in Section 4. Conclusions, including a summary of the contributions of this study, are provided in Section 5.

2 Related Research

2.1 Detection of Defects

Ambiguity arises if the SRS is flawed, meaning that a word or passage can be understood as having multiple meanings. In 2010, Rojas et al. [6] noticed that if an SRS contains logical conditions such as "and" and "or", there will be semantic ambiguity, pragmatic ambiguity and grammatical ambiguity; In 2015, Sandhu et al. [7] classified the ambiguity in SRS into lexical ambiguity and syntactic (structural) ambiguity. In order to solve the ambiguity caused by ambiguity, researchers have proposed three frameworks based on Unified Modeling Language (UML), Ontology-based Processing, and Natural Language Processing (NLP). Among them, NLP technology uses NL to process SRS. Recently, Sabriye et al. [8] proposed a framework for detecting ambiguity in SRS based on NLP, but this research is still at an early stage. The framework consists of three components: preprocessing, processing, and postprocessing, using an ambiguity detector to check the syntactic ambiguity of any sentence in the Stanford POS tagger; in 2018, Osman et al. [9] proposed an automated detection method that combines text mining and machine learning to detect ambiguous requirements specifications. However, the shortcomings of NLP technology are also obvious, such as the detected defect type is relatively single, and the accuracy rate is not high (misjudgment).

Furthermore, there are various review methods that can be used to detect various defect types (such as omissions, errors, inconsistencies, etc.): usage-based review (UBR), defect-based review (DBR), and perspective-based review (PBR). UBR is a relatively new technology that supports object-oriented system review by using use-case scenarios such as UML to guide inspection efforts, with each step checked in turn for each use-case scenario. However, more research is needed to determine how the effect of the review method varies with different kinds of "artifacts" (such as source code, blueprints, etc.), different changes in instructions, and different levels of prior experience with UBR [10]; DBR can help reviewers focus on different categories of defects, such as inconsistent data types or loss of functions. However, this method was originally developed as a formal notation for event-driven process control systems and is not widely used in the commercial field [11]. In PBR, stakeholders such as software users, software designers, and software developers will focus on the content of the SRS based on review guidelines from different perspectives [12].

2.2 Checklist-Based Review

CBR was proposed by Fagan [13] in the 1970s and has been in use till now, which is a classic and traditional method of examining SRS defects. CBR provides tips and advice for finding bugs, which also means the checklist is instructive. Heuristics tend to be simple and often memorable, well describing the basis for decisions about commonalities, with an implicit ability to how best to move the work forward [14]. In software review work, a corresponding checklist can be created at a specific stage of software development for quality review. There are six types of checklists (requirements checklist,

design checklist, code checklist, test checklist, documentation checklist, and process checklist) by stage. Among them, the checklist for requirements should be a list of problems analyzed to ensure the consistency, correctness, and completeness of requirements. In fact, such checklists are inherently more general than code checklists, since the work products that need to be reviewed are usually written in NL rather than programming languages [15]. The checklist should be based on the defined defects, and each checklist item is a type of defect, which is set by the important experience gained in the past review, and can be prioritized according to different defects. Software testing techniques (such as use case testing, and boundary value analysis) can quickly and accurately find various problems in computer software to help relevant personnel process and solve these problems centrally [16]. The relationship between inspection and testing should be understood as: inspection can find and eliminate faults (defects) more cost-effectively than testing [17]. The CBR uses the checklist as a guide, using this guide can significantly improve product quality and productivity because the longer a defect remains in a stage, the more failures the product will cost and the more expensive it will be to eliminate it. Defects are identified at an early stage, which improves the quality and productivity of the final product.

The quality of the CBR design is very important, but few reports have been published on empirical methods to improve the effectiveness of the checklist, which currently depends on the skill of the checklist creator [18,19]. The Institute for Defense Analysis of the United States [16] through a survey of 117 checklists from 24 different sources, pointed out that making an effective heuristic checklist should include the following characteristics:

- One test content in the checklist should not exceed 28 items (within one page)
- Checklists should be updated regularly based on defect analysis, especially for frequently occurring and missing defects
- Checklist items could be in the form of questions
- Checklist items should not be too general and should be tailored to a specific development environment

In the IT industry, software engineers often use CBR to guide software quality testing activities. Engineers advocate convenience and productivity, and their knowledge and experience form the basis for high-quality software testing, agreeing that CBR has the following advantages: 1) Versatility, 2) ease of creation, use, and maintenance, 3) ease of results analysis (task progress and completion status), 4) high flexibility, items can be added or removed as needed.

