Digital Life

Journal of Digital Life 2024,4,1

Article

Review matching task to diagnose basic review ability

Koki Saito [1] and Shohei Hidaka [2]

[1]BIPROGY Inc., 1-1-1 Toyosu, Koto-ku, Tokyo 135-8560, Japan
[2]Japan Advanced Institute of Science and Technology, 1-1 Asahidai, Nomi, Ishikawa, 923-1292, Japan

Abstract
Software development begins with writing a requirement definition document (RDD) specifying what software is to be built, and the RDD should define the necessary and sufficient conditions the software must satisfy. Preferably, the RDD is reviewed to guarantee its quality. However, the quality of reviews of such documents is not easy to evaluate, due to various review styles and the logical complexity of RDDs. Therefore, we regarded the review ability as the ability to match an RDD with a software and developed a game to make a matching task that can assess review quality. The task has four types of relationships, two-by-two classes of necessity (i.e., the RDD has no irrelevant sentences to define the given software) and sufficiency (i.e., the RDD covers all the parts of the software), between requirements and software, which are expressed in verbal and nonverbal forms. Results suggest that the game likely sufficiently simulated the process of making/reviewing RDD in the requirement definition process. Therefore, it is suggested that the matching task created through the game can be adequate to assess the review ability.

Keywords: requirement definition; necessary and sufficient; review; matching.

1. Introduction

Prior to any software development, developers need to formulate a requirement definition document (RDD) or software requirement specification (SRS) defining the requirements that the software needs to satisfy (IEEE, 1998). Software with the required functions will be developed according to such an RDD. The waterfall model (Royce, 1970) is often used, and the process takes place from the upstream to downstream (Figure 1). In the upstream phase, which occurs prior to software development, the quality of the RDD or SRS is important to ensure to develop a high-quality software in a short period of time, as RDD quality also affects the quality of deliverables in the downstream phase. Boehm and Basili (2001) have shown that it can be 100 times more expensive to correct defects in the downstream phase than in the upstream phase.

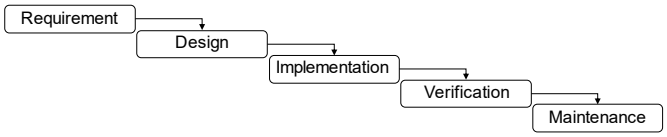


Figure 1: Waterfall model

Reviewing an RDD is a common method to ensure the quality of upstream deliverables; thus, various review methods have been developed. The purpose of reviewing is to detect defects (Ciolkowski et al., 2003), of which 31–93% (median 60%) can be detected by reviewing (Boehm & Basili, 2001). Several comparative studies have been conducted on the differences between and effectiveness of each method (Bernardez et al., 2004; Cantone et al., 2003; Thelin et al., 2003, 2004); however, the best review method for detecting defects has not been established.

Received: 29 March 2024, Revised: 21 April 2024, Accepted: 1 May 2024, Published: 5 June 2024
* Correspondence: koki.saito@biprogy.com

However, individual differences have been reported to have greater effects on review quality than the differences in review methods (Uwano et al., 2007). Thus, review quality likely depends more on the reviewer's ability than the review methods. Although several studies have reported that individual differences have a significant impact on the quality of software development, few have investigated the factors that contribute to individual differences (Uwano, 2011).

What measurement can be considered a good indicator of an individual reviewer's review ability? Previous research has treated *defect detection rate* and *review efficiency* as indicators to evaluate review quality (Thelin et al., 2003; Uwano, 2011). Defect detection rate is defined by the ratio of defects detected by review to the total number of defects, and review efficiency is defined by the number of defects detected per unit time. Defect detection rate cannot evaluate whether a reviewer incorrectly points out some features that are not defects, while review efficiency also depends on the quality of the target document. Namely, a low defect detection rate does not always mean low review ability or high quality of the target document, because some of necessary content to be written may miss in the target document. Therefore, it may be not reliable enough to evaluate individual differences in reviewing only by these indicators.

Another measurement proposed in previous research is *review experience* (Wong, 2003). Wong (2003) wrote "the experience (i.e. knowledge and skills) of reviewers is the most significant input influencing software review performance." (p.301) A study investigating the effects of role experience (review experience) and working experience on performance reported that work experience has a positive effect on performance, while role experience does not have a significant effect on performance (Wong, 2009). Therefore, in the case of reviewers with work experience, a reviewer with substantial review experience is not necessarily effective. If the software reviewed by a reviewer with considerable review experience is not high quality, the reviewer's review ability cannot be evaluated as being high.

