arXiv:1912.04719v4 [cs.HC] 25 Aug 2020

PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design

MICHAEL COBLENZ, Carnegie Mellon University, USA
GAURI KAMBHATLA*, University of Michigan, USA
PAULETTE KORONKEVICH*, University of British Columbia, Canada
JENNA L. WISE, Carnegie Mellon University, USA
CELESTE BARNABY*, Facebook, Inc., USA
JOSHUA SUNSHINE, Carnegie Mellon University, USA
JONATHAN ALDRICH, Carnegie Mellon University, USA
BRAD A. MYERS, Carnegie Mellon University, USA

Programming language design requires making many usability-related design decisions. However, existing HCI methods can be impractical to apply to programming languages: they have high iteration costs, programmers require significant learning time, and user performance has high variance. To address these problems, we adapted both formative and summative HCI methods to make them more suitable for programming language design. We integrated these methods into a new process, PLIERS, for designing programming languages in a user-centered way. We evaluated PLIERS by using it to design two new programming languages. Glacier extends Java to enable programmers to express immutability properties effectively and easily. Obsidian is a language for blockchains that includes verification of critical safety properties. Summative usability studies showed that programmers were able to program effectively in both languages after short training periods.

CCS Concepts: • **Human-centered computing** → *User studies*; *Usability testing*; *User centered design*; • **Software and its engineering** → *Object oriented languages*; *Data types and structures.*

Additional Key Words and Phrases: usability of programming languages, programming language design

1 INTRODUCTION

Programming languages serve as interfaces through which programmers and software engineers can create software. The ability of these users to achieve their goals, as with other kinds of interfaces, depends on the usability of the languages in which they do their work. For example, the presence of `null` in Java results in a particular kind of error-proneness, since programmers can easily accidentally write code that dereferences `null` [45]. These kinds of mistakes persist despite the training and experience that professional software engineers have.

*Work conducted while at Carnegie Mellon University.

Authors' addresses: Michael Coblenz, Computer Science Department, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA, **mcoblenz@cs.cmu.edu**; Gauri Kambhatla, Computer Science Department, University of Michigan, 500 S. State St., Ann Arbor, MI, 48109, USA, **gkambhat@umich.edu**; Paulette Koronkevich, Computer Science Department, University of British Columbia, 2329 West Mall, Vancouver, BC, V6T 1Z4, Canada, **pletrec@cs.ubc.ca**; Jenna L. Wise, Institute for Software Research, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA, **jlwise@cs.cmu.edu**; Celeste Barnaby, Facebook, Inc., 1 Hacker Way, Menlo Park, CA, 94025, USA, **celestebarnaby@gmail.com**; Joshua Sunshine, Institute for Software Research, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA, **joshua.sunshine@cs.cmu.edu**; Jonathan Aldrich, Institute for Software Research, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA, **jonathan.aldrich@cs.cmu.edu**; Brad A. Myers, Human-Computer Interaction Institute, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA, **bam@cs.cmu.edu**.

There is a long history of research on understanding how programmers' cognitive processes relate to the tools they use [55, 76, 78, 80]. This work was described in part in the proceedings of the Empirical Studies of Programmers (ESP) and Psychology of Programming Interest Group (PPIG) workshops. More recently, this line of work has continued at conferences such as CHI, ICSE, and VL/HCC. Our focus is a practical one: *How can programming language designers leverage data from users to improve language usability?*

Our high-level focus can be refined in terms of three research questions:

Naturalness: How can we obtain insights as to what language designs will be *natural* for programmers, given that we are trying to obtain particular static safety guarantees in the language?

Iteration: How can we iterate on a particular design so it continually gets to be more effective for users?

Comparison: How can we compare multiple language designs to see which are more effective for users?

Some authors, such as Stefik and Hanenberg, have focused on using quantitative approaches [82, 83]. Others have focused on in-depth case studies to evaluate their languages [1]. Our approach is to integrate a wide variety of both *qualitative* and *quantitative* user-focused methods with *formal theory-based methods* [67] to design programming languages [20, 56]. This approach allows us to integrate user research into many different stages of the design process.

In order to address our three research questions, we adapted traditional HCI methods to the context of the design of programming languages that target professional software engineers. Then, we applied those adaptations to the design process of two new languages, Glacier and Obsidian, which we used as testbeds for language design methods. Finally, we synthesized the methodology into a process we call *PLIERS*: **P**rogramming **L**anguage **I**terative **E**valuation and **R**efinement **S**ystem. This paper describes the methods and process we developed, and motivates them by showing the insights that we obtained by using the process on Glacier and Obsidian. The Obsidian work is new in this paper, and the Glacier work was partially discussed previously [22].

It is not enough to only use methods from HCI to design a programming language because, if a designer wants programs to have well-defined meanings (semantics) and well-understood safety properties (soundness), the programming language must also be subject to a collection of semantic constraints from the theory of programming languages. We have integrated strategic application of our adapted programming language design and evaluation methods into a process that also incorporates formal methods so that the resulting languages can be both usable and sound. Although the individual methods have been applied in the HCI literature in a variety of contexts, this paper shows how we have adapted the methods and combined them to obtain insights regarding programming languages that target professional software engineers. PLIERS is not a recipe for language design, just as *agile* is not a recipe for software engineering. Instead, PLIERS provides a process for organizing language design work around human-centered methods. For each step in the design process, PLIERS suggests ways of integrating human-centered methods to inform the designers.

Designing programming languages requires expertise in type systems, compilers or interpreters, and language runtimes. As such, PLIERS is aimed at showing people with those technical tools that it is feasible and effective to include user-centered methods. By doing so, the goal is that the languages will be more effective for programmers than they might be otherwise.

This paper makes three main contributions:

(1) We define the PLIERS programming language design process, which shows how user-centered methods can contribute to many different phases of programming language creation. We evaluated PLIERS by using it to develop two programming languages; we describe the benefits and areas for improvement that we observed in the process (Section 3.1).

(2) We show how we have adapted several *formative* study techniques, such as natural programming, Wizard of Oz, rapid prototyping, cognitive dimensions of notations analysis, and interview studies to inform the design of Glacier and Obsidian. We found that our adapted methods were effective when used in the context of PLIERS.
(3) We show how we conducted *summative* usability studies on new programming languages. By developing ways of teaching the languages efficiently, effectively, and consistently, we were able to conduct usability studies of programmers using novel programming languages.

We designed PLIERS for use with language designs that require developers to learn challenging programming concepts or think in a new way. The safety properties that motivated the two languages we discuss in this paper provided opportunities for language constructs that could potentially be confusing, which made them ideal testbeds for PLIERS. However, other contexts provide other kinds of conceptual challenges for programmers, and PLIERS may be similarly useful in those contexts. For example, multicore programming results in concurrency challenges; distributed programming requires managing asynchronous requests that may fail; and domain-specific languages (DSLs) may require mastery of domain concepts. We expect that PLIERS will be useful for helping language designers with any language that requires programmers to master difficult concepts.

In developing PLIERS, we initially intended to apply known HCI methods, such as *natural programming* [57], *Wizard of Oz* [28], *interviews*, and *rapid prototyping*. However, we found that the study design process was very challenging due to the nature of programming and the complexity of the design space. For example, some of the challenges we faced included:

Recruiting: how could we recruit participants who have sufficient programming skill and whose results would generalize beyond the population of students, despite having limited access to professional software engineers?
Training: how could we train participants in a new programming language in a short enough amount of time to make studies practical?
High prototyping cost: how could we conduct user studies on programming languages that have only informal designs and no implementations, since the cost of building working prototypes is high?
Variance and external validity: how would we mitigate high variance, which is typical in programming tasks, without constraining the tasks so much that they were no longer representative of real-world programming tasks?

The problems of variance and external validity were particularly relevant for quantitative studies, which needed to be practical in the context of our university setting. Programming tasks that are *not* extremely constrained tend to produce results with high variance, making statistical significance hard to obtain. On the other hand, tasks that *are* highly constrained suffer from low external validity, since real-world programming tasks are typically long and complex.

Therefore, we had to modify existing methods to address these challenges. Our study design contributions are summarized in Table 9. For example:

- By adapting the *natural programming* technique to allow *progressive prompting*, we were able to obtain both unbiased responses as well as data that were relevant to the particular designs we were considering.
- By *back-porting* language design questions to languages with which participants were familiar and by using the *Wizard of Oz* evaluation technique, we were able to obtain usability insights on incomplete designs, and *isolate* the design questions of interest from confounding variables.

- By dividing large tasks into multiple, smaller tasks, and by using pilot studies to set task time limits effectively in quantitative studies, we were able to reduce variance sufficiently to obtain meaningful results in complex programming tasks.
- By recruiting participants who were *representative of at least some junior-level professional developers*, we were able to maximize external validity in our studies while still conducting them practically at a university setting. We were also able to show usability impacts of the language designs under consideration.
- By developing incremental tutorials with integrated practice opportunities, we were able to *teach* the languages to participants in a short time (for Obsidian, about 90 minutes was typical).

In order to contextualize the methods we describe in this paper, Sections 1.1 and 1.2 explain the two languages that we used to develop the methods. Section 2 discusses related work, and Section 3 introduces PLIERS. The rest of the paper proceeds by showing how we used PLIERS in Glacier (Section 4) and Obsidian (Sections 5 to 7). Then, we discuss the challenges to effective study design that we observed while creating those two languages and how we addressed those challenges (Section 8). We propose future work and conclude in Sections 9 and 10.

1.1 Glacier

Glacier [22] is an extension to Java that supports *transitive class immutability*. Although security experts had recommended expressing state in an immutable way whenever possible, it was unclear how programming languages should support immutability. For example, Java includes the `final` keyword, but because `final` only restricts assignment to variables and not mutation of referenced state, actually enforcing immutability in Java is very difficult. The Java code below compiles without error despite the assignment and the use of `final`:

```
final int a[] = {0};
a[0] = 42;
```

In designing Glacier, we sought to show how a language design might support the use of immutability in practical programming languages. *Immutability* means that objects cannot be changed after they are created. Several organizations recommend the use of immutability to prevent security vulnerabilities in software. For example, Oracle's *Secure Coding Guidelines for Java* [71] and Microsoft's *Framework Design Guidelines* [70] both recommend using immutability for security reasons. However, we found that there were hundreds of different possible designs for immutability protection in programming languages, and it was unclear which approaches might be usable by programmers and which might actually support programmers' needs [26].

To determine a point in the design space that might be useful and effective, we conducted semi-structured interviews with eight software engineers. In those interviews, we asked questions about bugs, such as "Can you think of a bug you investigated or fixed that was caused by a data structure changing when it should not have?" [27]. All the participants who worked on software with significant amounts of state said that incorrect state change was a major source of bugs.

As a result, we hypothesized that *transitive* immutability might be a useful point in the design space to pursue. *Transitive* means that the restriction applies not just to a class, but recursively to all of its fields. *Immutability* means that objects to which the restriction applies cannot have any of their data modified through *any* reference, as opposed to the restriction only applying to certain references to a given object. This kind of immutability would provide strong guarantees, which developers could rely on to protect against improper changes to state.

We initially adapted an existing system, IGJ [96], to enforce transitivity. We found in a usability study, however, that there were significant usability challenges with this approach, which related to the flexibility provided by IGJ to apply

restrictions to individual objects. This led us to create Glacier, which supports transitive *class* immutability, in which classes that are declared immutable have all instances immutable. We were able to show that Glacier (a) could be used effectively by our study participants to specify immutability; and (b) detected improper-mutation bugs that participants frequently inserted in the codebase when they were using regular Java.

The final version of Glacier provides a new annotation, `@Immutable`, which can be applied to class definitions. `@Immutable` indicates that every instance of that class must be transitively immutable. The alternative to `@Immutable` is `@MaybeMutable`, which applies by default. The compiler checks classes that are annotated `@Immutable` and gives an error if any of the fields are references to classes that are themselves not `@Immutable`. Also, assignment is disallowed to fields of `@Immutable` classes except in their constructors. For example, the comments indicate that the compiler would emit errors for lines 3 and 6:

```
1  @Immutable class Person {
2      String name; // OK; String is @Immutable
3      Date birthdate; // Error; Date is @MaybeMutable
4
5      void setName(String n) {
6          name = n; // Error; cannot assign to fields of @Immutable classes
7      }
8  }
```

In Section 4, we describe how we used qualitative methods to ground our initial design in user data and how we conducted an RCT to compare Glacier to Java's `final` feature. In that study, we recruited 20 participants and found that although none of those assigned to use `final` were able to specify immutability in an existing program correctly, almost all of the Glacier participants were able to do so. We also gave the participants two maintenance tasks. Among 14 tasks that Glacier participants said they completed, all were done correctly. However, of 18 tasks that `final` participants said they completed, only seven were completed correctly.

1.2 Obsidian

Obsidian is targeted at programming *blockchains* [44], in which a decentralized network of computers maintains system state and executes transactions. Blockchains support deploying *smart contracts*, which are programs that maintain state. Typically, each deployment is an instance of a class, though in a blockchain context, the keyword `contract` is used instead of `class`. In contrast to most of the existing user-centered programming language work, which often focuses on novice or end-user programmers [48], Obsidian is intended for use by professional programmers and software engineers. This presents additional challenges, since we are interested in evaluating how the language will be used in the long term despite being limited in our ability to recruit software engineers to work for extended periods of time in our studies. After our design was complete, we found in a summative study that most of the participants were able to complete programming tasks successfully in Obsidian.

Blockchains, which have been proposed for high-stakes applications such as financial transactions, health care [42], supply chain management [46], and others [32], are an ideal testbed for a new language design process. The need for a safer language is motivated by the history of security vulnerabilities, through which over $80 million worth of virtual currency has been stolen [37, 79]. However, it is not realistic to assume, as some other projects do [3, 8, 43], that the developers will be experts in formal verification or that companies will invest the resources required to formally verify that their programs are correct. Instead, we seek a more *lightweight* approach that provides additional safety guarantees at low cost to developers.

We established several objectives in our design of Obsidian:

(1) Improve safety by detecting more bugs than current smart contract languages do, preferably at compile time, to prevent deployment of buggy programs.
(2) Maximize usability by ensuring that programmers can complete domain-appropriate programming tasks, ideally with little training in the language.
(3) Advance the science of programming language design by developing user-centered methods that can contribute to a more usable language.

Detecting bugs was our initial objective, so we considered bugs, such as the DAO hack [29], which resulted from a reentrant invocation in which a contract allowed itself to be invoked while in an inconsistent state. We also analyzed characteristics of proposed blockchain applications. In general, we observed that proposed blockchain applications typically maintain high-level state, which governs which operations are safe.

For example, a `Casino` can accept `bet` invocations only before the `Game` has been played. More generally, the authors of Solidity [36], a commonly-used smart contract language, observed that many contracts implement state machines [35]. Unfortunately, in Solidity, users must define states via enumerated types and then manually ensure that methods are only invoked when the target object is in an appropriate state. Although methods that can only be invoked in particular states are common [5], writing programs that only invoke methods when appropriate has been shown to be hard for users [86], and Solidity includes no mechanism to ensure safety.

Smart contracts commonly manipulate *assets*, which are objects that have value (such as cryptocurrencies). In Solidity, it is possible to lose track of money and other assets [30], resulting in their value being permanently irretrievable. We were interested in designing a language in which many kinds of asset loss could be detected by the compiler.

In order to leverage those observations, we became interested in a *typestate*-oriented approach [2], in which *states* of objects are incorporated into types. For example, rather than merely having a `LightSwitch` type, we can have `LightSwitch@On` be the type of a reference to an object that is in the `On` state. Then, if the user attempts an invalid operation, such as turning on a switch that is already on, the compiler can issue an error.

Typestate-based types are in a class called *linear types*. Unlike traditional types, linear types can change as operations are performed. For example, invoking `turnOff()` on a reference of type `LightSwitch@On` *changes the type* of the reference to `LightSwitch@Off`. Conveniently, linear types are also what are needed to ensure that assets are never lost. Obsidian includes *owned* objects: for each owned object, there is an object that owns it via an owning reference. If a local variable that owns an asset goes out of scope, the compiler emits an error message. Fields that own assets can only exist in contracts that are themselves assets. This way, each asset always has an owner.

We selected an object-oriented approach, since object-oriented approaches are well-suited for representing state and corresponding updates. We avoided inheritance, since we wanted to avoid the fragility that results [54]. For a full description of the language, please refer to Coblenz et al. [25]. However, Figure 1 shows some of the key features of the *final* version of Obsidian using the example of a *tiny vending machine* (TVM). `TVM` is a `main` contract, so it can be deployed independently to a blockchain. A `TVM` has a very small inventory: just one candy bar. It is either `Full`, with one candy bar in inventory, or `Empty`. Clients may invoke `buy` on a vending machine that is in `Full` state, passing a `Coin` as payment. When `buy` is invoked, the caller must initially *own* the `Coin`, but after `buy` returns, the caller no longer owns it. `buy` returns a `Candy` to the caller, which the caller then owns. After `buy` returns, the vending machine is in state `Empty`.

```
// TVM is a Tiny Vending Machine.
main asset contract TVM {
  Coins @ Owned coinBin;

  state Full {
    Candy @ Owned inventory;
  }

  // No candy if the machine is empty.
  state Empty;

  TVM() {
    // Start with no coins, and go to the Empty state.
    coinBin = new Coins();
    ->Empty;
  }

  // restock transitions from Empty to Full by taking ownership of candy.
  transaction restock(TVM @ Empty >> Full this,
                      Candy @ Owned >> Unowned candy)
  {
    ->Full(inventory = candy);
  }

  // buy transitions from Full to Empty by taking ownership of a coin.
  // buy returns the purchased candy.
  transaction buy(TVM @ Full >> Empty this,
                  Coin @ Owned >> Unowned coin)
                  returns Candy @ Owned
  {
    coinBin.deposit(coin);
    Candy result = inventory;
    ->Empty;
    return result;
  }

  // withdraw removes any accumulated coins and returns them to the caller.
  transaction withdraw() returns Coins @ Owned
  {
    Coins result = coinBin;
    coinBin = new Coins();
    return result;
  }
}
```

Fig. 1. A tiny vending machine that shows key features of Obsidian.