Since CBR is the most widely used in quality assurance engineering, we will compare the proposed method with CBR to verify the effectiveness.

3 Element Question Indicator: An SRS Quality Assessment and Defect Detection Method

3.1 SRS-based Elements and Questions

In the primary stage of software development, most SRSs are written in NL, and there are often more or less ambiguous elements in them, or elements that are not clearly expressed, thus affecting the software development and design. We surveyed the IT industry and found that a large number of engineers with many years of software development experience, were troubled by the lack of clarity and accuracy of SRSs, and the rework and function remaking caused by this was not uncommon, resulting in a huge waste of time, effort and money. In this study, we propose the concept of SRS Element Quality Indicator (EQI), which aims to assess the quality of SRS and find defects in SRS.

We consider the requirement content of SRS as consisting of various different elements. Suppose we design a personnel management system, which can be divided into: users (subdivision: login,

registration, data management, personnel rights, etc.), personnel management (subdivision: selection, use, training, appraisal, rewards, and punishments, etc.) from large aspects to fine aspects. We call all of these SRS elements. For the elements, we can understand them as "requirements analysis objects", which can be further divided according to their attributes or lexical nature. The Q (Question) in EQI is the question that is asked for the elements of SRS, and these questions will be categorized and counted. For example, during the development of this personnel management system, if the requirements are not very clear, the developer will feel confused and raise questions.

- How are users' permissions divided?
- Is the way users' data is managed related to permissions?
- Can information queries span different departments?

Obviously, there should be clearer instructions inside the SRS, otherwise, questions like these will be raised.

3.2 EQI-Based Review Process

We have designed the EQI-based SRS review process as follows.

1. Understand the background purpose of system development (software development) and should have the following basic qualities
    - Conduct a feasibility analysis of whether a requirement is correct and testable
    - Not all SRS content needs to be verified, so it is important to understand what information needs attention and what information does not need attention
    - A distinction should be made in the review between functional and non-functional requirements. Non-functional requirements often do not need to be very precise, just approximate
2. Understand what is an SRS element? What are the questions to ask about elements?

The content of SRS consists of different types of elements. We can understand the elements (e.g., user, appraisal) as the object of analysis, and then choose the analysis method according to the properties of the elements. For example, "user" is a noun, while "appraisal" is a verb, which is the issuance of an action. Obviously, the way of asking questions should be different for elements with different attributes.

- Questions about nouns should be directed at states and attributes. For example, what authority does the user have? This is asking a question about the attribute of the noun (user).
- The question to the verb should be directed to the opportunity and result. For example, how to classify the results of different appraisals? This is a question about the result of the verb (appraisals).

3. SRS Element-Based Question Extraction Method

In this study, SRS reviewers perform element extraction and element-based questioning from sentences or paragraphs of the SRS. However, some of the questions asked may not be informative. To improve the quality of reviewers' questions, we proposed the EQEM method to standardize the SRS element extraction and questioning procedures based on EQI. We divide EQEM into five components: ***key elements, target elements, target items, expected items, and exceptions***.

An example of EQEM is as follows.

Suppose an SRS documentation says "The security monitoring function must notify the device status at least every 60 s during normal time intervals". In this case, the following five elements can be extracted.

- Security monitoring
- During normal time intervals
- At least every 60 s
- Device status
- must notify

Based on the above elements, the process for confirming the clarity of SRS requirements can be constructed as follows: for each decomposed element, the "target item" should be checked first, then check whether the "expected items" and "exceptions" are clear. Table 1 shows the details of "expected items" and "exceptions" questioning methods.

Table 1: EQEM requirements confirmation form of security monitoring function

Key element	Decomposed element	Target items	Expected items	Exceptions
Security monitoring function	Security monitoring	Monitoring	What is being monitored?	What are the exceptions in monitoring?
	During normal time interval	Time interval	What is the upper limit?	What is an abnormal interval?
	At least every 60 s	At least	What is the upper limit?	What if the time interval is too long?
	Device Status	Status	What is status?	What is abnormal status?
	Must notify	Notify	What is the content and object of the notification?	What are the possible exceptions for the notification?