As mentioned above, few studies have focused on individual differences, and existing measurements are likely inadequate. To make an indicator of individual review ability, developing a task can be useful. Thelin et al. (2003) developed a task on taxi management systems including an RDD and a design document and compared the defect detection rate and review efficiency. The task could be useful in comparing individual differences because the total number of defects were controlled; however, it depended on specific domain knowledge and took more than 120 minutes on average for the participants to perform. Thus, using only this task to measure individual review ability could be difficult. Therefore, developing a task to assess review ability that does not depend on specific domain knowledge is necessary. To develop such a task, the cognitive process of reviewing RDD must be clarified.

In the waterfall model, reviewers review the deliverables of the current phase based on the deliverables of the previous phase. Thelin et al. (2003) evaluated this review method by presenting both an RDD and a design document in their evaluation study of design document review. During and after the design phase, the reviewers are able to check for defects such as omissions and errors based on the deliverables of the previous phase. In addition, during the requirement phase, the client requirements, which are input into the RDD, are likely regarded as to the deliverables of the previous phase, because the requirement phase is the first phase.

In the V-model, an RDD also corresponds to acceptance testing (Balaji & Murugaiyan, 2012), which verifies software behavior. Thus, the RDD defines the behavior of the software; the RDD is the logic of the computation and the software is the procedure (algorithm) of the computation. Therefore, an RDD is reviewed based on the client's software requirements, and the relationship between RDD and software is the logic and procedure of the computation. A task to assess reviewers' review ability should be developed to have the above characteristics.

In this paper, we developed the task having the above characteristics to assess the ability to review RDD in the software development. By using this task we developed, it can evaluate and quantify individual differences in reviewing.

2. Design

RDD can be interpreted as imperative statements and the software as its answer, because the software corresponds to the procedure of the computation and an answer is derived from the procedure. In addition, using an imperative statement to output multiple interpretable answers is unacceptable. The software required by the client is unique; thus, the RDD must be imperative statements that output unique software. Therefore, it is assumed that a reviewer verifies that an RDD is imperative statements that outputs the unique software that the client requires. A review can also be a task to match an RDD to the client requirements.

In the matching task, the RDD is a verbal object and the client requirement is a nonverbal object, because the RDD is imperative statements and the client requirement is not explicitly articulated. A reviewer likely imagines the software the client requires based on fragmented information (e.g., minutes summarizing the client statements and business flow), and checks whether the RDD outputs the imagined software. The client requirement is a nonverbal object because it indicates the software that the client requires, while the RDD is a verbal object. The reviewer likely matches between the nonverbal object (client requirement) and verbal object (RDD).

Considering the above characteristics of RDD and software, we can define an RDD an an "intensional definition" of a set of properties, and software is an example of an "extensional definition" of a set of those properties. RDD is a linguistic description of the software the client requires and software is a collection of many functions of properties. Intension and extension are terms used in set theory. The intensional definition of a set describes the properties of the elements of that set, and the extensional definition of a set specifically indicates all the elements of it. For example, in reference to the login function of a shopping website, an intensional definition would be a sentence that linguistically describes the condition that must be satisfied by the function, and an extensional definition would be a specific program that has the function (e.g., being able to login through a login page) or presenting such a program. Thus, software development is the process of deciphering the intensional definition that is RDDs describing the properties the client requires, and constructing the software that is an extensional definition.

In the match between the RDD and the software required by the client, four types of relationship exist between intensional and extensional definitions (Figure 2). Let **R (Requirement)** denote the set (product) of the instances (properties in the product) defined by the RDD, and let **P (Product)** denote the set of instances required by the client. In four types, any ideal RDD must be type NS to **P**, if the set **P** is the set of instances that the client requires. **R** and **P** are not sets that satisfy an order relation, but rather sets of conceptual properties.

Type N (lack of function): **R** is necessary, but not sufficient for **P**.
Type S (extra function): **R** is sufficient, but not necessary for **P**.
Type U (lack of function and extra function): **R** is neither necessary nor sufficient for **P**.
Type NS (full function): **R** is both necessary and sufficient.

