Pre-Print: Accepted at ICSE 2023

# Usability-Oriented Design of Liquid Types for Java

Catarina Gamboa*†, Paulo Santos*†, Christopher Steven Timperley* and Alcides Fonseca†
* School of Computer Science, Carnegie Mellon University, USA
† LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal
Email: *{cgamboa, pacsantos, ctimperl} @andrew.cmu.edu, †amfonseca@fc.ul.pt

***Abstract*—Developers want to detect bugs as early in the development lifecycle as possible, as the effort and cost to fix them increases with the incremental development of features. Ultimately, bugs that are only found in production can have catastrophic consequences.**

**Type systems are effective at detecting many classes of bugs during development, often providing immediate feedback both at compile-time and while typing due to editor integration. Unfortunately, more powerful static and dynamic analysis tools do not have the same success due to providing false positives, not being immediate or not being integrated into the language.**

**Liquid Types extend the language type system with predicates, augmenting the classes of bugs that the compiler or IDE can catch compared to the simpler type systems available in mainstream programming languages. However, previous implementations of Liquid Types have not used human-centered methods for designing or evaluating their extensions.Therefore, this paper investigates how Liquid Types can be integrated into a mainstream programming language, Java, by proposing a new design that aims to lower the barriers to entry and adapts to problems that Java developers commonly encounter at runtime. Following a participatory design methodology, we conducted a developer survey to design the syntax of LiquidJava, our prototype.**

**To evaluate if the added effort to write Liquid Types in Java would convince users to adopt them, we conducted a user study with 30 Java developers. The results show that LiquidJava helped users detect and fix more bugs, and that Liquid Types are easy to interpret and learn with few resources. At the end of the study, all users reported interest in adopting LiquidJava for their projects.**

***Index Terms*—Usability, Java, Refinement Types, Liquid Types**

## I. Introduction

Software quality is a major concern throughout software development [1]. Given the increased costs of finding and addressing bugs later in the development lifecycle, developers and organizations aim to identify issues as early as possible when they are cheaper and easier to address (“shifting left” [2]).

Strong type systems are present in many modern programming languages (e.g., Java, C#, Haskell), allowing developers to specify the expected type of operations and verify, at compile-time, if those types are respected. Code editors typically integrate this verification to provide developers with immediate feedback about the incorrect use of variables and values, simplifying the debugging process by helping developers to identify the errors more easily.

```
1 @Refinement("r >= 0 && r <= 255")
2 int r = 90; // Correct
3 r = 200 + 60; /* Correct in Java, but raises a Refinement Type Error:
4                  Type expected: (r >= 0 && r <= 255);
5                  Refinement found: (r == 200 + 60) */
```

Listing 1: Variable refinement and verification in LiquidJava.

However, the type systems of these programming languages restrict the expressiveness of domain-specific information that can be introduced in a program limiting the class of errors caught at compile time. Refinement Types are more expressive than popular type systems [3], as they extend the language with predicates that restrict the allowed values in variables and methods. For example, Listing 1 shows a simple refinement on the variable `r`, restricting the allowed values to the range of 0 and 255. Liquid Types [4] represent the decidable subset of refinements that allow automatic verification by SMT solvers.

Refinement Types have been used to detect simple division by zero errors and out-of-bounds array access bugs [5], as well as more complex security issues [6] and protocol violations [7]. Despite these advantages, Refinement and Liquid Types have not become mainstream, raising questions about their usability and usefulness.

Previous works on Liquid Types have not included a human-focused design or evaluation. Moreover, the initial implementation of Liquid Types targeted ML [8], and the most mature implementation (LiquidHaskell [9]) targets Haskell, both of which are functional programming languages with relatively low adoption in industry compared to languages such as Python, C, Java, and Javascript. Although Refinement Types have been added to C [10]and Javascript [11], these extensions have not been mature enough to be adopted in the industry.

In this work, we follow a user-oriented approach to design and evaluate Liquid Types for one of the most popular programming languages in the world, Java. Our approach is comprised of two parts: First, we design a Liquid Types extension for Java following a participatory design methodology, where users guide the decision on how to express Liquid Types. Then, we conduct a user study to evaluate if using this extension is more beneficial when compared to using plain Java. Therefore, our contributions are:

- The design of LiquidJava, an extension of Java that

This work is supported by the Fundação para a Ciência e a Tecnologia (FCT) under LASIGE Research Unit, ref. (UIDB/00408/2020) and (UIDP/00408/2020), the project DACOMICO ((PTDC/CCI-COM/2156/2021)), the CMU-Portugal project CAMELOT (POCI-01-0247-FEDER-045915), the RAP project (EXPL/CCI-COM/1306/2021), and AFRL (Award 19-PAF00747). The work is also co-financed by a Dual Degree Ph.D. Scholarship awarded by the Portuguese Foundation for Science and Technology through the Carnegie Mellon Portugal Program under the fellowship (SFRH/BD/151469/2021).

Any opinions, findings, or recommendations expressed are those of the authors and do not necessarily reflect those of the US Government.

supports Liquid Types, based on user feedback, and subsequent prototype implementation.
- A user study that evaluates the usability of LiquidJava, namely how understandable the code with refinements is, and whether it is more useful for developers to detect and fix bugs, when compared with plain Java.

*Data Availability:* The data supporting this paper is attached to the submission as supplementary material, and will be openly available if the paper is accepted. The online package includes the study guides, the data gathered during their execution and a virtual machine to simulate the required environment.

## II. LiquidJava Design

Designing interactive systems with the input of prospective users is a popular strategy in the field of HCI (Human-Computer Interaction) [12, 13]. Designers of new systems are encouraged to develop prototypes that aim to fulfill the users' needs and apply formative evaluations of the prototypes to include the users' feedback in their design solutions.

Programming languages are a form of human-computer interaction, and they should be designed to meet the developers' needs and expectations, having a clear focus on usability. While most of the effort in studying programming language interaction has been focused on learning (e.g., Hedy [14]), advances are being made in studying the interaction of experts. Coblenz et al. [15] discuss the difficulties of applying HCI methods to the design of languages and propose a process, PLIERS, for designing languages focused on the users. These principles were used in the development of two languages: Glacier [16], a language for immutability in Java, and Obsidian [17], a language to model blockchain protocols.