Based on the above, EQEM can be summarized as follows:

[Input] Provided key elements in an SRS

[Procedure]

1. According to the key elements provided, combine SRS sentences or paragraphs to decompose them into local elements.
2. Identify what the target item is from the element, and then create a requirements confirmation form (RCF).
3. For the extracted "target items", check whether the "expected items" and "exceptions" have been clarified in the SRS according to the RCF.
4. Mark ambiguous "expectations" and "exceptions" in the RCF (questions raised).

[Output] RCFs

3.3 EQI Definition

The EQI defines the ambiguity of the SRS in terms of the ratio of the number of EQEM-based questions to the extracted elements in the SRS. If the ratio is high, it indicates that the definition of SRS is low and the quality is poor; on the contrary, if the ratio is low, it can be considered that the content of the SRS has high definition and good quality. We define EQI as follows: Let $\boldsymbol{S_D}$ be the sum of the key elements of the SRS ($\boldsymbol{D}$), $\boldsymbol{Q_n}$ be the number of unclear items in "expected item" and "exceptions" in all EQEM-based RCFs, then the $\boldsymbol{EQI}$ ($\boldsymbol{D}$) is:

$$EQI\ (\boldsymbol{D}) = \frac{\boldsymbol{Q_n}}{\boldsymbol{S_D}} \tag{1}$$

EQI can also be defined as follows: Let $\boldsymbol{S_D}$ be the sum of the key elements of the SRS ($\boldsymbol{D}$), $\boldsymbol{Q_n}$ be the number of unclear questions in "Expected items" and "Exceptions" in all EQEM-based RCFs, and let $\boldsymbol{Q_p}$ be the number of items that have been identified for all key elements. Now, suppose a total of $\boldsymbol{i}$ reviewers participate in the quality assessment of SRS ($\boldsymbol{D}$) based on EQI, then $\boldsymbol{EQI}$ ($\boldsymbol{D}$) in this case is:

$$EQI\ (\boldsymbol{D}) = \frac{\sum \boldsymbol{EQI}\ (\boldsymbol{D_i})}{\boldsymbol{i}} = \frac{\sum \left(\frac{\boldsymbol{Q_n}}{\boldsymbol{S_D}} * \frac{\boldsymbol{Q_n}}{\boldsymbol{Q_p} + \boldsymbol{Q_n}} \right)_i}{\boldsymbol{i}} \tag{2}$$

According to formulas (1) and (2), the clarity and quality of the related SRS decreases as EQI (D) increases. If the number of questions to be clarified in RCFs is 0, then $\boldsymbol{EQI}\ (\boldsymbol{D}) = 0$, indicating that the SRS has a high degree of clarity and quality. Fig. 2 shows a schematic diagram of the correspondence between the elements in the EQI and the attributes of SRS in the IEEE 830 standard. As can be seen from this figure, the properties of SRS in the outer area can be confirmed by the EQI elements shown in the inner rectangular area.