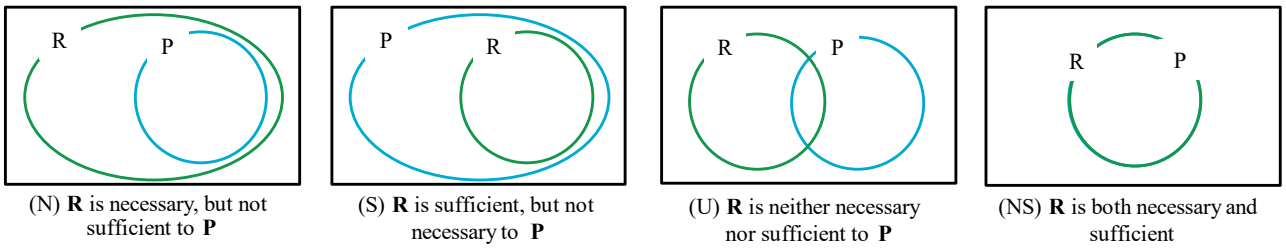


Figure 2: Four types of relationship between set **R** (Requirement) and set **P** (Product)

Therefore, a reviewer verifies whether an RDD is both necessary and sufficient (Type NS) to the software the client requires. In this review process, the reviewer transforms the RDD and software into a state in which they are matchable, because the RDD is verbal object and the software is nonverbal object, and they cannot simply be matched. Subsequently, the reviewer will likely detect/correct defects by judging whether the RDD necessarily and sufficiently satisfies the software requirements.

However, classifying actual RDDs into these four idealized types is difficult, as the complexity of the logical structure is quite high in the natural language and software framework spanned by a set of potential RDDs and software. Therefore, to develop a task to assess reviewers' review ability, we developed a game that simulates interactive communication between clients and developers (client-developer game, CD game) (Figure 3). This game's general scheme was inspired by the iterated learning paradigm in experimental semiotics (Smith et al., 2003). In this game, a set of geometric symbols is used to represent a set of software properties, and clients are supposed to verbally express their requirement definition (RDD). Developers are supposed to produce the set (product) of geometric symbols that the client requests. Through iterated interaction between clients and developers, initially poorly written RDDs would be expected to become good RDDs, which necessarily and sufficiently defines the requested product. As a result of this game, we will obtain a series of pairs of "RDDs" and "products" that vary from poor-matching (Types N, S, or U) to sophisticated (Type NS).

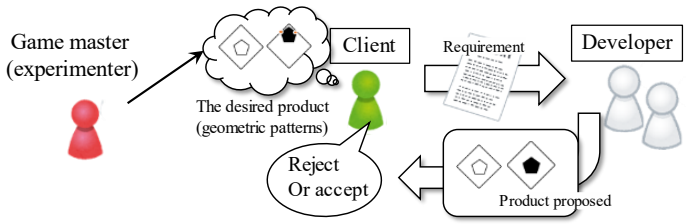


Figure 3: Overview of the one-round interaction between the client and developer

3. Experiment

The client requirement is provided in a linguistic form that describes the properties of the desired product (intensional definition: a set of propositions describing the intended set of objects), and the developer's generated product should be a set of "positive" objects, which are members of the set described by the intensional definition (extensional definition: a set of instances of objects).

In experiment, we employed a combination of geometric patterns, as shown in Figure 4, as a simulated "product" that a client has requested. We designed the desired products, represented by a set of geometric shapes, such that each product consists of the six different shapes, and an RDD, which exactly describes the product, consists of eight or nine sentences. With five pairs of products and correct RDDs, we conducted the CD game.

3.1. Participants

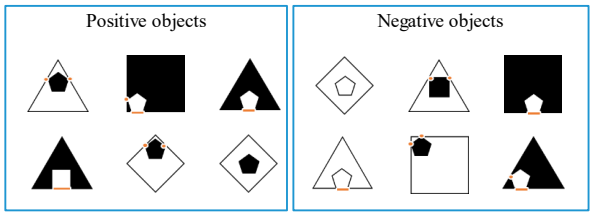
We recruited 10 Japanese adults as the participants (eight men and two women), with an average age of 45.3 years (SD = 9.4). Of the participants, four were in their 30s, one was in their 40s, four were in their 50s, and one was in their 60s. All participants were employed as system engineers at BIPROGY Inc..

