Technical report

I. User Study

Evaluating software engineering tools with user studies is not a recent practice, but it is still scarce. Mainly because most researchers find these experiments too difficult to design and conduct, have difficulties recruiting participants, and believe that the results might be inconclusive [1]. However, researchers who conducted user studies agree that user evaluations provide useful insights that generally outweigh the study costs and consider that studies increase the impact of the work [2].

In previous research, authors have employed user studies to compare different programming language designs. For example, comparing the benefits of using static versus dynamic type systems for maintainability and undocumented software, [3,4] analysing the usability and learnability of API aspects [5,6], and advanced language features such as lambdas [7] and garbage collection [8]. In PLIERS, the authors also present strategies for conducting summative usability studies and apply them to two novel programming languages. These studies either apply usability studies or randomized control trials (RCTs) [9]. Usability studies usually have participants complete tasks where relevant data is retrieved, such as the time spent on the task, the correctness of the answers, and the errors made. In RCTs, the configuration of the study aims to compare two or more designs options. Thus, one of the designs serves as a control condition, and each participant is assigned a random design to fulfil the tasks.

To evaluate LiquidJava, we have developed a user study that combines the two types of tests to answer the following research questions:

RQ1 Are refinements easy to understand?
RQ2 Is it easier and faster to find implementation errors using LiquidJava than with plain Java?
RQ3 Is it hard to annotate a program with refinements?
RQ4 Are developers open to using LiquidJava in their projects?

Given the research questions, we planned the study with tasks to assess the usability of LiquidJava, and also compare its benefits against Java. This section depicts the details of the study configuration (Section I-A), the participants' background (Section I-B), then the tasks and the obtained results are shown (Section I-C) and discussed (Section I-D), and, finally, the threats to validity of the study are presented (Section I-E).

A. Study Configuration

The study was designed to introduce participants to LiquidJava and answer the aforementioned research questions. Therefore, the study is divided into four programming tasks, one section of introduction to LiquidJava, and a final section to gather the participants' opinions on their experience of using LiquidJava.

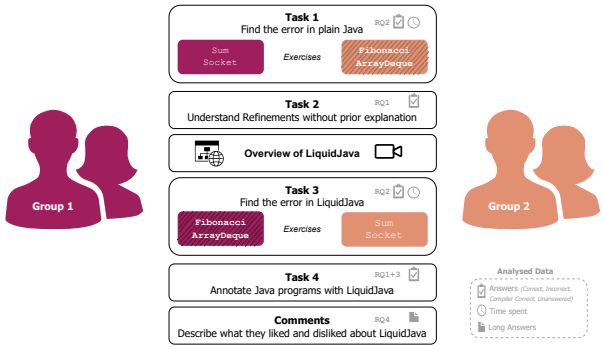


Fig. 1: Configuration of the evaluation study.

The study was conducted individually with each participant that signed up for it. Every participant received the study plan with the described tasks and the answer sheet to input their answers. The participants were free to quit the study at any time, and skip questions if they felt that they were not able to answer them, however, they could not go back to previous study sections to complete any unanswered questions.

Figure 1 schematizes the configuration of the study. When starting, each participant was randomly assigned to one group. All participants within a group follow the same order of exercises. However, the two groups had different exercises for two tasks to compare the performance of the participants while using Java and LiquidJava. Each section of the study is described as follows:

- **Task 1: Find the error in plain Java** – In this task, participants are asked to detect implementation errors in Java code, a language that they must be familiar with. The implementation errors on the programs make the execution incorrect against the informal documentation presented in the method's Javadoc, and participants must report the error and a possible correction. In this task, the exercises are split between the two groups. To evaluate the answers, we gathered the responses and the time participants spent on each exercise of the task.
- **Task 2: Interpreting refinements without prior explanation** – Although participants must be familiar with Java, they are not required to be familiar with refinements. Therefore, in this task, all participants had their first contact with LiquidJava, specifically with the language of the refinements. Here, the participants had to interpret the refinements present in different sections of the code (variables, methods, and classes) and provide a correct and incorrect use of the annotated code without ever being introduced to the language. This task aims to see if the participants find the refinements intuitive and easy to understand without a prior explanation. Thus, this study

section is related to **RQ1**, and the data gathered captures the correctness of the answers the participants gave to the correct and incorrect uses of the refinements.

- **Overview of LiquidJava** – Participants were exposed to a 4-minute video and a webpage explaining the concepts of LiquidJava using the examples of the previous task. Both resources were available for the participants to use in the remaining of the study. These resources help making study reproducible while reducing bias from the interviewers.
- **Task 3: Find the error with LiquidJava** – This was a repetition of Tasks 1, but using the LiquidJava plugin. To reduce the bias of already knowing the solution, there were two problems. Each half of the participants did one problem in Task 1, the control conditions, and the alternative problem in Task 3. Comparing the performance between the two tasks allows us to answer to **RQ2**.
- **Task 4: Annotate Java programs with LiquidJava** – Participants were presented with three plain Java programs and were asked to annotated them with LiquidJava specifications. This task aims to answer **RQ3** by asking participants how difficult it was to annotate variables, fields, methods and classes. Because this was a relatively short task, its success also answers **RQ1**.
- **Final Comments** – Finally, participants were asked about their overall opinion on using LiquidJava. They were also asked if they would like to use LiquidJava in their future projects, to answer the last research question, **RQ4**.

The exercises used in the different tasks are detailed in Section I-C, along with the obtained answers and result discussion.

