Imperative versus Declarative Collection Processing: An RCT on the Understandability of Traditional Loops versus the Stream API in Java

Nils Mehlhorn
Independent Consultant, Essen
mail@nils-mehlhorn.de

Stefan Hanenberg
Paluno - The Ruhr Institute for Software Technology,
University of Duisburg–Essen, Germany
stefan.hanenberg@uni-due.de

ABSTRACT

Java introduced in version 8 with the Stream API means to operate on collections using lambda expressions. Since then, this API is an alternative way to handle collections in a more declarative manner instead of the traditional, imperative style using loops. However, whether the Stream API is beneficial in comparison to loops in terms of usability is unclear. The present paper introduces a randomized control trial (RCT) on the understandability of collection operations performed on 20 participants with the dependent variables response time and correctness. As tasks, subjects had to determine the results for collection operations (either defined with the Stream API or with loops). The results indicate that the Stream API has a significant (p<.001) and large (η_p^2=.695; $\frac{M_{loop}}{M_{stream}} \sim 178\%$) positive effect on the response times. Furthermore, the usage of the Stream API caused significantly less errors. And finally, the participants perceived their speed with the Stream API higher compared to the loop-based code and the participants considered the code based on the Stream API as more readable. Hence, while existing studies found a negative effect of declarative constructs (in terms of lambda expressions) on the usability of a main stream programming language, the present study found the opposite: the present study gives evidence that declarative code on collections using the Stream API based on lambda expressions has a large, positive effect in comparison to traditional loops.

KEYWORDS

Programming Languages, Lambda Expressions, Declarative, Imperative, Java, Streams

ACM Reference Format:
Nils Mehlhorn and Stefan Hanenberg. 2022. Imperative versus Declarative Collection Processing: An RCT on the Understandability of Traditional Loops versus the Stream API in Java. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3519016

1 INTRODUCTION

In programming, the use of collections, i.e. lists or sets of values (or objects) is quite essential. However, different kinds of programming languages provide different means to operate on collections. In imperative programming languages, there are typically loop constructs that permit (in combination with additional operators) to iterate over a collection in order to, for example, select certain elements or to map the collection to some other values. A typical example of such an approach is the for-loop that comes in different styles and syntaxes and that can be found in almost all imperative programming languages from BASIC over C up to Java. Declarative programming languages, on the other hand, typically follow a different approach where the iteration over a collection is hidden by some abstraction that receives a condition that needs to hold on the results. In functional programming languages, such conditions are typically given using lambda expressions.[1]

While rather imperative programming languages such as Smalltalk supported lambda expressions right from the beginning, popular languages such as Java or C++ supported lambda expressions relatively late. In the case of Java, the alternative to the traditional loops appeared with Java 8 in 2013[2] where lambda expressions were introduced in addition to the Stream API that made use of lambda expressions.[3]

Figure 1 illustrates two alternatives to filter objects from a collection using either loops or the Stream API. The loop implementation requires first a definition of the result collection. Then, the input collection is iterated and for each element it is checked (via an if-statement) whether the age is 21 or higher. If this is the case, the object is added to the result collection. For the stream implementation, a stream has to be received from the collection by calling the method `stream()`. Then, the filter method is invoked with a lambda expression (representing the boolean expression that has to match each element of the result). Finally, the result is collected (in the example into a list).

In Java, the stream code is quite verbose because of the additional `stream()` and `collect()` invokations which is the result of a strict distinction between the Collection API and the Stream API. In other languages such as, for example, Smalltalk (that provides

[1]The theoretical background for functional programming is the lambda calculus that has lambda expressions as one of the core language features (see for example [31] for a general introduction into the lambda calculus and its application for modeling programming languages).
[2]The language specification was published in 2014 [16].
[3]Technically, Stream API does not necessarily require the application of lambda expressions, i.e. it is also possible to use inner classes. However, it seems quite common not to use inner classes for in the Stream API and use lambda expressions instead.

methods comparable to `filter()` in Java) the resulting code would be slightly shorter, because the two mentioned method calls are not necessary.

```
Collection<Person> persons = ...
Collection<Person> adults = ...;

// (a) Traditional filter operation using loops
for(Person person: persons)
  if(person.age()>=21)
    adults.add(person);

// (b) Filtering a collection using the Stream API
adults = persons.stream()
                .filter(p->p.age()>=21)
                .collect(Collectors.toList());
```

Figure 1: Filtering objects from a collection in Java using (a) loops and (b) the Stream API.

If we compare the code using loops and the Stream API in Figure 1, there is no larger difference in terms of lines of code (we omitted the brackets in the loop version which would make the code slightly longer). But the more complex operations on collections are, the longer seems to be the loop code in comparison to Stream API code. Although the usage of the Stream API has some other technical implications (such as a probably easier way to do iterations in parallel), both ways to write the code coexist. And it is unclear what way is preferable.

A first experiment that appeared in ICSE 2016 suggested that lambda expressions for iterating collections in C++ have problems when developers (novices or junior developers) need to write them while for senior developers at least no benefit was detected [37]. However, the study did not take into account that lambda expressions – once they have been finished – have a potential positive effect on the understandability of the code.

The present paper studies the possible effect of declarative constructs from a different angle: instead of writing expressions, participants read code that operated on a collection (either using loops or using the Stream API) and then decided, what the result of the code is. A randomized control trial that relied on the measurements of 20 volunteers revealed with strong evidence ($p<.001$) a large, positive effect of the use of the Stream API (η_p^2=.695): It took the subjects on average 78% longer to solve the tasks using loops in comparison to the Stream API, but the experiment also revealed that the difference between loops and the Stream API depends on the kinds of collection operations. An additional questionnaire given to the participants revealed that participants perceived their own speed in understanding the code using the Stream API higher than their speed with code using loops. This indicates that the positive effect of the Stream API is not only an effect that can be seen in a larger study, but an effect that developers can perceive via self-observations.

2 BACKGROUND

Conceptually, lambda expressions are anonymous functions (i.e. functions without a name) that can be passed as parameters, stored into variables or simply invoked. Lambda expressions are (and were) an essential and common language construct in functional programming languages since decades (see for example [1, 4]) and even some non-functional programming languages such as the object-oriented language Smalltalk [15] had lambda expressions right from start. But a number of main stream programming languages often did not have lambda expressions in their initial designs. We already mentioned that Java received lambda expressions about 20 years after the release of its first version. Other languages evolved in a similar way. For example, Python was originally released under the version 0.9 [38] but the introduction of lambda expressions appeared one version later (in version 1). C++ introduced lambda expressions in version 11 [35], C# introduced lambda expressions in version 3 [8], PHP received them in version 5.3, etc.

But although the concept of lambda expressions exists in different languages for years, there are still a number of differences between them. Examples of these differences are syntactical differences either between languages or even special syntax constructs within a language as well. Additionally, lambda expressions differ often with respect to their functionality (access to variables in outer scope, non-local returns[4], etc.).

```
Stream<Person> s ...
s.filter(p -> p.age()>=21);
s.filter(p -> { return p.age()>=21;});
s.filter((Person p) -> p.age()>=21);
s.filter((Person p) -> { return p.age()>=21;});
```

Figure 2: Syntactical differences of equivalent lambda expressions in Java.