3.2. Procedure

Each of the recruited participants was assigned at random to be either a "client" or "developer," resulting in five clients and five developers. Each client was paired with two developers, and played one game with a given visual array as their desired product. Each developer was paired with two distinct clients, and played two games with two distinct (initially unknown) products to generate requirements provided by their client. Thus, one game consisted of one client and two developers, and the union of the two developers' products was provided to the client in feedback on the product. The two developers worked together to ensure that they included the required geometric shapes in the feedback. The two leftmost columns of Table 4 show the player combination for each of the five games.

At the beginning of each game, each of the five clients was exposed to a visual array of positive (desired) and negative (non-desired) objects. Five different sets of objects were made, and each client was exposed to a different set. Figure 4(a) shows one of the used visual arrays of the positive and negative objects. The client was then asked to form a list of sentences describing the positive objects but excluding the negative ones. The detail of the rules for forming the list of sentences is described in Section 3.3.2. After the client formed the list, it was shown to the corresponding developer, and the developer was asked to form a visual array of geometric objects according to the rules, described in Section 3.3.1. Then, the union of the two visual arrays created by the two developers was sent back to the client. This exchange was one round. If the visual array created by the developer was the exact same as the "correct" visual array that was shown to the client (an example of the "correct" list of requirement for the product in Figure 4(a) is shown in Figure 4(b)), then the game ended at this round. Otherwise, the next round of exchange was repeated up to the fourth round. Both clients and developers were instructed on their goal and the rules in the game before they began.

(a)



(b)

The list of requirements

1. Large is an equilateral triangle or a quadrilateral
2. Small is a square or a regular pentagon
3. If large side and small side touch only one side , large is black painted
4. If large is a square , large side and small point touch only three points
5. If large is an equilateral triangle , large side and small point touch only two points
6. Large side and small point touch two or more , or do not touch
7. If large side and small side do not touch , small is black painted
8. If small is black painted , small is a regular pentagon

Figure 4: (a, left) The targeted list of positive objects, (a, right) an example list of negative objects, and (b) example of the list of requirements.

In the first round, two examples were provided for all client and developer groups to allow the participants to familiarize themselves with the experiment. These examples were shown to help participants understand the rules of the game. Thus, the clients and developers did not engage in any rounds of exchange. Clients formed a list of sentences, and then the example correct answers were presented. Developers created a visual array of geometric objects based on the client's list, and then the example correct answers were presented.

After the experiment, all participants were asked to answer a demographic questionnaire and were interviewed to collect personal characteristics.

3.3. Materials

To equalize game difficulties with different sets of products, we designed each visual array to consist of six pairs of unit shapes, which could be described with eight or nine sentences in the list of requirements.

3.3.1. Design principles for the positive list of objects

Allowing for the free-forming of geometric shapes could result in excessively long sentences or sentence exceptions to avoid ambiguity in the RDD. Therefore, to avoid these problems, the geometric shapes and sentences used in the experiment were restricted and equivalence groups were defined. Specifically, 20 types of unit shapes that could be used to create a "product" or a visual array of geometric shapes were defined (Figure 5). The client was allowed to use at most one unit shape for each type, and combined two to create an object (a set of six pairs of unit shapes makes a product).

For both positive and negative objects, we set the following rules for unit shape manipulation to reduce the number of possible combinations to generate an object:

1. Every object must consist of two unit shapes, one small and one large, and the large shape needs to contain the small one.
2. Only allowed manipulation of each unit shape is translation (no rotation, scaling, or mirroring transformation).
3. The large and small shapes can be either black or white, unless both are in black.
4. The large and small unit shapes can only be related in nine ways, (a)–(i) in Table 1, and the example as depicted in Figure 6. Only the difference in how large and small touch is considered as a combination, not the position of the touching points, sides.
   Table 1 shows the number of possible combinations for forming an object under the above rules.

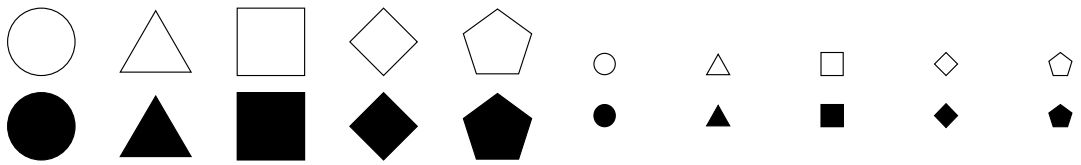


Figure 5: Twenty types of unit shapes

Table 1: Number of possible combinations to form an object

point and point	side and point	side and side	combination	number
touch 0 point	touch 0 point	touch 0 side	(a)	75
	touch 1 point	touch 0 side	(b)	48
	touch 2 point	touch 0 side	(c)	48
		touch 1 side	(d)	30
	touch 3 point	touch 1 side	(e)	9
touch 1 point	touch 0 point	touch 0 side	(f)	24
	touch 1 point	touch 1 side	(g)	9
	touch 2 point	touch 2 side	(h)	12
touch 2 point	touch 0 point	touch 0 side	(i)	12
			total	267

