Interdisciplinary Programming Language Design

DRAFT - Distributed for Comments

Michael Coblenz, Jonathan Aldrich, Brad Myers, Joshua Sunshine
Carnegie Mellon University
Pittsburgh, PA, USA
mcoblenz,jonathan.aldrich,bam,sunshine@cs.cmu.edu

Abstract

Traditional programming language design approaches center around theoretical and performance-oriented evaluation. Recently, researchers have been considering more approaches to language design, including the use of quantitative and qualitative user studies to evaluate how different designs affect users. In this paper, we argue for an interdisciplinary approach that incorporates many different methods in the creation and design of programming languages. We show how the addition of user-oriented design techniques can be helpful at many different stages in the programming language design process.

1 Introduction

Since the beginning of computing, programmers have argued about how they should express their programs. Some argue that the language should closely match the way users think about problems [23]. Others focus on training programmers to think in a way that facilitates correct reasoning about programs, and then they design languages to match this style of reasoning.

The stakes in this debate are high: programming languages are the most basic tool used by programmers, and therefore have a major impact on the software development process. Empirical studies suggest that the choice of programming language can significantly impact software quality and security [47], as well as performance and programmer productivity [41]. Understanding how to design languages better could clearly improve the way we engineer software.

In this paper, we discuss the question of *design process*, arguing that the most commonly used approaches fail to consider a broad enough range of different kinds of evidence. In contrast, we propose a variety of different approaches that can be used to create and evaluate programming languages.

Languages, like other kinds of software projects, frequently follow an iterative design process, summarized in Fig. 1:

Figure 1. A typical design process with suggested methods

1. In the *requirements elicitation and creation* phase, the designer studies the application domain for the language. The designer creates a draft version of the language, likely including a language specification and language implementation.
2. In the *evaluation* phase, the designer evaluates how well the language fulfills its requirements.

After evaluation, the design process may repeat to address shortcomings that were identified. We use the word *design* to refer to the entire process, including requirements analysis, specification, implementation, and evaluation. We use the word *creation* to refer to the part of the process that includes specifying the language, including its syntax and semantics, as well as the implementation phase, because these phases are usually intertwined.

Language designers face unique challenges relative to designers of other kinds of tools. Programming language designs must meet a unique set of interdisciplinary constraints and objectives, including mathematical foundations, performance characteristics of the created software, the ability of individual programmers to work efficiently (i.e. *usability*), and the ability of teams to construct large-scale software effectively (i.e. *software engineering*). All these considerations may conflict. For example, mathematical modeling of a language's design can be used to create a type-safe or memory-safe language, with significant software engineering benefits. However, implementing the necessary checks at run time imposes performance overhead, while implementing those checks at compile time can preserve performance but may make the language less usable. Unfortunately, language design is too often done in an ad-hoc way that ignores one or more disciplines that should inform it. For example, many languages are designed without user-centered evaluations [52], resulting in designs that may fulfill theoretical and

Draft, February 05–09, 2018, Dagstuhl, Germany
2018.

performance requirements but impose unnecessary burdens on their users.

In this paper, we argue that the large, complex design space of programming languages justifies treating language design as an engineering activity—one that makes principled tradeoffs among considerations from multiple disciplines. As with software development, language development should be iterative, and incorporate not just summative evaluation on completed designs but also formative methods during the design process itself. We show how we and other researchers have used a wide range of research and design methods to gain insight into how to design programming languages so that they are as effective as possible for programmers. Our account will emphasize human-centered methods, as these tend to receive less emphasis in the existing literature, but will also demonstrate synergies between these methods and traditional approaches such as type theory. Finally, we show how qualitative evaluation methods can complement quantitative methods to inform the search through the language design space.

Overall, we argue for an approach to language design that:

1. Uses a diverse array of complementary methods to address a variety of design questions and evaluate the design from a wide range of perspectives.
2. Prioritizes specific quality attributes of a language according to domain needs, rather than assuming that a particular set of attributes is best for all languages.
3. Strategically selects which methods to apply at each step in the design process.

We use the word *interdisciplinary* rather than *multidisciplinary* because it emphasizes the benefit of combining techniques and approaches into a unified method. In contrast, a *multidisciplinary* approach would emphasize using different approaches individually, perhaps by independent experts. The latter is insufficient because the lessons learned from one approach should affect both the process and the lessons learned from other approaches. For example, language theory should guide the set of designs that are tested with users, while studies of the software development process should affect which are the most important theorems to prove.