The study sessions were all conducted through the Zoom video platform, given the restrictions imposed by the COVID-19 pandemic, and the participants used their own environments to complete the study tasks. To ensure that these environments fulfilled the requirements to complete all tasks, participants were asked to come to the session with Visual Studio Code installed, and with the JDK11 and the Language Support for Java(TM) by Red Hat extension enabled so that the installation of these applications would not take time from the study session. During the study, participants had access to the GitHub repository with the study files and were asked to share their screen with the interviewer. With the participants' permission, the interviewer recorded the given answers and a video of the session.

B. Background of Participants

This study was designed to have 30 participants familiar with Java. Participants were recruited through social media channels, such as Twitter, Facebook and Instagram, and through direct contact via email.

Figure 2 shows that more than 90% of the participants considered themselves *Familiar* or *Very Familiar* with Java. The remaining participants, who considered themselves only *Vaguely Familiar* with Java, were only accepted into the study because they stated to be familiar with testing frameworks (e.g., JUnit). Of all the participants, 80% were *Vaguely Familiar* or

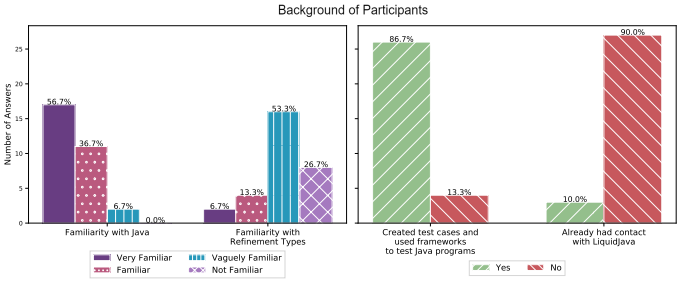


Fig. 2: Preferences on the Syntax for Methods' Refinements.

Not Familiar with Refinement Types which shows that despite their utility, Refinement Types are not widely known and used. The 3 participants that were familiar with LiquidJava, had attended a talk about it, but had not used it in any capacity. As for the participant's occupations, around 50% are university students, 26% work in the industry and the remaining work as faculty in academia, as described in the Table I.

All participants completed the study, and the gathered data results are analysed in the next section.

TABLE I: Occupations of study participants.

Occupation/Job	# Participants
Business	8 (26.7%)
Faculty	6 (20.0%)
PhD Students	5 (16.7%)
Masters Students	7 (23.3%)
Final-Year Bachelor Students	4 (13.3%)

C. Exercises and Results

This section presents the exercises used in each of the tasks of the study and the results obtained.

*1) **Interpreting Refinements without prior explanation**:* Since 90% of the participants had no previous contact with LiquidJava, and more than 80% were not familiar with Refinement Types, we wanted to understand if, without a prior explanation, the added specifications were intuitive to use. Thus, the study included a task with refinements examples that the participants needed to interpret and use in a correct and incorrect form. Specifically, we presented three code snippets with LiquidJava refinements with an increasing difficulty level (as showed in Listing 1) and asked the participants to implement a correct and incorrect usage for each of the represented features.

In the first exercise, participants had to assign a correct and incorrect value to the variable `x`, which restricted the range of values to the Earth's surface temperature limits. The second task asked the participants to implement a correct and incorrect invocation of `function1`, where the second parameter depends on the first. Finally, the last task presented a class protocol with three possible states and methods that modeled the object state, and the participants were asked to create a *MyObj* object and implement a correct and incorrect sequence of at least three invocations. The *MyObj* class aimed to represent a Vending Machine object with the three states *sX, sY* and *sZ* as *Show Items, Item Selected* and *Paid*, respectively. The anonymization

```
// 1 – Variable Refinement
@Refinement("-25 <= x && x <= 45")
int r;

// 2 – Function/Method Refinement
@Refinement("_ >= 0")
public static double function1(@Refinement("a >= 0") double a,
        @Refinement("b >= a") double b){
  return (a + b)/2;
}

// 3 – Class Protocol Refinement
@StateSet({"sX", "sY", "sZ"})
public class MyObj {

    @StateRefinement(to="sY(this)")
    public MyObj() {}

    @StateRefinement(from="sY(this)", to="sX(this)")
    public void select(int number) {}

    @StateRefinement(from="sX(this)", to="sZ(this)")
    public void pay(int account) {}

    @StateRefinement(from="sY(this)", to="sX(this)")
    @StateRefinement(from="sZ(this)", to="sX(this)")
    public void show() {}
}
```

Listing 1: Variable refinement in LiquidJava and verification of its assignments.

of the states and the class name were intentional to make the participants try to understand the refinements instead of calling the methods according to their mental idea of how a vending machine works.

Figure 3 shows the evaluation of the answers given by the participants. Each answer was classified as *Correct* if both the correct and incorrect usage of the specification were correct, *Incorrect* if at least one of the usages was incorrect, or *Unanswered* if the placeholder for answering was left blank. In the variable assignment, 86.7% of the participants answered correctly. The remaining participants understood the error when the examples were explained and claimed that the error was a pure distraction and misread the logical operators. The invocation of the annotated method had only one incorrect answer (3.3%). For the sequential methods' invocations, that depended on the class protocol described using the `@StateRefinements`, 46.7% of the answers were correct, and the remaining amount was split into incorrect and blank answers, showing that this example is less intuitive and harder to understand without a prior explanation, but still comprehensible by almost half of the participants.