In Java, a lambda expression might or might not have type declarations in its parameter list and the body might or might not have curly brackets. If there are no curly brackets, the body of a lambda expressions consists of a single expression which also represents the return value. In case a parameter list does not contain type declarations, Java infers the types, etc. I.e., a developer is typically free to decide whether or not he declares the types for parameters in a lambda expression. Figure 2 illustrates four equivalent lambda expressions passed to a stream's filter method – and there are studies that indicate that the different syntactic representations have an effect on readability (see for example [18]).

In Java's Stream API lambda expressions play an essential role and a number of methods in streams require lambda expressions to be passed.[5] We mentioned in the introduction already the method `filter()` that requires a lambda expression that returns a boolean value in order to determine whether or not one object should be selected from a list. Comparable methods are `anyMatch()` or `allMatch()` that return a boolean value if for any (or all) element(s)

[4]See for example [7, pp. 39] for a general explanation of local and non-local returns.
[5]Technically, this is not 100% correct, because the Stream API can be used with ordinary objects or anonymous inner classes as well. However, even Java's standard documentation on the Stream API proposes the use of lambda expressions.

in the collection the passed lambda expression returns true. In addition to that, the Stream API has methods for creating maps or groups or to convert a stream into a collection.

Figure 3 illustrates a more complex example that illustrates the more declarative nature of the Stream API in comparison to the traditional loops. In the example, a collection of objects is transformed into a map consisting of an integer key that is the age of a person from the collection and a list of all persons of that age. Using the Stream API one declares with a lambda expression the grouping criterion (`p->p.age()`) which defines the keys in the map. Another lambda expression describes what elements should be stored in the list. The same figure also contains an imperative implementation of the code where first the result map is created and then each element in the original collection is iterated. By using a conditional, the list objects are stored into the map and finally a person object is stored in the list. Although both ways result into a comparable result, the difference in style seems obvious. Still, the question is whether one way should be preferred over the other.[6]

```
Map<Integer, List<Person>> m;

// Map construction using the Stream API
m = personCollection.stream().collect(
      Collectors.groupingBy(p->p.age(),
        Collectors.mapping(p -> p,
                              Collectors.toList())));

// Map construction using loops
m = new HashMap<Integer, List<Person>>();
for(Person p : personCollection) {
  if(!m.containsKey(p.age()))
    m.put(p.age(), new Vector<Person>());
  m.get(p.age()).add(p);
}
```

Figure 3: Map construction from a list using the Stream API or using loops.

Additional to the evolution of programming languages there were other movements towards a more declarative style using lambda expressions. For example, the IDE IntelliJ visualized Java's anonymous inner classes already in 2009 as lambda expressions. I.e. although the underlying source code did not contain such expressions, IntelliJ's motivation was to improve the readability of code (under the assumption that lambda expressions can be easier read than anonymous inner classes).[7]

However, despite of the wide adoption of lambda expressions in programming languages over the last decade and the resulting more declarative means to write programs (in originally rather imperative languages), there is little empirical evidence on the benefits of using such constructs (despite the fact that there are indicators that the more functional style is increasingly accepted by developers, see for example the study by Mazinanian et al. [24]). At least one controlled experiment on lambda expressions seemed to suggest the opposite [37].

[6]In fact, the result is not identical, because the iterative style defines a hash map and a vector of persons while the concrete implementation used for the maps and the list are hidden inside the abstraction of the Stream API.

[7]https://blog.jetbrains.com/idea/2009/03/closure-folding-in-intellij-idea-9-maia/

3 EXPERIMENT

The goal of the present work is to study whether a more declarative way of operating on collections using the Stream API has any influence on the understandability. Our intention was to define a randomized control trial where participants needed to understand operations on collections that were either described using the traditional imperative style using loops or the rather declarative style using the Stream API with lambda expressions.

3.1 Initial Considerations

Before designing an experiment a number of concerns need to be taken into account.

Within-subject designs: One general concern is, whether the experiment should have within-subject variables, i.e. the general question is whether the experiment should be designed as a crossover study (see [34] for a general introduction into crossover trials). One reason to apply crossover trials is to overcome to a certain extent the so–called 10x problem (see [25]) which describes in general for programming related tasks that one can assume a high deviation between participants.[8] The analysis of a within-subject design permits to measure differences within a subject. This potentially compensates the effect of large deviations between subjects which is probably the reason why a large number of studies in software engineering are crossover trials.[9] However, crossover designs require to take possible carryover effects (respectively periodic effects) into account (see [23, p. 1984]) such as learning effects, fatigue effect, etc. And there is the necessity to check in the analysis whether the carryover effects actually occured (see for example [40] among many others).

Deviation in the non-iterating code: The obvious way to study differences in understandability is to give participants code with one treatment and code with a different treatment and then ask questions about it. The potential problem is the complexity of the code that is not in the focus of the present study. Such complexity could be caused by the code that defines and initializes the collections. If such code is too complex, there is the potential problem that it is the main cause for deviation in the measurements (and not the different treatments) which potentially hides the effect that should be studied: Such code needs to be read and understood before actually reading and understanding the code that operates on the collections. Hence, we think there is a need to represent the code that is not in the focus of the study in a more easy way.

Complexity of iterating code: Obviously, there are infinite possible ways to do operations on a collection. A simple code is a filter operation where for example objects from a collection are selected because one instance variable of the object corresponds to a certain literal (e.g., `person.age=42`). From our perspective, more complex operations are grouping or mapping operations (such as

[8]The evidence that participants differ by factor 10 is quite low (see [5, pp. 36]). Hence, 10x should be rather understood as a methaphore that the deviation can be high.

[9]Vegas et al. analyzed studies in software engineering and found that more than $\frac{1}{3}$ of of the analyzed studies were crossover trials [39, p. 123].

Table 1: Experiment source code (Tasks 1 – 5)

Task	Stream API	Loop

1	Map<String, Long> f(Collection<String> words) {
	  return words.stream().collect(
	    groupingBy(String::toLowerCase, counting())
	  );
	}
		Map<String, Long> f(Collection<String> words) {
		  Map<String, Long> groups = new HashMap<>();
		  for(String word: words) {
		    String lowercase = word.toLowerCase();
		    Long count = groups.getOrDefault(lowercase, 0L);
		    groups.put(lowercase, count + 1);
		  }
		  return groups;
		}

2	List<Employee> f(List<Long> ids, List<Employee> employees) {
	  return employees.stream()
	    .filter(employee -> ids.contains(employee.getId()))
	    .collect(toList());
	}
		List<Employee> f(List<Long> isa, List<Employee> employees) {
		  List<Employee> l = new ArrayList<>();
		  for (Employee employee : employees) {
		    if(ids.contains(employee.getId())) {
		      l.add(employee);
		    }
		  return l;
		}