For LiquidJava, we started by defining usability requirements (Section II-A). Then, we applied a formative study with users to drive the choice of the syntax based on our proposed alternatives (Section II-B). Based on the results of the formative study, we added Liquid Types to Java, covering local variables, fields, method definitions, and method invocations.

### A. Requirements

To promote the usability of Liquid Types in Java, we identified we three requirements to guide the language design. These requirements were based on previous implementations of Refinement Types and popular characteristics of successful static verification techniques, such as the `@NonNull` annotation [18], as well as feedback from developers [19, 20]. The requirements are the following:

**R.1 Refinements must be optional:** Without refinements, a Java program must be successfully validated by the liquid type-checker, allowing specifications to be introduced after the implementation, including to pre-existing codebases. This grants the developer the possibility of incrementally building the program's specification.

**R.2 Refinements need to be expressive:** Refinements need to cover a wide range of specifications while being written with a syntax similar to Java, to enable developers to write specifications without learning a completely new language.

**R.3 Refinement type-checking should be decidable:** The type-checking process should be decidable to prevent unnecessary overhead to the compilation process, provide interactive feedback to developers as they code, and reduce false positives — all concerns mentioned in previous static analysis tools [19, 20]. To this end, the refinements language is restricted to Liquid Types, which are verifiable by SMT solvers. Thus, the predicates are restricted to decidable logics using quantifier-free linear integer arithmetic with uninterpreted functions and accepting only SMT-decidable operations.

The design solution to fulfill the requirements can be split into two main topics: the design of the refinements language and how it is incorporated into Java programs; and the design of the verification system to validate these programs.

***Refinements Design:*** For the design of the refinements, we first had to decide how to introduce the predicates in the Java source code while following the aforementioned requirements, specifically regarding the need for a flexible and understandable language. Moreover, in previous refined typed languages [11, 3], variable and method declarations have been the main target of refinement annotations, however, classes are also a core concept in Java and a desirable target for refinements.

Considering this, we decided to encode the refinements as Java Annotations in the source code since annotations are optional, support all necessary targets, and have been commonly used in Java since their introduction in JDK 1.5. This decision fulfills **R.1** since these annotations are always optional as the Java type checker does not verify them. Additionally, annotations are attached to target code elements which include local variables, parameters, methods, fields and classes, allowing us to introduce refinements to all the desired code elements considered in this study.

Using annotations to express restrictions on variables has become more popular over the years, with `@NonNull` and `@NonEmpty` being present in many Android and enterprise applications, as well as in verification tools for Java (e.g., Jakarta Bean Validation [21], Checker Framework [22]). The popularity of annotations gives us some confidence that their usage within LiquidJava will not constitute a barrier to the system's adoption. Thus, a new `@Refinement` annotation was created to express refinements, and other annotations were also created as syntactic sugar (e.g., `@StateRefinement`).

However, annotations by themselves are not flexible or very expressive. Therefore, the refinements are written as strings inside the annotations and follow a custom language developed to be as similar to Java as possible, addressing **R.2** and reducing the obstacles in writing specifications. To improve the understandability of the language, we conducted an online survey to assess the best syntax for the features of the language, while keeping it verifiable by SMT Solvers (respecting **R.3**). The syntax survey is detailed later in this paper in Section II-B.

***System Design:*** To verify a Java program annotated with refinements, a LiquidJava verifier either proves that all refinements are respected throughout the program, or shows that there is a violation of the specification (design in Figure 1).

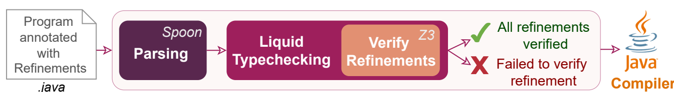


Fig. 1: Pipeline of the LiquidJava system.

The verifier receives as input a Java program annotated with refinements and performs a static verification before the execution of the program. The first step of the system involves parsing the input to an abstract syntax tree (AST) used for the verification, using the Spoon [23] framework. Taking the AST representation of the program, the liquid type checker traverses the AST and checks all expressions against their expected type through subtyping relationships. These relationships are then discharged to an SMT Solver, which proves their satisfiability. The details of the verification process are out of the scope of this paper but are available in online resources. [1]

### B. Syntax Survey

To design the refinement syntax we decided to get feedback from Java developers instead of following a personal preference, which is the most common approach in designing new languages. Therefore, to assess the best syntax for the language of refinements, we created and shared an online survey with possible syntaxes for different LiquidJava features, some based on other implementations of Refinement Types and others based on the Java language. These features include type refinements of variables and methods and the use of predicate aliases and anonymous variables. To minimize the time needed to complete the survey and have more participation, we proposed two to three syntax options for each language feature instead of asking the participants for their syntax proposal. Specifically, we asked the participants to evaluate each of the syntax options we designed with one of three preference levels: *Not Acceptable*, *Acceptable*, and *Preferable*.

The study was sent to academic students and researchers, and Java developers in the industry. As a result, we obtained 50 answers from participants acquainted with Java. The survey

[1] *removed for anonymity purposes*

**Refinements in Methods**

In this section we present syntax examples for refinements in methods, which includes refinements for the parameters and the return value.
These refinements express the following conditions:
- grade, the first parameter, is an int greater than or equal to 0;
- scale, the second parameter, is a positive int;
- the return value must be an int between 0 and 100.
Analyse each of the examples below.

**A**
```
@Refinement("\\v >= 0 && \\v <= 100")
public static int percentageFromGrade (@Refinement("grade >= 0") int grade,
                                       @Refinement("scale > 0")  int scale)
```

**B**
```
@Refinement("{grade >= 0} -> {scale > 0} -> {\\v >= 0 && \\v <= 100}")
public static int percentageFromGrade (int grade, int scale)
```

Evaluate your preference on each of the above syntaxes.

| | Not acceptable | Acceptable | Preferable |
|---|---|---|---|
| A | ○ | ○ | ○ |
| B | ○ | ○ | ○ |

Fig. 2: Question on preferable syntax for refinements in parameters and return value of methods.