In Section 5, we describe techniques we used to obtain data about individual language design choices. In the summative portion of the evaluation process, which we describe in Section 6, we recruited six participants. We trained them in Obsidian and then asked them to complete three programming tasks. Two of the participants inserted bugs that the Obsidian compiler detected, but which the Solidity compiler would not have been able to detect. Although we observed high variance in performance, two of the participants were able to complete all of the tasks.

We also conducted an experiment comparing Obsidian and Solidity, which we summarize in Section 7. In one task, Obsidian programmers were able to avoid losing an asset that the Solidity programmers frequently lost track of. In another, six of ten Obsidian participants were able to use ownership to fix a security problem. In a final, more open-ended task, we observed that surprisingly, the Obsidian programmers abused a feature of the language, resulting in asset loss,

and that Solidity programmers (who were not constrained by Obsidian's strong type system) were generally able to complete the task faster.

2 RELATED WORK

Newell and Card argued for the use of HCI methods in programming language design in 1985: "Millions for compilers but hardly a penny for understanding human programming language use [58]." Morrisett reiterated this problem in 2009, arguing that a programming language is a medium for communication among humans, but we lack principles for evaluating this aspect of languages [68]. Our earlier essay argued for using many different methods in language design [20]. Although that article promoted the use of formative methods (among others), this paper describes the methods in much more detail, giving recommendations for how other designers might use them in their own work. This paper also includes our experiences with Obsidian, including techniques we developed during that work. Finally, it describes PLIERS, which is our overall language design process.

The Empirical Studies of Programmers workshops focused in large part on a cognitive science approach to studying programmers: can we build models of cognition that explain programmer behavior? Key results include describing techniques used by programmers when working with code, such as identifying key lines (beacons), relating program details to the problem domain, and using both top-down and bottom-up understanding techniques. The ESP literature provided insights into the problems that people have when using existing languages.

Some of the work in ESP workshops studied professional programmers. For example, Pennington used theories of understanding of natural language text to model expert programmers' comprehension of programs [64], finding that procedural knowledge (rather than knowledge of functional units) dominated their understanding. The study was conducted on COBOL programs, which were likely structured substantially differently from modern software. Furthermore, no libraries or frameworks were used, so the fact that the programmers could see and consider all relevant code may have resulted in a substantially different kind of programming task than the ones that we consider today. However, the approach suggests that a cognitive modeling approach may help derive a theory of programmers that is relevant for designing programming languages. In this work, we rely on more recent, heuristic-based approaches, such as the Cognitive Dimensions of Notations [39], which are applicable in more general domains, and which came out of the cognitive science-based approach that was central in the ESP work.

Visser described a four-week observational study of one professional programmer working in a declarative, domain-specific language [93]. Visser noted that obtaining mental models was challenging because the programmer found think-aloud very difficult while working on problems, but made observations about the structure of the programmer's work. For example, the programmer used analogical reasoning and examples to help reason about the software; used both top-down and bottom-up work styles; and sought consistency in the program. This work provides an empirical foundation for some requirements of programming environments, such as allowing creation of interfaces separately from implementations, and providing tools to standardize notation (e.g. style linting tools).

Vans et al. conducted study of the comprehension process and information needs of programmers in industry doing maintenance tasks [90]. Some of the understanding techniques that the programmers used were similar to methods that were observed in novices as well, including top-down, bottom-up, and code-tracing methods, but the professionals used a much wider variety of techniques than had been observed in novices, such as generating and abandoning large numbers of hypotheses regarding the programs. This suggests that programming language design studies conducted with students can give some guidance regarding languages intended for professionals, but such studies may be limited in the kinds of techniques that the participants use. In many of the studies presented in this paper, we recruited

experienced students, who in many cases had several years of professional programming experience. This approach allowed us to broaden the set of techniques our participants would use to accomplish their tasks and make our results more generalizable relative to using only novice programmers.

Guindon et al. used protocol analysis [34] to analyze think-aloud protocols from three experienced programmers who were asked to design software to solve an elevator-scheduling problem [40]. Guindon et al. observed breakdowns in process that arose from lack of knowledge (e.g., of the problem domain) and from cognitive limitations (e.g., capacity of short-term memory). Because this work consisted of a think-aloud study of programmers, it shares several threats to validity with the qualitative work we describe here: a task that may not reflect real-world tasks; short duration of the task, which was concentrated in a lab-based two-hour session; and a sample of programmers that may not be representative of programmers in general. Our approaches to mitigating external validity were similar to theirs. We recruited from students who were likely to have some professional experience and do not claim that all programmers will encounter the same difficulties that they did. We do not claim that we observed all possible problems that users might have when using the tools we gave them. Instead, we argue that addressing the problems we observed is likely to help *some* relevant users be more effective in achieving their goals. In our approach to think-aloud studies, we analyzed notes taken by the experimenter and screen recordings of the participants doing the tasks.

Direct observations of work and interviews have both been used to understand how teams work together to develop software. Walz et al. analyzed videos of teams conducting a requirements analysis to study conflict patterns [94]; Krasner et al. interviewed members of 19 software development teams to understand team communication [51]. Although our work focused on individual developers, we used multiple methods where appropriate. This work shares threats to external validity with other small-sample studies, including the ones we conducted. Krasner et al. mitigated these risks by choosing diverse teams to study. Although Walz et al. only studied one team, they studied the team over 43 meetings over four months.

A substantial amount of prior work on the usability of programming languages focuses on novices. For example, HANDS [63], Helena [18], and Scratch [72] aimed to make it easier for novices to write programs. HANDS, in particular, introduced the *Natural Programming* technique, which we leveraged and adapted in this work. Stefik et al. also focused on novices, collecting quantitative data on their error rates [84]. Designing languages for novices is substantially different from designing languages for experienced programmers. For example, languages for novices typically focus on learnability. In contrast, languages for professionals commonly include additional complexity, in part resulting from the kinds of safety properties that are beneficial when building real systems.

Other work has focused on programming tools for end-user programmers, whose primary goal is not to write software but rather to accomplish goals in some particular domain [49]. For example, Blackwell and Burnett developed Attention Investment and applied it to a research spreadsheet tool, Forms/3 [11]. Peyton Jones et al. used Cognitive Dimensions [39] and Attention Investment to provide a new kind of user-defined functions in Excel [47]. Our work is focused on methods that address the unique challenges of complexity that result from targeting professional programmers and software engineers.

RCTs (randomized controlled trials) have been used to compare different programming language designs. For example, Uesbeck et al. investigated the impact of lambdas in C++ [88], and Endrikat et al. looked at static typing [33]. That work is a useful complement to this work, but the focus here is on using low-cost, practical *qualitative* methods to inform the entire language design process. In contrast, quantitative summative studies require high-fidelity prototypes in order to obtain measurements that can be expected to generalize to the final system. These prototypes can be very expensive to build for complex programming languages.

Another approach to programming language evaluation is to compare designs via crowdsourcing methods. Chamberlain [17] compared functional-style to literal-style approaches for specifying topology of streaming applications (i.e., pipes-and-filters style applications) using Mechanical Turk, finding that users were more likely to prefer literal-style specifications, and experienced programmers were more likely to understand the literal-style specifications than the functional-style ones. Wilson et al. [95] investigated crowdsourcing more esoteric language design decisions, finding low consistency (people did not give consistent answers when asked similar questions repeatedly) and low consensus (people did not agree with each other on which design choice was best). Crowdsourcing approaches can scale well, but typically require that the studies be of relatively short duration. This paper focuses on higher-bandwidth qualitative methods and on evaluation approaches for languages for professionals, and serves to complement crowdsourcing approaches.

The HCI literature includes many different language designs as well as other kinds of tools for programmers. For example, Dog/Jabberwocky [1], Protovis [14], Reactive Vega [75], and InterState [62] are all languages or APIs that make it easier for programmers to accomplish their goals. Those papers describe only the final designs of those systems and summative usability studies. This paper focuses on methods that can be used *during* the design process and gives recommendations that are useful in preparing a summative evaluation.

Finally, there is a variety of methodological guidance in SE and HCI that is applicable to studies of programming languages. Ko et al. discussed techniques for doing empirical studies of tools for software engineers [50]. Buse et al. conducted a systematic literature review, observing increasing use of user evaluations in software engineering research [16]. Verner et al. gave guidelines for industrial case studies in software engineering research [92]. Perry et al. gave a tutorial on case study methodology for software engineers [66]. Likewise, Shneiderman and Plaisant gave recommendations for using case studies for information visualization tools [77].

The technical details of the Obsidian language are described in a separate paper [25]; Glacier is also described in more detail separately [22]. This paper focuses on the user-centered process that we used to design and evaluate the languages.

3 PLIERS

PLIERS is summarized in Figure 2. User-centered design methodology [41] seeks to leverage data from users to improve the design of systems. PLIERS is a specialization of user-centered design for programming languages to enable designers to incorporate ideas from user studies as well as from the theory of programming languages. PLIERS consists of five phases: need finding, design conception, risk analysis, design refinement, and assessment. In each phase, the designer seeks and leverages input from or about the users that the designer is hoping will use the programming language. If work in any phase calls into question the work done in a previous phase, the designer may return to the previous phase and conduct more work according to the difficulties that were identified.

Need finding: The process begins by assessing the user's needs. What programming problems does the user have, and how might a new programming language help the user achieve their programming goals? Some have advocated that language designers should design languages for their own use [38]. In contrast, PLIERS uses user-centered methods, such as corpus study, interview, or contextual inquiry to understand the target audience and what their needs are. The designer chooses which particular user-centered methods to use according to the available resources and the goals of the design project. These user needs may be stated as hypotheses regarding what

kinds of languages might benefit users, what *benefit* means to those users, and how those benefits might be assessed.

Design conception: After assessing users' needs, the designer must conceive of initial language ideas that might satisfy those needs. As in other design situations, this process often requires significant creativity. The designer iterates between two kinds of work: theoretical work, in which the designer develops a theoretical foundation for the programming language (a *core calculus*), and prototyping work, in which the programmer directly works

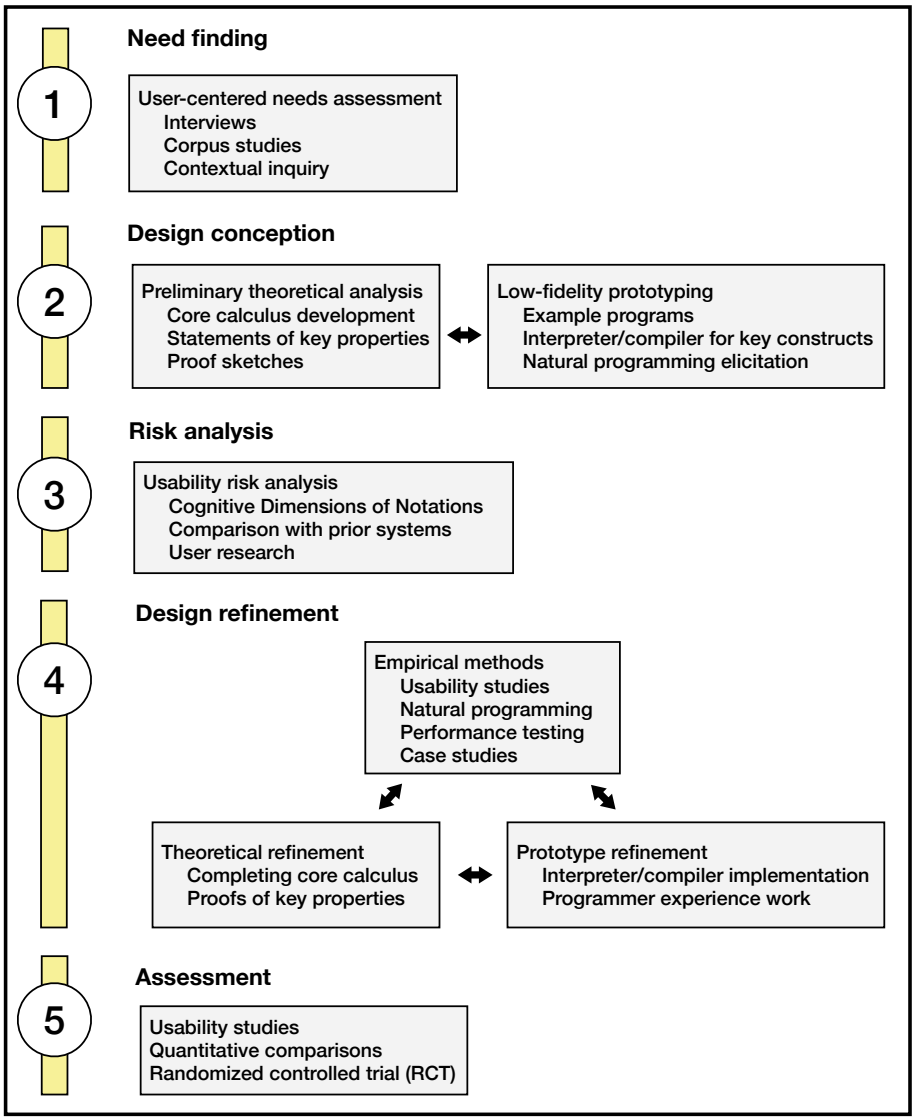


Fig. 2. The phases of the PLIERS process, showing activities conducted in each phase. Designers can return to previous phases if evaluation identifies opportunities for improvement.

on the language that programmers will see (the *surface language*). The result of this work is expressed as a low-fidelity prototype, such as a corpus of code samples (to demonstrate the surface syntax, by which users will write and edit programs) and a core calculus.

Risk analysis: In the risk analysis phase, the designer assesses and prioritizes usability risks in the proposed design. This step leverages theoretical models of cognition and usability inspection techniques to identify the aspects of the language design that are most likely to present difficulties to users. For example, cognitive dimensions of notations [39] can help identify usability risks that are worthy of further study.

User research might be needed in this phase to better understand the target audience. For example, if the designer is considering an approach that requires particular skills, then risk analysis might include assessing to what extent the target audience has those skills or whether those skills can be taught in an acceptable amount of time. If the language targets professionals, substantial training may be acceptable. If the language targets end-user programmers, the designer may want to limit the training that would be required.

The prototype design likely leverages some elements of existing languages, while creating new features that may be unfamiliar to users and have unknown usability characteristics. These new features may be particularly worth evaluating. Each risk corresponds to a design or research question, and provides an opportunity for learning more about how to make the programming language as usable as possible.

Design refinement: Using empirical methods, the designer assesses the usability risks identified in the prior phase. Then, the designer refines the prototype, successively increasing prototype fidelity as usability risks are addressed. The designer also considers other language requirements, such as *expressiveness* (can the language be used to write a particular kind of program?) and *performance* (does the program, when run, meet the designer's performance goals?). Then, the results are used to revise the theoretical model and the prototype. By using theory, the designer can ensure that any changes retain any formal guarantees that the language promises. Alternatively, the designer may choose to guarantee different properties in order to allow the desired modifications.

Because the theoretical model and prototype are related, changes in one frequently lead to changes in the other. Eventually, the theoretical analysis will include proofs of key properties, and the prototype will be high-fidelity, typically including IDE support and a compiler or interpreter.

Assessment: A summative usability study can assess whether the final design has achieved the designers' usability objectives. In contrast to the usability studies in the previous phase, which assess specific aspects of the language design, a summative study is intended to assess programmers' abilities to complete realistic programming tasks.

In this paper, we focus on usability-related objectives, but the designer may want to conduct performance testing as well. For performance evaluation, we refer readers to the SIGPLAN empirical evaluation checklist [6].

In developing the PLIERS process, we identified a collection of adaptations to traditional HCI methods, which helped us obtain useful information, primarily in the *design refinement* and *assessment* phases. Here, we describe some of the key ways in which we modified existing methods, which are further discussed in Section 8. The methods we have found useful are summarized in Table 1.

Back-porting design questions to existing languages: To study the usability of a design decision in isolation, we start from an existing language with which participants would already be familiar. For example, rather than asking participants in our early formative studies to learn a whole programming language, we told them that we were adding certain features to Java, and then asked them to complete programming tasks in the Java variant.

This substantially reduced the training time and allowed us to reason that any confusion was likely related to the new features, since our participants were already familiar with Java.

Selecting a target language for back-porting depends on several factors:

(1) Availability of participants skilled in the target language
(2) High-level similarity between the novel language and the target language (e.g., both object-oriented, both functional, etc.)
(3) For high-fidelity prototypes: development cost of modifications to target language

We also favored high-level design decisions that allowed us to attract participants who had relevant background. If we had tried to teach participants a completely novel language paradigm even though the basic assumptions of the language paradigm were not the targets of our research, we would have needed to try to distinguish the *relevant* mistakes from all the novice-level mistakes that the new programmers would be likely to make.

Wizard of Oz: Implementing a programming language is expensive. Rather than implementing a full compiler for each language variant we wanted to test, we adapted the Wizard of Oz technique [28]. In a classic Wizard of Oz study, an experimenter pretends a system is working by remote-controlling it in order to obtain insights about potential designs without having to actually build the system.

In early Obsidian studies, we gave participants a text editor, documentation, and programming tasks to do. Then, an experimenter verbally simulated compiler errors. Like a modern IDE, the experimenter could interject with errors, and could provide error messages when participants asked whether the compiler would emit any errors on their current code. For example, the experimenter might say "Suppose your compiler indicated that there was asset loss that occurred on line 42." If the error messages were unclear, the experimenter could revise them with more detail, helping us understand how to write clear error messages for the compiler. The technique we developed allowed efficient iteration on our design ideas, since design changes only required updating the documentation, not a potentially complex implementation. Unlike in a traditional Wizard of Oz study, participants were aware that the feedback was being provided manually, but we observed that this did not present an obstacle to the effectiveness of the technique.