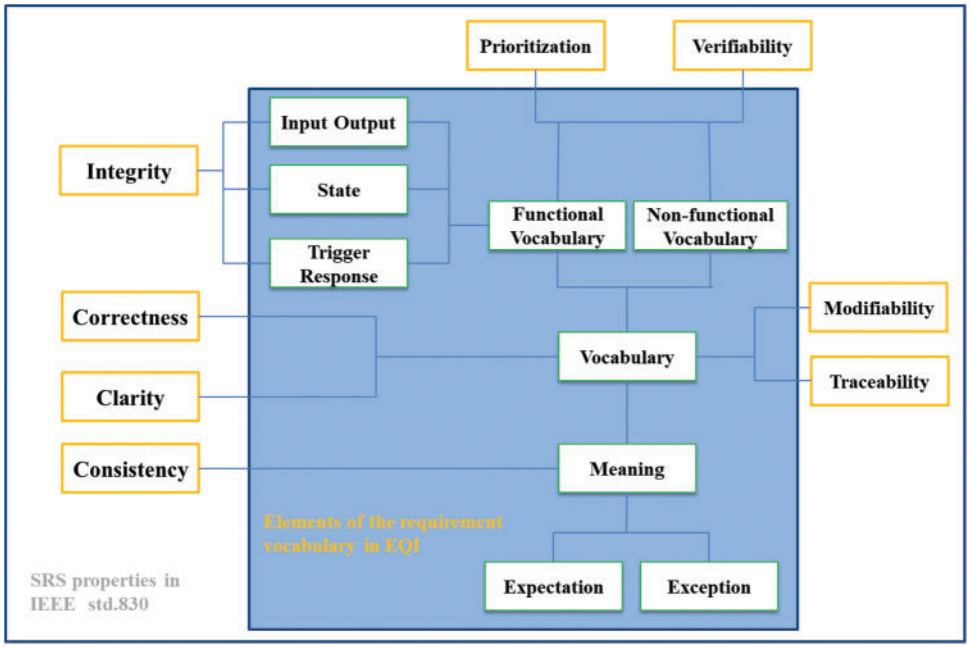

Figure 2: Relationship between EQI and IEEE 830 standard

4 Experiment

4.1 Experimental Design

In this experiment, two SRSs (A and B) with different contents and similar scales were selected, and they had similar numbers and types of defects. Table 2 gives the details of SRS and the corresponding review method.

Table 2: Experimental SRSs and review methods

SRS	Name	Review method
A	Student performance management system	EQI/CBR
B	Student life support system	EQI/CBR

A total of 40 graduate students majoring in software engineering participated in the experiment. The subjects were divided into four groups (X, Y, Z, W), there are ten subjects in each group and the average ability of each group member is similar. In order to reduce the influence of the learning effect of SRSs on the experimental results, we adopted the crossover design [20] in our experiment. Crossover is an experimental design in which all treatments are applied to each subject, but the order in which the treatments are applied is different for different subjects. Table 3 shows the design of this two rounds experiment.

Table 3: Experimental design

Group	First round		Second round	
	Review method	SRS	Review method	SRS
X	CBR	A	EQI	B
Y	EQI	A	CBR	B
Z	CBR	B	EQI	A
W	EQI	B	CBR	A

To design a high-quality and effective checklist, we follow the advice of the Institute for Defense Analysis of the United States [16] and employ a two-component structure [21] that combines statistical and empirical methods based on the content of the SRS. Simply put, the design of the checklist follows the rules that "where to look" and "how to detect", there are 5 items in the designed checklist, namely "Correctness and completeness", "Consistency", "Feasibility", "Testability", and "Meet standards". Table 4 gives the checklists used in this experiment. Additionally, Table 5 details the current mainstream defect types and detailed explanations, with a total of 5 categories.

Table 4: The designed checklist

Item no	Where to look?	How to detect?	
1	***Correctness and completeness***	Whether all functional capabilities are documented	ϒ

(Continued)

Table 4: Continued

Item no	Where to look?	How to detect?	
2		Whether reuse of existing software or use of COTS software is fully described	ϒ
3		Whether adaptation requirements (e.g., geographic parameters, platform variations) are identified	ϒ
4		Whether applicable timing, resource usage (e.g., CPU, memory, bandwidth), and associated system load requirements are identified	ϒ
5		Whether the effect of the operating system, executive, or COTS has been factored into the performance requirements and resource budgets	ϒ
6		Whether the applicable safety & security requirements are identified	ϒ
7		Whether applicable design constraint requirements (e.g., object-oriented design, language, support environment) are identified	ϒ
8		whether the applicable software quality attributes (e.g., reliability, maintainability, testability) are identified.	ϒ
9		Whether the applicable human performance/human engineering requirements are identified	ϒ
10		Whether applicable acceptance criteria (e.g., test, inspection, demonstration) are identified	ϒ
11		Whether requirements are traceable to requirements allocated to software	ϒ
12		Whether all functional data flows are specified, including sources and destinations	ϒ
13		Whether inputs and outputs of each requirement are necessary and sufficient for the specified processing	ϒ
14		Whether accuracy and precision requirements are defined	ϒ
15		Whether all software functions are considered (e.g., startup, restart, modes of operation, shutdown, normal terminations, abnormal conditions, performance monitoring and tuning).	ϒ
16		Whether operator interactions are considered	ϒ
17		Whether all functional processing requirements are specified for recognized error conditions (e.g., hardware faults, I/O errors, computational errors, processing overload, buffer overflow, events failing to occur, out-of-sequence events)	ϒ
18		Whether all test requirements are defined (e.g., test levels; provisions to inject test data, to adjust parameters, to control or trace the execution of test runs, to extract and reduce test results)	ϒ
1	***Consistency***	Whether all software requirements are derived from the system specification	ϒ