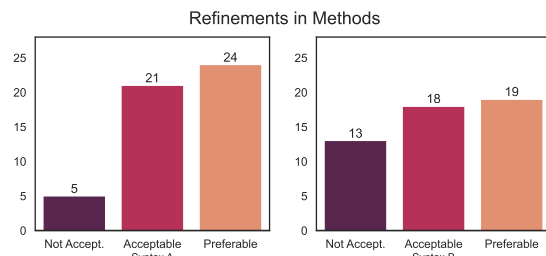


Fig. 3: Preferences on the Syntax for Methods' Refinements.

started with questioning the participants' background and briefly explaining the concept of Refinement Types, before questioning the preferred syntax options for LiquidJava features. The background answers of participants show that more than half of the participants (a total of 56%) are *Not Familiar* at all with Refinement Types and that only 2% consider themselves *Very Familiar* with the concept. These percentages show that most developers are not familiar with Refinement Types, highlighting that they are not widely spread.

Figure 2 represents one of the survey questions, where participants could read the description of the refinements for each of the parameters and the return type of the method `percentageFromGrade`, and analyse each of the proposed syntaxes to then evaluate with their preference. These syntax options follow two different designs. The first option attaches each refinement to the type of variable that it is refining. Therefore, the parameters have the refinements just before their basic type, and the return refinement is above the method, before the return type. The latter option has a syntax inspired by the type signatures used in functional languages, such as Haskell. Therefore, the parameters and return refinements are written in the same line, split by the `->` symbol, with a specific order starting with the parameters and finishing with the method's return type.

The participants evaluated their preference to both proposed syntaxes, and Figure 3 shows the gathered results. The first option had more *Preferable* answers and fewer *Not Acceptable* answers, which made us choose the first syntax for this feature.

The remaining features had similar syntax questions and preference evaluation (available in additional resources). We concluded that throughout the survey, participants preferred syntaxes with a flavor similar to Java, discarding syntaxes closer to previous implementations of Refinement Types that had a style closer to the syntax of functional languages.

### C. LiquidJava Features

LiquidJava supports the refinement of variables and fields whose assignments must always respect the introduced predicates. Furthermore, it is possible to add refinements to return values and parameters of methods, allowing the verification of the return value against the expected type and the verification of the method's invocations with the expected arguments.

Listing 2 depicts an example of these features. The `inRange` method receives two parameters, `a` and `b` where the first parameter has no annotation, staying with the default refinement of `true`. In contrast, the second parameter has a refinement

```
1 @Refinement ("a <= _ && _ <= b")
2 public static int inRange(int a, @Refinement("b > a") int b) {
3     return a + 1;
4 }
5 @Refinement ("x >= 10") int x;
6 x = inRange(10, 20 - 5); // Correct invocation and assignment
7 inRange(10, 2); /* Refinement Type Error
8                     Type expected: (b > a);
9                     Refinement found: (b == 2) && (a == 10) */
```

Listing 2: Refinement annotation of a method and a variable and verification of related operations.

dependent on the first,informing that its should be greater than the value of the first parameter. Above the method's signature (line 1) is the return value refinement, which informs that the return value (represented by the _ ) is expected to be within the range of both parameters. All return values are also checked against the declared return refinement.

In line 5, a variable `x` is declared with a refinement that restricts its value to integers greater or equal to 10. In the assignment that follows, the method invocation must respect the parameters refinements, and the return of the invocation should respect the variable refinement. In this case, both verifications hold, validating the operation. However, the second invocation produces an error since the second argument is not greater than the first one, and an error message depicting that information is shown to the user.

In LiquidJava, refinements can also be used to model the state of class objects. Although previous extensions of Refinement Types did not focus on object modelling, classes are considered a fundamental programming element of the Java language [24] and, therefore, we model them using type states [25].

Although classes themselves do not have a specific value that can be refined, they can have methods that produce changes to the internal state of the objects. Therefore, we can refine the object state by restricting in which state the object has to be when a method is invoked and what is the state of the object after the execution of the method. Both predicates are encoded in methods with the annotation `@StateRefinement(from="predicate", to="predicate")`, where the `from` argument indicates the object state from which the method can be called, and the `to` argument contains the resultant object state.

A method can have different combinations of source and destination states. These combinations are encoded as multiple `@StateRefinement` annotations above the method. To model the class state, one can create ghost functions that represent class properties (e.g., the size of a list) or define a set of states that the class objects can have. The states and the ghost properties are invoked inside the predicates as functions that take the current object as an argument using the `this` keyword. Moreover, if it is necessary to refer to the object's previous state, it is possible to use the `old` keyword with the current object, resulting in the expression `old(this)`.

Using `@StateRefinement` allows users to define protocols that a class must follow by encoding a finite state machine within the refinements. Java classes usually define protocols that the client programs must follow. However, these protocols are primarily defined through informal documentation using natural language (e.g., Javadoc). Therefore, most protocols are not enforced during code development, leading to runtime exceptions. By encoding the protocol definitions in LiquidJava, developers get feedback on the correct use of classes before the program execution, allowing them to, for example, use external libraries with more confidence.

```
1  @ExternalRefinementsFor("java.net.Socket")
2  @StateSet({"unconnected", "bound", "connected", "closed"})
3  public interface SocketRefinements {
4      @StateRefinement(to="unconnected(this)")
5      public void Socket();
6
7      @StateRefinement(from="unconnected(this)", to="bound(this)")
8      public void bind(SocketAddress add);
9
10     @StateRefinement(from="bound(this)", to="connected(this)")
11     public void connect(SocketAddress add, int timeout);
12
13     @StateRefinement(from="connected(this)")
14     public void sendUrgentData(int n);
15
16     @StateRefinement(from= "!closed(this)", to="closed(this)")
17     public void close();
18 }
```

Listing 3: Socket class object state refinement.

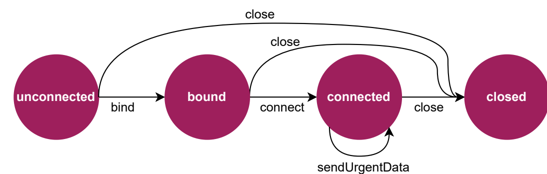


Fig. 4: DFA representing the Socket class states and transitions.

Listing 3 shows the annotation of the native library `java.net.Socket`.[2] The specification models the implicit state machine semantics (shown in Figure 4), documented only in natural language. The `@StateSet` annotation describes all possible states, and each method describes the states in which it is available (`from`) and the state in which the object is after the method execution (`to`). Using this specification, a program that invokes `sendUrgentData` without first invoking `connect` will not be accepted.