Multi-part tutorials: In our studies, we needed to teach participants a new programming language in a consistent and efficient way. A traditional course would not be effective, since we could not recruit our participants into a semester-long course. Instead, we developed multi-part language tutorials. By breaking the tutorial into sections (each about ten minutes long), including practice problems for participants to do, and having an experimenter available to answer questions, we were able to convey the knowledge we needed in a relatively short period of time. Our longest tutorial, for example, took participants an average of 1 hour, 35 minutes.

Next, we show how we applied PLIERS when designing Glacier (Section 4) and Obsidian (Sections 5 and 6). Then we discuss how our adaptations to traditional HCI methods helped address some of the challenges of using HCI methods on programming languages for software engineers (Section 8).

3.1 Evaluating PLIERS

To evaluate PLIERS, one might like to teach PLIERS to a collection of programming language designers and conduct a qualitative study regarding the insights the designers obtained. Better yet, one might like to recruit language designers and assign them to either use or not use PLIERS to design a language in a domain, and then conduct usability studies

Design Task	Method or Component	Details
Need finding and hypothesis generation	Interviews	• Interview practitioners (e.g., software engineers) to identify problems and formulate hypotheses
	Corpus analysis	• Analyze corpora of applications and bugs to identify common goals and obstacles
Formative design evaluation	Natural programming elicitation	• Ask participants to do programming problems without giving them syntax or identifiers in order to help design a syntax and vocabulary that matches their expectations
	Programming tasks	• Back-port design components to an existing language to isolate variables of interest • Break larger tasks into subtasks to constrain unproductive exploration • Include a range of task difficulties to obtain data from both more- and less- successful participants • Use low-fidelity prototypes to obtain early feedback on designs • Use Wizard of Oz to enable studies of incomplete prototypes
Summative design evaluation	Usability studies	• Assess what barriers users face when attempting to complete relevant programming tasks
	Randomized controlled trials (RCTs)	• Compare task times and success rates between different languages

Table 1. A summary of user-centered methods that we have found useful for studies in PLIERS.

of the resulting languages. Unfortunately, these approaches are impractical: language design and implementation is typically a long process, taking months or years, and language designers cannot be recruited for such studies.

Another approach might be to teach PLIERS to designers who have recently completed their designs and observe what changes the designers make as a result. However, a main benefit of PLIERS is that it provides a framework for the entire language creation process; many of the methods can be applied to incomplete prototypes or design concepts. Such an analysis, though useful, would only obtain insight on some of the components of PLIERS.

Because of these practical considerations, we evaluated PLIERS by using it ourselves to create Glacier and Obsidian. In the process, we observed how the approach helped us create and iterate on the language designs. This approach has significant limitations. Our evaluation does not show that other designers can use the process effectively, that it works on a wide variety of different programming languages, that languages produced with the method are necessarily superior to languages produced without the method, or even that the process does not make languages *worse*. However, using PLIERS ourselves was a necessary part of developing the process; in this paper, we leverage our experience creating PLIERS in the hope that others may benefit from it and iterate on its component methods. As this is the first work of

which we are aware that integrates HCI methods into the design of languages for professional software engineers, we view this is the first step toward creating a design process that will aid designers in making professional-level languages more usable.

Venable et al. provide a framework for evaluating design methodologies [91]. In that framework, our approach to evaluating PLIERS would be considered *ex ante* (formative, evaluating the design method before it is complete). The ex ante approach was appropriate for PLIERS since the work was conducted in large part to create and iterate on PLIERS. The evaluation included some aspects of *naturalistic* evaluations (it was evaluated in the context of real language design projects) and some aspects of *artificial* evaluations (the designers of the method used the method instead of third parties). This choice was driven by the impracticality of recruiting programming language designers other than ourselves to use our process over the long period of time required to conduct a language design and implementation project.

Blandford and Green have acknowledged the lack of an established path to acceptance for new methods and the difficulty of conducting rigorous evaluations of design methods [12]. We regard our work as the first step of many in evaluating PLIERS.

4 PLIERS FOR GLACIER

Formative studies are conducted before a system is built, or in the process of building the system, to learn about the target users or problem space, or to refine a design. *Summative* studies are conducted after an artifact is created to evaluate whether the designers have achieved their design goals. In Glacier, we conducted studies of both types. Before we developed Glacier, we conducted formative studies (Section 4.1) to develop a hypothesis regarding which particular kind of immutability might have direct benefits to programmers. We developed IGJ-T based on this hypothesis, but our risk analysis suggested that the flexibility of IGJ-T might result in usability barriers. We conducted a usability study of IGJ-T, which confirmed our hypothesis. Then, we refined our design to create Glacier, and conducted a summative study to answer two usability questions: first, can people easily specify immutability with Glacier, and if so, is it easier with Glacier than it is with Java? Second, does providing compile-time enforcement of immutability prevent bugs that would likely be inserted otherwise? Section 4.2 shows that the answers are in the affirmative.

4.1 Formative studies

We used the *Cognitive Dimensions of Notations* framework [39] to reason about some of the design choices. For example, including features that provided weaker guarantees than programmers actually needed could be *error-prone* if those features could be easily confused with stronger ones. Likewise, the inverse is error-prone too: if a programmer applied a weaker specification than could actually be applied, this could lead to undesirable tradeoffs. For example, if an interface is annotated to return a read-only object (indicating that the object potentially could be mutated through *other* references), the programmer might add locks to ensure safety in a concurrent context. But if the object is actually *immutable* (that is, *no* reference could be used to mutate the object), then the locks would be unnecessary and reduce performance. More details about our Cognitive Dimensions analysis appear in our earlier paper [26].

Although the Cognitive Dimensions analysis was lightweight, it did not answer some of our higher-level design questions. Cognitive Dimensions provides a vocabulary for discussing and analyzing tradeoffs, but it does not provide ground truth regarding how usable particular approaches will be for people. In order to narrow the space of possible language designs, we conducted semi-structured interviews with eight software engineers who were working on

large software projects at several organizations. Our participants had an average of fifteen years of experience, with a minimum of seven years, and had worked on projects with millions of lines of code and hundreds of people.

In order to both obtain unbiased data on problems with mutability in general as well as to obtain feedback on concrete language designs, we carefully ordered the interview questions. First we asked general questions, such as "How do you make sure that state in running programs remains valid?" We got wide-ranging answers, including ones such as "We've essentially done away with mutability to avoid security and concurrency problems" as well as recommendations for regular use of testing and assertions. Afterward, we asked about existing language features, such as `const` and `final` and their use. Then we asked about specific related areas, including concurrency and security. Finally, we asked about our own language design ideas, including immutable classes. The full set of interview questions is included in a previous paper [27].

Our interview participants said that bugs in which state changes when it is not supposed to are frequent. They also described how the language features they had available did not provide guarantees that were sufficient for their purposes. For example, when reusing existing code, participants could not typically tell whether the code was thread-safe, so they had to assume that it was not. If a component came with an appropriate compiler-checked immutability specification, then they could be confident of safety, but languages did not provide such a feature. We concluded that *transitive* immutability provided the strong safety properties that our interview participants requested: a transitively immutable object can be shared safely among threads without locks.

An interesting observation that came out of the interview studies is that typically, for a given class, either all instances are mutable or all instances are immutable. In contrast, some prior systems, such as IGJ [96], supported immutability at the object level of granularity (*object immutability*). We evaluated an initial prototype, IGJ-T, that extended IGJ with transitivity. We found that participants had great difficulty managing the complexity, which was in part because IGJ's syntax focused on object immutability, not class immutability. We reasoned that if we designed our system to support *class immutability* only, our system would be simpler and therefore likely easier to use without sacrificing much expressiveness. This motivated our new tool, Glacier, which was centered around *transitive class immutability*.

Figure 3 shows several design alternatives. #1 shows how in IGJ, immutability is a property of references, not of classes. Compared with IGJ, IGJ-T (#2) also enforces that fields of classes for which there are immutable instances must have all-immutable fields (enforcing transitivity). In Glacier (#3), immutability is a property of classes, not of references. Variant #4 explores a possible extension of Glacier, in which the compiler can automatically derive immutable versions of mutable classes. We elected not to pursue that approach, since our interview had indicated that most classes are used either in an immutable or a mutable way, but not both.

Triangulation [74], in which a designer combines results of multiple qualitative studies, was a key aspect of the design process. We sought designs that resembled the approaches participants proposed via natural programming and which also enabled participants to complete programming tasks as effectively as possible in our task-based studies. We also leveraged real-world evidence of security vulnerabilities to motivate our safety objectives. At the same time, we were guided by the theory of programming languages, which we used to ensure that our language would provide the guarantees that our design intended to achieve.

4.2 Summative studies

In addition to doing two case studies to evaluate *expressiveness* [25], we conducted a lab study to answer two research questions relating to our *comparison* question in §1 ("How can we compare multiple language designs to see which are more effective for users?"):

1: IGJ. Immutability is a property of references. Immutability is not transitive.

```
public class Game  {
  // No need for 'outcome' to be immutable, since immutability is not transitive.
  private Outcome outcome;

  // Constructor returns a reference to an immutable object.
  @Immutable Game (Outcome outcome) {
    this.outcome = outcome;
  }
}
```

2: IGJ-T. Immutability is a property of references. Immutability is transitive.

```
public class Game {
  // @Immutable is required when declaring 'outcome' because there is
  // at least one constructor that returns an @Immutable reference.
  private @Immutable Outcome outcome;

  // Constructor returns a reference to an immutable object.
  @Immutable Game (@Immutable Outcome outcome) {
    this.outcome = outcome;
  }
}
```

3: Glacier. Immutability is a property of classes. Immutability is transitive.

```
@Immutable public class Game {
  private Outcome outcome; // OK because Outcome class was declared @Immutable

  // Every instance of Game is immutable, so no need to specify immutability.
  Game (Outcome outcome) {
    this.outcome = outcome;
  }
}
```

4: An variant of Glacier, in which the compiler can synthesize immutable subsets of mutable classes.

```
// Assume Outcome is mutable, but the compiler can synthesize an
// immutable version by leaving out the mutating methods
@Immutable public class Game {
  Outcome outcome; // error: Outcome is mutable
  @Immutable private Outcome immutOutcome; // OK: use immutable subset

  Game (Outcome outcome) {
    // Assignment is always permitted in the constructor
    this.outcome = outcome;
  }

  void test () {
    this.immutOutcome.setOutcome(WON); // compile error: no such method
    this.outcome = ... // compile error because Game is @Immutable
  }
}
```

Fig. 3. Alternatives we considered for immutability systems.

(1) Can participants express immutability more successfully in Glacier than with Java's `final` keyword?
(2) Without Glacier (using only standard Java), are programmers likely to accidentally insert the kinds of bugs that Glacier detects?

We recruited 20 Java programmers. We randomly assigned participants to use either Glacier or `final`, and we gave participants a tutorial in their given tool (two pages for Glacier, three pages for `final`). In addition, we gave the `final`

	final	**Glacier**
Correctly enforced immutability in `Person`	0/10	10/10
Correctly enforced immutability in `Accounts`	0/10	9/10
Average time for enforcing immutability across both tasks	14 min.	11 min.
`FileRequest.execute()`		
Tasks without security vulnerabilities	4/8	7/7
Average time spent (among participants who finished)	14 min.	14 min.
`HashBucket.put()`		
Tasks without bugs	3/10	7/7
Average time spent (among participants who finished)	18 min.	14 min.

Table 2. Summary of summative study results for Glacier

participants a page from *Effective Java* [13] explaining how to safely enforce immutability with `final`. Then, to address the first question, we asked them to change one class in each of two small projects (`Person` and `Accounts`) so that those classes were immutable. None of the participants in the `final` condition were able to do their task successfully because it was too easy to forget to do one of the changes required, such as copying mutable inputs to constructors. Of the 20 Glacier tasks attempted, participants completed 19 correctly. The difference between success rates across conditions is significant in each task (Person: $p \approx 1.08 \times 10^{-5}$; Accounts: $p \approx 1.2 \times 10^{-4}$, Fisher's exact test). Among users who said they were done with both tasks, `final` users completed both annotation tasks in an average of 14 minutes. The average Glacier user completed the task in 11 minutes. This difference is significant with $p \approx 0.1$ (Wilcoxon rank sum test).

To address the second question, we asked our participants to do two programming tasks (`FileRequest.execute` and `HashBucket.put`) on two small immutable classes. Although we did not verbally tell them that the classes were immutable, the classes were adapted from real-world code, and the participants had just completed the tasks above pertaining to immutability. In the Glacier condition, each task was completed successfully by seven participants; of course, no one accidentally mutated immutable state because Glacier disallowed it. In the `final` condition, however, four of eight participants who finished the first task completed it successfully, and only three of ten participants who finished the second task completed it successfully. Fisher's exact test indicates these differences are significant at $p \approx .077$ for `FileRequest.execute` and $p \approx .0098$ for `HashBucket.put`. These results are summarized in Table 2.

These results seems surprising: although we tried to design the experiment to be as unbiased as possible, the programming tasks were actually biased toward the control condition (`final`) in that participants had just been trained to consider immutability. One would expect, then, that in a real-world scenario, programmers might perform even more poorly. The success of this study teaches us some lessons about study design:

Errors are frequent: Programming is so difficult that participants are likely to make errors very frequently, consistent with the *variance* challenge. Some of these errors will be ones that the experiment designer was hoping to observe, but many of them will be irrelevant. To mitigate this, ensure that participants are given enough time to correct their mistakes and actually finish tasks. Any task can be made difficult enough that participants will not finish it within a given amount of time, so it is imperative to pilot studies to identify an appropriate amount of time to allocate. A corollary, however, is that it is not difficult to run a study in which at least some participants make a particular error of interest.

Training may have limited effectiveness: In the Glacier study, participants in the `final` condition were unable to correctly follow the advice we gave them on using `final`, despite having both documentation and a relevant page from a textbook. This is an example of the *training* challenge. Likewise, in the second part of the study, they frequently failed to identify that the class they were working on was immutable, despite having just spent time studying immutability. This leads to two lessons. First, attempts to change programmer behavior with only training materials, without actually modifying the tools programmers use, may have limited effectiveness. Second, in retrospect, considering our observations teaching Obsidian (Section 6), the training might have been more effective if we had required participants to do exercises with the new knowledge rather than assuming that they could read documentation and follow directions.

Bias toward control condition may be acceptable: The effect of tool-based interventions can be so dramatic that it is much better to potentially bias the study toward the control condition than it is to introduce threats to validity or make the study harder to execute. For example, we might have seen even more errors in the second part of the experiment if we had not previously trained the participants in immutability, but that would have required either getting a second set of participants or changing the task order. If we had conducted the programming tasks first, then those tasks could have served to bias the *other* set of tasks. Because recruiting participants is challenging (*recruitment* challenge), we opted to do both the immutability-specification and the immutable-class-programming tasks with one set of participants.

Replication materials for the study can be found online [23].

5 FORMATIVE STUDIES FOR OBSIDIAN

Obsidian was a much larger language design project than Glacier was, so there were many more design questions to address. We started with an analysis of proposals for blockchain applications and by studying bugs that had significant implications on existing blockchain platforms. We identified hypotheses for technical approaches that would provide safety properties to address the key problems we identified. Then, we conducted formative studies to explore whether we could design a language to be as usable as possible while still achieving our safety goals. This section focuses on how those formative studies informed the language design. In the next section (Section 6), we show how we used summative studies to assess to what extent we had achieved our usability objectives.

We conducted formative studies, which consisted of traditional programming tasks as well as natural programming tasks. In various studies, we used our adapted Wizard of Oz method, back-ported our design to the context of Java to isolate our research questions, provided participants with tasks spanning a range of difficulties, and divided larger tasks into subtasks. The formative studies spanned a variety of central design questions, which corresponded to usability risks we identified in our early designs:

- How should lexical scoping should work for states? (Section 5.1)
- How should the language help programmers manage fields that enter and exit scope when state changes? (Section 5.2)
- How could programmers use *permissions* to express different kinds of references to objects? (Section 5.3)
- How should the language represent the relationship between typestate and permissions? (Section 5.4)

Due to the complexity of each design problem, we refer readers to the individual sections for details of the findings.

In this section, we describe studies that helped us identify a suitable design and iterate on our initial design ideas for Obsidian. For each study, we identify our research questions, methodology, and results. We started by assuming that

Topic	Participants	Methods
Basic design of typestate (Section 5.1)	P1–P12	Natural programming
Fields in states (Section 5.2)	P20, P66–P68	Natural programming; Usability study
Permissions (Section 5.3)	P14–P19	Natural programming; Usability study
Typestate and ownership approaches (Section 5.4)	P21–P25	Usability study
Summative usability study pilots	P26–P34	Usability study
Summative usability study (Section 6)	P35–P40	Usability study
RCT pilots	P41–P44	RCT
RCT (Section 7)	P45–P65	RCT

Table 3. Summary of all Obsidian user studies described in this paper and their participants. Participant P13 was in pilot studies. Participant numbering is consistent with prior distributed drafts, e.g., [21].

we would use *typestate* to achieve the desired safety guarantees but that expressing typestate in a usable way would require substantial iteration with users. The latter assumption was based on past work on typestate systems, such as Plural [10] and Plaid [87], which researchers had found were difficult for users to use. All of the studies were approved by our IRB. Because we needed skilled programmers, we recruited from appropriate academic programs, by posting flyers, and by contacting our acquaintances. Except where noted below, we paid participants $10/hour for participating. Materials used in the studies can be found in the replication package [24].

Although Figure 1 uses the final version of the language, because the formative studies were done earlier, they use code from earlier versions of the language. In this way, the reader can see how we changed the language as a result of the user studies. For example, Figure 6 shows different approaches that we considered using for declaring local variables.

Table 3 summarizes all the Obsidian user studies that are described in this paper.