3	boolean f(Collection<Result> results) {
	  return results.stream().allMatch(
	    r -> r.getPoints()>50
	  );
	}
		boolean f(Collection<Result> results) {
		  for (Result r : results) {
		    if (r.getPoints() <= 50) {
		      return false;
		    }
		  } **return** true;
		}

4	Map<Author, Book> f(Collection<Book> books) {
	  return books.stream().collect(
	        toMap(Book::getAuthor,
	             book -> book,
	             maxBy(comparing(Book::getSales)
	        )
	  );
	}
		Map<Author, Book> f(Collection<Book> books) {
		  Map<Author, Book> groups = new HashMap<>();
		  for(Book book : books) {
		    Book otherBook = groups.get(book.getAuthor());
		    if(otherBook == null || book.getSales()> otherBook.getSales()) {
		      groups.put(book.getAuthor(), book);
		    }
		  }
		  return groups;
		}

5	List<Double> f(Collection<BankAccount> accounts, double rate) {
	  return accounts.stream()
	    .filter(account -> !account.isFree()
	          && account.getBalance() <= 0)
	    .map(account -> {
	      double result = account.getBalance();
	      if (!account.isFixed()) {
	        double finalRate = account.isPremium() ? rate * 2 : rate;
	        result *= finalRate; } return result;
	      }
	    )
	   .collect(toList());
	}
		List<Double> f(Collection<BankAccount> accounts, double rate) {
		  List<Double> results = new ArrayList<>();
		  for (BankAccount account : accounts) {
		    double balance = account.getBalance();
		    if (account.isFree() || balance > 0) {
		      continue;
		    }
		    double result = balance;
		    if (!account.isFixed()) {
		      double finalRate = account.isPremium() ? rate * 2 : rate;
		      result *= finalRate;
		    }
		    results.add(result);
		  }
		  return results;
		}

the one given in Figure 3) and they probably have an effect on the potential differences. Hence, we see the need to study different kinds of operations on collections.

Deviation in the iterating code: We need to take into account that especially for the loops there is not one unique way to write down a collection operation. If we use again Figure 3 as an example, it is possible to define additional local variables (that store for example the result of the expression `person.age()`). Additionally, one needs to take into account that the naming of such variables

Table 2: Experiment source code (Tasks 6 – 7)

Task	Stream API	Loop

6	`Map<String, Integer> f(List<Product> products) {`
	`  return products.stream().collect(`
	`            toMap(Product::getDepartment,`
	`                  Product::getPrice,`
	`                  minBy(Integer::compareTo)`
	`            )`
	`  );`
	`}`
		`Map<String, Integer> f(List<Product> products) {`
		`  Map<String, List<Product> >`
		`  groups = new HashMap<>();`
		`  for (Product p : products) {`
		`    String department = p.getDepartment();`
		`    if (groups.containsKey(department)) {`
		`      groups.get(department).add(p);`
		`    } else {`
		`      List<Product> group = new ArrayList<>();`
		`      group.add(p);`
		`      groups.put(department, group);`
		`    }`
		`  }`
		`  Map<String, Integer> computed = new HashMap<>();`
		`  for (Map.Entry<String, List<Product> > group : groups.entrySet()) {`
		`    Integer c = null;`
		`    List<Product> departmentProducts = group.getValue();`
		`    for (Product product : departmentProducts) {`
		`      if (c == null || product.getPrice() < c) {`
		`        c = product.getPrice();`
		`      }`
		`    }`
		`    computed.put(group.getKey(), c);`
		`    }`
		`  return computed;`
		`}`

7	`boolean f(Collection<Artwork> artworks, int discount) {`
	`  return artworks.stream()`
	`                 .anyMatch(a -> a.getPrice() - discount < 5000);`
	`}`
		`boolean f(Collection<Artwork> artworks, int discount) {`
		`  for (Artwork a : artworks) {`
		`    if (a.getPrice() - discount < 5000) {`
		`      return true;`
		`    }`
		`  }`
		`  return false;`
		`}`

has an effect on the understandability of the resulting code.[10] Our conclusion on this is that the loop code should be plausible where local variables should be used whenever the corresponding expressions are too complex. Furthermore, variable names should be used that express best the content of these variables without actually announcing the semantics of the whole operation.

Deviation in lambda expressions: As states in Section 2, lambda expressions in Java can be written in different ways and it is not clear whether the different styles have an effect on the understandability. While it would be in principle possible to study the effect of different lambda styles as well, we believe that it is quite common in the Stream API to use lambda expressions without declared types and mainly with one single expression in their body whenever possible.

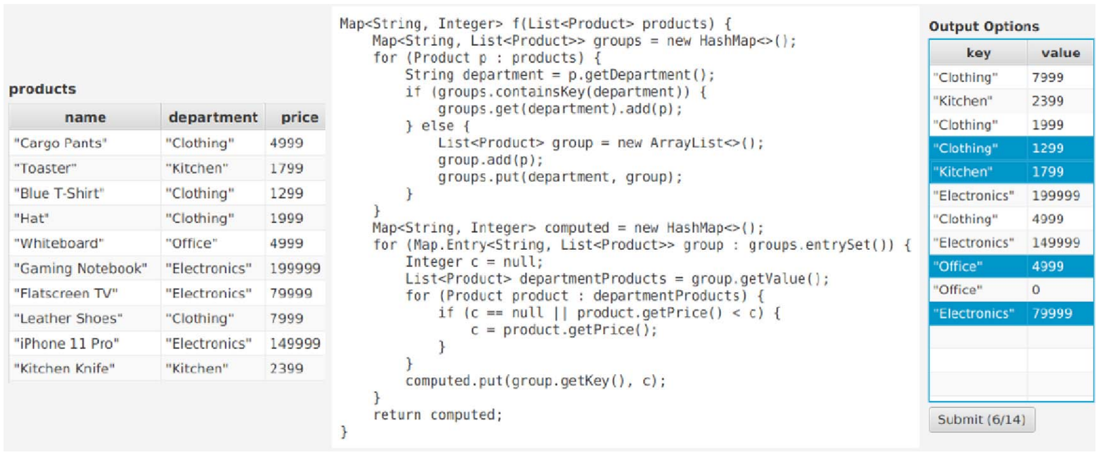
3.2 Tasks

Altogether, seven tasks were used in the experiment (see Table 1 and Table 2). Taking the initial considerations into account, we decided to give subjects code that consists of one single method where all objects required by a collection operation are passed to. While one initial consideration was to give meaningful names, this should not be the case of this single method: if the semantics of the method is already described in the method's name, participants would probably guess from the name the semantics of the code. As a consequence, we always gave the method the name `f`. But `f`'s parameter names are meaningful, i.e. their names indicate what the content of the variables is. For the local variables within the loop code, we used meaningful names as all.

For example, the code for Task 1 (with the not meaningul method name `f`) receives a collection of words as input (the parameter has the meaningful name `words`) and returns a map that contains how often a word (in lowercase) occured in the collection. In the loop code we decided to give the local variable for the result object the (meaningful) name `groups` and we decided to store the lowercase representation of a word as well as the counter in corresponding local variables (with meaningful names).

