

Comparing the understandability of iteration mechanisms over Collections in Java

Bogumiła Hnatkowska, Bartosz Krych*

Abstract. Source code understandability is a desirable quality factor affecting long-term code maintenance. Understandability of source code can be assessed in a variety of ways, including subjective evaluation of code fragments (perceived understandability), correctness, and response time to tasks performed. It can also be assessed using various source code metrics, such as cyclomatic complexity or cognitive complexity. Programming languages are evolving, giving programmers new ways to do the same things, e.g., iterating over collections. Functional solutions (lambda expressions and streams) are added to typical imperative constructs like iterators or *for-each* statements. This research aims to check if there is a correlation between perceived understandability, understandability measured by task correctness, and predicted by source code metrics for typical tasks that require iteration over collections implemented in Java. The answer is based on the results of an experiment. The experiment involved 99 participants of varying ages, declared Java knowledge and seniority measured in years. Functional code was perceived as the most understandable, but only in one case, the subjective assessment was confirmed by the correctness of answers. In two examples with the highest perceived understandability, streams received the worst correctness scores. Cognitive complexity and McCabe's complexity had the lowest values in all tasks for the functional approach, but – unfortunately – they did not correlate with answer correctness. The main finding is that the functional approach to collection manipulation is the best choice for the filter-map-reduce idiom and its alternatives (e.g., filter-only). It should not be used in more complex tasks, especially those with higher complexity metrics.

Keywords: source code quality, maintenance, understandability, perceived understandability, complexity metrics, Java, streams, for-each, iterator.

1. Introduction

Source code understandability is a desirable quality factor affecting long-term code maintenance. It can be defined as the degree to which the purpose of the code is clear to the

* Wrocław University of Science and Technology, Wybrzeże Wyspiańskiego 27, 50-370 Wrocław, Poland, bogumila.hnatkowska@pwr.edu.pl

inspector [1]. Greater understandability means greater source code analyzability and modifiability.

Understandability of source code can be assessed in several ways, including subjective evaluation of code fragments (perceived understandability), correctness, and response time to tasks performed. It can also be assessed by various source code metrics, such as cyclomatic complexity or cognitive complexity [1], [6], [7].

Programming languages are evolving, giving programmers new ways to do the same things. In Java, for example, programmers can iterate over collections using iterators (introduced in Java 1.2), *for-each* loop (introduced in Java 1.5), or streams (introduced in Java 8). The latter mechanism is representative of the functional paradigm in an imperative language. The question arises of which iteration mechanism is best understood and in which cases.

This research aims to check if there is a correlation between perceived understandability and understandability measured by task correctness and evaluated by some source code metrics for typical tasks implemented in different ways (iterator, *for-each*, stream). The results will allow making recommendations on the type of task and its understandability depending on the mechanisms used (see Section 5). They were formulated based on the results of an experiment. The experiment consisted of six tasks of increasing complexity, such as filtering a collection, finding the first element in the collection that satisfies a condition, applying a filter-map-reduce construction, or constructing a histogram. The experiment involved 99 participants of various ages, Java knowledge, and skill levels. Details presenting the research plan are given in Section 3, and a discussion of the results – is in Section 4.

2. Related works

The chapter tries to answer two research questions:

Q1: Which interfaces and their operations that manipulate over collections are most popular?

Q2: Which constructs (imperative/functional) are better understood?

The answer to the first question will make it possible to plan an experiment, while the answer to the second question will serve as a benchmark for the experiment results.

According to [4], the most used interfaces are *List* (56%) and *Map* (15%). The results were taken from nearly 11,000 Java projects. The most widely used implementation of the *List* interface is *ArrayList* (84%). Comparable results are presented in [10].

As for the lambda expressions used together with the stream interface, they are usually used as single elements (without nesting) [13]. The stream interface is part of the "filter-map-reduce" idiom, with at least some of its elements presented (in order: *map*, *collect*, *filter*) [8]. Quite often, the collect operation is substituted with another, selecting a single element from the input collection, namely *anyMatch* or *findFirst* [11][8]. In Java 17, the collect method was replaced by the *toList* method [13]. According to [13], developers rarely use parallel processing, and most of the parallel streams are applied on a simple *for-each* operation.

Many authors point to the dependence of understandability on complexity [5], [15], which can be measured with Line of Codes (LoC) [14], [8], McCabe complexity [6], Halstead complexity [5], or Cognitive Complexity [1]. Functional code is less complex than its object-

oriented version [6], [14], therefore, from that perspective, it seems to be more readable. However, some research, such as [6], does not confirm the link between the metrics and understandability.

Understandability comparison of functional and imperative constructs is the subject of much research, e.g., [6], [9]. The results of such a comparison presented in [6] and [9] are inconclusive. The perceived readability of streams and lambdas is better than of their object-oriented equivalents, which does not correlate with the quantitative assessment made by metrics [6]. According to [9], streams with lambda expressions are more readable for typical applications like the “filter-map-reduce” pattern [9]. The authors of [8] observed the increased trend of using streams and lambda expressions over the years and asked why they were introduced. In the first place, they got the answer “to make existing code more succinct and readable.”