5.1 Basic design of typestate

In order to minimize assumptions regarding how Obsidian should best represent typestate, we conducted a *natural programming* study, which we described in an earlier paper [4]. We focus here on two of the research questions we had:

- Are states a natural way of approaching the challenges that arise in blockchain programming?
- Which (if any) of our proposed ways of presenting states and state transitions is most understandable and usable by programmers?

These are examples of the *naturalness* research question in §1 (“How can we obtain insights as to what language designs will be natural for programmers?”). We gave participants a description of a voter registration system, in which we would investigate to what extent state machines were a natural way to write smart contracts. The first task used a natural programming methodology: we asked participants to implement the system using pseudocode using any language features they wanted to solve the problem. Next, we gave participants a state diagram that modeled the system, and asked them to modify their pseudocode to include states and state transitions. In the third task, we gave participants a two-page Obsidian tutorial that described state blocks. However, the tutorial omitted any description of how state transitions should be written; we gave participants an Obsidian program that implemented the voter registration system but which omitted state transitions. We asked participants to fill in the missing transitions by

```
1  contract C {
2    state Start {
3      transaction t(int x) {
4        ->S1{x1 = x};
5        toS2();
6      }
7    }
8
9    state S1 {
10     int x1;
11     transaction toS2() {
12       ->S2{x2 = x1};
13     }
14   }
15
16   state S2 {
17     int x2;
18   }
19 }
```

(a) Option 1. The dynamic state (not the lexical structure) governs which transactions may be called. For example, line 5 calls `toS2()` even though `t` is lexically in the `Start` state and `toS2()` is defined in `S1`.

```
1  contract C {
2    state Start {
3      transaction t(int x) {
4        ->S1({x1 = x})
5        if in S1 {
6          ->S2({x2 = x1})
7        }
8        if in S2 {
9          ...
10       }
11     }
12   }
13
14   state S1 {
15     int x1;
16   }
17
18   state S2 {
19     int x2;
20   }
21 }
```

(b) Option 3. Fields can only be referenced by code that *lexically* is in the state in which those fields are defined. An `if in` block can be used to enclose code that must reference fields of other states, as in line 6.

Fig. 4. Two of the options given to participants in the *basic design* study.

inventing their own syntax to do so. In the fourth task, we gave participants three options for the syntax and semantics of state transitions and asked them to use each option once in an example program we provided. The first and third options are explained in of Figure 4. An additional option involved a constructor in each state that would be invoked on transitions to that state and a rule that no code could follow a state transition.

Finally, in a fifth task, we asked participants to select one of the three options and use it to complete the voter registration program they started earlier.

We recruited a convenience sample of seven participants, most of whom were computer science undergraduates. Each participant was given a description of a program to implement and one hour to complete the implementation. We paid participants $10/hour for their time.

Only two participants invented syntax denoting states and state transitions; the rest used a conventional approach, such as an enumerated type. However, many of the approaches the remaining five participants used were unsafe, helping to justify using typestate to improve safety. For example, creating separate lists for unregistered and unregistered citizens results in the possibility of citizens appearing on both lists.

We asked six of the participants to modify their pseudocode to use states. Two created explicit state blocks with states and variables nested inside. The remaining four either maintained global state for each citizen, or gave each citizen a state field, or created empty, immutable states at the top of the program. Although the instructions forbid allowing duplicate registrations, several participants did not check for existing registrations before processing applications.

Transaction lexically *nested inside* state declaration:

```
contract Wallet {
   state Empty;
   state Full {
      int balance;

      // spend() is nested inside the declaration of the state it belongs to.
      transaction spend(Wallet@Full this) {
        // use 'balance'...
      }
   }
}
```

Transaction lexically *outside* state declaration:

```
contract Wallet {
   state Empty;
   state Full {
      int balance;
   }
   // spend() is not nested in Full, even though it can only be called in Full state
   transaction spend(Wallet@Full this) {
      // use 'balance'...
   }
}
```

Transitions when inside a state could be confusing:

```
contract Wallet {
   state Empty {
      transaction fill(int amount) {
        ->Full(balance = amount);
        // Now balance should be in scope, since new state is Full
      }
   }

   state Full {
      int balance;
      // ...
   }
}
```

Fig. 5. Although participants preferred to have transactions nested inside state declarations (the first alternative), this desire conflicted with the need for transactions only reference fields that were in lexical scope.

Regarding the syntactic choices we offered in the third task, three participants preferred state constructors (part (a) in Figure 4), one preferred nested state blocks (part (b) in Figure 4), and the remaining three either did not indicate a preference or did not complete this task.

Although most of this study focused on participant *behavior*, we took the opportunity to also ask participants for their syntactic *preferences*. Five participants preferred a syntax where all the actions of a state must be lexically encapsulated in that state, as in the first alternative in Figure 5. Likewise, P4 felt it should not be permitted to call transactions from one state while lexically in another state: "I'm calling `S1`'s transaction from code for `Start`."

This preference led to a conflict in the design. We found through work with example programs after the study that Obsidian needed to support transactions that could be executed in several different states. For example, in the third example of Figure 5, line 5 may reference `balance`, even though line 5 is lexically enclosed in the `Empty` state, in which `balance` is not in scope. This represents a conflict between a syntactic preference and an expressivity concern.

Participant	Time
P20	54m
P66	1h 39m
P67	1h 35m
P68	1h 41m

Table 4. Times spent by the four participants in the Fields in States study.

Another difficulty with the constructor-based approach is that states might be entered for a variety of different reasons, requiring different code to run after the transitions. This makes the constructor-based approach likely too inflexible. These challenges underscore the importance of sufficient example-based work before conducting user studies; it is easy to design a study that provides plausible options that turn out to not support critical use cases.

This problem led us to run the study described in Section 5.2. As a result of that study, we addressed the conflict by requiring that transactions are lexically *outside* of state declarations, like the second example in Figure 5. Future IDE tools could show all transactions that are possible for an object in a given state, even though their declarations are lexically outside that state's declaration.

5.2 Fields in states

States in contracts can have different sets of fields, so transitioning can cause some fields to exit scope and others to enter scope. For example, in Figure 1, the `Full` state has the `inventory` field, but the `Empty` state has no fields. This study used *natural programming* and *code understanding* methods to investigate how users specify cleanup of old fields and initialization of new fields when invoking state transitions.

We recruited participants until the data we were obtaining from recent participants duplicated data we had obtained earlier. This led to recruiting four participants. All were Ph.D. students studying software engineering. They had an average of seven years of programming experience (ranging from three to fifteen years) and an average of 1.5 years of Java experience. Two identified as male, and two identified as female. We did not limit their time in completing the tasks. The mean time to completion was of 1 hour, 27 minutes. The times per user are shown in Table 4. Due to a miscommunication between co-authors, the participant identifiers are not contiguous, but the experiments occurred sequentially over about a month.

In **Part 1**, we gave participants a state transition diagram for a `Wallet` object, which could hold a license and money, and which had four states corresponding to the possible combinations of contents. Participants were also given code partially implementing the `Wallet`, with several TODO comments asking participants to invent code to add money to the `Wallet`, remove money from the `Wallet`, etc. Participants were told that the money and license should be thought of as assets, so they could not be duplicated, used more than once, or lost. The code they were given was in a language similar to Obsidian but which used some keywords that would be more familiar to a Java programmer, such as `class` instead of `contract`. As such, this was a staged natural programming study, since we progressively gave participants more detail about the language we were designing.

All four participants prepared assets for a state transition before making the state transition (corresponding to option (2) in Part 2 below, `S::x = a1; ->S`). Two participants felt they needed to write code to handle failures during the asset preparation stage, which might lead to an improperly initialized state upon transition. One of them suggested a try-catch type wrapper for the asset preparation and transition phases.

In **Parts 2** through **4** of the study, participants were given several options. Then they were asked to implement each of the options within a given partially-implemented transaction. Finally, they were asked for their preferences.

Part 2 compared approaches for initializing fields in states during transitions. Options were:

(1) Assets are assigned to fields in the transition, e.g. `->S(x = a1)` assigns the value of `a1` to field `x` of state `S`.
(2) Assets are assigned to fields *before* the transition, e.g. `S::x = a1; ->S`.
(3) Assets are assigned to fields *before* the transition, but the fields are in local scope even though the state has not changed yet, e.g. `x = a1; ->S`.
(4) Assets are assigned to fields *after* the transition, e.g. `->S; x = a1`.

The participants successfully used all the approaches, but most of the participants preferred assigning assets to fields *before* the transition with destination state scoping (option 2). Before the study, Obsidian supported only atomic assignment (option 1, shown in Figure 1 on line 22). The results of these two parts motivated a language change: Obsidian now also supports option 2.

Part 3 presented two options for handling assets when transitioning from a state with an asset to a state without it:

(1) The transition evaluates to a collection containing the old assets, e.g. `x = ->S` indicates that `x` is assigned the leftover assets after the transition to state `S`. If the current state is unknown statically, the contents of the collection are determined dynamically.
(2) The transition evaluates to a tuple, e.g. `(x = a1) = ->S` indicates that `x` will be assigned the asset `a1` which is not present in state `S`.

There was consistent confusion about which leftover assets are assigned to option 1's collection after a transition. All participants understood the need for both options in certain cases, but would choose the tuple-like collection for more control and explicitness when the use of either approach is acceptable. We would like to implement this approach in the future but so far have not prioritized it, since the existing approach (described in Part 4, option 1), which requires that ownership of assets be surrendered before transitioning, has been effective for participants.

Part 4 focused on releasing assets owned by state fields when transitioning to states in which those fields do not exist. In contrast to *part 3*, this approach added the option of releasing assets before the transition. The choices were:

(1) Assets must be released before the transition, e.g. `release(a1); ->S`.
(2) The transition evaluates to a tuple of assets that are no longer owned, e.g. `a1 = ->S`. The tuple is necessary since there may be several asset-owning fields going out of scope, so there would be one element per field.

All the participants understood the options and implemented them without mistakes. Implementing using option 2 (evaluating to a tuple) enables both approaches, so participants were asked to indicate scenarios where one option would be preferred over the other. The participants consistently indicated that assets should be released before a transition if they are no longer needed; otherwise, they should evaluate to a tuple. This helped us prioritize our features, since releasing assets before the transition seemed to suffice.

5.3 Permissions: a qualitative study

Soundly enforcing typestate requires knowledge about all references to an object, which is afforded by a *permission system*. [9]. Permission systems allow the programmer to express what a particular reference can be used for (and therefore also what it *cannot* be used for). Is there a permission system that users can understand and use effectively (a question of *naturalness*)? If so, what can we learn from users about how to design it (a question of how to iterate on

designs)? In this work, we conducted the first studies (of which we are aware) in which people other than the designers of the system were asked to use a permission system to restrict references in a programming language. We found that our initial system design was surprisingly difficult to use, and iterated the design until it was more successful.

In order to study permissions while mitigating the *interdependency of features*, *training* and *recruiting* challenges, we extracted the permission system from Obsidian and re-cast it in Java as a set of annotations. We conducted a Wizard of Oz study where participants received documentation on a Java extension and the experimenter gave simulated compiler error messages. This approach minimized training time for participants, minimized implementation cost for ourselves, and allowed us to isolate this design decision from many others that would have otherwise distinguished the language from Java. At this point in the development of Obsidian, we assumed that it would be best to separate the notions of permissions and typestate; this approach was reflected in the study materials but may surprise a reader who has studied Fig. 1, which reflects the final Obsidian version, which combines the two. The training materials explained the annotations: `@Asset`, which applied to classes; and `@Owned`, `@Shared` (`@Shared` means there are no restrictions because the object has no owner), and `@ReadOnlyState` (restricting state modification), all of which applied to references.

Our objective was not to obtain as much data as possible about the current design, but rather to identify a forward path through the design process. Relevance of data depends on the fidelity of the prototypes used; because this study was with a relatively low-fidelity prototype, we conducted a relatively small number of trials to help us make key design decisions. As with the prior study, we continued recruiting participants until most of the new data duplicated earlier data and we had identified a concrete plan for continued language revision and evaluation. For this study, this resulted in recruiting six participants (P14–P19). They had a mean of six years of programming experience (ranging from three to nine years), a mean of one year of professional experience, and a mean of two years of Java experience. All identified as male.

The study included five parts. Since our goal was to identify as many usability problems as possible in each trial, we revised the design and instructions after each participant. This approach (of changing the tasks between participants) is an accepted practice in usability studies in order to obtain the most useful data from the study [31, 52]. The first three participants were given 1.5 hours to do the first four parts; the last three were given two hours to fit in a fifth part of the study. An experimenter was available to answer questions.

Part 1. To motivate the need for language features to prevent bugs, we gave participants a 163-line Java medical records system and asked the first two participants to find a bug in which a patient could refill the prescription more times than specified. The first participant did not find the bug within 30 minutes; the second did so just as time expired. To conserve time, we gave the other participants five minutes to inspect the code and then we explained the problem to them.

We conclude that at least some programmers who use traditional languages would have difficulty detecting the kind of bug that Obsidian prevents. This provides further evidence that if users use Obsidian, the compiler will help them detect bugs that otherwise might go undetected.

Part 2. We told participants we would prevent the previous bug by distinguishing between two kinds of references. "Considering an object *o*: *Kind #1*: There is only one reference of kind #1 to *o* at a time. *Kind #2*: There may be many references of kind #2 to *o* at a time." We asked participants to propose names for the two kinds of references. Note the careful language avoiding bias toward specific vocabulary. Participants' name suggestions included:

Kind #1: KeyReference, UniqueReference, Owned, Singleton reference, Resource handle, @default

Kind #2: DuplicateReference, ForeignKeyReference, Borrowed, Flyweight pattern reference, const pointer

The results were too inconsistent to justify an particular choice in the language; all the suggestions were distinct, and some of them were not appropriate in context (*unsound proposals* challenge). Obsidian uses `Owned`, which is at least consistent with one suggestion, and `Unowned`.

Part 3. To evaluate the usability of ownership, we gave participants an ownership tutorial and told them we had chosen [*no annotation*, `@ReadOnly`] (first participant) or [`@Owned`, *no annotation*] (later participants) as keywords. We asked them to modify the code from Part 1 to fix the bug. We hoped participants would require that `Prescriptions` deposited in a `Pharmacy` be owned and that the `Pharmacy` take ownership; thus, a deposited `Prescription` could not be deposited in a second `Pharmacy`. Completion times ranged from 3 minutes to 40 minutes (*variance* challenge). Two participants did not finish, one of whom we stopped after 38 minutes to prioritize other tasks.

We were surprised that many of the participants found this task very difficult. We expanded the tutorial to include a practice section for later participants. In general, participants were not prepared to use a type system to address a bug that they thought of in a dynamic way. For example, P16 wrote `if (@Owned prescription)`, attempting to indicate a dynamic check of ownership. We asked participants who wanted to use dynamic approaches for enforcement to use the language feature instead. P14 commented "I haven't seen…types that complex in an actual language …enforced at compile time."

P17 had trouble guessing what the compiler could know, expecting an interprocedural analysis (which would be non-modular). For example, in a case where an owned object was being consumed twice, P17 expected the compiler to give an error on the second `spend` invocation. Instead, because the second invocation was inside a helper method, the compiler reported the error on the invocation to the helper method, which took an owned argument and invoked the second `spend`.

P17, P18, and P19 had difficulty determining which variables should be annotated `@Owned`. In one case, a lookup method took an object to search for, but P17 specified that it should take an owned reference. Then he was stuck after invoking it: "How can I get the annotation back?" But this was impossible except via adding another method, since he had already given away ownership. Likewise P17 was confused by whether accessors should return owned references. Mistakes could be costly. For example, P19 unnecessarily annotated as `@Owned` a class that was contained in a collection, which caused a problem iterating through the collection. He made the reference to the current list element `@Owned`, which would require removing each item from the collection when iterating over it in code that was not supposed to modify the container at all.

Parameter-passing and assignment were common points of confusion. P18 asked what happens when passing an `@Owned` object to a method with an unowned formal parameter (ownership was not passed in this case). P19 said, "when I [annotate this constructor type `@Owned`], I'm not sure if I'm making a variable owned or I'm transferring ownership." P17 was surprised that assignment from an owned reference to an unowned-type variable did not transfer ownership. We later addressed this confusion by making assignment always transfer ownership; participants in later studies were generally not confused about which assignments transfer ownership.

From this portion of the study, we came to two general conclusions. First, the semantics of ownership needed to be as explicit and as simple as possible. This likely generalizes to many different kinds of complex language constructs: implicit behavior, although sometimes convenient for experts, can be baffling to novices. When the behavior can be made explicit without making the language inconvenient for experts, that should be done. Second, language design

decisions that have structural implications (as is the case for ownership) require substantial high-level training; we refined the training materials in future studies to give more explanation and examples.

Part 4 introduced the notion of *assets*. After a tutorial explaining the properties of assets, participants were asked to invent code that could indicate a particular owned reference was *intentionally* going out of scope. Two participants suggested `@Disown` and `free` to abandon owned references; the rest did not have time to answer or had no suggestions. We chose `disown` for Obsidian, since `free` has additional memory management connotations that are not relevant here.

Part 5 introduced typestate, starting with the fourth participant. Participants read 2.5 pages on typestate in Obsidian (as it existed then), including `@ReadOnlyState`, `@Shared`, and `@Borrowed` (which was for temporary ownership transfer in invocations). Ownership was the default, so no `@Owned` was needed. The tutorial also explained `available in` and `ends in`, which at the time specified state assumptions and guarantees for methods (before we changed to using `this` parameters instead, e.g., as on lines 19 and 27 of Figure 1). Then, they were asked to annotate uses of `Bond` in a 212-line Java program implementing a financial market. They were told to use ownership and state specifications whenever possible.

Consistent with Part 3, some participants were more comfortable with a dynamic perspective on ownership rather than a static one. P18 felt that `ends in` declarations were redundant with the transition code already in the method implementations, but these declarations allow separation of interface and implementation and modular checking. P19 wanted to use borrowing to represent the notion that the `BondMarket` owns a `Bond`, but an `Investor` borrows it for a while. In fact, borrowing was only appropriate for the duration of a method invocation. We later changed the design of the formal parameter syntax to remove the need for `@Borrowed`; now, if no ownership change is specified (via the » operator), ownership remains unchanged.