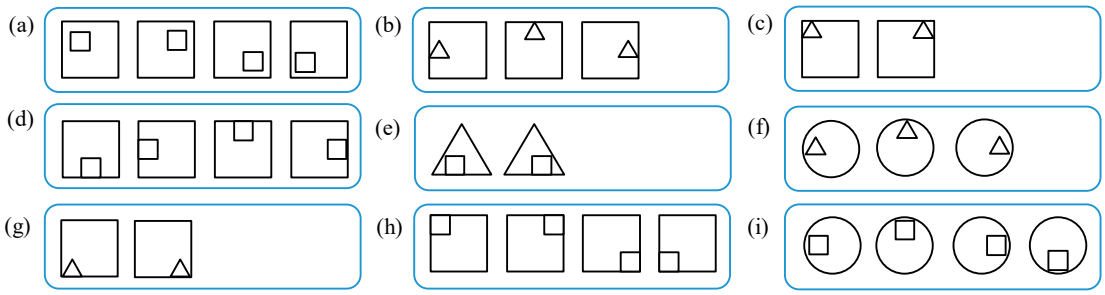


Figure 6: Example of an equivalence group (the objects in the blue frame are the same equivalence group)

3.3.2. Rules for verbal expression to form client requirements

In the verbal expression of client requirements, terms and description rules that the client could use were restricted by the rules listed below to create as few variations as possible in expressions among the participants.

The terms available for client requirements (translation from the original Japanese):

- Shape: *circle, equilateral triangle, square, rhombus, regular pentagon*
- Quadrilateral: *square, rhombus*
- Regular polygon: *equilateral triangle, square, regular pentagon*
- Polygon: *equilateral triangle, square, rhombus, regular pentagon*
- Size: *large, small*
- Paint: *white painted, black painted*
- Number of sides/vertices: circle is *zero*, equilateral triangle is *three*, square and rhombus are *four*, and regular pentagon is *five*
- Touch: if the borders of the unit shapes overlap, it is stipulated that they touch and the requirement is described in the nine ways (Rule 4)

The syntax available for client requirements:

- A is/is not B
- If A is/is not B, C is/is not D

In A, B, C and D, it was possible to describe only one type of requirement. If it is the same type of the requirements, it is possible to describe multiple with “or”. Table 2 shows the number of possible combinations to form “A is B” syntax under the rules above. It is possible to describe the same number of combinations in “A is not B” syntax and “C is/is not D” syntax. Thus, it is possible to describe a total of 18528 sentences: 192 sentences in the “A is/is not B” syntax and 18336 sentences in the “If A is/is not B, C is/is not D” syntax. However, the actual number of sentences that can be used is less than 18528, because some combinations of sentences contradict “A is/is not B” and “C is/is not D” in “If A is/is not B, C is/is not D” syntax.

Table 2: Number of possible combinations to form “A is B” syntax

	A	B	number
rules for shape	large	choose one of the five shapes	5
		connect two of the five shapes with "or"	10
		connect three of the five shapes with "or"	10
		connect four of the five shapes with "or"	5
		connect five of the five shapes with "or"	1
	small	choose one of the five shapes	5
		connect two of the five shapes with "or"	10
		connect three of the five shapes with "or"	10
		connect four of the five shapes with "or"	5
		connect five of the five shapes with "or"	1
	large and small	same shape	1
		different shape	1
rules for paint	large	white painted	1
		black painted	1
	small	white painted	1
		black painted	1
	large and small	same painted	1
		different painted	1
rules for touch	point and point	touch 0 point	1
		touch 1 point	1
		touch 2 point	1
		connect two of the touch condition with "or"	3
	side and point	touch 0 point	1
		touch 1 point	1
		touch 2 point	1
		touch 3 point	1
		connect two of the touch condition with "or"	6
		connect three of the touch condition with "or"	4
	side and side	touch 0 side	1
		touch 1 side	1
		touch 2 side	1
		connect two of the touch condition with "or"	3
		total	96

4. Results

4.1. Client group

Two of the five clients ended the game in the second round, one ended it in the third round, one ended it in the fourth round, and one did not have an acceptable product by the end of fourth round.