Overall, refinement annotations in variables and methods are intuitive and easy to understand. However, the annotation of classes and their methods with protocols is less intuitive, and, in half of the cases, the participants would need a previous explanation to use these annotations correctly.

*2) **Using LiquidJava to Detect Bugs**:* This section joins the first and third tasks, both related to finding implementation errors in Java code with or without the help of refinements. The pair of tasks aim to validate if using Java with Liquid Types

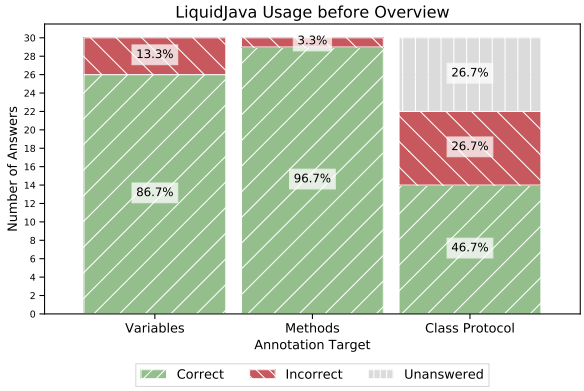


Fig. 3: Answers on interpreting LiquidJava refinements.

makes it easier and faster to detect and fix implementation errors in LiquidJava than in plain Java. To this end, we chose four exercises with implementation errors that the participants had to detect and fix. Firstly only using the plain Java code (*Task 1: Find the error in plain Java*) and then with the help of LiquidJava and its plugin for Visual Studio Code (*Task 3 - Find the error in LiquidJava*). Between both tasks, the participants had a small introduction to LiquidJava.

Each exercise had a plain Java version and a LiquidJava version with the same implementation errors to allow us to compare the number of participants that found and fixed the bug and the time taken in each version. The first group of participants started with exercises *Sum and Socket* while the second group started with the plain Java versions of the exercises *Fibonacci and ArrayDeque*. When moving to detect the errors using LiquidJava, the first group had the exercises *Fibonacci and ArrayDeque* whereas the second group got the exercises *Sum and Socket*. Therefore, one participant never used the same exercise in both tasks, avoiding tainting the second task with previous knowledge of the solution and allowing us to obtain plain Java baselines for every exercise. With this split, the maximum number of answers to each version is 15 since only half the participants viewed each exercise version.

In both *Task 1* and *Task 3*, we gathered the time spent in each exercise and the given answers. The answers were then evaluated into one of four possible categories: *Correct*, *Incorrect*, *Unanswered*, and *Compiler Correct*. The last category, *Compiler Correct*, represents the answers that, despite not having any error detected by the LiquidJava compiler, are not utterly correct according to the exercise.

Fibonacci Exercise: This exercise presents a recursive implementation of the method `fibonacci` that computes the nth Fibonacci number. However, the code contains an implementation error in the base case of the recursion, where the starting values of `F(0)= 1` and `F(1)= 1` are not respected. Listing 2 shows the method with the LiquidJava refinement annotations according to the informal documentation. The plain Java version has the same Java code but without the annotations on lines 1, 2, 10 and 11. The refinements were introduced to the parameter and the method's return following the Javadoc. In addition, the aliases were introduced to give more meaningful

```java
@RefinementAlias("Nat(int x) { x >= 0 }")
@RefinementAlias("GreaterEqualThan(int x, int y) { x >= y }")
public class Test1 {
  /**
   * Computes the fibonacci of index n
   * @param n The index of the required fibonacci number (greater or
       equal to 0)
   * @return The fibonacci nth number. The fibonacci sequence follows
       the formula
           Fn = Fn-1 + Fn-2 and has the starting values of F0 = 1 and
               F1 = 1
   */
  @Refinement( "_ >= 0 && GreaterEqualThan(_, n)")
  public static int fibonacci(@Refinement("Nat(n)") int n){
      if(n <= 1)
          return 0;
      else
          return fibonacci(n-1) + fibonacci(n-2);
  }
}
```

Listing 2: Fibonacci in LiquidJava.

names to the predicates and show another feature that can be used in LiquidJava.

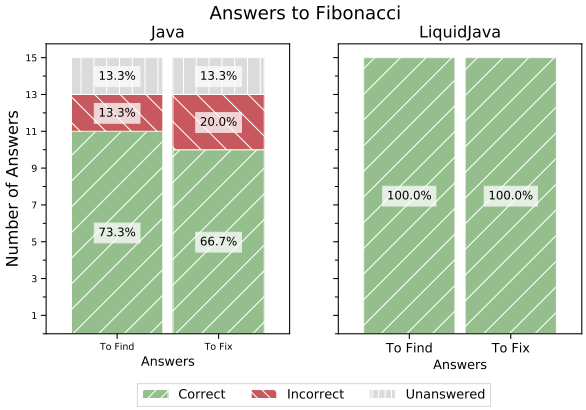
The answers and the time spent on the exercise in the plain Java and LiquidJava versions are plotted in Figure 4. We can see that all the participants could find and fix the exercise in LiquidJava, whereas in plain Java, only 73,3% found the error, and 66,7% were able to fix it. The answers accepted as *Correct* changed the return of the base case to `return 1`, and all the different answers were determined as incorrect since they did not fix the presented error. Additionally, in plain Java, 2 participants in total decided to skip this question without answering the possible location of the bug or proposing a fix.