P19 required significant prompting by the experimenter to make maximum use of typestate. First, P19 added annotations on methods but not on any variables. After prompting, he added dynamic checks in one place but required prompting to add static typestate specifications. This suggests that tools may be needed to help users obtain the most benefits from the language. On the other hand, P18 specified `@Asset` on `Bond` without being asked to do so, explaining "because it's something important and I don't want to get it out of scope…"

Overall, understanding the limitations of the type system and compiler may be an obstacle for some people. Users will need training to reason about what typestate can do, but the observations above motivated language changes that simplified the design without lowering the expressivity or safety. Tools could mitigate the limitations of traditional type systems by providing sophisticated static analyses rather than taking a traditional type checking approach (as Obsidian does), and by providing detailed, explanatory errors.

5.4 Comparing typestate and ownership approaches

We were interested in evaluating a new approach we invented, which was motivated by the confusion we observed in the prior study (in part a question of *naturalness* and in part a challenge of *training*). We invented a new approach: fuse the notions of ownership and typestate in order to simplify the type system, and the next study refined this design. This design has the benefit of eliminating `Shared` references that also specify typestate, which would then have to be disallowed to preserve soundness. Thus, the type `Bond@S` is always implicitly an owned reference for any state `S`, and users can write any permission instead of `S`, as in `Bond@Unowned`.

We were also interested in another usability concern. Consider *Approach 1* in Fig. 6. A reader of line 1 might expect that the type of `bond` would always be `Bond@Offered`. In fact, after line 2, the type is `Bond@Sold` due to the call to

Approach 1: traditional declarations

```
1  Bond@Offered bond = new Bond();
2  bond.buy(...);
```

Approach 2: types in variable names

```
1  Bond bond@Offered = new Bond();
2  bond@Offered.buy(...);
```

Approach 3: static assertions

```
1  Bond@Offered bond = new Bond();
2  bond.buy(...);
3  [bond@Sold];
```

Approach 4: no states in local variable declarations

```
1  Bond bond = new Bond();
2  bond.buy(...);
3  [bond@Sold];
```

Fig. 6. Variable declaration approaches

`buy`. A fundamental aspect of Obsidian is that ownership can change, so if a variable declaration includes any ownership information, the variable's ownership status may later be inconsistent with its declaration.

We initially invented two possible approaches to address this problem. One idea involved incorporating types into variable names, shown in *Approach 2*. The annotations pertain to the *current* type rather than the *new* type. The reader would have to look at only the most recent operation to infer the new type of a variable rather than having to potentially read the whole sequence since the declaration.

Approach 3 represents another idea: adding static assertions. Line 3 shows a static assertion that `bond` references an object in state `Sold`, which serves as documentation. Unlike traditional assertions, however, the compiler checks correctness. The intent is to make it easier for programmers to determine the types of variables.

We conducted studies with participants in the first three conditions. Inspired by observations of those participants, we invented *approach 4*. This approach is like approach 3 except that it *removes* state specifications from local variable declarations. The removal was not part of the original design but was inspired by early results of this study.

5.4.1 Participants. We required that participants be familiar with Java and we administered a simple Java pre-test. We recruited five students (P21–P25). Based on self-reports, they had an average of about four years of Java experience (ranging from one to ten years) and an average of one year of professional (paid) software development experience (ranging from zero to three years).

5.4.2 Procedure. Participants spent between 1 and 1.5 hours on the study. We used a Qualtrics survey to ask participants a series of questions regarding Obsidian programs, but the study took place in a lab and an experimenter was available to answer questions. The survey both taught aspects of the language and provided an opportunity for assessment. Most of the questions were typical *code understanding* questions, which gave snippets of code and asked whether the compiler would give an error or what the code meant. Rather than assigning participants to conditions randomly and ensuring equal numbers of participants in each condition, we conducted each trial according to the particular questions we wanted to obtain insight on at that time. We assigned P22 to approach 1, P21 to approach 2, P23 and P24 to approach 3, and P25 to approach 4.

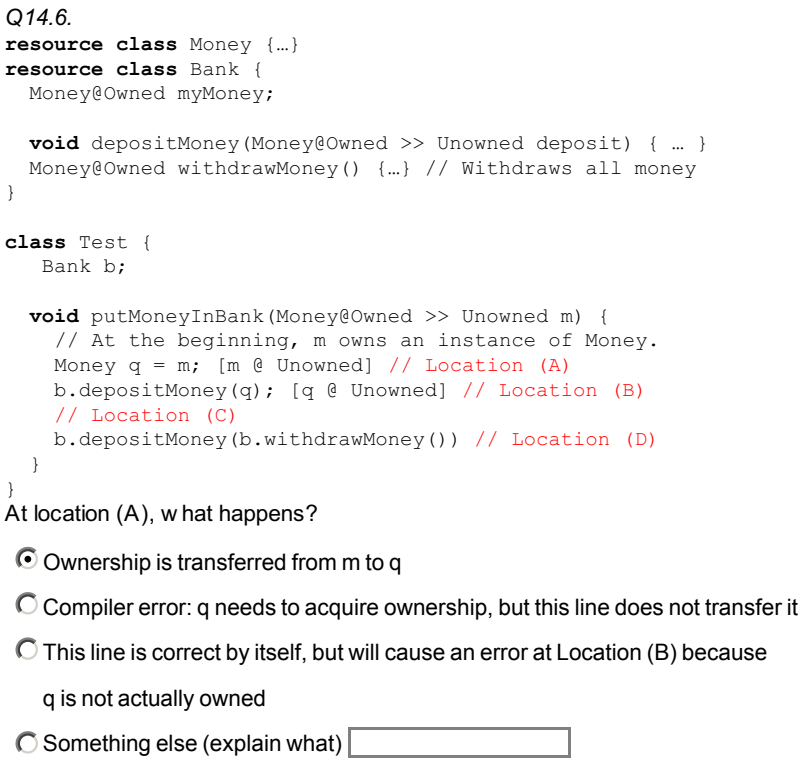


Fig. 7. An example question assessing understanding of ownership transfer. The correct answer is selected, since assignment transfers ownership.

5.4.3 Results and Discussion. P22, who was given approach 1 (with permissions and states specified only in declarations), tried to guess the compiler's behavior, saying things like "If the compiler was smart…". For example, P22 expected that the language would infer an implicit `@Off` in the declaration `LightSwitch s1 = new LightSwitch()`. P22 also expected that although changes of state were permitted via transactions, state-mismatching assignment to variables would be forbidden, even though approach 1 assumes that states can be inconsistent with type declarations. This approach would be inconsistent and P22's confusion suggests that the type-declaration approach is problematic.

Including types in variables names seemed to be confusing as well. P21 expected that ownership was *not* passed into method calls even when an owned reference was passed. P21 was also surprised that no ownership annotation meant that there was no ownership, instead expecting this to mean that ownership was unknown.

Participants in condition 3 seemed to do much better. For example, although the materials did not use the word *assertion*, P23 observed that the annotations were assertions. P23 liked the system, commenting "Perfect, I like this, this is very nice. I wish Java had this; it would have saved me a lot of bugs." As we obtained additional confidence in the value of approach #3, we added additional material. For P24, we changed assertions to use `@` rather than the initial » so that we could use » to specify type changes in transaction parameters. With P25, we used `?` to indicate lack of static state knowledge. We later simplified the system because this approach was ambiguous, leaving only notations `Owned`, `Unowned`, `Shared`, and unions of specific states (separated with `|`).

P24 was confused because state specifications on local variables were redundant. For example, in `LightSwitch@Off s = new LightSwitch()`, the `@Off` portion is redundant because the compiler already knows the state of the new object due to the constructor's declaration. To resolve this, we added approach 4, removing typestate and permission annotations from local variable declarations; in contrast, permissions are always specified for fields and formal parameters. In those cases, the annotations are important because they constrain types of variables at the end and beginning of transactions. This approach eliminated those as sources of confusion for P25. However, P25 was still confused by whether ownership was transferred in one case after an owned reference was passed to a transaction because of a failure to read the type of the transaction that was being invoked, in spite of a nearby annotation indicating the final permission. We concluded that this difficulty would likely be addressable with a small amount of training, so we proceeded with approach 4 in our final design.

In summary, this study motivated the removal of state specifications from local variable declarations and provided initial evidence that static assertions are likely to be a convenient way for programmers to specify states and permissions of local variables. We also obtained evidence that with these other changes, static state assertions are understandable by current Java users with little extra training.

5.5 Threats to validity

The studies share common threats to validity, many of which correspond to the *external validity* challenge: our participants may not be representative of the population of blockchain programmers; we had limited numbers of participants in each trial; and our tasks may not reflect the reality of blockchain programming. We believe, however, that the population of likely language users is *more* skilled than our participant population, which mostly consisted of students, so if the students are successful in completing tasks, that aspect of the result is likely to generalize. We did not seek to identify all possible usability problems, but rather to identify the most common and severe ones associated with particular design decisions so that we could try to address them. Because there were so many different design decisions, we focused on those for which we had prior evidence that there might be usability problems.

6 SUMMATIVE USABILITY STUDY OF OBSIDIAN

We finished a complete language design, including a formal proof that the design had the formal safety properties we claimed it did. We also completed implementation of the compiler and runtime environment. In order to assess whether our changes to Obsidian had resulted in a language in which programmers could be effective, we designed a summative usability study. As a usability study, this was a *qualitative* study that sought to identify remaining usability barriers as well as to find out whether participants could complete relevant programming tasks. In particular, our prior studies had identified serious usability problems, so we wanted to know whether completing relevant programming tasks was feasible at all for our participants. This study preceded an RCT comparing Obsidian to Solidity. That study is described in Section 7.

We gave six participants the complete Obsidian language, including its compiler, and asked them to complete three programming tasks. We were interested in whether the participants would experience the same usability problems as the prior participants and whether there were sufficiently serious usability problems left to prevent them from completing their tasks. All of the participants were able to complete the first programming task, but some of the participants ran out of time before completing the other tasks. The second task focused on ownership transfer, since our earlier study found significant usability problems in our earlier prototype. All of the participants who started the second task completed it, suggesting that we had successfully improved the usability of ownership. The third task, which was more open-ended,

presented additional challenges for participants in part because it required reasoning about how the states of different objects related to each other.

The design of the study was informed by several of our methodological contributions, and thus also served to assess their value. Key aspects of the PLIERS process that were helpful in designing the study included integrating training into the study to allow recruiting participants who only had Java language experience, not Obsidian experience; recruiting from a population that included many people with some professional experience; and using multiple programming tasks, rather than one long one.

6.1 Participants

We solicited experienced Java programmers to take a short screening test online, which took an average of about 9 minutes to complete. We accepted into the three-hour study only those who answered at least five of six basic Java questions correctly. The six questions concerned: Java constructor syntax; the definition of encapsulation; whether changes to a list through a reference would be visible through another reference; whether methods in interfaces may include bodies; whether abstract classes may be instantiated; and whether concrete subclasses of abstract classes must implement methods that were abstract in the superclass. Of 18 completed surveys, 11 people met our screening criteria. We got six participants (P35-P40), whom we compensated with $50 Amazon gift cards. The participants had an average of 9 years of programming experience, 2 years of professional experience, and 2 years of Java experience. One self-identified as female; the rest identified as male. Figure 8 shows an example question. A copy of the screening instrument is included in the supplement.

6.2 Procedure

The previous studies focused on particular aspects of the design, in many cases by giving participants languages that were not precisely Obsidian. To evaluate Obsidian, we conducted a usability evaluation. Because Obsidian provides stronger safety guarantees than existing languages such as Solidity, and because of our prior experience showing that it would be very challenging to develop a linear type system that would be usable at all, we focused the study

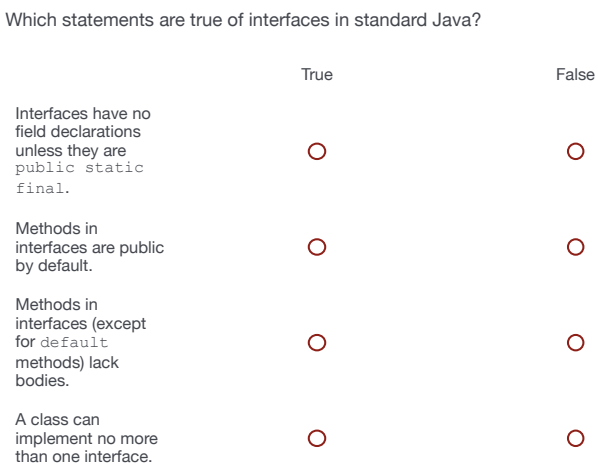


Fig. 8. An example question from the screening test.

on examining whether people could effectively complete tasks. Our randomized controlled trial (RCT), described in Section 7, focused on whether people could complete tasks faster than in existing languages.

The experimenter gave low-level guidance, such as how to invoke the compiler. Also, the experimenter provided assistance that simulated more mature tools. For example, when a participant attempted to debug an error that was reported on line 38 by examining line 33, the experimenter pointed out the discrepancy, since the IDE we provided did not highlight the appropriate line.

After completing the tutorial, which included seven programming exercises, we gave participants starter code for the three main tasks, described below. Although participants used the compiler, they were not given tests or a runtime environment, since the focus of our usability study was the type system (recall that Obsidian is designed to detect as many bugs as possible at compile time, since runtime detection may be too late to ensure safety). Although the first two tasks were short in order to reduce variance, we allowed the third task to be more open-ended to see whether participants would be able to complete a more challenging task.

The first task, *Auction*, simulated an English auction, in which bids are public, and the bidder who offers the highest price must pay that price for the item. We added the additional constraint that bids were required to come with `Money` so that bids could be guaranteed to be viable (a bidder could not issue a bid and then fail to pay for the item). As a starter task, we asked participants to finish implementing `createBid`, requiring them to invoke a constructor. They also needed to finish implementing `makeBid`, which records a new bid from a client. In `makeBid`, we were interested in whether they initially wrote code that accidentally lost the previous bid, which held the associated `Money` (before receiving a compiler error), indicating that Obsidian's typechecker had helped them avoid losing track of an asset. Figure 9 shows the Auction task.

The second task, *Prescription*, corresponded to the medical records system in the Permissions study section (§5.3); we were interested in whether our improvements enabled participants to reason more effectively about the code than we had observed in the previous studies. We asked participants to fill in the type signature for the `consumeRefill` and `depositPrescription` transactions, which mirrored the previous study. We also asked them to complete the implementation of `fillPrescription`.

The *Casino* task was more open-ended and included directions and requirements for what operations should be supported, as well as low-level starter code, such as implementations of `Money` and `Bet`. It asked participants to implement a `Casino` that takes bets on games. When games are complete, the casino enables winners to collect their winnings. The requirements were as follows:

(1) If a `Bettor` predicts the outcome correctly, the `Bettor` gets twice the `Money` they put down. For example, if `Bettor` b puts down 5 tokens on the correct outcome, they should receive 10 tokens after the Game is played.
(2) If the `Bettor` predicted incorrectly, the `Casino` keeps their tokens.
(3) Bets can only be made before the `Game` starts.
(4) Winnings can only be distributed after the `Game` is finished.
(5) `Bettor`s must collect winnings themselves from the `Casino` after a Game by calling code, which you need to write. Until winnings are collected, the `Casino` keeps track of them.
(6) A `Bettor` can have one active bet per game. If a `Bettor` bets more than once, their original bet should be replaced by the new one and any previous bet should be refunded.
(7) A `Bettor` MUST put down tokens at the same time that they're making a Bet.

```
1  main asset contract Auction {
2    Participant@Unowned seller;
3
4    state Open;
5    state BidsMade {
6      // the bidder who made the highest bid so far
7      Participant@Unowned maxBidder;
8      Money@Owned maxBid;
9    }
10   state Closed;
11
12   ...
13
14   transaction bid (Auction@Shared this, Money@Owned >> Unowned money, Participant@Unowned bidder) {
15       if (this in Open) {
16         // Initialize destination state, and then transition to it.
17         BidsMade::maxBidder = bidder;
18         BidsMade::maxBid = money;
19         ->BidsMade;
20       }
21       else {
22         if (this in BidsMade) {
23           //if the new bid  > current Bid
24           if (money.getAmt() > maxBid.getAmt()) {
25             //1. TODO: fill this in.
26             // Can call other transactions as needed.
27             maxBidder.receivePayment(maxBid);
28             maxBidder = bidder;
29             maxBid = money;
30           }
31           else {
32             //2. TODO: return money to the bidder,  since the new bid was too low.
33             // Can call other transactions as needed.
34             bidder.receivePayment(money);
35           }
36         }
37         else {
38           revert("Can't bid on closed auctions.");
39         }
40       }
41     }
42 }
```

Fig. 9. The Auction task. Code highlighted in yellow represents a correct solution; the rest was given to participants as starter code. Line 27 transfers ownership of the object referenced by `maxBid` to the `receivePayment` parameter. The new type of `maxBid` from then until line 29 is `Money@Unowned`. Line 29 re-establishes ownership in `maxBid` by transferring ownership from `money` to `maxBid`.

(8) If the `Casino` does not have enough tokens available to pay out winnings, the invocation to collect winnings can fail.

We also provided a sequence diagram to show participants what operations should be supported. In this way, we conveyed the requirements without also specifying the transaction signatures, since we wanted to see if the participants could infer those themselves.

We were primarily interested in participants' abilities to reason about ownership and typestate and to design architectures that could effectively use ownership.