The answer to the second research question is equivocal. The results of the code complexity metrics values (most of them) and the perceived readability assessment suggest that functional constructs are better understood. On the other side, objective values, such as task correctness or response time, may suggest otherwise.

3. Research plan

An experiment was planned and conducted to answer the research question about the correlation between the phenomena: perceived understandability, objective understandability, and source code selected metrics. A survey consisting of three parts was directed to programmers. The first part was designed to collect basic data about the attendants (age, gender, skills in programming in general, level of Java proficiency, and years of work experience). Participants with no programming skills were excluded from the experiment.

The second part contained 6 code snippets for which the programmer had to choose the correct answer from a list of suggestions (subject to objective understandability assessment). The suggestion list included a “don’t know” answer, which the participant could choose when he had no idea how a code fragment behaved and wanted to avoid guessing. The snippets and answers presented to participants are available in the Appendix.

The third part contained solutions to the same six problems written in three ways (iterator, *for-each* statement, stream with lambda expression). The attendants were asked to order the snippets from most to least understandable (subject to an assessment of perceived understandability). The choice of the mechanisms to check was based on the answers to the first research question about the most popular interfaces and their operations that manipulate over collections. The *iterable.forEach(lambda-exp)* construct has been intentionally omitted since this operation returns no value, and its application to tasks such as filtering or collection reduction is questionable. Moreover, it can be considered as a shortcut of *iterable.stream().forEach(lambda-exp)*.

The survey was prepared in three versions (with Microsoft Forms), for which the second part was structured differently. Participants were presented with a different task implementation depending on the survey version (group) – see Table 1. Every third person was assigned to the same group.

The survey was conducted between the 15th of April and the 15th of May 2022. Employees of IT companies, e.g., Unity Group, Nokia, and Capgemini, were asked to participate. Wrocław University of Science and Technology students were also encouraged to attend.

Table 1. Type of tasks in each group

	Group 1	Group 2	Group 3
Task 1	iterator	<i>for-each</i>	stream
Task 2	stream	iterator	<i>for-each</i>
Task 3	<i>for-each</i>	stream	iterator
Task 4	iterator	<i>for-each</i>	stream
Task 5	stream	iterator	<i>for-each</i>
Task 6	<i>for-each</i>	stream	iterator

The following metrics were chosen for the research: CogC (Cognitive Complexity), v(G) – McCabe complexity, and LoC – Lines of code. After the source code automatic reformatting, they were taken with the *MetricsReloaded* tool.

The second and third parts of the survey contained six tasks to be either solved (a correct answer from a list should be selected) or to be ordered in terms of perceived readability. These tasks are summarized in Table 2. Their content (source codes) is presented in Appendix.

Table 2. Tasks description

Task	Description
1. Collection filtering	Collection filtering is among the most often used [8], [13]. The same can be said of the sequence of operations when filtering results are returned as a new collection [8].
2. Searching for the first element satisfying a condition	A case is taken from [2].
3. Calculation of the sum of the collection elements	A case inspired by [3]. Includes a map operation in the stream version (the most often used according to [13][8]).
4. Counting the number of unique elements satisfying a condition	Inspired by [13]. The main difference is that the set interface was used to find unique elements in the iterator and <i>for-each</i> version.
5. Reducing a collection to a string of characters that are in the same position in an element of the collection as the element itself	Inspired by [2].
6. Histogram	Inspired by [2]. The original code was simplified by eliminating a set instance creation.

4. Results and discussion

4.1. Attendants

The experiment involved 99 programmers, 32-34 in each group. Only 10 (~10%) claimed to be unfamiliar with Java. A third (35 people) rated their knowledge of Java as very good and 9 as an expert.

Forty programmers had little (less than 3 years) experience; on the other side, 30 had significant experience (more than 5 years).

4.2. Source code metrics

The values of the metrics for the considered tasks can be seen in Table 3. As can be observed, the complexity values are equal for the iterator and *for-each* implementation. They differ only for the LoC metric (the former implementation is longer). In all cases, the metrics values are best for streams. The difference is noticeable, especially for tasks 1, 2, and 4.

Table 3. Metrics values

Task #	Interface	CogC	V(G)	LoC
1	iterator	3	3	11
	<i>for-each</i>	3	3	9
	stream	0	1	5
2	iterator	3	3	10
	<i>for-each</i>	3	3	8
	stream	0	1	6
3	iterator	1	2	8
	<i>for-each</i>	1	2	7
	stream	0	1	5
4	iterator	3	3	11
	<i>for-each</i>	3	3	9
	stream	0	1	7
5	iterator	3	3	13
	<i>for-each</i>	3	3	11
	stream	2	2	12
6	iterator	1	2	9
	<i>for-each</i>	1	2	8
	stream	0	1	5

4.3. Objective understandability

Correctness results (in percentage) for all tasks are given in Table 4. The table additionally includes the results of perceived understandability (the “Points” row), which is described in detail in the following subsection.