Table 3 shows the change in the type of this set-diagram relationship for each round. Although in some relationship types **R** was not necessary for **P** (type S and U) in Client Groups 3 and 5 in Round 1, clients showed a tendency to first form the list of sentences (requirements) that satisfied the necessary condition (type N). Then, clients modified the requirements to satisfy the sufficiency (type NS), based on visual array of geometric objects provided by the developer group. Type NS was always preceded by type N, and no client was able to create the requirement that was type NS without verification by the developer group. No pattern of changing from type N to type S or U was observed.

Figure 7 shows the changes in the number of sentences in the requirements per round for each client. In some cases, as the round progressed, the number of sentences in the requirements decreased; however, this remained constant or increased in many cases. When the relationship type changed from only necessary (type N) to necessary and sufficient (type NS), the number of sentences in the requirements did not decrease, but instead remained constant or increased. Therefore, the clients were likely to enumerate the client requirements to satisfy the positive objects, and thereafter define the requirements that satisfied the necessary and sufficient conditions by adding or modifying the requirements to satisfy the sufficient condition.

Table 3: Changes in type of relationship between set **R** and set **P**.

client group	type of relationship			
	round 1	round 2	round 3	round 4
1	N	NS	-	-
2	N	NS	-	-
3	U	N	N	NS
4	N	N	N	N
5	S	N	NS	-

N: Requirement is only necessary, S: Requirement is only sufficient, U: Requirement is neither necessary nor sufficient, NS: Requirement is both necessary and sufficient

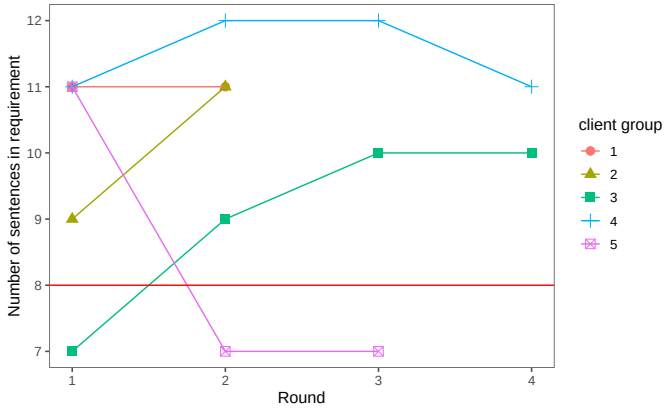


Figure 7: Change in the number of sentences in client requirements in each round (red line is the number of sentences that the experimenter set in advance)

4.2. Developer group

Table 4 shows the Jaccard index (Jaccard, 1912) indicating the degree of similarity between the objects formed by the developer group and the positive objects that can be made from the requirements in the client group. The Jaccard index indicates the similarity between two sets, defined as follows:

$$\mathrm{J}(R,P) = \frac{|R \cap P|}{|R \cup P|}, \tag{1}$$

where R is the set of positive objects that can be made from the requirements in the client group, and P is the set of objects formed by the developer group.

R is equal to six positive objects for type NS, and more than six positive objects for type N. In this experiment, the Jaccard index indicates the ability of the developer group to form objects from the list of sentences, and the closer the index is to one, the more the developer is able to form the necessary and sufficient objects that can be made from the list of sentences. The set R in each round is constant for the list of sentences, and the Jaccard index depends on the set P formed by the developer group; therefore, when the Jaccard index is one, $|R \cap P| = |R \cup P|$ and $R = P$ is true, and the Jaccard index is an indicator of being necessary and sufficient.

The Jaccard index tended to increase as the rounds progressed for many participants, indicating that the developers tended to be able to make the objects that satisfied the requirements defined by the client group necessary and sufficient as the rounds progressed. However, in some cases, the Jaccard index decreased significantly in the subsequent round. Thus, we conducted a paired t-test between the first and final rounds, and found no significant differences ($t = -1.271, p = 0.220$). In this experiment, the developer group did not receive a penalty even if the negative objects were mistakenly made as the positive objects, and such cases were not presented to the client group. For this reason, the participants were likely concerned that the game would not end, and made many objects, including negative ones, to ensure comprehensiveness, and as a result, the Jaccard index decreased.