6.3 Results and Discussion

Results for the tasks are summarized in Table 5. All the participants completed the first task (Auction). All the participants who spent less than two hours on the tutorial completed the second task (Prescription). All the participants who started the third task (Casino), which was substantially more complex than the other two, and had at least an hour available to work on it, finished it. Note that the two successful completion times for the third task were longer than the times that the other participants had available to spend on it. With P38, to assess to what extent the tutorial materials stood alone,

	Task completion times (hours:minutes)			
	Tutorial	**Auction**	**Prescription**	**Casino**
P35	1:31	0:13	0:18	1:01
P36	2:12	0:28	N/A	N/A
P37	1:03	0:33	0:46	0:36*
P38	2:18	0:46	N/A	N/A
P39	1:14	0:22	0:27	0:51*
P40	1:11	0:12	0:22	0:58

Table 5. Usability test results. * indicates insufficient time to finish the task. N/A indicates insufficient time to start the task.

the experimenter declined to answer Obsidian-related and debugging-related questions. However, this made the first task perhaps unrealistically difficult and lengthy, resulting in insufficient time for the other tasks.

In the **Auction** exercise, two of the six participants accidentally introduced a bug in which an asset was lost: they overwrote `maxBid`, which held money. The compiler gave an error message and they corrected their mistake, but if they had been using Solidity, its compiler would not have caught the bug. After P36, we slightly simplified the Auction exercise by removing a subtask and refactoring to inline a TODO that had been put in a helper transaction. The above times are adjusted to remove the extra time P35 and P36 spent on the removed task (1 and 8 minutes, respectively).

Some participants seemed to think carefully about ownership and wrote the correct code quickly. Others seemed to focus on satisfying the compiler, and their work took longer. For example, P38 got an error message after overwriting the owned `maxBid` reference, and "fixed" it with `disown`. This choice may be a result of weaker programming skills and lack of help in the tutorial; P38 took the longest on the tutorial, and was surprised to not be given a design diagram for the (< 300-line) Auction starter code. We changed the tutorial to emphasize that `disown` should be used to throw away assets.

In the **Prescription** task, as with other tasks, variance was large. For example, one reason for P38's long completion time was that P38 had used Python most recently and, despite the tutorial, sometimes wrote Python-like syntax, which did not parse (one example took four minutes to fix). At the time, we were hoping that participants would be able to complete the tasks entirely on their own, but in retrospect, we may obtain *more relevant* results by carefully providing appropriate help (which we provided to all the other participants).

We were interested in participants' ability to reason effectively about ownership. All of the participants who started Prescription were able to complete it. P37 encountered some difficulties due to shortcomings in Obsidian's support for dynamic state tests. Currently, Obsidian does not allow dynamic state tests to be used as arbitrary Boolean expressions, e.g. `if (x in S && e)` where `e` is an arbitrary Boolean expression. Likewise, `if (x not is Owned)` is not supported (perhaps this was inspired by Python's `is` operator). In the latter case, P37 developed some intuition: "Ownership doesn't feel like something I should be using in this way…" and restructured the code to check `if (maybeRecord in Full)`, which was correct. In another case, the compiler found a bug in which the code assumed that a collection must contain an element, a benefit of not allowing `null` in the language.

The *Casino* task was substantially more open-ended than the other tasks, requiring substantially more time, but participants who had a full hour for the task were able to finish it. Some participants defined states in the `Casino` contract (P35, P39), whereas others relied only on the states in the Game contract (P37, P40). Both approaches led to a lot of dynamic state tests, since the Casino object had to check to make sure the Game object was in an appropriate state. These checks could have been avoided if the different states of Casino had different typestate specifications for

	Solidity	Obsidian
Static solutions: correct solutions / attempts	N/A / 5	6 / 6
Dynamic solutions: correct solutions / attempts	2 / 6	1 / 3
Completed within time limit	3	9
Mean time among successful participants; [95% CI]	20 min.; [0, 45]	22 min. [12, 33]

Table 6. Summary of Prescription task results. $N = 10$ in each condition. Two Solidity participants tried both static and dynamic approaches, and one Solidity participant did not modify the starter code at all, resulting in 11 Solidity attempts.

their references to the Game, an idea that occurred to P40 in retrospect. This observation represents an opportunity for a future version of Obsidian in which states of owning objects are coupled to states of owned objects, reducing the need for dynamic checks.

We noticed that participants who did better on the "advanced Java" portion of our screening test seemed to complete tasks faster. We found that those test scores were positively correlated with completion speed in the Auction task ($r(4) = .96$, $p < .01$).

7 COMPARING OBSIDIAN TO SOLIDITY

We conducted a randomized control trial (RCT) comparing Obsidian to Solidity. The results of this RCT are described in detail in another paper [19], but in summary, we recruited 21 Java programmers and randomly assigned them to use either Obsidian or Solidity. This study used a tutorial and tasks that were similar to those in the summative usability study (Section 6); the Prescription task was modified to more exactly match the earlier permissions study described in Section 5.3.

Each study session lasted up to four hours and participants were compensated with a $75 gift certificate. We allowed them as much time as needed to complete the tutorial. Then, we gave 30 minutes for Auction, 35 minutes for Prescription, and any remaining time for Casino. Finally, we gave them a survey regarding their opinions.

7.1 Results and Discussion

In the **Auction** task, seven of ten Obsidian participants completed the task successfully. Two did not finish the task, and one did so incorrectly, accidentally refunding money to the wrong bidder. Two of the seven successful Obsidian participants had received compiler errors indicating that they had lost assets, suggesting that Obsidian had helped them avoid that bug. In contrast, of the ten Solidity participants, only two completed the task correctly. Seven said they finished the task but had bugs in their code; one ran out of time. The difference in success rates was significant with $p \approx .015$. That is, participants were more likely to finish successfully if they used Obsidian (odds ratio 0.053).

The Prescription task investigated whether participants could use ownership to statically address a security problem. Six of the ten Obsidian participants did so, suggesting that ownership is learnable. Six of the ten Solidity participants attempted a dynamic solution, but only two of them were able to finish it in the time available. In both conditions, some participants made `Prescription` mutable, even though that was explicitly disallowed by a comment in the program. We had selected an immutable design following standard security advice, but the results suggest that a *mutable* design for `Presciption` might have been more natural for some participants.

The **Casino** task was substantially more open-ended and offered more opportunities for mistakes. In part due to time spent on earlier tasks, only nine Solidity participants and five Obsidian participants had enough time to arrive at a

	Solidity ($N = 8$)	**Obsidian ($N = 4$)**
Had enough time to try Casino	9	5
Completed Casino with a program that compiled	8	4
Completed task correctly (no identified bugs)	1	0
Winnings collection emits error if Casino is out of tokens	5	2
Only used disown safely	N/A	0
Managed tokens correctly (not fabricating or losing them)	4	0
Mean completion time	37 min.	64 min.

Table 7. Summary of Casino task results among completed programs that compiled, showing correct solution rates among errors made by more than one participant.

solution with which they were satisfied within their four-hour time window; we did not set a separate limit for the task. We discarded data from one Obsidian participant who encountered a compiler bug.

Results are summarized in Table 7. Notably, the four Obsidian participants who finished the task all abused the `disown` keyword, despite verbiage in the training materials warning about this. The result is that they lost track of assets, since their usage suppressed errors that the compiler would have otherwise given. This warrants further investigation in how to safely provide language features that are necessary for some kinds of programs, but which nonetheless can be used unsafely.

The Obsidian participants spent significantly longer on the task than the Solidity participants did ($p \approx 0.02$, Mann-Whitney U test, $d \approx 1.9$). This gives an approximation of the cost of the stronger type system in implementing a software prototype (but perhaps not in implementing a production-quality system). Of course, this cost may be worth bearing, since the safety guarantees may result in a more efficient and safer software creation process by reducing the testing burden. However, this benefit may require either language modifications or more training to avoid the risk of abusing `disown`. Future work will need to investigate mitigating this risk and whether additional training and practice mitigate the cost of the stronger type system.

We asked participants several questions about their opinions of the languages they used in the study. Participants who were assigned to use Obsidian rated ownership as *more useful* than participants who used Solidity ($p \approx 0.002$, $d \approx 2.5$). However, the Solidity participants indicated that they felt they understood states better than the Obsidian participants did ($p \approx 0.04$, $d \approx 1.3$). Results are shown in Table 8.

	Solidity (N=6)	**Obsidian (N=8)**
How much did you like the language you used?	3.7 (0.82)	4.0 (0.53)
How well do you feel you understand the concept of ownership?	3.8 (0.98)	3.75 (0.99)
*How useful do you think ownership is?	3.0 (1.1)	4.88 (0.36)
*How well do you feel you understand the concept of states?	4.8 (0.41)	4.1 (0.64)
How useful do you think states are?	4.3 (0.81)	4.1 (0.64)

Table 8. Perceptions of ownership, states, and assets on a 1–5 scale (5 is best). Cells show average (standard deviation). * indicates that a Mann-Whitney U test shows a significant difference at $p < 0.05$.

The survey also asked for additional comments. Three Solidity participants wrote that they wished ownership were checked by the compiler (as is the case in Obsidian). Some participants using Solidity wished they had a notion of state. For example, one wrote:

> It also seemed like there should be some syntactic sugar for writing things like:
> ```
> enum State { Foo, Bar, Buzz }
> State s
> ```
> since they are so common.

Some Obsidian participants wrote that they appreciated the tutorial and exercises, and that ownership seemed natural after some practice.

7.2 Implications on PLIERS

The randomized controlled trial comparing Obsidian and Solidity served in part to evaluate Obsidian and in part to evaluate PLIERS. In this RCT, we were able to show a safety benefit of Obsidian in the Auction task, and were able to show that most of the Obsidian participants were able to use ownership successfully in the Prescription task. This shows that the tutorial method was mostly successful (though more success could likely have been obtained with more practice) and that the language design was effective overall (modulo the abuse of `disown` that we observed). Every study design involves making tradeoffs. The results here may show a tradeoff between training time and success rates; users of PLIERS will need to decide, based on their own design and research goals, how to balance the risks when designing their studies. However, the overall PLIERS design process did result in a language that had significant benefits relative to the status quo, which we were able to measure in a relatively low-cost study.

In retrospect, since only one of 20 participants completed the Casino task successfully (across both conditions), that task was too hard for the amount of time we allowed. We recommend that users of PLIERS carefully select success criteria in pilot studies in order to set appropriate task time limits and difficulties.

8 STUDY DESIGN CHALLENGES AND SOLUTIONS

Table 9 summarizes the challenges that PLIERS addresses. Our primary interest is in programmers' abilities to achieve their goals *after* they have become proficient in the programming language, not on how easy it is for novices to *learn* the language. Thus, our evaluation approach requires first teaching people a language and then observing their performance on tasks.

When we initially tried to apply HCI methods in our language design work, we were thwarted by several challenges, described in the introduction: *training*, *recruiting*, *high prototyping cost*, and *variance*. We also encountered additional challenges, such as *interdependence of features*, *time management* in studies, *participant bias toward familiar languages*, and *unsound proposals by participants*. In this section, we describe techniques we used when designing user studies in order to address each challenge.

8.1 Training

Evaluating a programming language requires first *teaching* the programming language. Many universities offer term-length courses in specific programming languages or techniques; requiring this kind of time commitment would make it extremely difficult to recruit participants. Furthermore, most courses ensure a consistent experience for all students by having all students learn the material in parallel (for example, with one session per topic, where all students participate at the same time). In contrast, our design approach was iterative, consistent with design methods used in other areas of

Challenge	Approaches
Training	• Include knowledge assessments and practice problems in tutorial • Divide tutorial into small pieces • Answer questions during training phase of study • Automatically provide feedback for wrong answers
Recruiting	• In academic settings, recruit master's students, who frequently have professional experience that may be representative of many practitioners • Recruit professionals, but only when their expertise is needed • Appeal to professionals' altruism for recruiting (they may not be incentivized by typical study budgets) • Screen participants carefully; set a high bar for student participation • Evaluate language design research questions in the context of a language with which many possible participants are familiar
High prototyping cost	• Back-port language design questions to existing languages (also helps isolate effects of independent variables) • Use Wizard of Oz to simulate tools that do not exist yet: use a plain text editor rather than a real IDE, and have an experimenter provide feedback in lieu of a real compiler or interpreter
Interdependence of features	• Isolate design questions by back-porting them to a familiar language • Mitigate non-orthogonality risk with summative studies
Variance and external validity	• Triangulate with multiple study types • Break tasks into subtasks • Recruit from populations with sufficient programming skills and knowledge; pre-screen participants.
Time management	• Pilot repeatedly to assess how long tasks usually take • Set cutoff times so that most people will succeed at most tasks • Allow participants extra time when possible, then report these successes separately from the "within time limit" results
Bias toward familiar languages	• Staged natural programming approach: sequentially expose additional constraints to participants • Request that participants do tasks using specific language designs that are being evaluated
Unsound proposals	• Provide sound alternatives and ask participants to use them • Provide participants with expert feedback on design ideas

Table 9. How PLIERS addresses common challenges in running user studies on programming languages.

HCI [31]. We were interested in addressing a variant of our *training* challenge that asks: what would be an effective way to teach a programming language in a consistent way to many participants in sequence?

Initially, we created a textual guide to the new programming language, and asked participants to read it before doing the tasks relevant to each study. The guide was relatively short; it could be read thoroughly in under an hour.

Unfortunately, this approach had very significant limitations. Although it was effective for some participants, others only skimmed the material and were then unable to complete the programming tasks. Because the guide was not structured as reference material and it included substantial conceptual information, skimming the guide was insufficient.

We were able to solve the problem with two adaptations: (1) break the guide into much smaller pieces; (2) ask participants to answer questions or complete small tasks to assure they had absorbed the material of each piece. For example, we broke the Obsidian tutorial into ten parts, and still the average participant completed it in under 90 minutes. We found that we were able to design tasks that checked understanding that were brief and did not require substantial experimenter intervention (helpful for ensuring consistency). We used a web survey tool (Qualtrics [69]) to guide participants through the tutorial and ask questions to check understanding. The tool also offered automatic feedback on participants' answers to multiple-choice questions. For example, Figure 10 shows a question about a code fragment with the correct answer selected. The relevant language details are explained in §1.2.

```
contract Money {
  int amount;
  transaction getAmount() returns int {
    return amount;
  }
}
contract Wallet {
  Money@Owned m;
  Wallet@Owned() {
    m = new Money();
  }

  transaction spendMoney() {
    ...
  }

  transaction receiveMoney(Money@Owned >> Unowned mon) returns Money@Owned
    Money temp = m;
    m = mon;
    return temp;
  }

  transaction checkMoney() returns Money@Owned {
    return m;
  }
}
```

Q15. Would we get a compiler error with the checkMoney function?

- ○ Yes, you cannot return a field of a contract in a transaction
- ○ No, the return type matches the type of m
- ◉ Yes, returning m makes it Unowned, which doesn't match the ownership status of m's declaration
- ○ No, m is of type Owned, which matches the ownership status of m's declaration
- ○ None of the above

Fig. 10. One question from the Obsidian tutorial. The question assesses whether the participant has understood that at the ends of transactions, fields must be have types that match their declarations, and that returning a variable consumes any ownership in the variable. If a participant submits an incorrect answer, the survey tool informs them of their error so they can fix their misunderstanding.

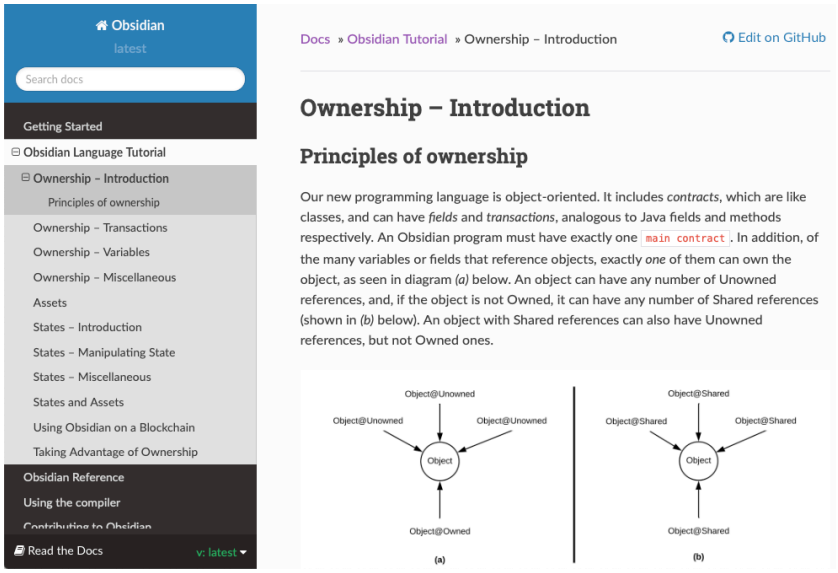


Fig. 11. A page from the Obsidian tutorial. The navigation bar at left shows how the tutorial was divided into 11 sections.

Although we originally wanted to make the tutorial stand alone so that every participant would have the same experience, we found that to be impractical; participants inevitably had questions about the materials, and forcing them to continue without having their questions answered resulted in them being unable to complete the tasks. However, we found that if an experimenter was available to answer questions, most participants asked only a small number of questions, which could be addressed rapidly. This approach is arguably more similar to a real-world language learning experience than an approach in which no questions are answered; normally, learners can search the Internet for answers to their questions, ask friends for help, etc.

In summary, although our initial tutorial was not an effective way of teaching the language, and the final tutorial was not sufficient by itself, dividing the tutorial into small pieces, providing tasks to help participants check and reinforce their understanding, and having an expert who could answer questions allowed most of our participants to learn the needed material in a short period of time. Figure 11 shows how the tutorial was broken into 11 different sections, each of which was followed by exercises for participants to complete.

8.2 Recruiting

Evaluation requires participants who are sufficiently skilled that they can rapidly learn a new programming language and then complete tasks using the new language. This would seem to require lengthy user studies with skilled participants, who can be challenging to recruit and retain for the required period of time. *Iterative* evaluation requires a large number of participants, since participants who learned an earlier version of the language can no longer provide fresh perspectives on new ideas. Although some user interfaces for experts in other domains require recruiting members of a small population, many of those interfaces are for short-term, focused tasks rather than lengthy problem-solving tasks. Furthermore, although it is typical to conduct studies with students, this relates to our *external validity* challenge: to what extent do results from students apply to the professional software engineers that are the target of our language?