The results show that the *for-each* statement, found in many programming languages, is well understood (for 5 out of 6 cases, the sum of incorrect and “don’t know” responses were the lowest; 4 out of 6 cases got the highest number of correct answers).

Unfortunately, the nominal scale used in the objective assessment of understandability limited the possibility of statistical analysis to frequency analysis only.

Table 4. Assessment of objective and subjective understandability

Task #	Answer	Iterator	For-each	Stream	Total
1	Correct	94.1%	90.6%	81.8%	88.9%
	Incorrect	2.9%	6.3%	18.2%	9.1%
	Don’t know	2.9%	3.1%	0.0%	2.0%
	Points	23	115	159	
2	Correct	59.4%	72.7%	58.8%	63.6%
	Incorrect	34.4%	27.3%	38.2%	33.3%
	Don’t know	6.3%	0.0%	2.9%	3.0%
	Points	26	138	132	
3	Correct	75.8%	85.3%	81.3%	80.8%
	Incorrect	18.2%	14.7%	6.3%	13.1%
	Don’t know	6.1%	0.0%	12.5%	6.1%
	Points	33	126	138	
4	Correct	61.8%	46.9%	84.8%	64.4%
	Incorrect	32.4%	46.9%	12.1%	30.3%
	Don’t know	5.9%	6.3%	3.0%	5.1%
	Points	25	122	150	
5	Correct	75.0%	75.8%	58.8%	69.7%
	Incorrect	9.4%	15.2%	17.6%	14.1%
	Don’t know	15.6%	9.1%	23.5%	16.2%
	Points	68	176	53	
6	Correct	63.6%	70.6%	62.5%	65.7%
	Incorrect	6.1%	14.7%	9.4%	10.1%
	Don’t know	30.3%	14.7%	28.1%	24.2%
	Points	39	124	135	

Streams scored worse in 4 cases regarding correct answers and 3 cases regarding the highest number of incorrect answers. A detailed analysis is provided below.

Task 1 was the easiest, with the number of correct answers at a general level of about 88%. This is the only case for which the iterator got the best results (the largest number of correct answers and the smallest sum of incorrect and do not know). Surprisingly, the idiomatic version of filtering a collection earned the worst results (the smallest percentage of correct answers and the largest percentage of incorrect).

Task 2 was the trickiest – it got the highest number of incorrect answers, similar in all versions ($33 \pm 6\%$). Here, the *for-each* statement did the best score (zero “don’t know” responses and the highest number of correct ones).

For-each statement and streams were equally assessed in Task 3. Both performed better than iterators.

Task 4 got the most confusing results with the largest differences between extremes (almost 38 percentage points for correct answers and 35 percentage points for incorrect answers in favor of streams).

The opposite result for Task 4 was achieved for Task 5. Here the imperative implementations worked equally, while streams performed notably worse.

The last Task 6 was the least readable with the highest number of “don’t know” answers – here, the iterator and stream were scored similarly.

The results should be interpreted in context. Figure 1 presents the number of responses falling into a specific category depending on the participant's declared Java expertise.

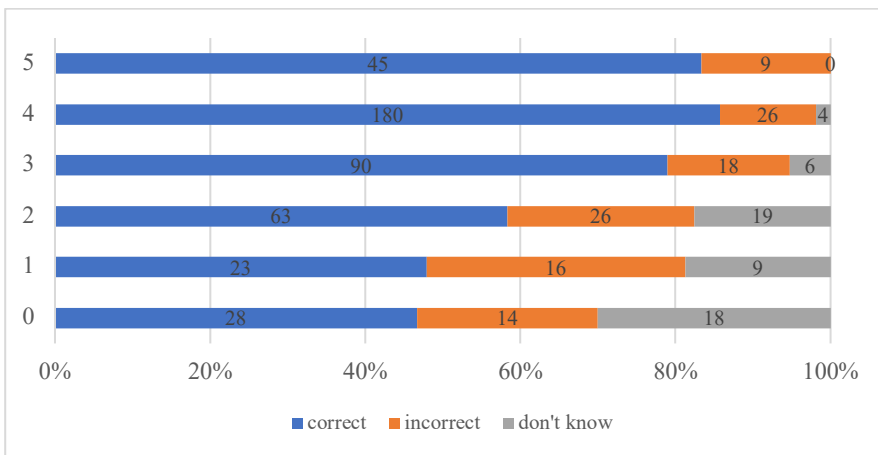


Figure 1. Answer correctness (percentage, nominative) vs. declared Java knowledge (0-5)

More experienced people were less likely to give “don’t know” answers, and the answer accuracy was higher than in other groups (a slight anomaly was observed in the expert group). The tasks (2, 4, 5) with the highest number of incorrect and “don’t know” answers were further analyzed.

4.3.1. Detailed analysis of Task 2

Figure 2 presents a matrix with the number of incorrect answers (the two most often were chosen) depending on the version of the implementation and the declared level of Java knowledge (0 – novice, 5 – expert) for Task 2. The level of language knowledge is hardly recognizable (slightly better results are observed in the groups marked as 4-5), but the implementation ratings clearly differ (8 incorrect answers for the iterator, 4 – for *for-each* statement, 12 – for stream).