We describe in §2 the goals for programming language designs that we believe most designers have. Next (§3), we discuss current approaches to programming language design, and argue for a holistic approach that prominently includes human-centered methods. §4 explains why no single method is sufficient in general, and why combining multiple approaches is more likely to lead to high-quality designs. §5 describes the methods that we and others have found useful in the creation and evaluation of programming languages. §6 discusses approaches for choosing which methods to use, and §7 describes two particular language designs in which different design and evaluation methods complemented one another.

2 Desiderata of Programming Languages

Quality attributes [31] describe properties that are used to evaluate software systems. For example, the *maintainability* quality attribute refers to a class of scenarios pertaining to how easy or difficult it is for maintainers of a system to modify it. Design decisions frequently involve tradeoffs among different quality attributes. A strong, static type system may guarantee the absence of certain kinds of bugs (*correctness*), but if it is too hard to use, a non-safety-critical system may be better served by a less type-safe language (*modifiability*). Furthermore, it is impossible to evaluate a programming language without knowing what one wants to evaluate it *for* – that is, what should be optimized? And evaluation of the attainment of different quality attributes requires different evaluation *methods*. For example, measuring *correctness* can be evaluated with formal methods or with testing; *learnability* requires evaluation with novices; *performance* typically requires executing benchmarks.

Below, we show a partial list of relevant considerations; practically every software quality attribute may be affected by a language design. Our purpose in this section is to highlight how particular characteristics of programs relate to programming language design and how they trade off with each other. As a result, we argue that programming language designers should be intentional and explicit about their priorities, and they should select from a diverse set of methods according to their design goals.

2.1 Traditional goals

Correctness concerns the question of whether a particular program has specific desirable properties, such as adherence to a specification and the absence of certain classes of bugs. Languages typically support correctness through formal approaches such as type systems or proof systems. Researchers evaluate the correctness of the formalism using proofs of appropriate soundness theorems.

Performance (of the resulting code) is typically evaluated with benchmarks on the particular hardware of interest. Performance is amenable to benchmarking methods, which have been well-described for particular performance evaluation domains [22].

Expressiveness is the ability of a programmer to express their intent explicitly in the language.[1] For example, a type system adds expressiveness compared to an untyped language in the sense of communicating the programmer's intended restrictions on the contents of variables that are annotated with types. However, that same type system may rule

[1]For a more formal definition that is consistent with the more intuitive arguments made in this paragraph, we point the interested reader to Felleisen [20]

out certain desirable program structures, limiting expressiveness in a different sense. Some forms of expressiveness may come at a usability cost, e.g. because users are forced to express decisions that they would prefer to postpone or prefer not to express formally. Additional expressiveness may facilitate some kinds of modifiability (e.g. changing the type of a function may allow the compiler to find all the calls that need to be updated) while inhibiting others (e.g. a large modification cannot be even partially tested until it is completely done). Researchers typically evaluate the expressiveness of the formal system by giving examples of the properties it can verify on sample programs.

Speed of compiling was a major concern when computers were less powerful, and remains important for providing quick feedback in IDEs and when compiling very large systems. The module structure of Go, for example, was designed to improve compilation times compared to C++ [44].

2.2 User-centered goals

Understandability is the property of how easy it is for a reader of the code of a program to understand it. Programmers typically spend far more time reading code than writing it, with one estimate suggesting a 10:1 ratio of time spent reading to writing [33]. Researchers and practitioners have proposed a variety of measures of and proxies for understandability. One approach involves user studies in which participants are given source code and asked to answer questions about program behavior. Some languages have been evaluated by observing that the designers were able to implement a particular program in fewer lines of code using their language than using prior languages, but it is not necessarily the case that shorter programs are always easier to understand or maintain than longer programs [2]. Our primary interest is in the ability of programmers to correctly and quickly answer important questions about their programs (in service of their goals).

Ease of reasoning is the user-focused analog of correctness: for each form of correctness desired of a program, one might ask how easy or difficult it is for users to show a particular program is correct. This question is answered much less often than the questions about correctness itself, but it can be evaluated through user experiments [15]. Arguably, if programmers cannot use the correctness features correctly, then the resulting code will not be correct.