The time spent on the Java version was, on average, smaller than the time spent in LiquidJava. The average time spent in the plain Java exercise was 2 minutes and 52 seconds, while the time spent on LiquidJava was 3 minutes and 22 seconds, reaching a difference of 30 seconds. This result suggests that participants are already used to Fibonacci's plain implementation given its popularity. When the new refinements were added, participants spent more time understanding the different sections of the code.

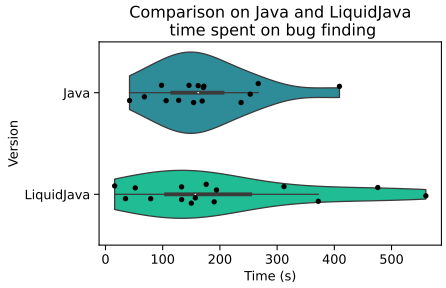
Sum Exercise: The `sum` exercise, presented in Listing 3, contains a recursive method that should implement the sum of all numbers between 0 and the given parameter. However, it contains a bug in the base case since the method returns 0 if the gotten parameter is 1. This exercise was inspired by the recursion example presented in *Refinement Types: A Tutorial*[10].

Listing 3 represents the LiquidJava version of the exercise. Again, the plain version is similar but without the annotations for the refinements (lines 1 and 6). The informal documentation does not specify any conditions to the parameter, leading to the omission of its refinement. However, it specifies the return conditions that can be simplified into the return refinement presented, using the same expression as the tutorial.

The answers and the time spent in this exercise are plotted in Figure 5. It is possible to see that in the plain Java version, only one participant could not find and fix the bug, while the



(a) Correctness.



(b) Duration.

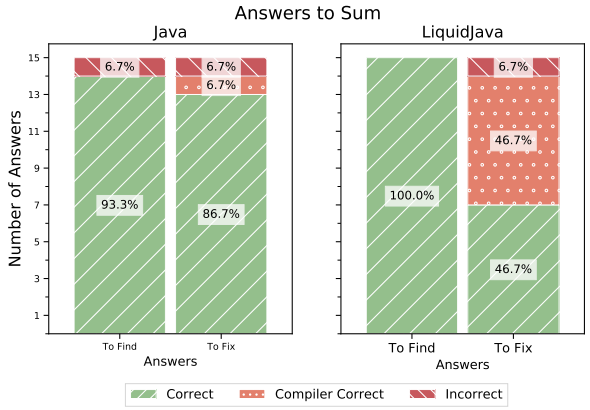
Fig. 4: Answers correctness and duration of the Fibonacci exercise, in both versions.

```java
@RefinementAlias("Nat(int x) {x >= 0}")
public class Test1 {
  /** The sum of all numbers between 0 and n
   * @param n
   * @return a positive value that represents the sum of all numbers
       between 0 and n, or 0 if n is negative */
  @Refinement("Nat(_) && _ >= n")
  public static int sum(int n) {
    if(n <= 1)
      return 0;
    else {
      int t1 = sum(n-1);
      return n + t1;
    }
  }
}
```

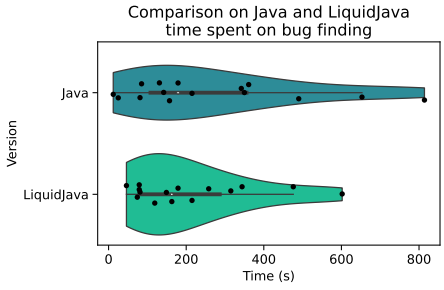
Listing 3: Sum exercise in LiquidJava.

14 others were able to find the error, and 13 correctly fixed it. On the other hand, in the LiquidJava version, every participant was able to locate the error. However, only 7 participants were able to fix it correctly, and the same number answered with *Compiler Correct* options.

The answers accepted as *Correct* adjusted the base case to one of the following versions: `if(n <= 0)` or `if(n < 1)`. The answers considered *Compiler Correct* are the ones that silenced the compiler error but are not correct according to the informal specification. Among these answers, 6 out of the 7 participants



(a) Correctness.



(b) Duration.

Fig. 5: Answers and time spent on the Sum exercise, in both versions.

changed the base case's return value, changing it to `return 1`, while the other participant answered `return Math.abs(n)`.

The 46.7% of *Compiler Correct* answers suggests that the participants were more focused on silencing the compiler error than on correcting the program according to the informal specification. Additionally, we may relate the 40% of `return 1` answers with the Task 1 of these participants, which included the Fibonacci exercise where the correct fix was changing the return line to `return 1`. This line of thought might indicate that the participants were biased by the previous sections of the study and opted for the same answer as they used in the beginning.

The time plot shown in Figure 5 contains two similar distributions, with a shorter average time in the LiquidJava version. In minutes, the participants spent 4 minutes and 30 seconds on average finding and fixing the error in plain Java, whereas in LiquidJava, they spent an average of 3 minutes and 32 seconds, being faster by almost 1 minute.