In order to represent different kinds of complexities, we selected code snippets based on our personal perception.

[10]For example, the studies by Binkley et al. [3] or by Hofmeister et al. [19] showed that the style as well as the length of identifiers has an effect on readabilty.



products

name	department	price
"Cargo Pants"	"Clothing"	4999
"Toaster"	"Kitchen"	1799
"Blue T-Shirt"	"Clothing"	1299
"Hat"	"Clothing"	1999
"Whiteboard"	"Office"	4999
"Gaming Notebook"	"Electronics"	199999
"Flatscreen TV"	"Electronics"	79999
"Leather Shoes"	"Clothing"	7999
"iPhone 11 Pro"	"Electronics"	149999
"Kitchen Knife"	"Kitchen"	2399

```
Map<String, Integer> f(List<Product> products) {
    Map<String, List<Product>> groups = new HashMap<>();
    for (Product p : products) {
        String department = p.getDepartment();
        if (groups.containsKey(department)) {
            groups.get(department).add(p);
        } else {
            List<Product> group = new ArrayList<>();
            group.add(p);
            groups.put(department, group);
        }
    }
    Map<String, Integer> computed = new HashMap<>();
    for (Map.Entry<String, List<Product>> group : groups.entrySet()) {
        Integer c = null;
        List<Product> departmentProducts = group.getValue();
        for (Product product : departmentProducts) {
            if (c == null || product.getPrice() < c) {
                c = product.getPrice();
            }
        }
        computed.put(group.getKey(), c);
    }
    return computed;
}
```

Output Options

key	value
"Clothing"	7999
"Kitchen"	2399
"Clothing"	1999
"Clothing"	1299
"Kitchen"	1799
"Electronics"	199999
"Clothing"	4999
"Electronics"	149999
"Office"	4999
"Office"	0
"Electronics"	79999

Figure 4: Application used for the data selection (task 6 with technique loop).

- **Simple (Task 3, 7):** We used two simple tasks where only one conditions needed to be checked (either whether the condition holds for one or for all elements). In both cases, the loop contained a condition with a return statement as well as an additional return statement after the loop while for the Stream API just the corresponding methods were invoked. For the simple tasks, we did not expect any positive effect of the Stream API and possibly even a negative effect: the loop code looks from our perspective quite trivial.
- **Medium (Task 1, 2, 4, 6)**: We considered tasks with grouping criteria as medium. We think that especially for loops the code is no longer that trivial to understand, because one has to handle the definition of subgroups and the assignment of elements to that group. Task 4 and 6 are from our perspective special cases. In task 4, the loop code defines the subgroups and the mapping (the maximum) in one step. We were not sure whether it is easier for developers to define the mapping in a separate loop. Hence, Task 4 and 6 differ mainly in the loop code where the mapping occurs either in one single loop (Task 4) or in two loops (Task 6). For both cases, the Stream API code is comparable.
- **Complex (Task 5)**: From our perspective a task is complex if the filter citerion and the mapping criterion cannot be simply defined by referring to an attribute. Task 5 is from our perspective such a task that contains a more complex definition of the mapping criterion: even in the Stream API the lambda expression passed to the `map` contains an additional condition. In the loop implementation, the loop's body is from our perspective more complex due to the additional conditional.

Note that the previous classification is based on our personal perception and only used for explaining the different kinds of tasks. Hence, we do not use this (from our perspective) rather weak classification as an independent variable in the experiment.

Because of the initial considerations (see Section 3.1) we decided to give subjects a more graphical representation of the input variables for the code. Figure 4 illustrates the application we used in the experiment (in the example for Task 6 in the loop representation).[11] In the center of the screen the code is shown whose result it to be determined. On the left hand side, the input data is given, on the right hand side the possible output is given. The input is a table where the table's name matches the parameter name of the method. Additionally, the table's column names represent the instance variables of the corresponding object. For example, the code in Figure 4 uses a collection of products where each product has the instance variables `name`, `department`, and `price`.

3.3 Experiment Design and Layout

The experiment was designed as a crossover design where each participant was tested under all treatments. We divided the participants into two groups (A and B). Both groups received the tasks in identical order (Task 1–7, then, again, Task 1–7) but with alternating techniques (loops vs. Stream API). For group A, Task 1 was given in the loop representation, Task 2 in the Stream API representation, Task 3 in the loop representation, etc. Group B started with the Stream API. After finishing the seven tasks, the tasks were again given, this time with the other technique (i.e., group A received task 1 with the Stream API, Task 2 with the loop, etc.) and with different inputs. The participants were not informed about the experiment design, i.e. the participants just knew that 14 tasks had to be done.

For each task, the participants were asked to rate their subjective impressions from the task with respect to three criteria (each rated on a 10 point Likert scale, 1=lowest; 10=highest): overall performance (i.e. how good a participant from his perspective performed the task), speed (i.e. how fast he was able to solve the task), and readability (how well from the participants's perspective the code reflected its intention)[12].

The experiment studies the following hypotheses:

(1) **H0**$_{technique}$: There is no difference in time for answering what the results of a loop-based or Stream API-based source code are.

[11] https://github.com/nilsmehlhorn/loop-stream-rct/releases
[12] Actually, the participants were asked to judge the code with respect to aesthetics instead of readability. We use the term readability here instead to make the terminology more clear.

(2) **H0**$_{correctness}$: There is no difference in the correctness of the given responses for a loop-based or Stream API-based source code.
(3) **H0**$_{perception}$: There is no difference in the participant's perception (overall performance, speed, and readability) of source code based on loops or the Stream API.

Each hypothesis comes with three different response variables:

(1) **Response time (ratio scale)**: The time it took a participant to answer a question.
(2) **Correctness (true/false)**: Whether or not a response matched the actual result of the code.
(3) **Perception (ordinal scale, 1–10)**: The participant's perception of each task with respect to overall performance, speed, and asthetics.

The independent variables in the experiment were:

(1) **Task**: The concrete code given to the subjects (Task 1 – 7).
(2) **Technique**: The technique in which the code was written (loop or Stream API).

While the variable time and correctness are from our perspective the important response variables, the participant's perception is interesting with respect to whether the potential outcomes of the experiment were also perceived by the participants.

The time measurement started right from the moment when a task was shown. The participant had to chose 0–n elements from the output table and then had to click on the submit button. The application stopped the time (and assigns the corresponding time to the task) and asked the participant to fill out a questionnaire for the current task. After finishing the questionnaire, the participant explicitely needed to start the next task.

3.4 Execution

We selected 20 volunteers (10 master students and 10 professionals) and measured each one in an individual session. Based on our subjective impression, 20 participants would be already enough to detect an effect. The participants were chosen based on purposive sampling [30] in the following way[13]: The authors contacted professional developers and asked them whether they are familiar with the Stream API and whether they are willing to participate. Once 10 developers were found we repeated the same process for master students.

Despite the fact that all volunteers stated that they were familiar with the Stream API, we still gave each one a short introduction into the iteration of collections using loops and the Stream API and emphasized especially for the Stream API specific operations to ensure similar levels of knowledge among participants. This short refreshment training took about 15 minutes depending on a participant's prior knowledge and depending on a participant's specific questions.