Modifiability captures the ease of making particular changes to programs. Developers spend more time maintaining and evolving programs than creating them in the first place, so modifiability has long been a major concern in the field of software engineering. Parnas suggested that information hiding, in service of modifiability, should be the main criteria for decomposing systems into modules [42], and so module systems have been a central way that language designers support this goal.

Module systems can also support ease of reasoning, since when appropriate specifications are added to module boundaries, reasoning can be done more locally. Languages may need to support modifiability in other ways, too, since modular decomposition sometimes has a deleterious effect on performance. Small-scale modifiability can be evaluated with user studies, while modifiability at larger scales requires case studies or repository mining methodologies.

Learnability is an important practical concern for adoption: to what extent can existing programmers use the language with little training? It is often evaluated in studies with novices.

3 Current Perspectives on Language Design

For comparison and discussion purposes, we describe several existing approaches to language design. Programming languages are designed by various people in various contexts (e.g., in universities or corporations) for many purposes. Our intent here is to promote discussion and draw contrasts between important language design styles, recognizing that in practice individual language designers may (as we advocate) use a combination of the approaches below.

A **logician** is primarily concerned with developing logical systems that are relevant to computation. Viewing programming as the practice of writing *correct* programs — that is, programs that meet particular mathematical specifications — the logician is focused on concise, convenient, correct expression of algorithms. Programming is considered to be a task that is best suited to experts, who can be thoroughly trained in the appropriate mathematics so that they can write correct programs. Since programming is considered to be primarily a mathematical pursuit, the best language for programming is likely to be very close to the language of mathematics (and presumably close to the way the logician is thinking, as in the *closeness of mapping* heuristic [23]). Future discoveries of mathematical principles may lead to better programming languages — ones in which programs can be expressed more beautifully and with stronger guarantees of correctness.

The **industrialist** is interested in designing languages that are effective for writing large software systems in order to achieve various commercial goals. As such, performance and adoption (which depends on many different attributes, including learnability [34] and interoperability) are often priorities.

The **empiricist** views programming languages as critical tools for programmer performance. The focus is on using carefully-designed experiments to show concrete effects of specific design decisions on programmers' success on programming tasks. The empiricist expects that by doing a large number of experiments, researchers will learn how language designs affect users; after gathering sufficient data, language designers will be able to make a large portion of their design

choices on the basis of experiments that show the impact of those particular design choices on people. The Quorum programming language claims to be the first evidence-oriented programming language [54].

A **teacher** focuses on pedagogical benefits of programming languages. This approach has been taken in the design of many programming languages [17]. Some, such as Alice and Scratch, use structured editors to address the barrier typically imposed by formal syntax. Others, such as Logo, use a graphical environment to make abstract concepts more concrete and fun.

4 Interdisciplinary Design

We view each of the stylized approaches above to be useful for language design, but we view them as being individually too limited. Instead, our approach is to combine many different methods according to the design goals of the language. A logical approach (*the logician*) forms a sound basis for language design. It is useful for quickly eliminating from consideration many designs that are not internally consistent. However, formal methods do not specify which of many different, sound languages will be best for programmers, who are *people* [35]. While formal methods can link language constructs to program properties, they cannot directly tell us which program properties are the most important, and therefore should be the focus of a type system or proof system design.

An *industrial* approach is practically useful for designing serviceable languages. As researchers, however, we focus on longer-term goals of improving productivity at scale, as unconstrained as possible by the market forces and technological experience of current programmers. This points to the need to consider *software engineering theory* to understand how language constructs such as modules affect the software development process as programs scale up in size.

In industry, risk aversion frequently results in the selection of well-proven techniques, such as object-oriented and imperative programming. It is not necessary to show that the design is the best possible one, since a high-quality design that is of practical use suffices. Knowing what aspects of the design contribute to or detract from programmer success may be of lower priority than creating a design quickly and cost-effectively in which it is practical to implement interesting, commercially-relevant systems. Over time, as the community gains experience with the language, the design will be modified to make writing certain programs more convenient. However, it will be difficult to fix major design flaws in a deployed language due to backwards compatibility constraints, so users will have to learn workarounds for deficiencies.