Table 5 shows the coverage indicating how many objects made by the developer group cover **R**. The coverage is defined as follows:

$$C(R, P) = \frac{|R \cap P|}{|R|}. \quad (2)$$

Although the Jaccard index indicates the ability to accurately form objects based on client requirements, some developers made many negative objects as positive objects to increase the comprehensiveness because making such objects did not carry a penalty. In such cases, the Jaccard index decreases. When the Jaccard index was small, whether the developers were making many negative objects as positive objects or not making the necessary positive objects was difficult to judge. Thus, we introduced coverage that indicated the coverage rate to evaluate whether the developers were making the necessary positive objects that could be made from the client requirements. The closer the coverage is to one, the more the developer is able to form the necessary objects that can be made from the list of sentences. Therefore, when the coverage is one, $|R \cap P| = |R|$ and $R \subseteq P$ is true, the coverage is an indicator of sufficiency. A good developer has high values for both the Jaccard index and coverage.

These results indicate that one participant verifying the client requirements is not enough, because individual coverage varied and was too low in some cases. However, the union coverage of two participants was over 86% in all rounds. Hence, two people verifying the client requirements was likely enough. We also conducted a paired t-test between the first and final rounds and found a significant difference ($t = -2.167, p = 0.0474$). This suggests that the quality of the developer review and of the requirements is enhanced by repeating reviews (rounds).

Table 4: Jaccard index (distance between two sets) of the sets of objects generated by the developer and client

client group	developer group	Jaccard index			
		round 1	round 2	round 3	round 4
1	10	1.00	1.00	-	-
	9	0.88	0.86	-	-
2	9	0.60	1.00	-	-
	8	0.39	0.55	-	-
3	8	0.61	1.00	0.43	1.00
	7	0.75	0.36	0.75	0.63
4	7	0.18	0.22	0.44	0.71
	6	0.78	0.50	1.00	1.00
5	6	0.75	0.94	0.35	-
	10	0.67	0.94	0.83	-

Table 5: Coverage of the positive (desired) objects made by the developer for set **R**

client group	developer group	coverage							
		round 1		round 2		round 3		round 4	
		each	union	each	union	each	union	each	union
1	10	100%	100%	100%	100%	-	-	-	-
	9	88%		100%		-		-	
2	9	67%	89%	100%	100%	-	-	-	-
	8	78%		100%		-		-	
3	8	79%	86%	100%	100%	100%	100%	100%	100%
	7	86%		71%		100%		83%	
4	7	38%	88%	29%	86%	57%	100%	71%	100%
	6	88%		86%		100%		100%	
5	6	100%	100%	100%	100%	100%	100%	-	-
	10	67%		94%		83%		-	

5. Discussion

No clients were able to form type NS requirements in the first round without verification by the developers. This suggests that forming a list of sentences (RDD) that satisfies a client's requirements without any reviews is difficult, even for a relatively simple task, such as creating requirements that satisfy necessary and sufficient conditions for geometric objects. In addition, most clients were able to create a type NS requirement in the final round by the verification of the developers. The coverage significantly increased in the final round compared with in the first round, and repeating reviews likely improved the quality of the requirements. This experiment's results indicate that reviews have a positive effect on the quality of at least symbolized products, and reviews are effective for RDDs. Furthermore, we found that the Jaccard index is an indicator of necessity and sufficiency, and coverage is an indicator of sufficiency.

In addition, we found that multiple reviewers were more effective than a single reviewer because some developers showed low coverage. Although the minimum union coverage of two developers was 86%, the minimum individual coverage was 29%, indicating the limitations of a single review.

Therefore, the results indicate that multiple rounds and verification by multiple people are needed to satisfy the requirement of necessary and sufficient conditions. In the actual requirement definition process, the same applies to making an RDD. The client/developer group in this experiment appeared to play a role like the requirement definition maker/reviewer in the requirement definition process. Therefore, this experiment likely sufficiently simulates the process of making/reviewing RDDs in the requirement definition process.

6. Conclusions

In this study, we developed the review matching task to assess the ability to review RDD in the software development through CD game. We defined review ability as the ability to match an RDD with client requirements and found that the relationship between the two can be divided into four types (Figure 2). The results suggest that the game has the similarities to the process of making RDD and the matching task created through the game can be used to evaluate people's review ability.