	Java expertise	<i>for-each</i>	iterator	stream	Sum
0		1	2	1	4
	[Krych, Krych]	1	1	1	3
	[Krych]		1		1
1		1	2	3	6
	[Krych, Krych]	1	2	3	6
2		1	1	2	4
	[Krych, Krych]	1			1
	[Krych]		1	2	3
3		1	2	2	5
	[Krych, Krych]		2		2
	[Krych]	1		2	3
4				2	2
	[Krych]			2	2
5			1	2	3
	[Krych, Krych]		1		1
	[Krych]			2	2
Sum		4	8	12	24

Figure 2. Number of incorrect answers grouped by Java language and incorrect answer per iteration mechanism

It might be that the reason for such many incorrect answers for the stream implementation was caused by hast and inattention. As many as 8 people in the stream group chose the wrong answer “[Krych]” instead of the correct “Krych.”

4.3.2. Detailed analysis of Task 4

In Task 4, the dominant wrong answer was “7” (19 out of 35, 18 for iterator and *for-each*), representing the collection size after filtering without assuring that the collection should contain unique elements only.

The worst result for *for-each* statement was due to the use of the *Set* interface to achieve the uniqueness of elements, which the survey participants did not notice. The same interface was also used in the iterator implementation, but with fewer negative side effects.

As in the previous case, seniority or Java proficiency did not affect the results.

4.3.3. Detailed analysis of Task 5

Task 5 got a considerable number of “don’t know” answers (8 – streams, 5 – iterator, 3 – *for-each*), so the authors tried to find the possible reasons for this phenomenon. Figure 3 presents how the number breaks down into groups defined by seniority level and declared Java knowledge. The stream implementation was complex for commercially inexperienced people to understand. The most readable – from this perspective – was the *for-each* statement.

Regarding the knowledge level of Java, those with lower levels had problems with iterators, which is not surprising. Again, streams were the worst and, *for-each* the best readable construct.

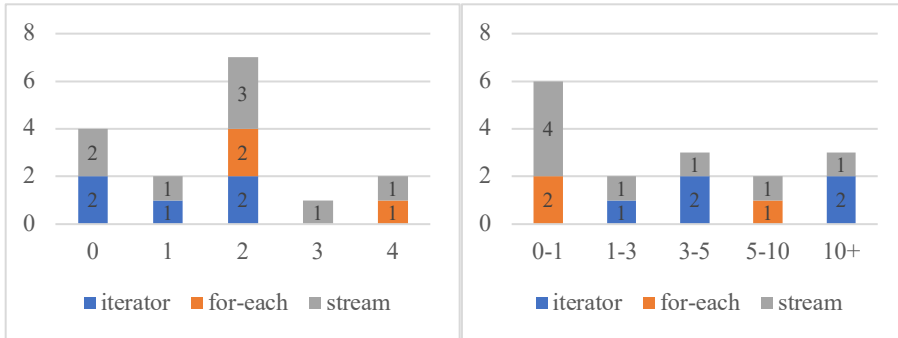


Figure 3. Number of “don’t know” answers dependently on declared Java knowledge (on the left) and seniority (on the right) for Task 5

4.3.4. Detailed analysis of Task 6

Task 6 received the highest number of “don’t know” answers (9 – streams, 10 – iterator, 5 – *for-each*). Figure 4 shows how the numbers break down depending on the declared Java knowledge.

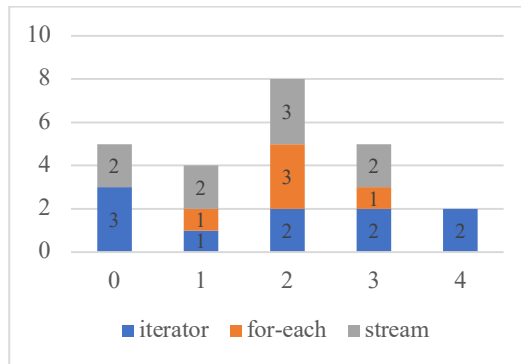


Figure 4. Number of “don’t know” answers dependently on declared Java knowledge (0-5)

One can observe that iterator is present in all groups (except experts) and should not be further recommended.

4.3.5. Objective understandability versus metrics

The complexity metrics calculated for tasks 1, 2, and 4 were the same. Slight differences were observed only in the number of lines of code. The imperative solution takes $200 \pm 50\%$ of the length of the functional solution. The values suggest that the stream should be an unquestionable winner in objective readability assessment. Unfortunately, the survey results do not support that claim. Streams scored best only in Task 4.

Tasks 3 and 6 also have the same complexity metrics with similar dependencies for source code length. The metrics are not correlated with the obtained correctness of answers for these tasks.

The most interesting is Task 5, for which the complexity metrics for streams got higher values (2). The complexity of this implementation was confirmed by the survey in which the understandability of streams was far from the other versions.