Once the participant started with the experiment, all 14 tasks were done one after another but the participant was permitted to take a break between two tasks.

[13]Purposive sampling is a non-probability sampling technique that is quite often applied in software engineering. The study by Baltes and Ralph [2] revealed that about 70% of empirical studies in software engineering use purposive sampling.

4 RESULTS

4.1 Response Times

The data was analyzed using an ANOVA with repeated measurements on the dependent variable time and the within-subject variables task (1–7), and technique (loop, Stream API) and the between-subject variable group (A, B).[14] Table 3 contains a more detailed description of the statistical results.

The variable group is not significant (p=.440) and there is no interaction between group and task (p=.176) and group and technique (p=.746). Hence, we found no indicator for carryover effects which indicates that the counterbalanced design worked out.

Both main effects (technique and task) are significant (both p<.001) with relatively large effect sizes (both η_p^2>=.695). But it turns out that there is a significant interaction effect between both variables (p<.001, η_p^2=.160).

Figure 5 illustrates the interaction between technique and task (including the 95% confidence intervals). The figure suggests that the differences between loop and Stream API depend on the tasks: while for three tasks (2, 3, and 7) the differences are rather small, the difference is rather large for Task 1, Task 4, and Task 6.

We need to keep in mind that Task 6 played a special role. We introduced it in addition to Task 4 because we were not sure whether the mapping in the imperative style could be easier understood if two loops are used. Taking the difference between Task 4 and 6 in the imperative style into account, we can exclude this possibility.

Pairwise comparisons[15] of loops and the Stream API per task show for all tasks a positive effect of the Stream API[16] which is also illustrated in the interaction diagram: for each task the mean for the Stream API is lower than the mean for the loop. Looking at the ratios of means, we see that they vary from 123% (Task 2) up to 282% (Task 6). I.e. the usage of loops required at least 23% more time and at most 182%. Due to the interaction it is valid to say that a positive effect of the Stream API exists independent of the task, but how large the difference in time actually is depends on the tasks.

We should take into account that we introduced Task 6 under the assumption that using two loops for a grouping criterion might help understanding the iterative code. And this task revealed the highest difference between the Stream API and the loop. Including Task 6 the ratio $\frac{M_{loop}}{M_{stream}}$ is 178%, i.e. the loop required 78% more time to understand. If we exclude Task 6, we receive $\frac{M_{loop}}{M_{stream}} = \frac{114s}{74s} = 154\%$ (while all other results are comparable except the effect size for T*TA which is without Task 6 higher with η_p^2=.239). Hence, even if we remove one disputable task, the loop variant still required 54% more time to understand. And we think it is noteworthy that even the very simply tasks 3 and 7 showed a positive effect of the Stream API although we considered it as possible that the Stream API could have a negative effect.

Consequently, we reject H0$_{technique}$.

[14]All following analyses were done using SPSS v27.
[15]Since the the tasks were significantly different ($p_{algorithm}$ <.001), we intentionally did here a pairwise t-test without a correcting the p-value (using, for example Bonferroni).
[16]For task 5 and 7 the difference is only significant for an alpha level of .1, not for an alpha level of .05.

Table 3: ANOVA on 20 subjects on the response variable time. We report the interaction T*TA via pairwise t-tests for each task (such as Δ_1 for the comparison between loop and Stream API for Task 1) instead of the values for all treatment combinations. M, $CI_{95\%}$ and SD are rounded to full seconds.

	df	F	p	η_p^2	Treatement	N	$CI_{95\%}$	M	SD	Ratios
Technique (T)	1	40.977	<.001	.695	loop	140	107; 167	137	118	$\frac{M_{loop}}{M_{stream}} = \frac{137}{77}$=178%
					Stream API	140	64; 82	77	56	
Task (TA)	3.406	50.813	<.001	.738	*omitted*					
Group	1	.440	.516	.024	*omitted due to insignificant results*					
T * TA	6	8.020	<.001	.160	Δ_1^{medium}	20	30;116 (p=.002)	73	91	$\frac{M_{loop}}{M_{stream}} = \frac{179}{105}$=170%
					Δ_2^{medium}	20	3; 22 (p=.008)	13	20	$\frac{M_{loop}}{M_{stream}} = \frac{70}{57}$=123%
					Δ_3^{simple}	20	3;16 (p=.009)	10	15	$\frac{M_{loop}}{M_{stream}} = \frac{41}{32}$=128%
					Δ_4^{medium}	20	26; 14 (p=.006)	82	129	$\frac{M_{loop}}{M_{stream}} = \frac{194}{112}$=173%
					$\Delta_5^{complex}$	20	-4; 95 (p=.067)	46	106	$\frac{M_{loop}}{M_{stream}} = \frac{151}{106}$=142%
					Δ_6^{medium}	20	132; 223 (p=.000)	18	97	$\frac{M_{loop}}{M_{stream}} = \frac{276}{98}$=282%
					Δ_7^{simple}	20	-1; 24 (p=.070)	12	27	$\frac{M_{loop}}{M_{stream}} = \frac{46}{34}$=132%

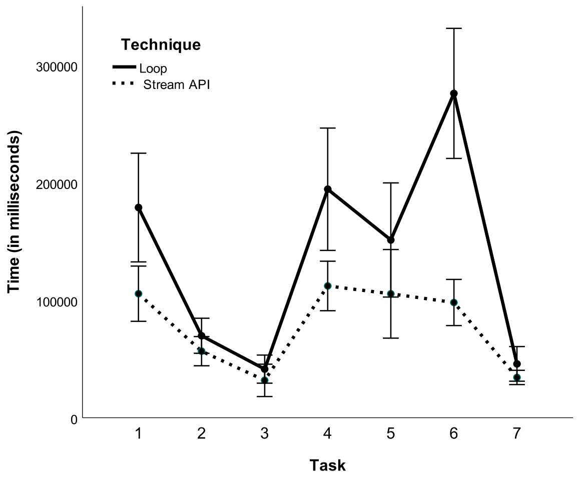


Figure 5: Interaction Task*Technique on response times

In order to check, whether the subject's background (master student versus professional) potentially influenced the results, we repeat the previous ANOVA with background as an additional dependent variable – being aware that this reduces the sample size for each treatment combination (which potentially leads to the situation that variables that were previously significant are now insignificant). Actually it turned out that background had a positive effect and students were on average .91 seconds (respectively 14%) slower than professionals (p<.001, η_p^2=.053, $M_{student}$ = 7.238, $M_{professional}$ = 6.330, $\frac{M_{student}}{M_{professional}}$ =1.14). And while the variable technique is still significant (p=.012, η_p^2=.030), the interaction between technique and task is no longer significant (p=.101, η_p^2=.049). However, from our perspective the latter result is rather the effect of the previously mentioned reduced sample size per treatment combination instead of an indidator that the result does not depend on the tasks.

4.2 Correctness

Because of the different nature of the response variable correctness (which simply states whether or not a result was correct), we cannot run an ANOVA. Instead, we run a χ^2-test first on the variables correctness and technique in order to check whether (independent of the tasks) the correctness of responses differed between the techniques.