We find the *empirical* approach compelling for conducting summative evaluations of systems; traditional quantitative methods from social science can be applied effectively to show, for example, that certain static type systems have certain benefits over dynamic type systems [19]. However, summative evaluations are only useful on systems that are complete enough to withstand user tests, which can require significant engineering work; furthermore, of the thousands of design decisions involved in a particular programming language design, a particular experiment can only consider a small set of options. For example, a 2 x 2 factorial study studies two design options in each of two dimensions, and even this would require a large number of participants if one wants statistically significant results. In cases where design choices interact—something we have observed to be very common in language design—it quickly becomes impossible to evaluate the cross product of the possible choices. These interactions between design features make it difficult to go from study results to holistic language designs. In contrast, language designers need approaches that allow them to explore and evaluate a larger portion of the design space. Additional challenges include the difficulty of studying longer and more complex tasks in a controlled, laboratory setting; and the difficulty of recruiting a representative sample of software engineers and retaining them in a laboratory environment long enough to obtain results.

Medicine is another discipline that seeks to inform its recommendations with empirical data. Some researchers have argued that the programming languages community might look to the field of medicine for insight regarding appropriate evidence in scientific fields [52, 53]. Evidence-based medicine rests on three pillars: individual clinical expertise; external clinical evidence from systematic research (particularly from controlled trials when considering therapeutic options, when available); and patient values, preferences, and characteristics. [18, 48, 51] Notably, controlled trials form only one component of three; the medical community considers other relevant aspects of a clinical situation when recommending treatment. Even if language designers were to use a medical approach, then, they would need to consider arguments beyond those which are directly supported by controlled experiments. However, although clinicians can typically choose to *not* recommend a treatment, this option is not available to programming language designers, whose closest moral equivalent might be to abandon the pursuit of language design (instead recommending that users use existing languages). Language designers are frequently forced to make decisions or recommendations lacking direct experimental evidence.

The number of design decisions involved in designing a particular programming language is immense; we hope that future work will analyze this space more formally, but our experience suggests that there are at least thousands of decisions that are made in the design of any given language. For example, high-level decisions such as what paradigms and type systems to use, medium-level decisions such as what control structures and modularity features to provide in the

language, and lower level decisions such as the concrete syntax and which reserved words to use. In practice, designers complete their work by making many decisions on the basis of prior successful systems and their own intuition and experience. Although orthogonality of *constructs* is one of the canonical recommendations for language designers [45, 49], it is our experience that many language design *decisions* are not orthogonal. We argue, then, that it is risky to combine the results of individual experiments without some more holistic evaluation that either provides evidence that the decisions are in fact orthogonal, or provides enough guidance about how the decisions interact to properly interpret the experimental results.

Instead of relying on exhaustive experimentation, then, we propose to use many different methods from the field of Design to *triangulate* when making design decisions; although a particular method might only suggest a particular region in the design space, we can obtain further guidance helping us narrow it further by using different methods. Although this approach lacks the statistical satisfaction of randomized controlled trials, it has the benefit of producing evidence grounded in real users that can be obtained practicably and applied to wide variety of different language designs. We show in §5 examples of how various techniques have been used to obtain insight about programming languages.

The perspective of the *teacher* is useful in the design of practical languages because languages that are difficult to learn are less likely to be adopted. Insights from pedagogy may also provide hints as to which approaches are more or less *natural* for users. However, languages that focus on pedagogical goals may not be ideal for creating large, complex systems; instead, a teacher's focus is on teaching particular aspects of programming so that students can be effective when using other languages.

An important aspect of an interdisciplinary approach is that it allows a collection of detailed qualitative results regarding different designs. Rather than focusing on *whether* a particular design promotes faster task completion times compared to another, we seek to learn *why* [29]: when programmers are confused, what is the cause of the confusion? What concrete improvements can we make to the language, the programming environment, and the training materials to improve task performance?

We seek to use human-centered approaches broadly in order to first obtain lower-cost, *qualitative* knowledge about designs, and then to obtain quantitative results showing how new designs compare to existing ones. Our assumption is that we are likely to obtain a better design (one for which a quantitative evaluation is likely to show a superior result) if we take user data into account throughout the design process [36] rather than focusing the use of user-oriented methods only at the end of the process.

In general, the discipline of *design* is about creating tools that help people achieve their goals while considering practical constraints [12]. Design is applicable to large design spaces, such as that of programming languages, including in high-stakes situations. For example, an airplane cockpit is designed taking human factors into account in order to reduce error rates to improve airplane safety [63]. The design recommendations are drawn from a variety of sources, including human factors texts and industry standards. The aviation industry learns how to design safe cockpits with a interdisciplinary approach; it does not restrict itself to quantitative studies of pilots with candidate interfaces.