The observations presented above were confirmed by statistical analysis. To see if there was a relationship between the correctness of the answers and the metrics, the responses selected by the participants were coded with values of 0 (wrong answer or don't know) and 1 (correct answer). Since this creates a nominal scale, Cramer's V coefficient was used to check the dependency. No dependencies were observed for the iterator and *for-each* construct. The only interesting result was obtained for streams: a weak correlation between correctness and the LoC metric (V-Cramer = 0.22 with p-value = 0.02 calculated from the χ^2 test).

4.4. Perceived understandability

To assess the perceived understandability, a simple transformation was used. The positions of constructs were translated to points, where the first place was equal to 2 points, the second was equal to 1 point, and the third – 0 points. If two constructs were in the same place, they got the same number of points. The sum of points for tasks/constructs can be seen in Table 4.

Basic statistics for all tasks (N=594) are shown in Fig. 5. The mean value for iterator (IT) is designingly lower than for *for-each* statement (FE) and streams (ST).

Variable	Descriptive Statistics (Total values)				
	Valid N	Mean	Minimum	Maximum	Std.Dev.
FE	594	1,348485	0,00	2,000000	0,593477
IT	594	0,358586	0,00	2,000000	0,563985
ST	594	1,291246	0,00	2,000000	0,846194

Figure 5. Basic statistics for perceived understandability

The relations among the groups were checked with the Friedman Anova test, which can be used when the rankings of a few objects (more than 2) came from various sources (e.g., judges). The obtained result (p-value < 0.000001) for $\alpha = 0.05$ means a statistically that a significant difference exists between at least two groups. The post-hoc Dunn analysis with the Bonferroni correction indicates differences between iterators and streams, and iterators and *for-each* statements (p-value < 0.000001 for these pairs).

The Friedman Anova test for dependent groups also confirmed the statistical difference between the considered groups (IT – iterator, FE – *for-each*, ST – stream) for each task separately.

Streams were selected as the best understandable 4 out of 6 times (for tasks 1, 3, 4, 6), but only for Task 1 the post-hoc Bonferroni test confirmed their statistical advantage compared with *for-each* statement (p-value = 0,006).

Iterator definitively has the worse perceived understandability. For Task 5, it jumped streams up, but the difference was not significant (p-value = 0.95 for Bonferroni correction). What is interesting, Task 5 had the highest values of complexity metrics.

4.4.1. Subjective understandability versus metrics

Subjective understandability was determined on an ordinal scale, while the complexity and LoC metrics were measured on an interval scale. Both have a limited number of values. Therefore, Kendall's tau correlation coefficient was calculated for each mechanism to assess whether they are correlated. The Z statistics was used to determine whether the variables were statistically independent. The results are summarized in Table 5.

Table 5. Kendall's tau correlation coefficient and Z p-value between perceived understandability and metrics per mechanism

	Tau/p-value (for z statistic)		
	CogC	v(G)	LoC
iterator	0.01 / 0.68	0.01 / 0.68	0.14 / 0.00001
<i>for-each</i>	0.01 / 0.0005	0.01 / 0.0005	0.16 / < 0.000001
stream	-0.07 / 0.01	-0.07 / 0.01	-0.17 / < 0.000001

Statistical tests failed to reject at the significance level (0.05) the null hypothesis of independence of complexity metrics and perceived understandability for the iterator.

One can observe a very weak correlation between subjective assessment and LoC metric. The direction of the coefficient for *for-each* statement is surprising and suggests that understandability increases when the number of lines increases. That may be a side effect of the fact that lengths/complexity of the code snippets were very similar, and that tasks were assessed by experiment participants in the context (compared to other tasks).

Therefore, a similar analysis was performed for each task separately. Here we observe the dependance of all pairs of variables with p-value < 0.00001. The correlation coefficient ranged from -0.495 (Task 1) to -0.27 (Task 2) for complexity metrics and from -0.65 (Task 1) to -0.45 (Task 6) for LoC, what means at least average correlation. The only anomaly was observed for Task 5, when tau=0.379 for complexity metrics. That was because *for-each* construct with the highest complexity got much better understandability results than the streams.

4.4.2. Objective understandability versus perceived understandability

The results of perceived understandability were in total agreement with objective assessment in terms of correct answers in only one Task 5, partial agreement (winner) for tasks 2 and 4, and partial agreement (loser) for tasks 3 and 5.

For Task 1, the results were contradictory – the iterator with the highest correctness score was chosen as the least readable, and conversely, the stream with the lowest correctness score was chosen as the most readable. However, it should be noted that the differences in the objective assessment were not large. For typical filter-map-reduce tasks, both *for-each* and streams performed comparably, so considering the perceived understandability, streams should be recommended.

The *for-each* statement did the best with Task 2, in which the reduction returns a single element. This suggests that a balance must be kept between streams and imperative solutions. Streams are not a golden bullet to be used at all costs. These conclusions confirm the results for Task 5, in which the complexity of the functional solution negatively impacts the correctness of answers and the perceived readability of the solution.