In the future, we will assess the effectiveness of the matching task and investigate differences in review ability for the four types of relationships.

Author Contributions

Conceptualization, K.S. and S.H.; methodology, K.S. and S.H.; software, K.S.; validation, K.S. and S.H.; formal analysis, K.S.; investigation, K.S.; resources, K.S.; data curation, K.S.; writing—original draft preparation, K.S.; writing—review and editing, K.S. and S.H.; visualization, K.S.; supervision, S.H.; project administration, K.S. All authors have read and agreed to the published version of the manuscript.

Funding

This research received no external funding.

Institutional Review Board Statement

The study was approved by the Ethics Committee of Japanese Association for the Promotion of State of the Art in Medicine (2019-06).

Informed Consent Statement

Informed consent was obtained from all subjects involved in the study.

Conflicts of Interest

The authors declare no conflict of interest.

References

Balaji, S., & Murugaiyan, M. S. (2012). Waterfall vs. V-Model vs. Agile: A comparative study on SDLC. *International Journal of Information Technology and Business Management*, *2*(1), 26–30.

Bernardez, B., Genero, M., Duran, A., & Toro, M. (2004). A controlled experiment for evaluating a metric-based reading technique for requirements inspection. *10th International Symposium on Software Metrics, 2004. Proceedings.*, 257–268. https://doi.org/10.1109/METRIC.2004.1357908

Boehm, B., & Basili, V. R. (2001). Top 10 list [software development]. *Computer*, *34*(1), 135–137. https://doi.org/10.1109/2.962984

Cantone, G., Colasanti, L., Abdulnabi, Z. A., Lomartire, A., & Calavaro, G. (2003). *Evaluating Checklist-Based and Use-Case-Driven Reading Techniques as Applied to Software Analysis and Design UML Artifacts BT - Empirical Methods and Studies in Software Engineering: Experiences from ESERNET* (R. Conradi & A. I. Wang (eds.); pp. 142–165). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-45143-3_9

Ciolkowski, M., Laitenberger, O., & Biffl, S. (2003). Software reviews, the state of the practice. *IEEE Software*, *20*(6), 46–51. https://doi.org/10.1109/MS.2003.1241366

IEEE. (1998). IEEE 830: Recommended Practice for Software Requirements Specifications. In *IEEE Std 830-1998* (Vol. 1998). https://doi.org/10.1109/IEEESTD.1998.88286

Jaccard, P. (1912). The distribution of the flora in the Alpine Zone.1. *New Phytologist*, *11*(2), 37–50. https://doi.org/10.1111/j.1469-8137.1912.tb05611.x

Royce, W. W. (1970). Managing the development of large software systems. proceedings of IEEE WESCON. *Los Angeles*, 328–388.

Smith, K., Kirby, S., & Brighton, H. (2003). Iterated learning: A framework for the emergence of language. *Artificial Life*, *9*(4), 371–386. https://doi.org/10.1162/106454603322694825

Thelin, T., Andersson, C., Runeson, P., & Dzamashvili-Fogelstrom, N. (2004). A replicated experiment of usage-based and checklist-based reading. *10th International Symposium on Software Metrics, 2004. Proceedings*, 246–256. https://doi.org/10.1109/METRIC.2004.1357907

Thelin, T., Runeson, P., & Wohlin, C. (2003). An experimental comparison of usage-based and checklist-based reading. *IEEE Transactions on Software Engineering*, *29*(8), 687–704. https://doi.org/10.1109/TSE.2003.1223644

Uwano, H. (2011). Measuring and characterizing eye movements for performance evaluation of software review. *Eye Movement: Theory, Interpretation, and Disorders*.

Uwano, H., Nakamura, M., Monden, A., & Matsumoto, K. (2007). Exploiting eye movements for evaluating reviewer's performance in software review. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, *E90-A*(10), 317–328. https://doi.org/10.1093/ietfec/e90-a.10.2290

Wong, Y.-K. (2003). An exploratory study of software review in practice. *PICMET '03: Portland International Conference on Management of Engineering and Technology Technology Management for Reshaping the World, 2003.*, 301–308. https://doi.org/10.1109/PICMET.2003.1222807

Wong, Y.-K. (2009). Software quality and group performance. *AI & Society*, *23*(4), 559–573. https://doi.org/10.1007/s00146-007-0174-6