### D. IDE Integration

To enhance the usability of LiquidJava, we developed an IDE plugin to provide immediate liquid typechecking feedback, while developers are programming. It also provides localization of bugs by underlining the relevant incorrect code. Figure 5 shows how errors are reported. While we support it as a plugin for VSCode, it is available as a Language Server [26], which can be easily integrated in other IDEs.

## III. User Study

Evaluating software engineering tools with user studies is not a recent practice, but it is still scarce. Mainly because

[2]https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html

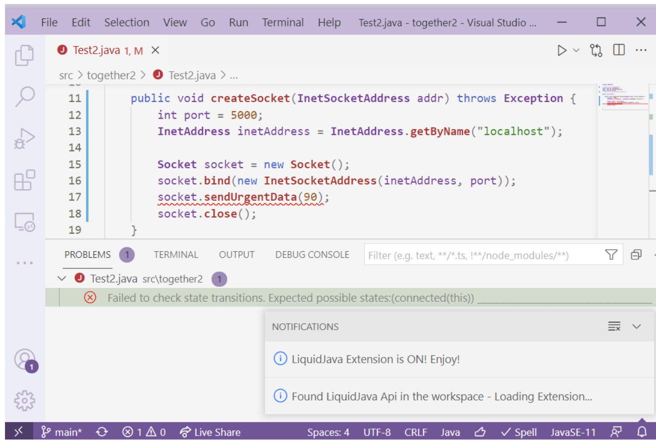


Fig. 5: IDE Plugin reporting an error in the incorrect usage of the Socket class.

most researchers find these experiments too difficult to design and conduct, have difficulties recruiting participants, and believe that the results might be inconclusive [27]. However, researchers who conducted user studies agree that these evaluations provide useful insights that outweigh the study costs while increasing the impact of the work [28].

In previous research, authors have employed user studies to understand how programmers write code [29, 30] and compare different programming language designs. For example, comparing the benefits of using static versus dynamic type systems for maintainability and undocumented software, [31, 32] analysing the usability and learnability of API aspects [33, 34], and advanced language features such as lambdas [35] and garbage collection [36]. 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) [15]. 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, having one option as a control condition, and assigning each participant a random design to fulfill the tasks.

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

**Q1** Are refinements easy to understand?
**Q2** Is it easier and faster to find implementation errors using LiquidJava than with plain Java?
**Q3** Is it hard to annotate a program with refinements?
**Q4** Are developers open to using LiquidJava in their projects?

Given these research questions, we planned the study with tasks to assess the usability of LiquidJava, and compare its benefits against Java. This section depicts the study configuration (Section III-A), the participants' background (Section III-B), and the detailed tasks with results (Section III-C) and their discussion (Section III-D). At the end, the threats to validity of the study are presented (Section III-E).

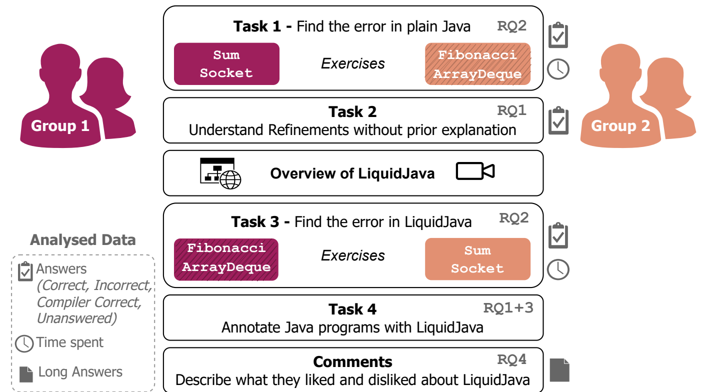


Fig. 6: Configuration of the user study.

## A. *Study Configuration*

The study was designed to introduce participants to LiquidJava and answer the aforementioned research questions.

The study was conducted individually with each participant and everyone 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.

Figure 6 schematizes the configuration of the study. When starting, each participant was randomly assigned to one group. All participants within a group followed 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** – Participants were asked to detect implementation errors in Java code, and provide a possible correction. The implementation errors make the execution incorrect against the informal documentation presented in the method's Javadoc. The participants could use all the resources within the IDE environment including read documentation, change the code, and even execute the code.
- **Task 2: Interpreting refinements without prior explanation** – Although participants must be familiar with Java, they are not required to be familiar with refinements. Thus, all participants had their first contact with LiquidJava in this task. 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 use without a prior explanation. Thus, this study section is related to **Q1**, 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

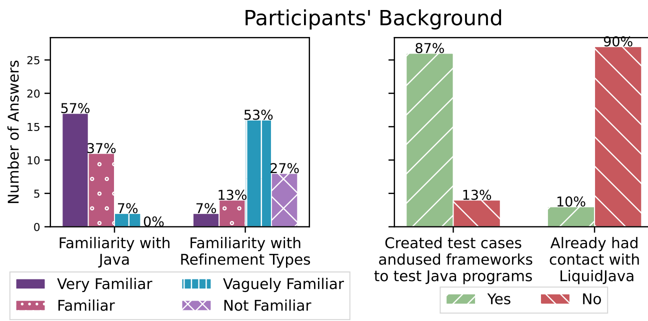


Fig. 7: Answers to participants' background.

resources were available for the participants to use in the remaining of the study. These materials 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 exercises. 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 **Q2**.
- **Task 4: Annotate Java programs with LiquidJava** – Participants were presented with three Java programs and were asked to annotated them with LiquidJava specifications. This task aims to answer **Q3** 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 **Q1**.
- **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, **Q4**.

The study sessions were all conducted through the Zoom video platform, and participants used their own environments to complete the study tasks. To ensure that these environments fulfilled the requirements to complete all tasks, participants installed Visual Studio Code with JDK11 and the Language Support for Java(TM) by Red Hat extension. During the study, participants had access to the GitHub repository with the study files, a webpage with information on LiquidJava and to all features of the IDE provided by the plugins.

### *B. Background of Participants*

We designed the study for 30 participants familiar with Java and recruited them through social media channels, such as Twitter and Instagram, and through direct contact via email.

Figure 7 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 *Not Familiar* with Refinement Types which shows that despite their utility, Refinement Types are not widely known and used.