(Continued)

Table 4: Continued

Item no	Where to look?	How to detect?	
2		Whether each object is referred to by a unique name	ϒ
3		whether each object is defined by a set of non-conflicting characteristics	ϒ
4		Whether all requirements are free of logical/timing conflicts	ϒ
5		Whether requirements do not conflict with each other	ϒ
6		Whether all data/messages/requirements are specified only once	ϒ
7		Whether data flows are consistent with the specified inputs and outputs of the relevant requirements	ϒ
8		Whether data flow notations are used consistently	ϒ
9		Whether message data attributes are consistent with the inputs and outputs of relevant requirements	ϒ
10		Whether loads used to allocate resource budgets are consistently specified for all functions	ϒ
11		Whether function names used in diagrams are consistent with the requirements text	ϒ
12		Whether requirements are consistent with the operational context	ϒ
1	***Feasibility***	Whether data expected from external sources exist at those sources	ϒ
2		Whether data sent to external destinations are expected at those destinations	ϒ
3		Whether requirements are achievable with available technology and necessary implementation tools are available	ϒ
4		Whether the scope of requirements is realistic, considering software estimates, schedules, and support facility plans	ϒ
5		Whether the performance requirements are realistic based upon available facts or modeling information, (e.g., response times, accuracies, processing capacities)	ϒ
6		Whether resource budgets are realistic (e.g., CPU time, I/O utilization, memory, worst-case loads, data storage)	ϒ
1	***Testability***	Whether all requirements are specified against the software (i.e., not against the hardware or the operator).	ϒ
2		Whether all requirements can be verified by some (implicit, explicit, analytical, empirical) means.	ϒ
3		Whether test results can be evaluated against predetermined acceptance criteria.	ϒ
1	***Meets Standards***	Whether major software functions are described in relation to system operation	ϒ
2		Whether requirements are clearly stated and unambiguous	ϒ
3		Whether terminology is understandable and consistent	ϒ
4		Whether all notation and naming conventions are defined	ϒ

(Continued)

Table 4: Continued

Item no	Where to look?	How to detect?	
5		Whether glossary is adequate	ϒ
6		Whether requirements are clearly numbered or otherwise marked	ϒ
7		Whether requirements terminology (i.e., "shall", "will", "may", etc.) is used correctly	ϒ

Table 5: Defect classification

	Defect category	Description
I	Incorrect	Some information in the SRS contradicts relevant information in the domain knowledge or conflicts with previous documentation
II	Description problem	Description error and format error
III	Ambiguity	Ambiguous, unclear requirements
IV	Omission	Necessary information in the SRS related to the problem the software needs to solve is omitted or incomplete
V	Feature Suggestion	Suggestions for adding optional features

4.2 Experiment Procedure

Before starting the experiment, we divided the subjects into 4 groups with similar average abilities according to their professional performance. Then the subjects were given the appropriate experimental training, which was explained by professionals to ensure that the subjects could master the appropriate knowledge and skills. There are two rounds for the experiment, after each round of the experiment, subjects' review materials were recovered at the end of each round for analysis of the experimental results. The experimental design process is shown in Fig. 3.