5 Methods

We divide the methods into those that are primarily oriented around eliciting and iterating on design ideas (without needing a prototype to evaluate) and those that are oriented around evaluation (requiring a prototype).

5.1 Methods for requirements and creation

Surveys are a useful way to assess opinions and experience among a large sample, for example for assessing whether a proposed problem is one that a large fraction of practitioners face, or assessing whether which problems are the most important to solve from a practitioner's point of view. Some researchers have also used surveys to get direct insight into programming language designs [60], but the results have been inconclusive regarding specific design guidance. Most surveys ask people what they believe, but in some cases people's beliefs do not lead to designs that benefit users in practice. Furthermore, survey results can be difficult to interpret or clouded with noise. Sometimes, little verifiable information is known about participants, and there may be motives that detract from data validity (e.g. Mechanical Turk workers may want to complete the survey as fast as possible to maximize their hourly wage).

Interviews can be a valuable source of information for areas in which researchers can find experts. These can be a useful approach to quickly obtain knowledge about existing problems and their existing solutions. For example, we interviewed experienced software engineers and API designers to understand how practitioners use immutability in their software designs [16]; the insights led to a new tool, Glacier [15], which is designed around the needs of real users instead of around maximizing expressiveness. Glacier extends Java to support transitive class immutability, a kind of immutability that the interviewees expressed was useful in real software. Interviews are limited in external validity because it may be difficult or impossible to interview a representative sample of any particular population. The results strongly depend on the participants themselves as well as the skill of the interviewer in eliciting as much useful information as possible with minimal bias.

Corpus studies can show the prevalence of particular patterns in existing code, including patterns of bugs in bug databases. For example, Callaú et al. [13] investigated the use of dynamic features in Smalltalk programs, Malayeri et al. [32] investigated whether programs might benefit from a language that supported structural subtyping, and we studied how Java programmers used exception handling features [26]. Corpus studies can show that a particular problem occurs often enough that it might be worth addressing; they can also show how broadly a particular solution applies to real-world programs, as in Unkel and Lam's analysis of stationary fields in Java [62]. However, it can be difficult to obtain a representative corpus. For example, though GitHub contains many open source projects, they can be difficult to build; it can be difficult to sample in an unbiased way; and open source code may not be representative of closed source code.

Natural Programming [36] is a technique to elicit how people express solutions to problems without any special training. It aims to find out how people might "naturally" write programs. These approaches have been useful for HANDS [40], a programming environment for children, as well as professionally-targeted languages, such as blockchain programming languages [3]. However, the results are biased by participants' prior experience and education, and results depend on careful choice of prompts to avoid biased language.

Rapid prototyping is commonly used in many different areas of HCI, and can be used for language design as well [35]. Low-fidelity prototypes, such as paper prototypes, can be used to obtain feedback from users on early-stage designs ideas. Wizard-of-Oz testing involves an experimenter substituting for a missing or insufficient implementation. For example, when evaluating possible designs for a type system for a blockchain programming language, we gave participants brief documentation on a language proposal and asked them to do tasks in a text editor. Because there was no typechecker implemented, the experimenter gave verbal feedback when participants wrote ill-typed code. This allowed us to learn about the usability of various designs without the expense of implementing designs that were about to be revised anyway. However, low fidelity prototypes may differ in substantive ways from polished systems, misleading participants. The results depend on the skill and perspectives of the experimenter and the participants, resulting in limited external validity.

Programming language and aoftware engineering theory provide a useful guide when considering the requirements for a programming language. For example, the guarantees that a transitive immutability system can provide in the areas of both security and concurrency—which have been well-established in the programming language theory literature—were key reasons that we chose this semantics for the Glacier type system [16]. Similarly, an understanding of how modularity affects modifiability from the software engineering literature [42] motivates the module systems present in many languages, and more recent theories about how software architecture [50] influences software development motivated our design of the ArchJava language [1]. However, theoretical guarantees that pertain to optional language features will not be obtained if the features are misunderstood or not used. Furthermore, guarantees can be compromised by bugs in unverified tool implementations.