Altogether, the participants gave 40 incorrect answers (loops=30/Stream API=10), which is a statistically significant result (χ^2(1, N=280)=11.667, p=.001).

Taking the results of the previous section into account, it seems reasonable to run the test for each task in separation. However, only for Tasks 5 and 6 the differences were significant: 11 incorrect answers were given for Task 5(10/1; χ^2(1, N=40)=10.157, p=.001), 10 incorrect answers were given for Task 6 (8/2; χ^2(1, N=40)=4.800, p=.028). The only task where the number of incorrect answers using the Stream API in comparison to loops was higher was Task 1 (2/4), but the difference was not significant (p=.376). For all other tasks, the number of incorrect responses were higher for loops than for the Stream API. Again, removing Task 6 from the analysis leads to the same results. Hence, we reject hypothesis $H0_{correctness}$.

4.3 Subjective Ratings

In order to analyze the subjective ratings[17] we run again a repeated measures ANOVA, this time with the dependent variable rating and the within-subject variables task, technique and question.[18] Likewise to the time analysis, all independent variables are significant (p<.001) with different effect sizes ($\eta_p^2(task)$ = .606; $\eta_p^2(technique)$ = .779; $\eta_p^2(question)$ = .231). Additionally, all combinations of the independent variables are statistically significant (p<=.001).

[17]Because of a technical error, the subjective ratings of 3 subjects were not recorded. Hence, the following analysis is based on the data of 17 subjects.

[18]Because of space limitations we do not provide the complete data of the ANOVA here and report instead only the most essential results.

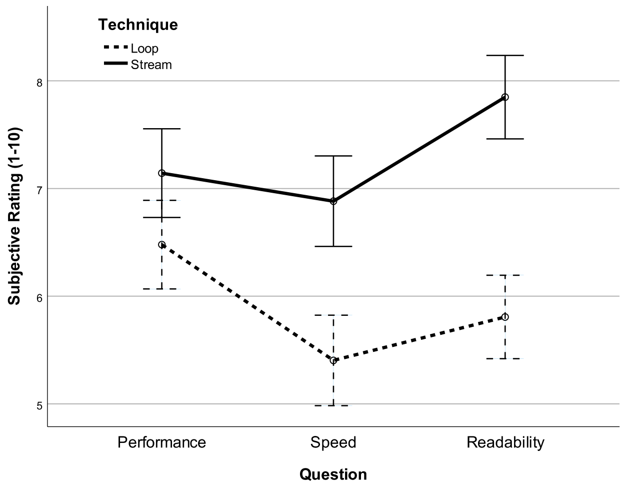


Figure 6: Interaction Question * Technique on ratings

Figure 6 illustrates the interaction diagram for question*technique. It shows, that participants always rated the Stream API higher than the loops and the largest difference between both techniques exists for readability. Consequently, we reject hypothesis $H0_{perception}$.

We think that the second and the third question play the most important roles, because the question on the speed indicates whether subjects were able to perceive the difference in time we detected in section 4.1 and readability could direct into the same direction (although the rating criterion is less precise). Figure 7 illustrates the interaction between the speed ratings and the tasks. For all tasks the means for the Stream API are higher than for the loops. We do not show the interaction diagram for the question on readability which looks similar to the interaction diagram of the speed ratings.

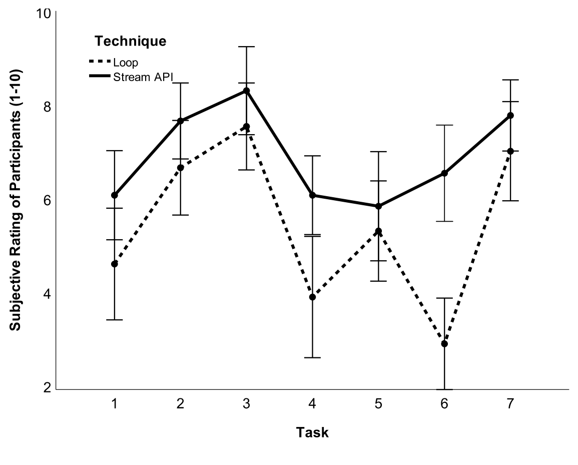


Figure 7: Interaction Task * Technique on speed ratings

Because of the similarities between speed ratings and time measurements it is plausible to run a multiple linear regression on the variables speed rating and technique to explain the response times. The resulting regression equation is significant (F(3, 234)=85.900, p<.001) with R^2=.524. In other words, the differences in time were not only measured and measurable, they were also perceptible by the participants. Repeating the same for the question on readability results – for understandable reasons – in a weaker model (F(3, 234)=34.688, p<.001) with R^2=.308: the readability perception says less about the understandability (in terms of times) than the speed perception. In other words: whether or not someone considers some piece of code more readable is rather a weak indicator for the code's understandability.

5 THREATS TO VALIDITY

Experiments should document their potential threats to validity (see for example [41]) and we see the following threats worth considering.

Code representation: The code given to the participants is obviously a threat and the code could be written in a different way. For the Stream API anonymous inner classes could have been used instead of lambda expressions, the different expressions could be stored in local variables, etc. Something similar could be done for the loops. In all cases, the corresponding names could more clearly express what they stand for. To a certain extent, a variable that expresses too precisely its meaning could be problematic for the experiment, because participants could just rely on the names in the code instead of the semantics.

Chosen tasks: We think that there are much more complicated means to specify operations on collections – and they have probably an impact on the difference between the Stream API and loops. We think that the differences become larger if multiple grouping operations are applied, but the present study cannot give evidence for this statement.

Training phase / selection of participants: The subjects were chosen using purposive sampling, which might or might not influence the results. However, potential participants were asked upfront whether they are familiar with the Stream API in Java. Actually, we think that just asking people whether or not they are familiar with a certain tecnique is potentially problematic, because "being familiar with some technique" could mean different things to participants. We also think that the short refreshment into the Stream API respectively into the iteration of collections has a potential effect on the results – but we cannot estimate, how large this effect is (or whether it existed in the present study at all).

Measurements and measurement technique: The experiment used a graphical representation for the input and output of each task – and it used the moment when the participant pressed the submit button for the time measurement. The representation of the input data was chosen to ease the reading of the code. If this assumption holds, it means that in a practical setting the differences between the loop and Stream API would be probably less because the reading of the additional code would cause additional deviation. Next, the answers were given by clicking on the appropriate lines. In principle one can argue that it would have been a better measurement to stop the time recording when the first mouse button was pressed. But in preliminary experiments we made the experience that people do not start to choose answers when they know the whole answer to a question. From that perspective, we think that the chosen measurement technique is appropriate although it still contains some additional time for navigating with the mouse, etc.