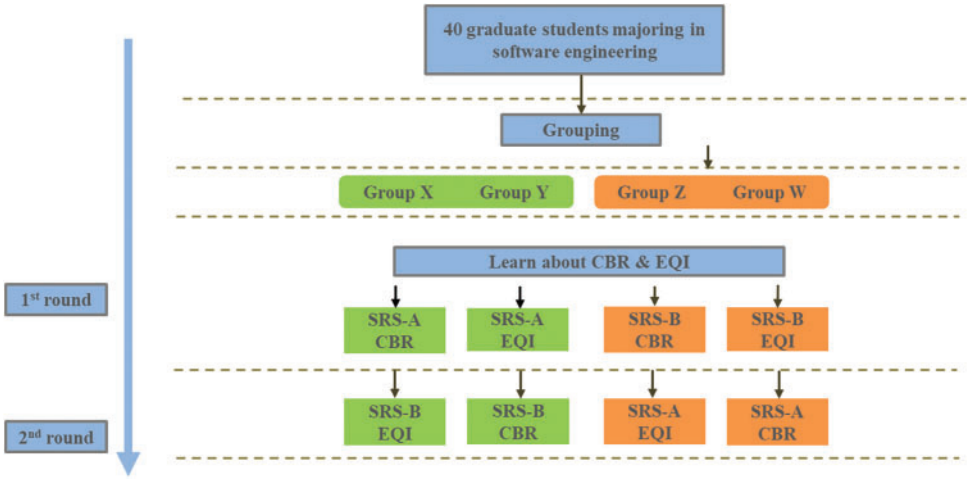

Figure 3: Experimental procedure

4.3 Experimental Results and Analysis

4.3.1 The Number of Defects

We categorized the statistics of defects found in each group of subjects. According to Table 5, we classified the defect types into omission type, description problem type, ambiguous type, feature suggestions, and incorrect type. We divided the experimental data into two datasets based on SRSs A and B, and the total number of defects found in each dataset is shown in Fig. 4.

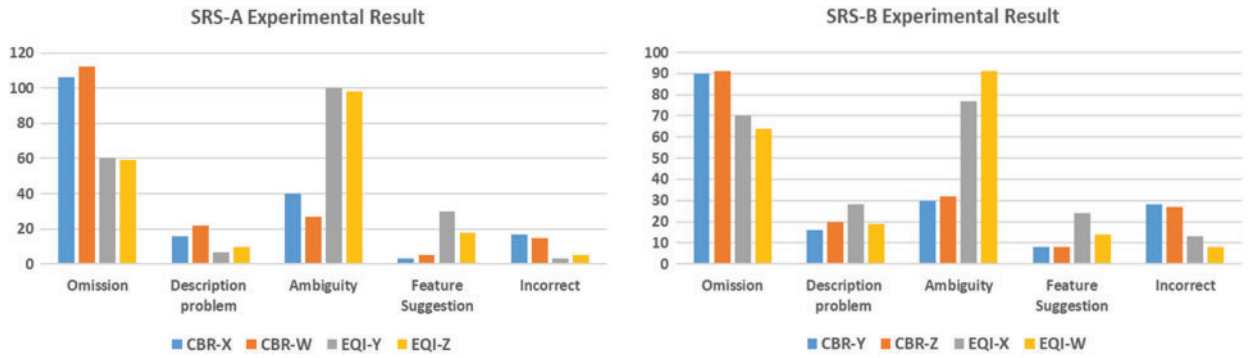

Figure 4: Analysis of experimental result

According to the procedure of the experimental design, we counted the experimental data of 40 subjects (4 groups, 10 subjects in each group). Since the experiment adopts a crossover design, it can effectively reduce the influence of the learning effect while solving the problem of small sample sizes. Assume the subjects of the same method are treated as a team (20 subjects), according to the mean of the number of defects detected for SRS (A) and SRS (B), EQI is better than CBR (Table 6). Table 7 lists the average number of defects detected by each group.

Table 6: The average number of defects counted as a team

SRS	Review method	Average
A	CBR	18.2
A	EQI	19.5
B	CBR	17.5
B	EQI	20.4

Table 7: The average number of defects in each group

SRS	Group	Review method	Average
A	X	CBR	18.2
	Y	EQI	20
	W	CBR	18.1
	Z	EQI	19
B	Z	CBR	17.2
	W	EQI	19.6

(Continued)

Table 7: Continued

SRS	Group	Review method	Average
	Y	CBR	17.8
	X	EQI	21.2

4.3.2 Proportion of Each Defect

Fig. 5 shows the proportion of each defect detected by different groups based on different SRS. In the review results of CBR, the defects of the omission type accounted for the largest proportion, and in the review of EQI, the ambiguity defects accounted for the largest proportion.