The three 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. Specifically, we presented three code snippets with LiquidJava refinements with an increasing difficulty level (as showed in Listing 4) 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 was restricted by the limits of Earth's surface temperature. The second task had participants implement correct and incorrect invocations of `function1`, where the second parameter depends on the first. The last task presented a class protocol with three states and methods that model the object state. Here, 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 with the three states *sX, sY* and *sZ* as *Show Items, Item Selected* and *Paid*, respectively. The anonymization 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 8 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'

```
1  // 1 − Variable Refinement
2  @Refinement(”−25 <= x && x <= 45”) int x;
3
4  // 2 − Function/Method Refinement
5  @Refinement(”_ >= 0”)
6  public static double function1(@Refinement(”a >= 0”) double a,
         @Refinement(”b >= a”) double b)
7    { return (a + b)/2; }
8
9  // 3 − Class Protocol Refinement
10 @StateSet({”sX”, ”sY”, ”sZ”})
11 public class MyObj {
12     @StateRefinement(to=”sY(this)”)
13     public MyObj() {}
14
15     @StateRefinement(from=”sY(this)”, to=”sX(this)”)
16     public void select(int number) {}
17
18     @StateRefinement(from=”sX(this)”, to=”sZ(this)”)
19     public void pay(int account) {}
20
21     @StateRefinement(from=”sY(this)”, to=”sX(this)”)
22     @StateRefinement(from=”sZ(this)”, to=”sX(this)”)
23     public void show() {}
24 }
```

Listing 4: Exercises to interpret refinements.

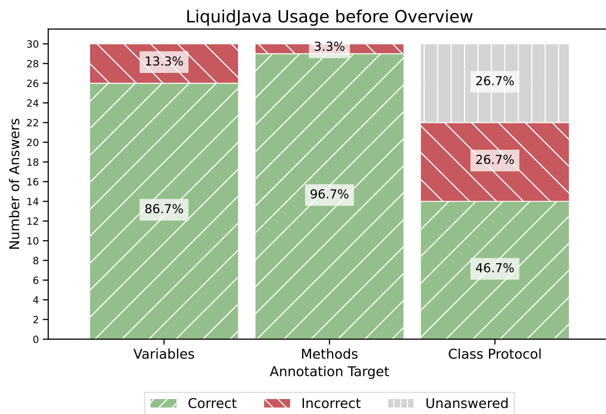


Fig. 8: Answers on interpreting LiquidJava refinements.

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, hinting 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 seem to be 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**:* To compare the effort of detecting and fixing bugs while using Java and LiquidJava, participants had similar exercises in the first and third tasks. The four exercises used in the study are versions of *Fibonacci*, a *Sum* of all numbers until a given input, and invocations to the *ArrayDeque* and *Socket* libraries. The exercises are simple, with

| Exercise | Average Time Java | Average Time LiquidJava |
|---|---|---|
| **Fibonacci** | 2 mins 52 secs | 3 mins 22 secs |
| **Sum** | 4 mins 30 secs | 3 mins 32 secs |
| **ArrayDeque** | 3 mins 41 secs | 2 mins 56 secs |
| **Socket** | 5 mins 35 secs | 4 mins 42 secs |

TABLE II: Average times to complete the exercises.

a maximum of 15 lines of meaningful code (e.g., disregarding empty lines), to ensure that the participants can reason about the programs with their Java background and the introduced information about Liquid Types. Since all participants are already used to working in Java, they all start with the plain Java exercises, and, after a brief introduction to LiquidJava, they move to detecting bugs with the custom plugin activated.

Each exercise has two versions, a plain Java and a LiquidJava version with the same implementation errors that allow us to compare the number of participants that found and fixed the bug and the time taken by the participants to complete the exercises. A group of participants started with the 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.

For each exercise we gathered the time spent, and the written answers for the incorrect line(s) and the proposed fixes. The answers were then evaluated into one of four possible categories: *Correct*, *Incorrect*, *Unanswered*, and *Compiler Correct*. The last category represents the answers that silenced the compiler error despite not being utterly correct according to what the exercise asked.

The results on finding and fixing bugs in the four exercises are displayed in Figure 9. Moreover, the figure includes two plots on the distribution of time taken by participants in each exercises, and the average times for each exercise are in Table II. The code given to participants and their answers is available in supplemental material.

All exercises have in common that 100% of participants found the error location while using LiquidJava, which was expected since the plugin underlines the error found. However, the plugin does not inform about a possible fix and in most cases participants outperform in the LiquidJava version for fixing the error. Additionally, in all but one exercise (*Fibonacci*), participants were faster on average in the LiquidJava versions.

Both ***Fibonacci*** and ***Sum*** exercises are implemented using recursion with an error in the base case. Inspecting each exercise individually, we can see that in the ***Fibonacci*** exercise all participants fixed the error using LiquidJava, while only 66.7% fixed the error in the plain Java version. Regarding the