Imperative (*for-each*) and functional solutions have comparable subjective and perceived understandability for tasks in which the elements of the filtered collection are somehow processed, e.g., summarized (Task 3). In such cases, one can decide based on their preferences.

The results for Task 4 show the dominance of the functional paradigm, in which the name of the operation (*distinct*) performed on the elements of a collection indicates its effect. The use of the Set interface has gone unnoticed by many survey attendants, leading to the worse correctness for the *for-each* statement.

Partially contradictory results were also obtained for Task 6. The assessment of perceived understandability points out the application of streams as the best solution for creating a histogram. On the contrary, streams got the lowest number of correct answers and the highest sum of incorrect and “don’t know” responses. In this case, the streams were used for an unusual task, which seems to be the reason for the low rating.

4.5. Threats to validity

The identified threats to external validity are as follows:

- Participants, except students, worked unsupervised, so they might be tempted to cheat or not to focus properly on the solution (e.g., the answer “I don’t know” chosen too eagerly) – that risk was mitigated by the fact that the survey was directed to mature practitioners who should know the importance of research; under typical conditions, i.e., in IT companies, programmers work under time pressure. The efficiency of the prevention actions was proved by the zero “don’t know” responses selected by participants with the highest claimed Java knowledge.
- Knowledge of the declared Java language varied and could have affected the understandability assessment – the number of participants mitigated this risk; more than one-third (41) claimed a very high or expert level; only 10 did not know Java at all, but even in this group the number of correct answers was higher than “don’t know” or incorrect (see Figure 1).

The identified threats to internal and content validity are as follows:

- Selection bias and group construction – the risk was limited by the number of participants and the fact that code fragments in the groups were similar.
- Only one measure (correctness) was used to assess understandability objectively. This was due to the limitations of the tool used. In the future, we intend to measure response time as well.
- Other constructs than the compared iteration mechanisms could have affected the understandability results – we tried to mitigate the risk by reusing and adapting examples from the literature; however, one such adaptation (Task 4, iterators and *for-each*), in which we decided to use *Set* to assure collection elements uniqueness and to keep the source code as short as possible, resulted in an increased number of incorrect answers. We could only predict that the overall results for the constructs involved might be slightly better. Nevertheless, this observation did not change the final recommendations.
- Participants could guess the correct answer without paying attention – the high number of answers given mitigated this risk.

5. Summary and further works

The paper deals with the problem of the understandability of three iteration mechanisms over collections in Java. Understandability can be variously predicted or assessed, using metrics, objectively by solving the tasks, or subjectively by programmers. This research aimed to find out the possible correlations between these perspectives and the definition of recommendations.

Functional code was perceived as most understandable by programmers in four cases, but only in one case, the subjective assessment was confirmed by the correctness of answers. In the other two examples with the highest perceived understandability, streams received the worst correctness scores. Cognitive complexity and McCabe's complexity had the lowest values in all tasks for the functional approach, but – unfortunately – they did not correlate with answer correctness. Cognitive complexity greater than 1 for streams resulted in the worst correctness values and the highest percentage of "don't know" responses. These results were confirmed by the perceived understandability scores. The threshold values of these metrics should be established separately for imperative and functional constructs.

Therefore, the final recommendations are as follows. The functional approach to collection manipulation is the best choice for the filter-map-reduce idiom (especially when the result of the reduce operation is another collection) and its alternatives (e.g., filter-only). It should not be used in more complex tasks, especially those with higher values of complexity metrics. A viable option to stream is still the *for-each* statement. Its understandability, in most cases, is comparable to the functional approach.

Iterator should be treated as a legacy statement. It is not recommended for further use.

In the future, we intend to extend the assessment of understandability to include time response values for individual tasks. The conducted experiment showed that the readability of streams is sensitive to complexity. Therefore, finding a threshold of complexity metrics for which a functional approach is not recommended would also be interesting.

Appendix

Task 1. Collection filtering - iterator, for-each, and stream implementations with a set of parameters and answers to select from.

```
public static List<String> iteratorMethod(List<String> stringList,
    String containsStr) {

    List<String> response = new ArrayList<>();
    Iterator<String> it = stringList.iterator();
    while (it.hasNext()) {
        String str = it.next();
        if (str.contains(containsStr)) {
            response.add(str);
        }
    }
    return response;
}

public static List<String> forEachMethod(List<String> stringList,
    String containsStr) {

    List<String> response = new ArrayList<>();
    for (String str : stringList) {
        if (str.contains(containsStr)) {
            response.add(str);
        }
    }
    return response;
}

public static List<String> streamMethod(List<String> stringList,
    String containsStr) {

    return stringList.stream()
        .filter(str -> str.contains(containsStr))
        .toList();
}
```

Parameters:

```
List<String> stringList =
    List.of("Bartosz", "Jan", "Krych", "235333");
String containsStr = "a";
```

Set of answers: **[Bartosz, Jan]**, Error, Don't know, [Bartosz, Jan, Krych, 235333], [Bartosz], [Bartosz, Jan, Krych], [Bartosz], [Bartosz, Jan, Krych], [Jan, Bartosz], Bartosz