6 RELATED WORK

Although the discussions whether or not functional languages in general or functional language constructs are helpful in addition to imperative language construct are vivid since decades (see for example [6, Chapter 4][19], [21, 32] among many others) and while there are approaches avaible to translate imperative code into a more functional style (see for example [9, 13, 17, 36]), there is not much evidence on the possible effects of the usability of functional language constructs in general, the combination of functional language constructs into other kinds of language or, even more particular, the introduction of lambda expressions into today's main stream languages. On the other hand, the technical implications of for example lambda expressions in comparison to existing constructs are often studied (see for example [28]).

Uesbeck et al. [37] provided a controlled experiment on lambda expressions for replacing while loops with lambda expressions in C++. In the study, three tasks were designed and applied to 54 participants (from freshmen to junior up to senior developers) who needed to write corresponding algorithms. The response variables in the experiment were correctness, time, and number of compilation errors. The independent variables were technique (lambda versus loop) and the participants' experience level. The study revealed a positive effect of loops in comparison to lambda expressions on all response variables: programmers using lambdas were on average about 50% slower and the success rate of loops was higher. The only noteworthy exception were experienced developers for whom no differences between lambda expressions and loops were detected.

One controlled experiment that compared a declarative style with a more traditional style (although not explicitly related to loops) was performed by Salvaneschi et al. [33] where reactive programming (which is a more declarative style of programming, see [10]) was compared to the application of the observer design pattern [14]. 127 participants read the code of 10 code snippets and answered questions about them. The response variables were reaction time and correctness. It turned out that reactive code took less time and increased the correctness of the answers. Additionally, the participants expressed their perception of the two programming styles and there seem to be some agreement that the participants considered reactive code as more desirable and more readable.

Pankratius et al. studied the possible impact of a rather functional style in comparison to a rather imperative style for multicore programming [29]. Thirteen programmers generated 39 Java and 39 Scala programs. While the main focus of the study is on the technical aspects of the programs, programmers were also asked about their perception of the programs. The developers perceived the Java programs more often readable than the Scale programs. Additionally, while all programmers considered the Java programming model as easy, this statement was only received from 30% of the developers for Scala.

Other studies based on the analysis of code repositories also give first indications for a comparison of declarative and imperative programming languages. For example, the study by Nanz et al. [26] compares different languages from different perspectives, mainly based on their technical characteristics (runtime, memory consumption, etc). However, the study also contains statements about errors found in the projects and one of the findings was, that Java programs were more error prone than for example programs written in the functional language Haskell. A comparable study by Ray et al. [32] on a larger sample came to opposite results.

A study that focused on Java was performed by Lucas et al. [22] who compared the usage and usability of lambda expressions in comparison to anonymous inner classes. The authors evaluated code snippets before and after the introduction of lambdas and asked 28 developers about the acceptance of lambda expressions where about 50% considered the introduction of lambda expressions as an improvement of Java. While the application of readability and understandability metrics did not find significant differences, developers perceived some code migrations towards lambda expressions negatively, especially when for-loops were replaced.

Another study that focused on the acceptance of lambda expressions in Java was performed by Mazinanian et al. [24] who investigated the use of lambda expressions in open-source projects. They analyzed how the projects evolved and applied static source code analyses while also interviewing 98 developers in these projects. Half of the analyzed projects showed a significant increase in adoption while core developers introduced more often lambdas than others per LOC. Furthermore, lambda expressions were mostly used as a replacement for anonymous inner classes where developer perceived a better readability of lambda expressions over anonymous inner classes.

Another analysis of lambda expressions was performed by Nielebock et al. [27] who examined 2923 open-source projects written in C#, C++, and Java with regard to the use of lambda expressions in concurrent code. Their findings did not show that lambda expressions are applied more frequently in concurrent contexts than in general, rather the opposite. Additionally, they investigated other use cases for lambda expressions and found them to be used above average in user-interface and testing code as well as implementations of generic algorithms such as sorting.

We ran a study on the readability on lambda expressions in comparison to anonymous inner classes before [18]. Actually, that study was motivated by the fact that it was unclear to us whether lambda expressions should be used in the present study, i.e. whether lambda expressions should be used with or without type annotations, or whether traditional Java constructs such as ordinary objects, respectively anonymous inner classes should be used. In the previous study it turned out that lambda expressions without type annotations improved the readability up to 35%. Consequently, we felt convinced that the proposal found in Oracle's Stream API documentation, where lambda expressions were used as parameters for the Stream API, was better suited than using any other alternative.

7 DISCUSSION AND CONCLUSION

The present work was motivated by the tendency of main stream (imperative) programming languages to integrate declarative means via lambda expressions in conjunction with corresponding APIs that permit the ability to operate on collections in a more declarative way instead of the traditional way using loops (in conjunction with local

[19]An english translation is available under https://caml.inria.fr/pub/docs/oreilly-book/

variables, conditions, etc.). More precisely, we were interested in whether the application of the Stream API (which makes use of the rather declarative lambda expressions) in Java had a measureable effect on the understandability of operations on collections.

The controlled experiment on 20 participants and 7 tasks (where participants had to determine the result of code) revealed that overall the Stream API improved the understandability (in terms of response times): loops required 78% more time (even if we removed a debatable task, the effect was still 54%). On an individual basis, only for two tasks the difference were not significant on an alpha level of .05 (but on an alpha level of .1) which leads us to the generalization that for all tasks a positive effect of the Stream API was measured. It is noteworthy that the participants (who considered the Stream API code more readable than the loop code) were able to perceive the differences in time. But the user perception also said that the readability perception might be only a weak indicator for the understandability.

What makes this result interesting is that a different experiment in the literature already studied the effect of lambda expressions on collections and found a rather negative effect when participants had to write code [37]. There, only senior developers had no negative effect of the more declarative style. Hence, the present experiment casts a new light on the discussion about whether or not functional language constructs might be beneficial in imperative languages. It might be the case that the application of lambda expressions might be harder than the traditional application of loops (argument writability), but that the resulting code with lambda expressions is more understandable (argument understandability / readability). Consequently, the question of whether or not functional constructs are helpful in imperative languages could maybe not be simply answered by a "yes" or "not". It might be the case, that it is beneficial for reading and understanding code, but the opposite for writing code (but one should keep in mind that IDE support might help in the latter case). In addition to that the experiment revealed that the difference between the understandability of both styles depends on the concrete code snippets under investigation: in some cases the benefit in understanding might be just 20% in time, in other cases much more (up to 78%, respectively – ignoring a disputable task – 54% more).

However, we should not overgeneralize the results with respect to the imperative versus functional language discussion. Actually, the introduction of lambda expressions into an imperative language is just one way to consider a more declarative style. Other languages such as SQL are declarative as well, although their language constructs are quite different from lambda expressions. Additionally, one should keep in mind that the syntax of lambda expressions is quite different between languages: although C++, Java, and Python are object–oriented languages where loops and method calls are syntactically quite similar, their notations or lambda expressions are quite different. It is possible that the results of the present paper depend much on the chosen lambda syntax in Java (taking into account that already the different notations of lambda notations in Java have a measurable effect [18]).