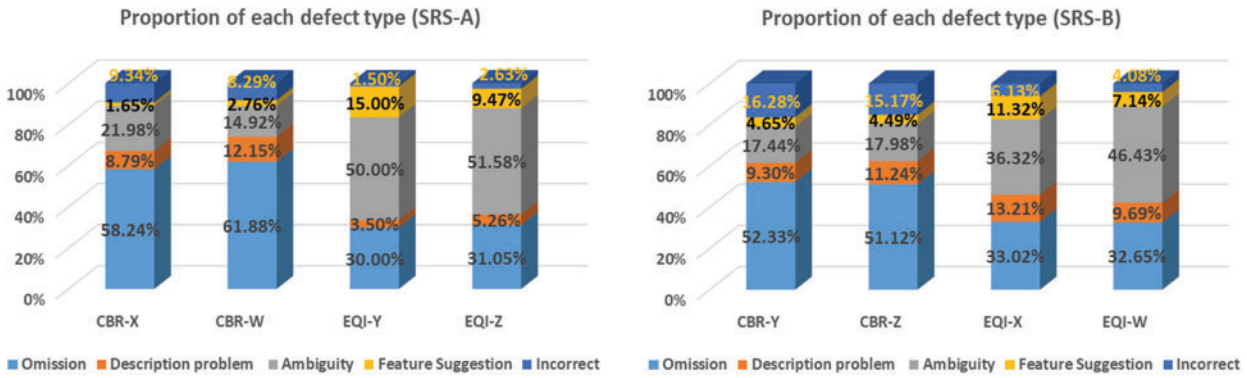

Figure 5: Proportion of each defect in the experiment

4.3.3 Time for Experiment

We note that the experimental time for EQI is greater than that for CBR. However, CBR requires considerable time to prepare the guideline (checklist), and the guideline preparation time is not included in the experimental time. In contrast, EQI does not require guidelines, but generates RCFs based on EQEM. The experimental time of the two methods is shown in Fig. 6.

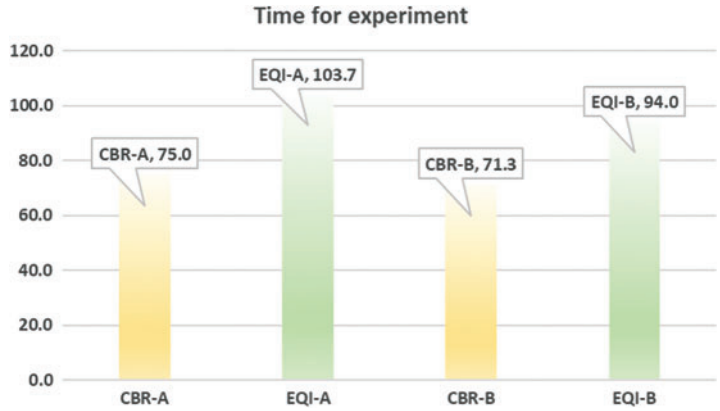

Figure 6: Experimental time required for CBR and EQI (min)

5 Conclusion

In this paper, we proposed an SRS quality assessment and defect detection method–EQI. The experimental results of the comparison between CBR and EQI show that most of the valid questions in the RCFs of EQI can directly point out defects. According to our SRS defect classification, EQI was able to detect more defects than CBR, and the experimental data were statistically significant ($P < 0.05$). However, EQI is less effective than CBR in terms of omission-type defects, and omission-type defects are considered to be the most critical defects in SRS.

In addition, the EQI review does not require guidelines, which saves man-hours and costs for the production of guidelines, but the time cost of EQI implementation is higher than that of CBR. The original intention of EQI is to give a clear numerical indicator to evaluate the quality of SRS, but it has not been empirically studied, which will be future work. We suggest combining EQI with other professional defect detection methods (such as CBR, PBR, or other non-artificial review methods) for SRS review, and applying the EQI method before other review methods may have a better effect on the improvement of SRS quality.

Data Availability Statements: The datasets generated and analyzed during the current study are available in the [Zhi-JSNU/EQI] repository, [https://github.com/Zhi-JSNU/EQI].

Funding Statement: This work was partially supported by the Natural Science Foundation of Jiangsu Province under Grant No. BK20201462, and partially supported by the Scientific Research Support Project of Jiangsu Normal University under Grant No. 21XSRX001.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] L. Wu, N. Che Pa, R. Abdullah and W. N. W. Ab. Rahman, "An analysis of knowledge sharing behaviors in requirement engineering through social media," in *Malaysian Software Engineering Conf. (MySEC)*, Kuala Lumpur, Malaysia, pp. 93–98, 2015.