Task 2. Searching for the first element satisfying a condition – iterator, *for-each*, and stream implementations with the set of parameters and answers to select from.

```
public static String iteratorMethod(List<String> stringList,
    String startStr) {

    Iterator<String> it = stringList.iterator();
    while (it.hasNext()) {
        String str = it.next();
```

```

        if (str.startsWith(startStr)) {
            return str;
        }
    }
    return startStr;
}

public static String forEachMethod(List<String> stringList,
    String startStr) {

    for (String str : stringList) {
        if (str.startsWith(startStr)) {
            return str;
        }
    }
    return startStr;
}

public static String streamMethod(List<String> stringList,
    String startStr) {

    return stringList.stream()
        .filter(str -> str.startsWith(startStr))
        .findFirst()
        .orElse(startStr);
}

Parameters:
List<String> stringList =
    List.of("Bartosz", "Jan", "Krych", "235333", "Krych");
String startStr = "Kr";

```

Set of answers: **Krych**, [Krych, Krych], [Krych], Kr, Don't know, Error, [Kr], [Bartosz, Jan, Krych, 235333]

Task 3. Calculate the sum of the collection elements – iterator, for-each, and stream implementations with the set of parameters and answers to select from.

```

public static int iteratorMethod(List<Integer> ints) {
    Iterator<Integer> it = ints.iterator();
    int sum = 0;
    while (it.hasNext()) {
        sum += it.next() + 1;
    }
    return sum;
}

public static int forEachMethod(List<Integer> ints) {
    int sum = 0;
    for (int i : ints) {
        sum += i + 1;
    }
    return sum;
}

public static int streamMethod(List<Integer> ints) {
    return ints.stream()
        .mapToInt(i -> i + 1)

```

```
        .sum();
    }
}
```

Parameters:

```
List<Integer> ints = List.of(0, 1, 2, 3, 4, 5);
```

Set of answers: 21, 15, Don't know, Error, 6, 16, 012345, 0

Task 4. Counting the number of unique elements satisfying a condition – iterator, for-each, and stream implementations with the set of parameters and answers to select from.

```
public static long iteratorMethod(List<String> stringList,
    String containsStr) {

    Set<Integer> helper = new HashSet<>();
    Iterator<String> it = stringList.iterator();
    while (it.hasNext()) {
        String str = it.next();
        if (str.contains(containsStr)) {
            helper.add(str.length());
        }
    }
    return helper.size();
}

public static long forEachMethod(List<String> stringList,
    String containsStr) {

    Set<Integer> helper = new HashSet<>();
    for (String str : stringList) {
        if (str.contains(containsStr)) {
            helper.add(str.length());
        }
    }
    return helper.size();
}

public static long streamMethod(List<String> stringList,
    String containsStr) {

    return stringList.stream()
        .filter(str -> str.contains(containsStr))
        .map(String::length)
        .distinct()
        .count();
}
//
Parameters:
List<String> stringList = List.of("Bartosz", "Bartosz", "Jan", "Bartosz",
    "Jan", "Krych", "Bartosz", "Jan", "Krych", "235333");
String containsStr = "a";
```

Set of answers: 2, 7, 4, Don't know, 10, 3, Error, 0

Task 5. Reducing a collection to a string of characters that are in the same position in an element of collection as the element itself – iterator, for-each, and stream implementations with the set of parameters and answers to select from.

```
public static String iteratorMethod(List<String> stringList) {
    StringBuilder response = new StringBuilder();
    int i = 0;
    Iterator<String> it = stringList.iterator();
    while (it.hasNext()) {
        String str = it.next();
        if (i < str.length()) {
            response.append(str.charAt(i));
        }
        i++;
    }
    return response.toString();
}

public static String forEachMethod(List<String> stringList) {
    StringBuilder response = new StringBuilder();
    int i = 0;
    for (String str : stringList) {
        if (i < str.length()) {
            response.append(str.charAt(i));
        }
        i++;
    }
    return response.toString();
}

public static String streamMethod(List<String> stringList) {
    return IntStream.range(0, stringList.size())
        .mapToObj(i -> {
            if (i < stringList.get(i).length()) {
                return ((Character) stringList.get(i)
                    .charAt(i)).toString();
            }
            return "";
        })
        .filter(s -> !s.isEmpty())
        .reduce("", (s, c) -> s + c);
}
```

Parameters:

```
List<String> stringList = List.of("Bartosz", "Bartosz", "Jan", "Bartosz",
    "Jan", "Krych", "Bartosz", "Jan", "Krych", "235333");
```

Set of answers: **Bantz**, Don't know, Bantnhz, BBJBJKBJK2, 2an, Bay3, Bantnhznh3