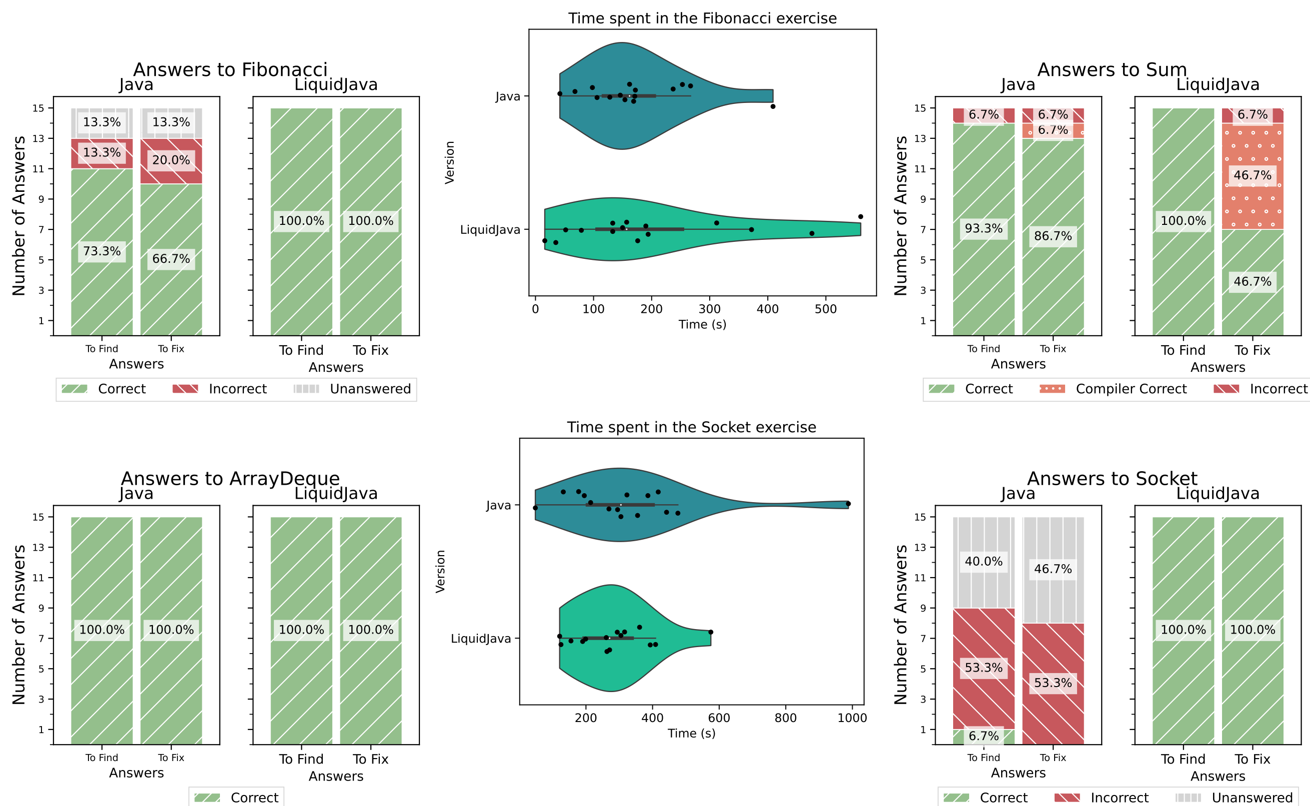

Fig. 9: Results on finding and fixing errors using Java and LiquidJava.

time spent on both versions, on average participants were faster in the plain Java exercise when compared to the LiquidJava. This result suggests that participants might be already used to Fibonacci's plain implementation given the popularity of the algorithm in introductory programming classes, and when the new refinements were added participants spent more time understanding the different sections of code.

For the ***Sum*** exercise, 7 participants correctly fixed the program, and other 7 provided a *Compiler Correct* answer. These last answers silenced the compiler error since they complied with the liquid specification but did not comply with the informal specification of the method. After reviewing these answers, it is possible to see that 6 out of the 7 participants fixed the error with the same code as they used in the first exercise in plain Java (*Fibonacci*), and while it was the correct fix before, it was not the required fixed in this exercise. This line of thought might indicate that participants were biased by the previous sections of the study and opted for the same answer as they used in the beginning.

In the ***ArrayDeque*** exercise all participants fixed the error in both versions by fixing the order of invocations of popular methods to add, remove, and retrieve elements from an ArrayDeque object depending on its number of elements.

The ***Socket*** exercise is the one with the largest difference in the number of correct answers while using Java and LiquidJava. In the Java version none of the participants were able to fix the error, giving a wrong fix or leaving the answer blank. However, while using LiquidJava every participant understood that there was a missing invocation in the order of methods to make the invocations valid. In both cases the participants had access to the informal documentation of the library through the Java plugin active in the IDE. The time spent in this exercise shows that participants were faster by 52 seconds in LiquidJava, in average, with a higher rate of correct answers.

*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 10 and are analysed along with each exercise below.

The first and more straightforward exercise just included

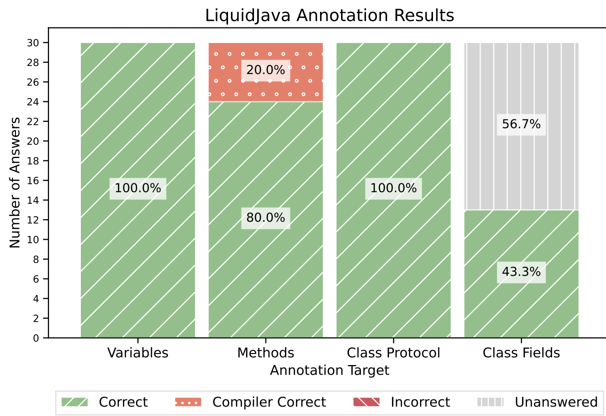


Fig. 10: Results of the annotations with LiquidJava.

a variable named `currentMonth` that should be restricted with the lower and upper bound of month values, resulting in a code similar to `@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. The second exercise presented the method `inRange` 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 11]", 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.

All participants correctly modeled the class by declaring the starting states and the state transitions of each method. This percentage constitutes a significant increase in the understanding of class protocols 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 participants annotated the class fields. The remaining ones did not add any refinement to fields, leaving an incorrect assignment inside a method's body. There might be several reasons for this occurrence. One might be that they misinterpreted the exercise, not realising the need for these annotations. Another possible explanation is that participants did not consider important to add these 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

```
1  public class TrafficLight {
2    private int r;
3    private int g;
4    private int b;
5
6    public TrafficLight() { r = 76; g = 187; b = 23; }
7    public void transitionToGreen() { r = 76; g = 187; b = 23; }
8    public void transitionToAmber() { r = 255; g = 120; b = 0; }
9    public void transitionToRed() { r = 230; g = 0; b = −1; }
10 }
11
12 //Correct Test − different file
13 TrafficLight tl = new TrafficLight();
14 tl.transitionToAmber(); tl.transitionToRed();
15 tl.transitionToGreen(); tl.transitionToAmber();
16
17 //Incorrect Test − different file
18 TrafficLight tl = new TrafficLight();
19 tl.transitionToAmber();
20 tl.transitionToRed();
21 tl.transitionToAmber();
22 tl.transitionToGreen();
```

Fig. 11: Program to annotate with the protocol showed in the diagram, and with the limit ranges on fields.

that adding the annotations was *Very Easy*, while the remaining 40% considered the task *Easy*, leading us to conclude that 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?

Figure 12 summarizes the answers to the first two questions with the codes obtained through a qualitative coding [37] approach and quotes of participants. 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. As for the disliked topics, only 26 participants answered the question, from which 8 explicitly mentioned there was nothing they did not like. The remaining negative aspects are related to some syntax options and plugin features to improve.

The last question of the study asked if participants 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 (Q1)** – Refinements in variables and methods are easy to understand without a prior explanation, and even though the features to model

Fig. 12: Comments on what participants liked and disliked.

classes are not very intuitive at first, they are easy to understand with few resources.