5.2 Methods for evaluation

Qualitative user studies have been used to evaluate many different kinds of tools, including programming languages [16, 40], APIs [37], and development environments [28]. Some of these consist of *usability analyses*, in which participants are given tasks to complete with a set of tools and the experimenter collects data regarding obstacles the participants encounter while performing the tasks. Unlike randomized controlled trials, these are usually not comparative; that analysis is left to a future study. Instead, they focus on learning as much as possible in a short amount of time in order to *test feasibility* of a particular approach and *improve the tool* for a future iteration of the design process.

Another qualitative approach involves *participatory design* [10, 39], in which participants are asked to help explore the design space and analyze tradeoffs. The assumption is that domain experts are likely to give feedback and suggestions that are of practical use in designing tools. Unlike usability studies, which evaluate existing prototypes, this approach is *formative*, intended to help inform the exploration of a design space.

Qualitative user studies can also be used to understand a problem that a language design is intended to solve, and help to guide other research methods used to evaluate the eventual solution. We studied programmers solving protocol-related programming problems that were gleaned from real StackOverflow questions in order to understand the barriers developers face when using stateful libraries [57]. The results of the study were useful in developing a language and its associated tools, and produced a set of tasks that were used in a later user experiment. Because of the qualitative user study, we knew these tasks were the most time-consuming component of real-world programming problems, mitigating the most significant external threat to the validity of the user experiment.

Qualitative user studies are usually limited to short-duration tasks with participants that researchers can find. In practice, this sometimes limits the sizes of the programs that the tasks concern because larger programs typically require more sophisticated participants and more participant time. Although a typical qualitative user study might only take an hour or two per participant, even a small real-world programming task might take a day or more.

Case studies show *expressiveness*: a solution to a particular programming problem can be expressed in the language in question. Many case studies aim to show *concision*, observing that the solution is expressible with a short program, particularly in comparison to the length of a typical solution in a comparison language. Case studies are particularly helpful when the language imposes restrictions that might cause a reader to wonder whether the restrictions prevent application of the language to real problems.

Case studies can also be used to learn about how a programming language design works in practice. For example, we used exploratory case studies on ArchJava, an extension of Java with software architecture constructs, to learn about the strengths and limitations of the language design and to generate hypotheses about how the approach might affect the software engineering process [1].

Case studies have limited external validity because they necessarily only consider a small set of use cases (perhaps just one). As a result, the conclusions are biased by the selection of the cases. Furthermore, the results may not generalize to typical users, since the case studies may be done by expert users of the system under evaluation.

Expert evaluation methods, such as Cognitive Dimensions of Notations [23] and Nielsen's Heuristic Analysis [38], provide a vocabulary for discussing design tradeoffs. Although they do not definitively specify what decision to make in any particular case because each option may trade off with another, they provide a validated mechanism for identifying advantages and disadvantages of various approaches. This approach has been widely used in the visual languages community. However, expert evaluation requires access to experts and a validated and relevant set of criteria. The traditional criteria, such as Cognitive Dimensions of Notations, have not yet been validated against traditional textual languages by showing that their results are correlated with quantitative experiments.

Performance evaluation, typically via benchmarks, is well-accepted for comparing languages and tools. Performance evaluation can be critical if it is relevant to the claims made about a language, but many popular languages are not as fast as alternatives (consider Python vs. C), so it is important to decide how much performance is required. SIGPLAN released a checklist [6] for empirical evaluations of programming languages; although is title is only "Empirical Evaluation Checklist," it only describes performance evaluations. The checklist hints at limitations of this approach, such as mismatch between benchmark suite and real-world applications; an insufficient number of trials; and unrealistic input.

User experiments, also known as *randomized controlled trials* (RCTs) have been used to address a variety of programming language design questions, such as the benefits of C++ lambdas [61], static type systems [19], and typechecking [46]. In some ways, these represent the gold standard for summative evaluations. However, they do not always lead to insights that can be used to design or improve systems, and unless they are supplemented by theory (e.g. gleaned from qualitative studies), it can be difficult to be certain that results on a narrow problem studied in the laboratory will apply to a more complex real-world setting. For example, Uesbeck et al. discuss in what contexts their conclusions might apply [61], but not how one might improve C++ lambdas to retain possible advantages but mitigate identified shortcomings.

Formalism and proof are traditional tools for showing that a specific language design has particular properties, such as type soundness [43]. In many languages, a formal model provides key insight that inspires a new language design; in these cases, the formal analysis might be the *first* step in a language design. However, in other languages, a formalism serves primarily to provide a specification and a safety guarantee, in which case this work might be done much later.