We found in our work on Glacier and Obsidian that we were able to usefully combine results from different populations. Rather than trying to exclusively obtain professional software engineers, we found that we could design studies that yield meaningful results from students; for other aspects of the research, we recruited limited numbers of professionals. For example, when we wanted to interview software engineers to find out their experiences of using immutability constructs in the Glacier work, we recruited senior-level professional software engineers. However, for the other studies, we made three observations that enabled us to do our studies with various kinds of students.

First, about 41% of professional developers have been programming professionally for less than five years [81]. Many graduate students have some professional experience. For example, students at the Professional Master's program in Computer Science & Engineering at the University of Washington were reported to have an average of five years of professional experience [61]. Similarly, the Carnegie Mellon Master of Software Engineering program requires all students to have at least two years of experience [89]. By recruiting from graduate students, we were able to attract a population that is similar to a significant fraction of professional programmers and software engineers.

Second, in usability studies, it is typical to assume that usability problems encountered by even one user may be experienced by many others. Not every usability problem can be addressed without risking introducing new usability problems, but our experience is that many can be. For example, error messages, documentation, and keywords can be interpreted in ways that were not intended by the author; clarifying the text can prevent others from being confused in the same way. Syntax borrowed from other languages can be evocative in useful ways, but when the semantics do not match precisely, confusion can result; this can be addressed by choosing distinct syntax. On the other hand, semantic or structural changes can have consequences on users that are hard to predict, especially since one high-level change may necessitate a series of lower-level changes, which each have their own impact. For example, in Obsidian, moving transactions so that they were no longer lexically scoped in states necessitated adding special syntax for specifying the initial type of the receiver, `this`. We had selected that approach based on consistency with Java, which already uses that design. Unfortunately, that approach was surprising to some of our participants.

By addressing problems that student participants encounter, we prevent professionals from encountering those problems as well. Of course, some of the problems may not be ones that professionals would encounter, but nonetheless, addressing them may improve learnability, making the system better overall. When changes that would improve the system for the participants might degrade performance for experienced users, then the designer can make an informed tradeoff, potentially addressing the problem in training materials rather than in a design change.

Third, for Obsidian studies, we developed a screening instrument so that we could include only participants who had appropriate programming skills. The instrument, which is a web-based survey, takes most participants under ten minutes to complete. The instrument also included more difficult questions; because of the difficulty, we did not use this portion for screening. However, we found that performance on the more difficult portion of the instrument was positively correlated with speed in one of our programming tasks even in a small, six-participant study. At the time of the Glacier studies, we had not yet developed this instrument; the more-complex programming tasks in the Obsidian studies motivated us to screen our participants more carefully.

Designing a screening instrument (or deciding not to use one) depends on an assessment of what knowledge and skills are required of participants, and of how honest the prospective participants will be in their self-assessment. If prospective participants can reliably self-assess preparedness for the study, and they can be assumed to be honest, then screening may be unnecessary. On the other hand, even in this case, assessing programming knowledge and skills can be useful for understanding how these relate to task performance. In the Obsidian studies, we invited participants based on a "basic Java" portion of the screening instrument and observed that performance on one of the tasks was correlated

with performance on the "advanced Java" portion of the instrument, suggesting that Java programming knowledge was a significant influence on task performance. This is somewhat surprising, since the screening test examined language-specific knowledge, but did not give any actual programming tasks. We would encourage others to consider using this kind of screening instrument, since it is low-cost (generally under 10 minutes per participant), resulted in participants who were generally capable Java programmers, and portions of it correlated with task performance.

We found that relatively small incentives were sufficient to motivate students to participate in our studies. For three-hour studies, we offered a $50 Amazon gift card; for four-hour studies, we offered a $75 Amazon gift card. For shorter studies, we paid $10/hour. We recruited professionals from among our personal networks and did not offer them a specific incentive to participate.

8.3 High prototyping cost

Programming language designers are accustomed to creating high-cost implementations, not low-cost *prototypes*, but traditional HCI methods assume that low-cost prototypes can be created. Traditional ways of evaluating programming languages typically require a compiler or interpreter as well as theoretical work to create a sound design (informally, one in which programs mean what they are supposed to mean and the safety guarantees that the type system claims to provide can actually be provided). If one insists on creating a sound, formal model of the language before evaluating it with users, iteration can require so much time that it is impractical. Furthermore, the cost is increased by the expectation of sophisticated language-dependent tooling in IDEs: syntax highlighting, autocomplete, high-quality error messages, and the like.

Instead, we do not insist on doing this work at the beginning. We outline a potentially sound underlying formalism without proving all the relevant properties. Then, we design a surface language and evaluate it with users so that we can obtain feedback early. In doing so, we accept the risk that the formal system cannot be made sound without invalidating the data we gathered from users, but in practice, we found that usually any mistakes are minor and can be corrected without having to redo the user studies.

Late in the project, we found that designing and running user studies of low-level features typically required much more time than implementing the features; for those, it make sense to implement the alternatives rather than simulating them. On other other hand, early in the project, many high-level design decisions would have required substantial design and implementation work. Among those, we carefully selected questions for which user input would be the most impactful. A key approach in minimizing cost of language changes was to re-use training materials across phases of the studies to the extent possible, allowing us to amortize the cost of their development across multiple studies. The training materials co-evolved with the implementation and represented a significant investment.

8.4 Interdependence of features

Suppose a comparison between two languages showed that one allowed participants to complete tasks faster or more successfully. If the two languages were very different from each other, it would be unclear which aspects of the new language were actually helpful. For example, a comparison between a particular functional language and a particular object-oriented language would not result in fine-grained, actionable design guidance for a new language. Furthermore, if the study was done in the context of a language that was new to participants, confusion might be due to unfamiliar aspects of the language that are unrelated to the design question of interest.

By using the *back-porting* approach described above, we isolated particular design questions in the context of an existing language. Although this does not enable us to address very high-level design questions, such as whether

the language should be object-oriented or functional, it allowed us to obtain actionable data about particular design decisions.

Theoretical refinement is another approach that helps address feature interactions, since frequently, key theoretical issues relate to interactions between language features. Likewise, case studies, natural programming, and usability studies with appropriate tasks can lead to insight regarding cross-cutting concerns.

Of course, it is still the case that the design choices are not orthogonal. To address this, we integrate the results into a new language and conduct summative studies on the completed language as a whole as well.

8.5 High variance and external validity

The nature of programming is that there is huge variance in performance on tasks among different programmers [59]. When asking participants to complete programming tasks to help a designer iterate on a language design, participants frequently get stuck on problems that are not of interest to the designer. For example, in one Obsidian study, a participant spent significant time writing code to recurse through a data structure, even though code had been provided to do exactly that. Issues involving the details of the data structure were intended to be out of scope for the study. On the other hand, constraining tasks too much may result in artificial tasks that do not represent the complexity of real-world programming problems, which limits the external validity of the studies.

We use three techniques to address these problems. First, we combine the results of different kinds of studies (*triangulation* [74]). Qualitative studies of varied tasks with varied participants, in which timing is not an important dependent variable, can identify usability problems, and an experimenter can guide participants away from problems that are not intended to be part of the study. Quantitative studies typically involve fairly constrained tasks, but we can hope to obtain statistical significance in a comparison between two different designs. Finally, although this paper does not focus on our case study work, we also used case studies to address questions of *expressiveness*: elucidating what happens when the language is used to solve a larger programming problem, which cannot be completed in a single-session user study; for more information, see Coblenz et al. [22] and Coblenz et al. [25].

Second, particularly in RCTs (in which the experimenter cannot provide any guidance), we give several independent tasks rather than one long task. Then, we analyze the tasks separately, although of course the performance on the tasks is not independent because the same participant completed all of the tasks. Furthermore, dividing tasks into multiple pieces enables separate analysis of complete vs. incomplete tasks. For example, in the Glacier studies, we gave both simple and complex immutability specification tasks rather than one combined task. In the Obsidian studies, we separated a complex task, Casino, from simpler tasks, Auction and Prescription, even though the research questions overlapped. This allowed participants to succeed in the simpler task even if the more complex task was too difficult for them. It also offered an opportunity for participants to apply knowledge gained in the simpler task when working on the more complex task.

Third, recruiting from a constrained population reduces the impact of uninteresting noise. The primary technique to use is a screening survey, which participants must complete before being selected to participate in the study. This allows the experimenter to ensure that programmers have sufficient programming skills and knowledge. Of course, one must be careful to avoid screening out participants that may, in fact, be representative of the population to which the results should generalize.

In qualitative studies, it is sometimes unclear how many participants to recruit. Nielsen and Landauer found that the best benefit/cost ratio occurred at 3.2 participants in a set of their usability studies, which were of a medium-large

software project [60]. We found it was effective to consider the following factors in assessing when to stop recruiting more participants:

- To what extent new data (from the most recent participants) duplicates existing data
- To what extent the researcher is willing to tolerate risk of missing usability problems
- Fidelity of the current prototype (it may not be worth exhaustively testing low-fidelity prototypes)
- Specific research objectives: have the primary research questions been addressed yet?

8.6 Time management

As a practical matter, one needs to keep each participant's commitment brief in order to be able to recruit and retain enough participants and to minimize study cost. However, the experiment designer needs to allow enough time for most participants to finish the given programming tasks (at least in some of the experimental conditions). To address this problem, we conducted enough pilot studies (when preparing for an RCT, we found typically five or so pilots sufficed) that we could estimate the range of times that most participants would spend on each task. We found that we could allow participants enough time such that when the participant did not finish in the allotted time, the experimenter usually believed that even given substantial additional time, the participant would not have completed the task. This belief was driven by observing the difficulties that participants were facing at the end of the time window. Sometimes the problem was a design choice by the participant that made the problem much more challenging than anticipated; other times we believe it was due to lack of programming skill, since we observed some participants making basic programming errors. Of course, it is difficult to generalize about participants who do not finish particular tasks when we expected them to have enough time, but we found that the above approach resulted in studies that were practical to run and which yielded useful results.

The choice of study pre-screening method introduces a tradeoff. A lax pre-screening procedure makes it easier to obtain enough participants from a population that generalizes to a broader community. A strict pre-screening procedure that admits only the most expert participants may reduce times as well as variance, but may make it difficult to recruit participants and harder to generalize the findings. In university settings, with many novices, we advise erring on the stricter end of the spectrum, since most real practitioners will be more skilled than most students.

Rather than giving fixed limits for each task in advance, we aimed to maximize effective use of participants' time. When participants had additional time remaining in their commitment (for example, in one study, we told participants that the study would take four hours), we could let the participants spend longer than budgeted on the later tasks if their earlier tasks took less time than expected. Then, when reporting results, we could consider what the success rate would have been if everyone had had only the time available of the participant with the minimum time window for that task. In addition, we could report which participants succeeded given the additional time. This allowed us to make the best use of our participants' time while maintaining experimental validity.

8.7 Bias toward familiar languages

In a user study of a new programming language in which the participants are experienced programmers, one might expect that the language that performs "best" might be one with which participants are already familiar. Furthermore, when asked to join in participatory design exercises, perhaps participants might be likely to guide the design toward languages with which they are already familiar.

We used three techniques to address this problem. First, to find out what approaches might be easily learnable and would make immediate sense to participants, we adapted the *natural programming* elicitation technique [57]. In it, participants are given blank paper or a text editor and asked to write programs without being given a specific language to use. As a form of participatory design, the goal is to elicit from participants the way they would *naturally* express the ideas in question. Although traditional natural programming studies give the programmer no training at all, we took a *staged* approach. First, we asked participants to write programs on a blank screen with no training. Then, we told them information about the language design, and asked them to do additional programming tasks with the new (but still underspecified) design. For example, we gave participants a state transition diagram and asked them to write a program that expresses the state transitions. By scaffolding the participants' work in stages, we were able to answer both questions about participants' initial expectations as well as identifying what approaches might be most natural given our preliminary language design assumptions.

Second, in most of the studies, we constrained the participants' work according to our design ideas. Because the languages were designed to provide particular formal safety guarantees, we were interested in the impact of the language features related to those properties. In general, providing stronger guarantees requires that programmers enable the compiler to prove safety properties, which may require additional work from programmers. We were interested, then, in whether participants could complete tasks in the language even though they were obtaining stronger safety guarantees.

Third, we focused on observing and understanding behavior rather than preferences. By doing so, participants' prior experience was not an obstacle to overcome, but instead background we could leverage in teaching participants our language.

To encourage innovative responses (rather than ones that merely reflected prior training), we used natural programming for situations in which commonly-used languages could not directly represent the requirements we gave participants. We also used natural programming for low-level syntactic choices (e.g. keyword selection). We also instructed participants explicitly to be creative and not write in any particular existing language. Finally, we were careful to interpret the results in the context of participants' prior knowledge. For example, when participants use curly braces to denote blocks, the content of the blocks may be interesting even though the choice of curly braces is not.

8.8 Unsound proposals by participants

Another common limitation of natural programming is that participants lack expertise in language design, resulting in unsound proposals. This problem occurs with participatory design in other domains as well, and the usual solution is to use participant ideas as input to an expert-led design process [65], which applies here as well.

Our language design process typically involves writing multiple example programs, each of which assumes a particular language design and explores a particular kind of programming problem. The examples typically expose tradeoffs in language design; choosing which tradeoff to make can be informed with user input. We were able to use some of these prototypes to develop user studies, in which we presented participants with several options rather than expecting them to compose designs from scratch. In some cases, we tried to generate all feasible options in a particular design space (due to various technical constraints, this might result in three or four options), and then narrowed this down to the most promising approaches based on the tradeoffs that were apparent. We asked participants to *complete tasks* using the best candidates so that we could come to an informed conclusion about which of the options were best, rather than merely asking participants for their opinions. This allowed us to focus the process on designs that would fulfill the technical requirements while still obtaining relevant design insights.

9 FUTURE WORK

In this paper, we evaluated PLIERS by applying it to two different language designs. This approach greatly facilitated developing and iterating on the PLIERS process. However, in the future, we would like to show that these methods can be used by language designers with a variety of backgrounds and goals. As a practical matter, recruiting language designers to participate in a language study is a challenging and heavyweight endeavor, but future work may identify promising contexts in which to evaluate PLIERS more broadly. We have begun by teaching PLIERS to students in an undergraduate programming language design class, and found that the students were able to use some of the methods to help them iterate on their language designs.

One particularly challenging aspect of applying PLIERS is that the theoretical aspects of the design work require substantial background. Perhaps in the future, mechanized tools could help those who are not programming language experts design safe languages in their own application domains. By using program synthesis techniques, a language synthesis tool might be able to search the space of languages that have particular formal properties to help users identify safe design candidates. Creating Obsidian required spending months developing the underlying core calculus and proving it sound; if this effort could be mechanized, language iteration would be much faster, and could be feasible for those without formal programming language training.

The Wizard of Oz approach that we proposed in this paper relies on an experimenter who can accurately simulate the kinds of error messages that a compiler might generate. We envision a utility that would help promote consistency and improve reliability. Such a tool would accept error messages entered by experimenters during experiments and deliver them to participants in a realistic way. By recording the errors that were delivered, the experimenter could re-use existing error messages as well as record participants' reactions. This approach might lead toward refined error messages that are clearer for users to understand.

Another limitation of the methods we describe in this paper is that although one can do studies that assess the usability of particular language design choices, in some cases design choices interact with each other. As a result, it is not clear that designers can combine the results of different studies and expect that the resulting language will be usable. In this work, we mitigate this threat in two ways. Summative studies address the integrated language and can reveal problems that arise from combinations of design choices. Likewise, by triangulating design through multiple kinds of studies, some of which crosscut multiple language design choices, we obtain different perspectives on various combinations of features. However, future method development work may be able to address this problem more directly.

In the future, we hope to explore how PLIERS could be used to develop tools for other problem-solving contexts beyond programming. Attributes of programming that are shared with other kinds of problem-solving activities include:

High variance: Problem-solving can be unpredictable [53]; in user studies, some participants typically complete tasks almost instantly whereas others can spend hours working and still not finish. This large variance makes running quantitative user studies very challenging.

Range of working styles: Bergström and Blackwell described a diverse collection of different approaches to programming problems [7], such as *bricolage/tinkering* and *engineering*. These different styles may be used even by different people using the *same language*, impeding a designer's attempts to anticipate a user's strategy or behavior.

High stakes: Errors when programming can contribute to serious real-world safety problems, e.g., in avionics or health care systems.

For example, CAD tools affect their users' creative processes [73]; likewise with process engineering tools [15] and even drug design tools [85]. All of these domains involve expert problem-solving by a variety of different people with high costs of failure. As such, they might be amenable to use the PLIERS process to help designers in those domains create tools that are effective for their target users.

10 CONCLUSION

PLIERS represents a new approach to designing programming languages for software engineers. PLIERS is exemplified in Glacier and Obsidian, which reflect a new way of designing programming languages that integrates user-centered techniques into many stages of the design process. By incorporating feedback from users, we obtained insights that led to two languages in which programmers can be effective at obtaining stronger safety guarantees than prior languages provided. We expect our new approach to language design is applicable to the design of other programming languages, and even to the design of a wide variety of different kinds of problem-solving tools.

11 ACKNOWLEDGMENTS

We appreciate the help of Eliezer Kanal at the Software Engineering Institute, who helped start this project, as well as Jim Laredo, Rick Hull, Petr Novotny, and Yunhui Zheng at IBM, who provided useful technical and real-world insight.