ArrayDeque Exercise: This exercise presented a program that uses the `ArrayDeque` class from the `java.util` library. The class contains popular methods to add, remove, and retrieve elements from an ArrayDeque object. These operations may depend on the number of elements present on the deque, and if those dependencies are not respected, exceptions will arise. With the Java plugin enabled on the IDE, participants were able to access the documentation of the class. Listing 4 presents a program that includes sequential invocations of methods of

```
1  ArrayDeque<Integer> p =
2    new ArrayDeque<>();
3  p.add(2);
4  p.remove();
5  p.offerFirst(6);
6  p.getLast();
7  p.remove();
8  p.getLast();
9  p.add(78);
10 p.add(8);
11 p.getFirst();
```

Listing 4: Client code that uses the ArrayDeque class.

```
1  @ExternalRefinementsFor("java.util.ArrayDeque")
2  @Ghost("int size")
3  public interface ArrayDequeRefinements<E> {
4    public void ArrayDeque();
5
6    @StateRefinement(to="size(this) == (size(old(this)) + 1)")
7    public boolean add(E elem);
8
9    @StateRefinement(to="size(this) == (size(old(this)) + 1)")
10   public boolean offerFirst(E elem);
11
12   @StateRefinement(from="size(this) > 0", to = "size(this) == (size(old(
         this)))")
13   public E getFirst();
14
15   @StateRefinement(from="size(this) > 0", to = "size(this) == (size(old(
         this)))")
16   public E getLast();
17
18   @StateRefinement(from="size(this)> 0", to="size(this) == (size(old(this))
         - 1)")
19   public void remove();
20
21   @Refinement("_ == size(this)")
22   public int size();
23 }
```

Listing 5: ArrayDeque Exercise.

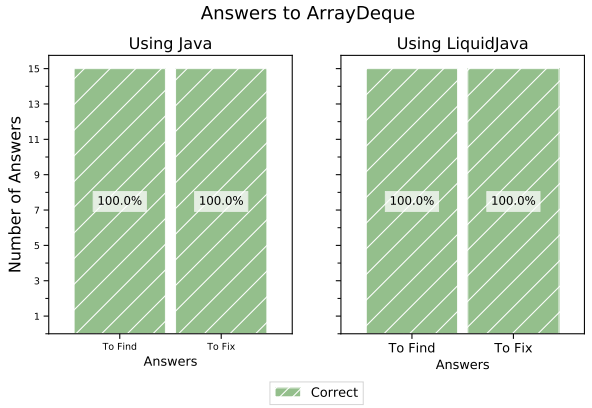
`ArrayDeque`. However, the invocation `p.getLast()` on line 8 produces an exception since it is called when the object is empty.

The code presented in the LiquidJava version, includes the client of Listing 4 and a separate file with the modelling of ArrayDeque with the ghost variable `size` as shown in Listing 5.

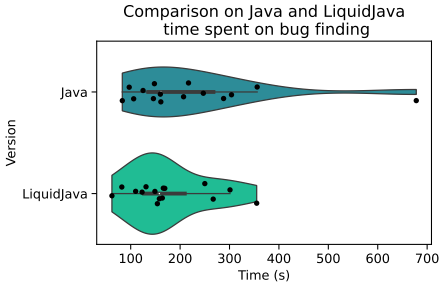
Figure 6 presents the results of the answers and the time spent on each version of the exercise. In this case, all the participants could correctly find and fix the error in both plain Java and LiquidJava. There were multiple options to fix the code to prevent the raising of the exception; the accepted answers included removing `p.getLast()` on line 8, verifying if the queue is empty with `if(!p.isEmpty)` right before line 8, changing the line to `p.peekLast()`, among others.

The average time participants took in this exercise in LiquidJava (2 minutes and 56 seconds) was 45 seconds less than using plain Java (3 minutes and 41 seconds).

Socket Exercise: The last exercise uses the `Socket` class from the external library `java.net`. The class was modeled as mentioned before (**??**) and a client method `createSocket(...)` was created with incorrect usage of the invocations' order (Listing 6). In this exercise, the error lies in the

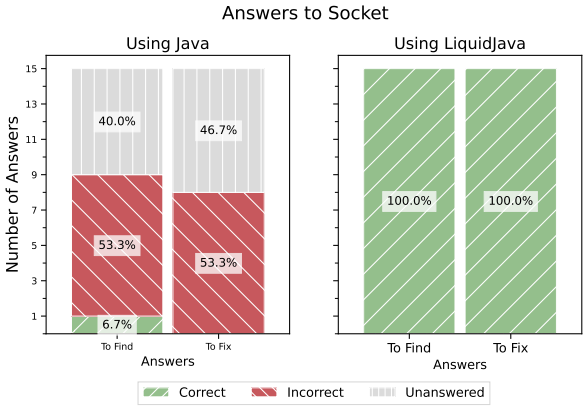


(a) Correctness.

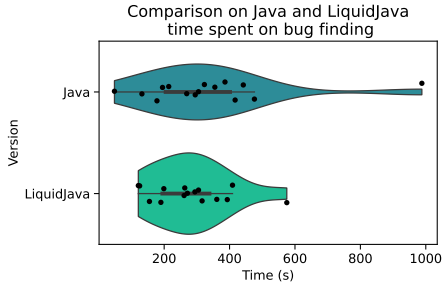


(b) Duration.

Fig. 6: Answers and time spent on the ArrayDeque exercise, in both versions.



(a) Correctness.



(b) Duration.

Fig. 7: Answers and time spent on the Socket exercise, in both versions.

```
1  //Exercise 2
2  public void createSocket(InetSocketAddress addr) throws IOException {
3      int port = 5000;
4      InetAddress inetAddress = InetAddress.getByName("localhost");
5
6      Socket socket = new Socket();
7      socket.bind(new InetSocketAddress(inetAddress, port));
8      socket.sendUrgentData(90);
9      socket.close();
10 }
```

Listing 6: Client code of Socket class.

invocation of `socket.sendUrgentData(90)` before the socket being connected to a server. The same client code was shown to participants in both Tasks 1 and 3. However, the code in Task 3 contained a file with the `Socket` class annotations of states and the allowed state transitions between the methods. Again, the informal documentation of the class from the library could be reached using the enabled Java plugin.