Task 6. Histogram - iterator, for-each, and stream implementations with the set of parameters and answers to select from.

```
public static Map<String, Long> iteratorMethod(List<String> stringList){
    Map<String, Long> strLongMap = new HashMap<>();
    Iterator<String> it = stringList.iterator();
    while (it.hasNext()) {
```

```

        String str = it.next();
        strLongMap.put(str, (long) Collections.frequency(stringList, str));
    }
    return strLongMap;
}

public static Map<String, Long> forEachMethod(List<String> stringList){
    Map<String, Long> strLongMap = new HashMap<>();
    for (String str : stringList) {
        strLongMap.put(str, (long)
            Collections.frequency(stringList, str));
    }
    return strLongMap;
}

public static Map<String, Long> streamMethod(List<String> stringList){
    return stringList.stream()
        .collect(Collectors.groupingBy(str -> str,
            Collectors.counting()));
}

```

Parameters:

```
List<String> stringList = List.of("Bartosz", "Bartosz", "Jan", "Bartosz",
    "Jan", "Krych", "Bartosz", "Jan", "Krych", "235333");
```

Set of answers: {**Krych=2, Jan=3, 235333=1, Bartosz=4**}, Don't know, Error, [[Bartosz, Bartosz, Bartosz, Bartosz], [Jan, Jan, Jan], [Krych, Krych], [235333]], {Krych=0, Jan=1, 235333=2, Bartosz=3}, [Bartosz, Jan, Krych, 235333], [], [Bartosz, Bartosz, Jan, Bartosz, Jan, Krych, Bartosz, Jan, Krych, 235333]

Bibliography

- [1] Baron, M. M., Wirich, M., Wagner, S.: An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability, in Proceedings of the 14th ACM/IEEE International Symposium of Empirical Software Engineering and Measurement, p. 1-12, (2020).
- [2] Bexell, A.: Comparing functional to imperative Java with regards to readability, complexity, and verbosity, Växjö, (2017).
- [3] Cheon, Y., Escobar De La Torre, A.: Impacts of Java Language Features on the Memory Performances of Android Apps, ScholarWorks@UTEP, Impacts of Java Language Features on the Memory Performances of Android Apps - CORE (2017).
- [4] Costa, D. E., Andrzejak, A., Seboek, J., Lo, D.: Empirical Study of Usage and Performance of Java Collections, Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, (2017).
- [5] Flauzino, M., Veríssimo, J., Terra, R., Cirilo, E., Durelli, V., Durelli, R.: Are you still smelling it?: A comparative study between Java and Kotlin Language, Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse, pp. 23-32, (2018).
- [6] Holst, G., Dobslaw, F.: On the Importance and Shortcomings of Code Readability Metrics: A Case Study on Reactive Programming, ArXiv, abs/2110.15246, (2021).

- [7] Lucas, W., Bonifácio, R., Dias Canedo, E., Marcílio, D., Lima, F.: Does the Introduction of Lambda Expressions Improve the Comprehension of Java Programs?, *Proceedings of the XXXIII Brazilian Symposium on Software Engineering, Association for Computing Machinery*, pp. 187-196 (2019).
- [8] Mazinanian, D., Ketkar, A., Tsantalis, N., Dig, D.: Understanding the use of lambda expressions in Java, *Proceedings of the ACM on Programming Languages*, vol. 1, pp. 1-31, (2017).
- [9] Monteiro Mendonça, W.L., Fortes, J., Lopes, F.V., Marcílio, D., De Almeida, R.B., Dias Canedo, E., Lima, F., Saraiva, J.: Understanding the Impact of Introducing Lambda Expressions in Java Programs, *Journal of Software Engineering Research and Development*, vol. 8, pp. 1-22, (2020).
- [10] Nanz, S., Furia, C.A.: A Comparative Study of Programming Languages in Rosetta Code, *37th IEEE International Conference on Software Engineering*, vol. 1, pp. 778-788, (2015).
- [11] Nostas, J., Alcocer, J.P.S., Costa, D.E., Bergel, A.: How Do Developers Use the Java Stream API?, In: *Computational Science and Its Applications – ICCSA 2021. ICCSA 2021. Lecture Notes in Computer Science*, vol. 12955, Springer, Cham, (2021).
- [12] Oliveira, W., Oliveira, R., Filho, F.C., Fernandes, B., Lima Pinto G.H., *Recommending Energy-efficient Java Collections*, *Proceedings of the 16th International Conference on Mining Software Repositories*, (2019).
- [13] Oracle, Stream (Java SE 17 & JDK 17) – `toList()`, [https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Stream.html#toList\(\)](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Stream.html#toList())
- [14] Pankratius, V., Schmidt, F., Garretón, G.: Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java, *34th International Conference on Software Engineering*, pp. 123-133, (2012).
- [15] Pantiuchina, J., Lanza, M., Bavota, G., *Improving Code: The (Mis)perception of Quality Metrics*, *IEEE International Conference on Software Maintenance and Evolution*, pp. 80-91, (2018).
- [16] Tanaka, H., Matsumoto, S., Kusumoto, S.: A Study on the Current Status of Functional Idioms in Java, *IEICE Transactions on Information and Systems*, E102-D(12), pp. 2414-2422, (2019).

Received 14.09.2022, Accepted 02.12.2022