Under the assumption that language designers consider understandability of code as valuable one needs to ask why current main stream languages evolved in a way as they did. Because the usage of a more declarative style in rather imperative programming languages is far from being new. For example, the Collection API in the programming language Smalltalk – a language that was designed more than 40 years ago – already contained methods that received lambda expressions as parameters: the API has a number of parallels to the Stream API we find today in Java. From that perspective, it makes sense to suggest that language designers and API designers should apply usability studies right from the beginning of the language design or API design process. Because such studies could influence the perspective of language and API designers on the usability of certain language constructs or APIs.

In addition to that, we think that it is desirable to study in more detail the role of type inference in the Stream API. We think that one reason the Stream API code is more understandable is the quite simple syntax of lambda expressions in Java where not even type declarations are required. Although we are aware that there are studies in the literature that found a rather positive effect of static types (see for example [11, 12, 20, 32]), we still think that the rather compact syntax of lambda expressions in Java without type declarations (but still statically typed) is helpful (which requires additional studies), respectively, there are already some initial studies available that give evidence that code without type annotations is helpful [18].

And finally, the present study might be used to think about whether teaching of main stream language could be improved by focussing more on lambda expressions for iterations instead of traditional loops. But especially in that case, it is desirable to have more empirical data on the learnability of lambda expressions.

In general, the present study could be considered as a contribution to the ongoing discussion about functional versus imperative programming languages and it could be used as an argument for a rather functional style. However, we think that the present study should be handled with care with respect to such an argument. Because on the one hand the focus of the present study was just on operations on collections and on the other hand it would be desirable to have a larger basis of empirical facts for that discussion.

REFERENCES

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, USA, 1985.

[2] Sebastian Baltes and Paul Ralph. Sampling in software engineering research: A critical review and guidelines. *CoRR*, abs/2002.07764, 2020.

[3] Dave W. Binkley, Marcia Davis, Dawn J. Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. *Empir. Softw. Eng.*, 18(2):219–276, 2013.

[4] Richard Bird and Philip Wadler. *An Introduction to Functional Programming.* Prentice Hall International (UK) Ltd., GBR, 1988.

[5] L. Bossavit. *The Leprechauns of Software Engineering.* Laurent Bossavit, 2015.

[6] E. Chailloux, B. Pagano, and P. Manoury. *Développement d'applications avec Objective Caml.* O'Reilly, 2000.

[7] Craig Chambers. Object-oriented multi-methods in cecil. In Ole Lehrmann Madsen, editor, *ECOOP '92 European Conference on Object-Oriented Programming*, pages 33–56, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[8] Microsoft Cooperation. *C# Language Specification Version 3.0.* 2007.

[9] R. Dantas, A. Carvalho, D. MarcÃlio, L. Fantin, U. Silva, W. Lucas, and R. BonifÃ¡cio. Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 497–501, 2018.

[10] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. Association for Computing Machinery.

[11] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. How do api documentation and static typing affect api usability? In *Proceedings of the*

36th International Conference on Software Engineering, ICSE 2014, pages 632–642, New York, NY, USA, 2014. ACM.

[12] Lars Fischer and Stefan Hanenberg. An empirical investigation of the effects of type systems and code completion on api usability using typescript and javascript in ms visual studio. *SIGPLAN Not.*, 51(2):154–167, oct 2015.

[13] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. Lambdaficator: From imperative to functional programming through automated refactoring. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1287–1290. IEEE, 2013.

[14] Erich Gamma, Richard Helm, and Ralph E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1st ed., reprint. edition, 1994.

[15] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., USA, 1983.

[16] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition.* Addison-Wesley Professional, 1st edition, 2014.

[17] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. Crossing the gap from imperative to functional programming through refactoring. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 543–553. ACM, 2013.

[18] Stefan Hanenberg and Nils Mehlhorn. Two n-of-1 self-trials on readability differences between anonymous inner classes (aics) and lambda expressions (les) on java code snippets. *Empirical Software Engineering*, 27(2):33, Dec 2021.

[19] Johannes C. Hofmeister, Janet Siegmund, and Daniel V. Holt. Shorter identifier names take longer to comprehend. *Empir. Softw. Eng.*, 24(1):417–443, 2019.

[20] Michael Hoppe and Stefan Hanenberg. Do developers benefit from generic types? an empirical comparison of generic and raw types in java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 457–474, New York, NY, USA, 2013. Association for Computing Machinery.

[21] Shriram Krishnamurthi and Kathi Fisler. Programming paradigms and beyond (chapter 13). In S.A. Fincher and A.V. Robins, editors, *The Cambridge Handbook of Computing Education Research*, pages 377–413. Cambridge University Press, 2019.

[22] Walter Lucas, Rodrigo Bonifácio, Edna Dias Canedo, Diego Marcílio, and Fernanda Lima. Does the introduction of lambda expressions improve the comprehension of java programs? In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, SBES 2019, pages 187–196, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Lech Madeyski and Barbara A. Kitchenham. Effect sizes and their variance for AB/BA crossover design studies. *Empir. Softw. Eng.*, 23(4):1982–2017, 2018.

[24] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in java. *Proc. ACM Program. Lang.*, 1(OOPSLA):85:1–85:31, 2017.

[25] Steve McConnell. What does 10x mean? measuring variations in programmer productivity. In Andy Oram and Greg Wilson, editors, *Making Software - What Really Works, and Why We Believe It*, Theory in practice, pages 567–574. O'Reilly, 2011.

[26] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in rosetta code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 778–788. IEEE Press, 2015.

[27] Sebastian Nielebock, Robert HeumÃŒller, and Frank Ortmeier. Programmers do not favor lambda expressions for concurrent object-oriented code. *Empirical Software Engineering*, 24(1):103–138, 2019.

[28] F. Ortin, P. Conde, D. Fernandez-Lanvin, and R. Izquierdo. The runtime performance of invokedynamic: An evaluation with a java library. *IEEE Software*, 31(4):82–90, 2014.

[29] Victor Pankratius, Felix Schmidt, and Gilda Garreton. Combining functional and imperative programming for multicore software: An empirical study evaluating scala and java. *Proceedings - International Conference on Software Engineering*, pages 123–133, 06 2012.

[30] M.Q. Patton. *Qualitative Research & Evaluation Methods: Integrating Theory and Practice.* SAGE Publications, 2014.

[31] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

[32] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. *Commun. ACM*, 60(10):91–100, September 2017.

[33] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering*, 43(12):1125–1143, 2017.

[34] S.S. Senn. *Cross-over Trials in Clinical Research.* Statistics in Practice. Wiley, 2002.

[35] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Professional, 4th edition, 2013.

[36] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 60–70. IEEE Press, 2017.

[37] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. An empirical study on the impact of c++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 760–771, 2016.

[38] Guido Van Rossum. *Documentation of the Python Programming Language 0.9.* 1991.

[39] S. Vegas, C. Apa, and N. Juristo. Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering*, 42(2):120–135, 2016.

[40] Stefan Wellek and Maria Blettner. On the proper use of the crossover design in clinical trials part 18 of a series on evaluation of scientific publications. *Deutsches Ärzteblatt international*, 109:276–81, 04 2012.

[41] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering.* Computer Science. Springer Berlin Heidelberg, 2012.