Figure 7 shows the evaluation of the answers and the time spent in the Socket exercise. Here, it is possible to see that only one participant was able to pinpoint the location of the error while using Java, and no participant was able to fix the error correctly. Additionally, we can see that the percentage of blank answers was higher in this exercise than in all others given the 40% to 46% of the participants that decided to move forward without answering.

However, in LiquidJava, all the participants could find the error and add the missing invocation to the `connect` method. The correct answer to fix the client code relied on the addition of the line `socket.connect(addr);` between lines 8 and 9, where `addr` is the server address passed as an argument. However, we did not specify that the parameter `addr` was the one supposed to be used in the invocation, therefore we accepted any answer that invoked the `connect` method regardless of the argument used in the call.

Regarding the time spent on this exercise, in the Java version, the participants spent an average of 5 minutes and 35 seconds, compared to the 4 minutes and 42 seconds spent in the LiquidJava version, showing that the participants were faster by 52 seconds in LiquidJava with a higher rate of correct answers.

Discussion on Java and LiquidJava exercises: With the results gathered from Task 1 and Task 3, we can conclude that LiquidJava consistently helped the participants find the bugs present in the code since the percentage of participants who found the bugs was always higher, or equal, in the LiquidJava version when compared to the plain Java version. LiquidJava also helped the participants fix the bugs according to the error information provided. However, the participants focused on silencing the errors, which led to some answers that were only considered *Compiler Correct* since the error was not totally fixed.

The task of finding and fixing the bugs was faster in

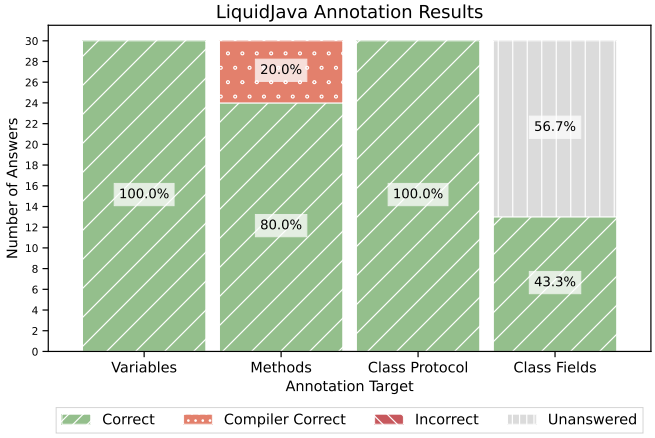


Fig. 8: Results of the annotations with LiquidJava.

LiquidJava in all but one exercise. The latter refers to the Fibonacci example, which may have had a shorter time because of its popularity and the fact that most developers are familiar with its plain Java version. From all the exercises, the one that benefited the most from the LiquidJava version was the `Socket` client, where we moved from having only one participant finding the error in plain Java to 100% in LiquidJava, and from no participant fixing the bug to 100% fixing it. These percentages may indicate that LiquidJava is more useful when used in more complex programs, with classes that have protocols less-known by developers.

Overall, LiquidJava helped participants find and fix the bugs, and in some cases, it helped them do it faster.

*3) **Adding LiquidJava Annotations**:* In Task 4, participants were asked to add LiquidJava annotations to the implemented code according to the informal documentation written in the program as comments. In this step, participants could use the website and the video to help writing the refinements.

Participants had to annotate programs with increasing order of difficulty. The first program relied only on the annotation of a variable with its bounds. The second program expected the annotation of a method by specifying the parameters and return refinements. Finally, the third program required the annotation of a class protocol and the class fields. For each program, we presented an example of a correct usage of the refinement and another example of its incorrect usage to help the developers test their refinements.

The participants shared their proposals for the annotation of each exercise, and we evaluated them with the four categories used in the previous section, of *Correct*, *Incorrect*, *Unanswered* and *Compiler Correct*. The results of the annotations are in Figure 8 and will be analysed along with each exercise below.

The first and more straightforward program is presented in Listing 7 and contained the simple task of restricting the value of the variable with an upper and lower bound, which could be accomplished with the annotation: `@Refinement("currentMonth >= 1 && currentMonth <= 12 ")`. In the first exercise, all the participants used the website as a resource to look for the right syntax, and 100% of them annotated the variable correctly.

```
/* A month needs to have a value between 1 and 12*/
int currentMonth;

currentMonth = 13; // Error
currentMonth = 5; // Correct
```

Listing 7: Variable to be annotated with LiquidJava and two assignments to test the refinement.

```
/**
* Returns a value within the range
* @param a The minimum border
* @param b The maximum border, greater than a
* @return A value in the interval [a, b] (including the border values)
*/
public static int inRange(int a, int b) {
    return a + 1;
}
...
inRange(10, 11); //Correct
inRange(10, 9); //Error
```

Listing 8: Method to be annotated with LiquidJava and two invocations to test the refinements.

Listing 8 shows the method presented in the second exercise where participants should add a refinement to the second parameter, changing the signature of the method to `public static int inRange(int a, @Refinement("b > a")int b)`, and refine the return type of the method, adding the refinement `@Refinement("_ >= a && _ <= b")` above the method's signature.

24 out of 30 participants were able to add the expected annotations leading to 80% of `Correct` answers. However, 20% only added the annotations to the parameter, silencing the example error but not completing the exercise in its totality, which lead to the *Compiler Correct* answers.