This material is based upon work supported by the National Science Foundation under Grants CNS-1423054 and CCF-1814826, by the U.S. Department of Defense, and by Ripple. In addition, the first author is supported by an IBM PhD Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Salman Ahmad, Alexis Battle, Zahan Malkani, and Sepander Kamvar. 2011. The Jabberwocky programming environment for structured social computing. In *User interface software and technology (UIST '11)*. ACM, 53–64. **https://doi.org/10.1145/2047196.2047203**

[2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented Programming. In *Companion of Object Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*. 1015âĂŞ1022. **https://doi.org/10.1145/1639950.1640073**

[3] Leonardo Alt and Christian Reitwiessner. 2018. SMT-Based Verification of Solidity Smart Contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice.*

[4] Celeste Barnaby, Michael Coblenz, Tyler Etzel, Eliezer Kanal, Joshua Sunshine, Brad Myers, and Jonathan Aldrich. 2017. A User Study to Inform the Design of the Obsidian Blockchain DSL. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU '17)*.

[5] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. 2011. An Empirical Study of Object Protocols in the Wild. In *European Conference on Object-oriented Programming (ECOOP '11)*. 2âĂŞ26. **http://dl.acm.org/citation.cfm?id=2032497.2032501**

[6] E. D. Berger, S. M. Blackburn, M. Hauswirth, and M. Hicks. 2018. Empirical Evaluation Checklist (beta). **http://www.sigplan.org/Resources/EmpiricalEvaluation/**

[7] Ilias Bergström and Alan F Blackwell. 2016. The practices of programming. In *Visual Languages and Human-Centric Computing (VL/HCC '16)*. IEEE, IEEE, 190–198. **https://doi.org/10.1109/VLHCC.2016.7739684**

[8] Karthikeyan Bhargavan, Nikhil Swamy, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, and Thomas Sibut-Pinote. 2016. Formal Verification of Smart Contracts. In *ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. **https://doi.org/10.1145/2993600.2993611**

[9] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular Typestate Checking of Aliased Objects. In *Object-oriented programming systems, languages, and applications (OOPSLA '07)*. 301âĂŞ320. **https://doi.org/10.1145/1297027.1297050**

[10] Kevin Bierhoff and Jonathan Aldrich. 2008. PLURAL: Checking Protocol Compliance Under Aliasing. In *Companion of International Conference on Software Engineering (ICSE Companion '08)*. 971âĂŞ972. **https://doi.org/10.1145/1370175.1370213**

[11] Alan Blackwell and Margaret Burnett. 2002. Applying Attention Investment to End-User Programming. In *Human Centric Computing Languages and Environments (HCC '02)*. IEEE Computer Society, Washington, DC, USA, 28–30. **https://doi.org/10.1109/HCC.2002.1046337**

[12] Ann Blandford and Thomas Green. 2008. *Methodological Development.* Cambridge University Press, 158–174. **https://doi.org/10.1017/CBO9780511814570**

[13] Joshua Bloch. 2008. *Effective Java, Second Edition.* Addison-Wesley.

[14] Michael Bostock and Jeffrey Heer. 2009. Protovis: A graphical toolkit for visualization. *IEEE transactions on visualization and computer graphics* 15, 6 (2009), 1121–1128. **https://doi.org/10.1109/TVCG.2009.174**

[15] Bertrand Braunschweig and Rafiqul Gani. 2002. *Software architectures and tools for computer aided process engineering*. Computer Aided Chemical Engineering, Vol. 11. Elsevier.

[16] Raymond PL Buse, Caitlin Sadowski, and Westley Weimer. 2011. Benefits and barriers of user evaluation in software engineering research. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*. 643âĂŞ656. **https://doi.org/10.1145/2076021.2048117**

[17] Roger D. Chamberlain. 2017. Assessing User Preferences in Programming Language Design. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. Association for Computing Machinery, 18âĂŞ29. **https://doi.org/10.1145/3133850.3133851**

[18] Sarah Chasins. 2017. Helena: Web Automation for End Users. **http://helena-lang.org/**

[19] Michael Coblenz, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2020. Can Advanced Type Systems Be Usable? An Empirical Study of Ownership, Assets, and Typestate in Obsidian. In *Object-oriented programming systems, languages, and applications (OOPSLA '20)*. To appear.

[20] Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2018. Interdisciplinary Programming Language Design. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '18)*. 133âĂŞ146. **https://doi.org/10.1145/3276954.3276965**

[21] Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2019. Usability Methods for Designing Programming Languages for Software Engineers. arXiv:1912.04719

[22] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2017. Glacier: Transitive Class Immutability for Java. In *International Conference on Software Engineering (ICSE '17)*. IEEE Press, 496âĂŞ506. **https://doi.org/10.1109/ICSE.2017.52**

[23] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2020. Glacier software and user study replication package. (7 2020). **https://doi.org/10.1184/R1/12108693.v1**

[24] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2020. Obsidian software and user study replication package. (8 2020). **https://doi.org/10.1184/R1/12771074**

[25] Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. 2019. Obsidian: Typestate and Assets for Safer Blockchain Programming. arXiv:1909.03523

[26] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. 2016. Exploring Language Support for Immutability. In *International Conference on Software Engineering (ICSE '16)*. ACM, 736âĂŞ747. **https://doi.org/10.1145/2884781.2884798**

[27] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. 2016. *Exploring Language Support for Immutability*. Technical Report CMU-ISR-16-106. Carnegie Mellon University.

[28] Nils Dahlbäck, Arne Jönsson, and Lars Ahrenberg. 1993. Wizard of Oz studies - why and how. *Knowledge-based systems* 6, 4 (1993), 258–266. **https://doi.org/10.1016/0950-7051(93)90017-N**

[29] Phil Daian. 2016. Analysis of the DAO exploit. Retrieved August 21, 2018 from **http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/**

[30] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2016. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International conference on financial cryptography and data security*. **https://doi.org/10.1007/978-3-662-53357-4_6**

[31] Joseph S Dumas and Janice Redish. 1999. *A practical guide to usability testing*. Intellect books.

[32] Chris Elsden, Arthi Manohar, Jo Briggs, Mike Harding, Chris Speed, and John Vines. 2018. Making Sense of Blockchain Applications: A Typology for HCI. In *CHI Conference on Human Factors in Computing Systems (CHI '18)*. 1–14. **https://doi.org/10.1145/3173574.3174032**

[33] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 2014. How Do API Documentation and Static Typing Affect API Usability?. In *International Conference on Software Engineering (ICSE '14)*. ACM, 632–642. **https://doi.org/10.1145/2568225.2568299**

[34] K Anders Ericsson and Herbert A Simon. 1984. *Protocol analysis: Verbal reports as data*. the MIT Press.

[35] Ethereum Foundation. 2020. Common Patterns. Retrieved February 18, 2020 from **http://solidity.readthedocs.io/en/develop/common-patterns.html**

[36] Ethereum Foundation. 2020. Solidity. Retrieved February 18, 2020 from **https://solidity.readthedocs.io/en/develop/**

[37] Luke Graham. 2017. $32 million worth of digital currency ether stolen by hackers. Retrieved November 2, 2017 from **https://www.cnbc.com/2017/07/20/32-million-worth-of-digital-currency-ether-stolen-by-hackers.html**

[38] Paul Graham. 2001. Five Questions about Language Design. **http://www.paulgraham.com/langdes.html**

[39] Thomas R. G. Green and Marian Petre. 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131–174.

[40] Raymonde Guindon, Herb Krasner, Bill Curtis, et al. 1987. Breakdowns and processes during the early activities of software design by professionals. In *Empirical studies of programmers: Second Workshop*. 65–82.

[41] Jan Gulliksen, Bengt Göransson, Inger Boivie, Stefan Blomkvist, Jenny Persson, and Åsa Cajander. 2003. Key principles for user-centred systems design. *Behaviour and Information Technology* 22, 6 (2003), 397–409. **https://doi.org/10.1080/01449290310001624329**

[42] Harvard Business Review. 2017. The Potential for Blockchain to Transform Electronic Health Records. Retrieved February 18, 2020 from **https://hbr.org/2017/03/the-potential-for-blockchain-to-transform-electronic-health-records**

[43] Dominik Harz and William Knottenbelt. 2018. Towards Safer Smart Contracts: A Survey of Languages and Verification Methods. arXiv:1809.09805

[44] Maurice Herlihy. 2019. Blockchains from a distributed computing perspective. *Commun. ACM* 62, 2 (2019), 78–85.

[45] C. A. R. Hoare. 2009. Null References: The Billion Dollar Mistake. Retrieved February 18, 2020 from **https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/**

[46] IBM. 2019. Blockchain for supply chain. Retrieved March 31, 2019 from **https://www.ibm.com/blockchain/supply-chain/**

[47] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. 2003. A User-centred Approach to Functions in Excel. In *International Conference on Functional Programming (ICFP '03)*. ACM, 165–176. **https://doi.org/10.1145/944705.944721**

[48] Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)* 37, 2 (2005), 83–137.

[49] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)* 43, 3, Article 21 (2011), 44 pages. **https://doi.org/10.1145/1922649.1922658**

[50] Amy J. Ko, Thomas D. LaToza, and Margaret M. Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141. **https://doi.org/10.1007/s10664-013-9279-3**

[51] Herb Krasner, Bill Curtis, and Neil Iscoe. 1987. *Communication Breakdowns and Boundary Spanning Activities on Large Programming Projects.* 47âĂŞ64.

[52] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. 2010. *Research Methods in Human-Computer Interaction.* Wiley Publishing.

[53] Dastyni Loksa, Amy J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *SIGCHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, 1449–1461. **https://doi.org/10.1145/2858036.2858252**

[54] Leonid Mikhajlov and Emil Sekerinski. 1998. A Study of The Fragile Base Class Problem. In *European Conference on Object-Oriented Programming (ECOOP 1998)*. 355âĂŞ382.

[55] Lance A. Miller. 1974. Programming by Non-programmers. *International Journal of Man-Machine Studies* (1974). **https://doi.org/10.1016/S0020-7373(74)80004-0**

[56] Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7 (July 2016), 44–52. **https://doi.org/10.1109/MC.2016.200**

[57] Brad A. Myers, John F. Pane, and Amy J. Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47 (2004), 47–52. Issue 9. **https://doi.org/10.1145/1015864.1015888**

[58] Allen Newell and Stuart K. Card. 1985. The Prospects for Psychological Science in Human-computer Interaction. *Hum.-Comput. Interact.* 1, 3 (Sept. 1985), 209–242. **https://doi.org/10.1207/s15327051hci0103_1**

[59] W. R. Nichols. 2019. The End to the Myth of Individual Programmer Productivity. *IEEE Software* 36, 5 (2019), 71–75.

[60] Jakob Nielsen and Thomas K. Landauer. 1993. A Mathematical Model of the Finding of Usability Problems. In *Proceedings of the INTERACT âĂŹ93 and CHI âĂŹ93 Conference on Human Factors in Computing Systems (CHI âĂŹ93)*. Association for Computing Machinery, New York, NY, USA, 206âĂŞ213. **https://doi.org/10.1145/169059.169166**

[61] University of Washington. 2019. Professional Master's Program. Retrieved February 18, 2020 from **https://www.cs.washington.edu/academics/pmp**

[62] Stephen Oney, Brad A. Myers, and Joel Brandt. 2014. InterState: a language and environment for expressing interface behavior. In *User interface software and technology (UIST '14)*. ACM, 263–272. **https://doi.org/10.1145/2642918.2647358**

[63] John F. Pane, Brad A. Myers, and Leah B. Miller. 2002. Using HCI techniques to design a more usable programming system. In *Human Centric Computing Languages and Environments (HCC '02)*. 198–206. **https://doi.org/10.1109/HCC.2002.1046372**

[64] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295 – 341. **https://doi.org/10.1016/0010-0285(87)90007-7**

[65] Kara Pernice and Kathryn Whintenton. 2017. How to Deal With Bad Design Suggestions. Retrieved February 18, 2020 from **https://www.nngroup.com/articles/bad-design-suggestions/**

[66] Dewayne E. Perry, Susan Elliott Sim, and Steve M. Easterbrook. 2004. Case studies for software engineers. In *International Conference on Software Engineering (ICSE '04)*. IEEE, 736–738.

[67] Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press.

[68] Benjamin C. Pierce and Yitzhak Mandelbaum. 2009. PL Grand Challenges. Retrieved February 18, 2020 from **http://plgrand.blogspot.com**

[69] Qualtrics. 2020. Qualtrics Software. **http://www.qualtrics.com/**

[70] Microsoft Corp. 2008. Framework Design Guidelines. Retrieved February 18, 2020 from **https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/struct**

[71] Oracle Corp. 2019. Secure Coding Guidelines for the Java SE, version 4.0. Retrieved February 18, 2020 from **https://www.oracle.com/technetwork/java/seccodeguide-139067.html**

[72] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. 2009. Scratch: Programming for all. *Commun. Acm* 52, 11 (2009), 60–67. **https://doi.org/10.1145/1592761.1592779**

[73] BF Robertson and DF Radcliffe. 2009. Impact of CAD tools on creative problem solving in engineering design. *Computer-Aided Design* 41, 3 (2009), 136–146. **https://doi.org/10.1016/j.cad.2008.06.007**

[74] Paulette M. Rothbauer. 2008. *The Sage encyclopedia of qualitative research methods*. SAGE.

[75] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2015. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE transactions on visualization and computer graphics* 22, 1 (2015), 659–668. **https://doi.org/10.1109/TVCG.2015.2467091**

[76] Ben Shneiderman. 1986. Empirical Studies of Programmers: The Territory, Paths, and Destinations. *Empirical Studies of Programmers* (1986), 1–12.

[77] Ben Shneiderman and Catherine Plaisant. 2006. Strategies for evaluating information visualization tools: multi-dimensional in-depth long-term case studies. In *AVI workshop on BEyond time and errors: novel evaluation methods for information visualization*. ACM, ACM, 1–7. **https://doi.org/10.1145/1168149.1168158**

[78] Max E. Sime, Thomas R. G. Green, and DJ Guest. 1977. Scope marking in computer conditionalsâĂŤa psychological evaluation. *International Journal of Man-Machine Studies* 9, 1 (1977), 107–118.

[79] Emin Gün Sirer. 2016. Thoughts on The DAO Hack. Retrieved February 18, 2020 from **http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/**

[80] Elliot Soloway and Kate Ehrlich. 1984. Empirical studies of programming knowledge. *IEEE Transactions on software engineering* 5 (1984), 595–609.

[81] Stack Overflow. 2019. Developer Survey Results 2019. Retrieved February 18, 2020 from **https://insights.stackoverflow.com/survey/2019**

[82] Andreas Stefik and Stefan Hanenberg. 2014. The Programming Language Wars: Questions and Responsibilities for the Programming Language Community. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2014)*. 283–299. **https://doi.org/10.1145/2661136.2661156**

[83] Andreas Stefik and Stefan Hanenberg. 2017. Methodological irregularities in programming-language research. *Computer* 50, 8 (2017), 60–63. **https://doi.org/10.1109/MC.2017.3001257**

[84] Andreas Stefik, Susanna Siebert, Melissa Stefik, and Kim Slattery. 2011. An Empirical Comparison of the Accuracy Rates of Novices Using the Quorum, Perl, and Randomo Programming Languages. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU '11)*. ACM, 3–8. **https://doi.org/10.1145/2089155.2089159**

[85] Kent D. Stewart, Melisa Shiroda, and Craig A. James. 2006. Drug Guru: a computer software program for drug design using medicinal chemistry rules. *Bioorganic & medicinal chemistry* 14, 20 (2006), 7011–7022.

[86] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. 2015. Searching the State Space: A Qualitative Study of API Protocol Usability. In *International Conference on Program Comprehension (ICPC '15)*. IEEE Press, Piscataway, NJ, USA, 82–93. **http://dl.acm.org/citation.cfm?id=2820282.2820295**

[87] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. 2011. First-class state change in Plaid. In *Object Oriented Programming Systems, Languages, and Applications (OOPSLA '11)*. **https://doi.org/10.1145/2076021.2048122**

[88] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. 2016. An Empirical Study on the Impact of C++ Lambdas and Programmer Experience. In *International Conference on Software Engineering (ICSE '16)*. ACM, 760–771. **https://doi.org/10.1145/2884781.2884849**

[89] Carnegie Mellon University. 2019. Master's Programs. Retrieved February 18, 2020 from **https://www.cs.cmu.edu/masters-programs**

[90] A. Marie Vans, Anneliese von Mayrhauser, and Gabriel Somlo. 1999. Program understanding behavior during corrective maintenance of large-scale software. *International Journal of Human-Computer Studies* 51, 1 (1999), 31 – 70. **https://doi.org/10.1006/ijhc.1999.0268**

[91] John Venable, Jan Pries-Heje, and Richard Baskerville. 2012. A Comprehensive Framework for Evaluation in Design Science Research. In *Design Science Research in Information Systems. Advances in Theory and Practice*. 423–438. **https://doi.org/10.1007/978-3-642-29863-9_31**

[92] June M. Verner, Jennifer Sampson, Vladimir Tosic, N.A. Abu Bakar, and Barbara A. Kitchenham. 2009. Guidelines for industrially-based multiple case studies in software engineering. In *International Conference on Research Challenges in Information Science*. IEEE, 313–324. **https://doi.org/10.1109/RCIS.2009.5089295**

[93] Willemien Visser. 1987. Strategies in programming programmable controllers: A field study on a professional programmer. In *Empirical Studies of Programmers: Second workshop (ESP2)*. 217–230. **https://hal.inria.fr/hal-00641376/document**

[94] Diane B. Walz, Joyce J. Elam, Herb Krasner, and Bill Curtis. 1987. A Methodology for Studying Software Design Teams: An Investigation of Conflict Behaviors in the Requirements Definition Phase. In *Empirical Studies of Programmers: Second Workshop*. 83âĂŞ99.

[95] Preston Tunnell Wilson, Justin Pombrio, and Shriram Krishnamurthi. 2017. Can We Crowdsource Language Design?. In *Symposium on New Ideas in Programming and Reflections on Software (Onward! 2017)*. 1–17. **https://doi.org/10.1145/3133850.3133863**

[96] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kielun, and Michael D. Ernst. 2007. Object and reference immutability using Java generics. In *Foundations of Software Engineering (FSE '07)*. ACM, 75–84.