- **Detecting and fixing implementation errors in Java and in LiquidJava (Q2)** – From Task 1 (Section III-A) and 3 (Section III-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 for the time taken in each exercise, participants generally finished the LiquidJava exercises faster.
- **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 fix 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 (Q3)** – All participants were able to add refinements to variables and to model class protocols. 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 led to some participants changing the code to respect the specification, disregarding the program's intent, silencing the compiler error, and passing the verification without fully fixing the program. For example, in the *Sum* exercise and the annotation of methods, participants gave answers that were correct according to the refinements but incorrect according to the informal documentation, producing the *Compiler Correct* category of classification. This result might show that partial specifications can mislead developers if they do not capture the specification's meaning and the program's expected behaviour.
- **Would participants use LiquidJava (Q4)**– Achieving the desired number of participants to the study shows that developers are open to participate in studies to discover new approaches to improve their code quality. Moreover, the affirmative answers on developers' willingness to use LiquidJava give us confidence that participants are open to using this approach. 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 [16]). 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.

## IV. Related Work

Freeman and Pfenning [8], introduced the concept of Refinement Types inside ML, a strongly-typed functional programming language, allowing the detection of more errors at compile time. Liquid Types (*Logically Qualified Data Types*) [4], proposed in 2008, represent a subset of Refinement Types that use predicates over a decidable logic to ensure the inference and type checking decidability. Liquid Types have a more practical implementation in LiquidHaskell [38], featuring type aliases to improve predicates' brevity and readability.

Liquid Types have been used in more imperative settings. CSolve [10] uses refinement type checking to verify heap layout and pointer usage within C programs, using macros to express the liquid refinements. Kazerounian et al.[39] introduced refinements in Ruby, an object-oriented and dynamic scripting language. Liquid Types were also added to a typed extension of Javascript, DJS [40], that handles extensible objects, prototype inheritance, flow-and sensitive strong updates on mutable variables, which LiquidJava also supports.

Refinements were also introduced in Scala [41] and Typescript [42] as class invariants on immutable class fields, which allow reflecting immutable fields in the specification of mutable fields. In Java, Stein et al. [43] applied a specific set of Refinement Types to stream-based processing but used a fixed type hierarchy, limiting the expressiveness of the refinements without using logical predicates to qualify the types.

However, none of these previous works designed the refinements with users' input or applied any user study to understand the usability of the refinements and the behaviour of developers when using these light software verification techniques. In that sense, this is the first study that we are aware of that aims to study the usability of Refinement Types.

Despite being scarce, there are works that integrate usability testing into the design of interactive verification techniques. For example, authors of KeY [44], a project that integrates verification and analysis within the Java language, developed a questionnaire to evaluate the cognitive dimensions of the tool and improved it based on the study conclusions [45]. In later work [46], the authors conducted focus groups to understand the users' typical interaction with theorem provers. The results obtained were used to implement changes in a prototype, which was then evaluated through an exploratory user study to assess if the tool improvements were helpful. In another work, Kadoda [47] aimed to compare the differences between the designers' and the users' perspectives of using theorem proving assistants (e.g., HOL [48]) using questionnaires.

## V. Conclusion and Future work

Our participatory design and evaluation of LiquidJava allowed us to conclude that refinements can be added to a mainstream, imperative and object-oriented language. The formative study for the language design identified that Java developers prefer having refinements closer to the code units they refer to (e.g., annotations closer to method's parameters), which is not the popular approach in other embedded verification languages (e.g., JML [49]).

In the summative user study, we concluded that refinements are intuitive, with more than 86% of the participants being able to use refinements in variables and methods without a prior introduction to Refinement Types. However, we found object state refinements harder to understand and use, as only 46% of participants used them correctly. Nevertheless, after a 4-minute video, 100% of the participants could correctly introduce the protocol specification and considered the task easy.

More importantly, LiquidJava allowed participants to detect and fix more bugs than plain Java, especially when classes have states not expressed in plain Java types. As a result, all participants showed interest in adopting LiquidJava, mostly due to the fast and early error reporting.

This study also identifies that this lightweight verification might give users a false sense of safety by assuming that if a program typechecks, it is correct. Because our approach provides only gradual verification, it is important to present users with the exact guarantees they can expect from a type-safe program. We plan to address this concern in future work.

## References


[1] P. Upadhyay, "The role of verification and validation in system development life cycle," *IOSR Journal of Computer Engineering*, vol. 5, no. 1, pp. 17–20, 2012.

[2] H. Wright, T. D. Winters, and T. Manshreck, *Software Engineering at Google*, 2020.

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

[4] P. M. Rondon, M. Kawaguchi, and R. Jhala, "Liquid types," in *Conference on Programming Language Design and Implementation*, R. Gupta and S. P. Amarasinghe, Eds. ACM, 2008, pp. 159–169.

[5] H. Xi and F. Pfenning, "Eliminating array bound checking through dependent types," in *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 1998, pp. 249–257.

[6] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, "Refinement types for secure implementations," in *21st IEEE Computer Security Foundations Symposium, CSF 2008*. IEEE Computer Society, 2008, pp. 17–32.

[7] N. Burnay, A. Lopes, and V. T. Vasconcelos, "Statically checking REST API consumers," in *Software Engineering and Formal Methods*, F. S. de Boer and A. Cerone, Eds., vol. 12310, 2020, pp. 265–283.

[8] T. S. Freeman and F. Pfenning, "Refinement types for ML," in *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, D. S. Wise, Ed. ACM, 1991, pp. 268–277.

[9] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for haskell," in *ACM SIGPLAN Notices*, vol. 49, no. 9, 2014, pp. 269–282.

[10] P. M. Rondon, A. Bakst, M. Kawaguchi, and R. Jhala, "Csolve: Verifying C with liquid types," in *Computer Aided Verification - 24th International Conference*, ser. Lecture Notes in Computer Science, P. Madhusudan and S. A. Seshia, Eds., vol. 7358. Springer, 2012, pp. 744–750.

[11] Chugh R., Herman D., Jhala R., "Dependent Types for JavaScript," http://goto.ucsd.edu/ ravi/research/oopsla12-djs.pdf, 2012.