For the last exercise we asked participants to "Annotate the class TrafficLight, that uses rgb values (between 0 and 255) to define the color of the light, and follows the protocol defined by the image [in Figure 9]", as announced in the study guide. The evaluation of this exercise was split into two: the addition of the refinements to model the class protocol, and the specification on the class private fields.

100% of the participants correctly modeled the class by declaring the starting states and the state transitions allowed in each method. This percentage constitutes a significant increase in the understanding of the class protocols when compared to the first time participants tried to understand the protocol in Task 1, where half of the participants could not interpret the meaning of the annotations.

However, only 43.3% of the participants annotated the class fields, and the remaining participants did not add any refinement to them, leaving an incorrect assignment in the body of one class method. There might be several reasons for why participants did not annotate the class fields. A possible explanation is that they misinterpreted the exercise, not realising the need for these annotations, another possible explanation is that participants did not consider it important to add these

```
public class TrafficLight {
   private int r;
   private int g;
   private int b;

   public TrafficLight() { r = 76; g = 187; b = 23; }
   public void transitionToGreen() { r = 76; g = 187; b = 23; }
   public void transitionToAmber() { r = 255; g = 120; b = 0; }
   public void transitionToRed() { r = 230; g = 0; b = −1; }
}

//Correct Test – different file
TrafficLight tl = new TrafficLight();
tl.transitionToAmber();
tl.transitionToRed();
tl.transitionToGreen();
tl.transitionToAmber();

//Incorrect Test – different file
TrafficLight tl = new TrafficLight();
tl.transitionToAmber();
tl.transitionToRed();
tl.transitionToAmber();
tl.transitionToGreen();
```

Listing 9: Variable refinement in LiquidJava and verification of its assignments.

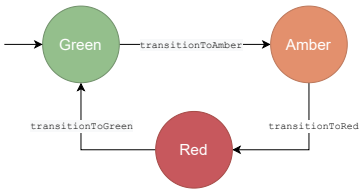


Fig. 9: Protocol of the TrafficLight that must be followed to the annotation.

annotations to the code.

After being introduced to the annotations, the participants had to evaluate the ease of adding annotations from *0 - Very Difficult* to *5 - Very Easy*. 60% of the participants considered that adding the annotations was *Very Easy*, while the remaining 40% considered the task *Easy*, leading us to conclude that the refinements are simple to add to implemented code.

*4) **Final Comments**:* At the end of the tasks, we asked the participants about the overall experience of using LiquidJava using three questions:

- What did you enjoy the most while using LiquidJava?
- What did you dislike the most while using LiquidJava?
- Would you use LiquidJava in your projects?

Table II presents the results to the first question, obtained using a qualitative coding[11] approach. We used inductive coding to create the codes, which were then used to review all passages and to identify the main topics of each answer, leading to a cohesive and systematic view of the results. We found that participants mostly appreciated the error reporting, state refinements to model objects and the intuitive and non-intrusive syntax.

Table III presents the topics that 26 participants reported they did not like. The remaining 4 participants did not identify any negative aspect. Of the 26, 8 explicitly mentioned there was nothing they did not like. The remaining negative aspects were about the syntax and some plugin features.

Topic	# References	Representative Quotes
Error Reporting	12	"[...] the fact it reported un-compliances with the specifications"; "[with] refinement types we have an assurance that the program is logically consistent at the time of coding."; "Helps to define the program's logic and avoid future errors";
State Refinements	11	"the state refinement it is very useful"; "Specification of states, because enforcing correct state transitions in the implementations is tedious and error-prone";
Syntax	6	"The simple syntax based on annotations", "Very intuitive syntax", "[the refinements addition is] non-intrusive";
Plugin	5	"Excellent integration with VSCode", "Very well assisted by the custom vs code extension";
Understandability	5	"Very intuitive", "Easy to understand in terms of semantics";
Useful	4	"Highly useful", "It helped me a lot to known which values/methods are correct when I'm coding";
Resources	3	"Using the video and the examples, it was easy to understand how the refinements work", "The examples and the video were very elucidative";
Flexibility	2	"I liked the flexibility of scenarios in which I could use LiquidJava", "+1 to the diverse set of refinements that can be written in different parts of the code".

TABLE II: Topics and representative quotes of what participants liked about LiquidJava.

The last question of the study asked the participants if they would use LiquidJava in their projects, to which **all the participants answered affirmatively**. However, in the final suggestions, one participant declared that they would only use it in critical parts of the project, and two other participants stated that they are not currently using Java in any project but would like to have similar verifications in other programming languages.

D. Study Conclusions

The major takeaways of the study can be summarized in the following points.

- **Interpretation of refinements (RQ1)** – Refinements in variables and methods are easy to understand without a prior explanation, and even though the features to model classes are not very intuitive at first, they are easy to understand with few resources, in this case the 4-minute video and the website with examples.
- **Detecting and fixing implementation errors in Java and in LiquidJava (RQ2)** – From Task 1 (Section I-A) and 3 (Section I-A), it is possible to assess that LiquidJava helped developers find the error present in the code. For fixing the bugs, LiquidJava helped in all but one case, since developers focused on silencing compiler errors disregarding the reasoning behind the changes applied. As

Topic	# References	Representative Quotes
Nothing	8	"nothing, it is quite straightforward to be used";
Syntax	6	"[in the state refinement] repetition of this", "the usage of _ for the return value";
Plugin features	4	"Improve the usability of the plugin -remove double quotes; - use auto-complete inside the refinements", "not being able to correct multiple files at the same time";
Not Intuitive	3	"Hard to understand without access to documentation, mainly DFA protocols";
Error Messages	3	"Some error messages are not straight to the point";
Verbose	3	"Makes Java more verbose";
Other	2	"The state transition is a bit harder to get but paired with the given documentation is fine.", "The installation process was not very user friendly".