[2] I. Inayat, S. Salim, S. Marczak, M. Daneva and S. Shamshirband, "A systematic literature review on agile requirements engineering practices and challenges," *Computers in Human Behavior*, vol. 51 (Part B), pp. 915–929, 2015.

[3] J. Polpinij, "An ontology-based text processing approach for simplifying ambiguity of requirement specifications," in *2009 IEEE Asia-Pacific Services Computing Conf. (APSCC)*, Singapore, pp. 219–226, 2009.

[4] A. Takoshima and M. Aoyama, "Assessing the quality of software requirements specifications for automotive software systems," in *2015 Asia-Pacific Software Engineering Conf. (APSEC)*, New Delhi, India, pp. 393–400, 2015.

[5] A. A. Porter, L. G. Votta and V. R. Basili, "Comparing detection methods for software requirements inspections: A replicated experiment," *IEEE Transactions on Software Engineering*, vol. 21, no. 6, pp. 563–575, 1995.

[6] A. B. Rojas and E. G. B. Sliesarieva, "Automated detection of language issues affecting accuracy, ambiguity and verifiability in software requirements written in natural language," in *Proc. of the NAACL HLT 2010 Young Investigators Workshop on Computational Approaches to Languages of the Americas*, Los Angeles, California, pp. 100–108, 2010.

[7] G. Sandhu and S. Sikka, "State-of-art practices to detect inconsistencies and ambiguities from software requirements," in *Int. Conf. on Computing, Communication & Automation*, Greater Noida, India, pp. 812–817, 2015.

[8] A. O. J. Sabriye and W. M. N. W. Zainon, "A framework for detecting ambiguity in software requirement specification," in *2017 8th Int. Conf. on Information Technology (ICIT)*, Amman, Jordan, pp. 209–213, 2017.
[9] M. H. Osman and M. F. Zaharin, "Ambiguous software requirement specification detection: An automated approach," in *IEEE/ACM Int. Workshop on Requirements Engineering and Testing (RET)*, Gothenburg, Sweden, pp. 33–40, 2018.
[10] D. J. A. Cooper, B. R. von Konsky, M. C. Robey and D. A. McMeekin, "Obstacles to comprehension in usage based reading," in *2007 Australian Software Engineering Conf. (ASWEC'07)*, Melbourne, VIC, Australia, pp. 233–244, 2007.
[11] T. Y. Chen, P. -L. Poon, S. -F. Tang, T. H. Tse and Y. T. Yu, "Towards a problem-driven approach to perspective-based reading," in *7th IEEE Int. Symp. on High Assurance Systems Engineering*, Tokyo, Japan, pp. 221–229, 2002.
[12] O. Laitenberger, K. El Emam and T. G. Harbich, "An internally replicated quasi-experimental comparison of checklist and perspective-based reading of code documents," *IEEE Transactions on Software Engineering*, vol. 27, no. 5, pp. 387–421, 2001.
[13] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.
[14] H. Dar, M. I. Lali, H. Ashraf, M. Ramzan, T. Amjad *et al.,* "A systematic study on software requirements elicitation techniques and its challenges in mobile application development," *IEEE Access*, vol. 6, pp. 63859–63867, 2018.
[15] S. Gregory, "It depends": Heuristics for common-enough requirements practice," *IEEE Software*, vol. 35, no. 4, pp. 12–15, 2018.
[16] B. Brykczynski, "A survey of software inspection checklists," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 1, pp. 82–89, 1999.
[17] Z. Peng, T. -H. Chen and J. Yang, "Revisiting test impact analysis in continuous testing from the perspective of code dependencies," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 1979–1993, 2022.
[18] E. Cibir and T. E. Ayyildiz, "An empirical study on software test effort estimation for defense projects," *IEEE Access*, vol. 10, pp. 48082–48087, 2022.
[19] "Institute of Electrical and Electronics Engineers (IEEE), International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), "International standard for software and systems engineering–Software testing–Part 3:Test documentation," *ISO/IEC/IEEE 29119-3:2021(E)*, pp. 1–98, 2021.
[20] S. Vegas, C. Apa and N. Juristo, "Crossover designs in software engineering experiments: Benefits and perils," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 120–135, 2016.
[21] Y. Chernak, "A statistical approach to the inspection checklist formal synthesis and improvement," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 866–874, 1996.