[12] A. Dix, J. E. Finlay, G. D. Abowd, and R. Beale, *Human-Computer Interaction (3rd Edition)*. Prentice-Hall, Inc., 2003.

[13] M. J. Fonseca, P. Campos, and D. Gonçalves, *Introdução ao Design de Interfaces*, 10 2012.

[14] F. Hermans, "Hedy: A gradual language for programming education," in *ICER 2020: International Computing Education Research Conference*, A. V. Robins, A. Moskal, A. J. Ko, and R. McCauley, Eds. ACM, 2020, pp. 259–270.

[15] 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.

[16] 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*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 496–506.

[17] M. J. Coblenz, J. Aldrich, B. A. Myers, and J. Sunshine, "Can advanced type systems be usable? an empirical study

of ownership, assets, and typestate in obsidian," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 132:1–132:28, 2020.

[18] M. Fähndrich and K. R. M. Leino, "Declaring and checking non-null types in an object-oriented language," in *2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003*, R. Crocker and G. L. S. Jr., Eds. ACM, 2003, pp. 302–312.

[19] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[20] L. N. Q. Do, J. R. Wright, and K. Ali, "Why do software developers use static analysis tools? a user-centered study of developer needs and motivations," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 835–847, 2022.

[21] G. Morling, "Jakarta bean validation specification." 2019. [Online]. Available: https://jakarta.ee/specifications/bean-validation/2.0/bean-validation_2.0.pdf

[22] M. M. Papi, M. Ali, T. L. C. Jr., J. H. Perkins, and M. D. Ernst, "Practical pluggable types for java," in *ACM/SIGSOFT International Symposium on Software Testing and Analysis*, B. G. Ryder and A. Zeller, Eds. ACM, 2008, pp. 201–212.

[23] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "SPOON: A library for implementing analyses and transformations of java source code," *Softw. Pract. Exp.*, vol. 46, no. 9, pp. 1155–1179, 2016.

[24] D. H. Ken Arnold, James Gosling, *THE Java™ Programming Language, Fourth Edition*. Addison Wesley Professional, 2005.

[25] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks, "Typestate-oriented programming," in *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, S. Arora and G. T. Leavens, Eds. ACM, 2009, pp. 1015–1022.

[26] Microsoft, "Language server protocol," 2022. [Online]. Available: https://microsoft.github.io/language-server-protocol/

[27] 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.

[28] 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*, C. V. Lopes and K. Fisher, Eds. ACM, 2011, pp. 643–656.

[29] J. Lubin and S. E. Chasins, "How statically-typed functional programmers write code," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–30, 2021. [Online]. Available: https://doi.org/10.1145/3485532

[30] K. Thayer, S. E. Chasins, and A. J. Ko, "A theory of robust API knowledge," *ACM Trans. Comput. Educ.*, vol. 21, no. 1, pp. 8:1–8:32, 2021. [Online]. Available: https://doi.org/10.1145/3444945

[31] 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.

[32] 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*, G. T. Leavens and M. B. Dwyer, Eds. ACM, 2012, pp. 683–702.

[33] B. Ellis, J. Stylos, and B. A. Myers, "The factory pattern in API design: A usability evaluation," in *29th International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 302–312.

[34] 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*, M. J. Harrold and G. C. Murphy, Eds. ACM, 2008, pp. 105–112.

[35] 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 *International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 760–771.

[36] 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.

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

[38] N. Vazou, E. L. Seidel, and R. Jhala, "Liquidhaskell: experience with refinement types in the real world," in *2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, W. Swierstra, Ed. ACM, 2014, pp. 39–51.

[39] M. Kazerounian, N. Vazou, A. Bourgerie, J. S. Foster, and E. Torlak, "Refinement types for ruby," in *Verification, Model Checking, and Abstract Interpretation - 19th International Conference*, ser. Lecture Notes in Computer Science, I. Dillig and J. Palsberg, Eds., vol. 10747. Springer, 2018, pp. 269–290.

[40] R. Chugh, D. Herman, and R. Jhala, "Dependent types for javascript," in *27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, G. T. Leavens and M. B. Dwyer, Eds. ACM, 2012, pp. 587–606.

[41] G. S. Schmid and V. Kuncak, "Smt-based checking of predicate-qualified types for scala," in *7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH*, A. Biboudis, M. Jonnalagedda, S. Stucki, and V. Ureche, Eds. ACM, 2016, pp. 31–40.

[42] P. Vekris, B. Cosman, and R. Jhala, "Refinement types for typescript," in *37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*, C. Krintz and E. Berger, Eds. ACM, 2016, pp. 310–325.

[43] B. Stein, L. Clapp, M. Sridharan, and B. E. Chang, "Safe stream-based programming with refinement types," *CoRR*, vol. abs/1808.02998, 2018.

[44] W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, M. Herda, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich, "The key platform for verification and analysis of java programs," in *Verified Software: Theories, Tools and Experiments - 6th International Conference*, ser. Lecture Notes in Computer Science, D. Giannakopoulou and D. Kroening, Eds., vol. 8471. Springer, 2014, pp. 55–71.

[45] B. Beckert and S. Grebing, "Evaluating the usability of interactive verification systems," in *1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, United Kingdom, June 30, 2012*, ser. CEUR Workshop Proceedings, V. Klebanov, B. Beckert, A. Biere, and G. Sutcliffe, Eds., vol. 873. CEUR-WS.org, 2012, pp. 3–17.

[46] B. Beckert, S. Grebing, and F. Böhl, "A usability evaluation of interactive theorem provers using focus groups," in *Software Engineering and Formal Methods - SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, September 1-2, 2014, Revised Selected Papers*, ser. Lecture Notes in Computer Science, C. Canal and A. Idani, Eds., vol. 8938. Springer, 2014, pp. 3–19.

[47] G. F. Kadoda, "A cognitive dimensions view of the differences between designers and users of theorem proving assistants," in *12th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2000, Cosenza, Italy, April 10-13, 2000*. Psychology of Programming Interest Group, 2000, p. 9.

[48] K. Aksoy, S. Tahar, and Y. Zeren, "Introduction to hol4 theorem prover," *Sigma Journal of Engineering and Natural Sciences*, vol. 10, no. 2, pp. 237–243, 2019.

[49] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: a Java modeling language," in *Formal Underpinnings of Java Workshop (at OOPSLA'98)*. Citeseer, 1998, pp. 404–420.