TABLE III: Topics and quotes of what participants disliked about LiquidJava.

for the time taken in each exercise, participants generally finished the LiquidJava exercises faster. The one outlier to this was the Fibonacci exercise, probably because it is a traditional algorithm that developers are used to seeing in its plain form and not with the annotations.

- **Best result for detecting and fixing error in LiquidJava** – From all exercises, the one that had most improvements while using LiquidJava was the `Socket` client, where no participant was able to find the error in plain Java but all participants were able to detect and fix the error using LiquidJava. This result might show that LiquidJava is more useful when applied to lesser-known classes and protocols than to mainstream classes or simple code.
- **Annotate Programs (RQ3)** – All participants were able to add refinements to variables and to model class protocols. Furthermore, 80% were able to add refinements to methods correctly (the remaining silenced the errors), and 43% were able to introduce refinements in class fields (the remaining participants left the answer blank). Participants also classified the annotation process as *Easy* or *Very Easy*. Thus, we can conclude that refinements are easy to add to the code to model the desired behaviour of programs.
- **Compiler Correct answers** – Having partial specifications on the code lead to some participants changing the code to respect the specification, and therefore pass the verification, disregarding the intent of the program. In the `Sum` exercise and the annotation of methods, participants gave answers correct according to the refinements but incorrect according to the informal documentation, producing the *Compiler Correct* category of answer classification. This result might show that partial specifications can mislead developers if they do not capture the meaning of the specification and the expected behavior of the program.
- **Would participants use LiquidJava (RQ4)**– The fact that we got all the desired participants to the study, already helps answering this question, since it shows that Java developers are open to participate in studies to discover new approaches to improve their code quality. However, the last question of the study is the one that gives us confidence that participants are open to using LiquidJava since we asked them if they would use it in their projects, and all participants answered affirmatively. Therefore, we are confident that participants find LiquidJava accessible for its gains and are ready to use new useful verification tools.

E. Threats to Validity

This study shares threats to validity with other empirical studies (e.g., Glacier[12]). The first threat is the limited number of participants, and the fact that they may not represent the population of Java developers. Their occupation in Table I identifies the population for which this study could be generalizable. The exercises used for the tasks may also not be demonstrative of real-world tasks since they are small and simple to allow the sessions to be under 1h30min. However, these tasks were designed based on problems that also occur in larger projects. The study sessions themselves may not represent the development environment that developers are used to, and they might have less interest in fulfilling all tasks correctly. However, the environment was the same for Java and LiquidJava, and while VSCode is not on par with other IDEs, it has the necessary features for the small tasks presented.

There is also a potential learning effect between Tasks 1 and 3. This threat was addressed by using different problems, as described in the previous section.

REFERENCES

[1] A. J. Ko, T. D. LaToza, and M. M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empir. Softw. Eng.*, vol. 20, no. 1, pp. 110–141, 2015.

[2] R. P. L. Buse, C. Sadowski, and W. Weimer, "Benefits and barriers of user evaluation in software engineering research," in *26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, C. V. Lopes and K. Fisher, Eds. ACM, 2011, pp. 643–656.

[3] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," *Empir. Softw. Eng.*, vol. 19, no. 5, pp. 1335–1382, 2014.

[4] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik, "An empirical study of the influence of static type systems on the usability of undocumented software," in *27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, G. T. Leavens and M. B. Dwyer, Eds. ACM, 2012, pp. 683–702.

[5] B. Ellis, J. Stylos, and B. A. Myers, "The factory pattern in API design: A usability evaluation," in *29th International Conference on Software Engineering (ICSE*

2007), Minneapolis, MN, USA, May 20-26, 2007. IEEE Computer Society, 2007, pp. 302–312.

[6] J. Stylos and B. A. Myers, "The implications of method placement on API learnability," in *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, M. J. Harrold and G. C. Murphy, Eds. ACM, 2008, pp. 105–112.

[7] P. M. Uesbeck, A. Stefik, S. Hanenberg, J. Pedersen, and P. Daleiden, "An empirical study on the impact of C++ lambdas and programmer experience," in *38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 760–771.

[8] M. Coblenz, M. Mazurek, and M. Hicks, "Does the bronze garbage collector make rust easier to use? a controlled experiment," *arXiv preprint arXiv:2110.01098*, 2021.

[9] M. J. Coblenz, G. Kambhatla, P. Koronkevich, J. L. Wise, C. Barnaby, J. Sunshine, J. Aldrich, and B. A. Myers, "PLIERS: A process that integrates user-centered methods into programming language design," *ACM Trans. Comput. Hum. Interact.*, vol. 28, no. 4, pp. 28:1–28:53, 2021.

[10] R. Jhala and N. Vazou, "Refinement types: A tutorial," *Found. Trends Program. Lang.*, vol. 6, no. 3-4, pp. 159–317, 2021.

[11] J. Saldaña, *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2009.

[12] M. J. Coblenz, W. Nelson, J. Aldrich, B. A. Myers, and J. Sunshine, "Glacier: transitive class immutability for java," in *39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 496–506.