A typechecker provides some safety guarantees once a program typechecks, but one must compare the difficulty of writing a type-correct program to the difficulty of obtaining safety some other way (for example, with runtime checks, at the cost of deferring verification to runtime) and to the option of not providing the guarantee at all. In some systems, safety guarantees are not necessary; for example, the consequences of a bug in a video game may be smaller than the consequences of a bug in avionics software.

Formal verification via tools such as Dafny [30] or Coq [7] can provide even stronger guarantees, likely at greater implementation cost. However, if the tools are too difficult to use, programmers may not obtain the guarantees because they may circumvent the tools (e.g. by implementing difficult procedures in a lower-level language) or because they may fail to complete their projects within their cost and time constraints.

In practice, there is typically a gap between what is actually specified in a formal specification of correctness and what is desired by the programmer. For example, a programmer may specify the correct output of a factorial function in a recursive way, implement the function iteratively to avoid overflowing the stack for large input, and leave the specification that *the program shall not overflow the stack for input within the expressible range of machine-size integers* unwritten.

6 Strategy

Language designers must be *strategic* in their selection and order of method application. We propose using approaches according to likely risk reduction: prefer techniques that are most likely to provide insight that most reduces the risk of failing to achieve the design goals. Languages that are inspired by the desire to investigate how a particular logical system corresponds with computation are likely to start with

a formal analysis. The design of a domain-specific language might start by using formative methods to understand the appropriate domain and its users. The domain-specific language design project may do the formal analysis *last* because although the designers want to show that their language is sound, they do not expect the results of the formal analysis to significantly impact the design of the language.

Research-oriented language designers may only use a small number of different methods in order to achieve the research objectives. Researchers are typically interested in languages that are novel along particular dimensions but standard along others; they should therefore use methods that target the novel aspects of their language rather than attempting to evaluate all aspects of their language designs. Application of various methods requires special expertise, and individual researchers will likely specialize in one or more methods. This offers an opportunity for collaboration to apply the appropriate methods to each language design.

Production-oriented language designers may need a wider variety of methods because their designs must have high quality along many different axes. However, they may also be likely to make traditional choices in a wide variety of aspects of the design; as such, they may not be interested in design methods that address most of the aspects of the design. For example, designers of a new object-oriented language need not conduct evaluations on the usability of object-oriented programming, but they might conduct evaluations of their particular syntax and compiler error messages.

7 Examples

In this section, we show how combinations of the above methods have been helpful in two examples of programming language designs.

Typestate is a way of tracking the conceptual states of objects in a type system, ensuring that state-sensitive operations such as *read* on a *File* are not applied when the object is in an inappropriate state, such as *closed* [55]. Two of us were involved in a decade-long interdisciplinary research project that illustrates how different research methods complement one another in exploring language and type system support for typestate.[2]

We wanted to know how common it is in practice to have state protocols, so we carried out a code corpus study identifying and classifying classes that define protocols in Java library and application code [5]. Our study was a bit unusual for corpus studies: while we used tools to identify code that might define protocols, because the definition of protocols includes the notion of abstract states, we had to manually examine each candidate identified by the tool to verify that it really defined a protocol. We found that at least 7% of types in our corpus defined protocols, and that nearly all these protocols naturally fall into one of seven protocol categories.

That suggests that protocols are reasonably common, but do they cause problems for developers? To answer that question, we carried out another study which identified Stack Overflow questions about object protocols and then carried out a think-aloud study watching programmers perform tasks abstracted from those questions [57]. We found that when performing these tasks, developers spent 71% of their time answering four types of protocol-related questions. These two studies are complementary: one suggests that protocols are reasonably common, the other that there are real development problems that programmers struggle with involving protocols. By combining these studies we gain more confidence that object protocols are an important problem to work on than we would with either study in isolation.

We designed typestate support both as a set of annotations and analysis on top of Java, and as a separate language, Plaid [58]. Formal models of a typestate checking system in each setting were proved sound [9, 21]. Although some of the design and formal work was done before publishing the papers above, the design of these formal systems was driven by examples from the Java libraries that were included in those studies, ensuring that the formal approach had real-world applications. This was further verified by case studies in Java, verifying that our tool could successfully check real uses of typestate in 100,000 lines of Java code [4, 8], and in Plaid, verifying that the language could express complicated state machines in real examples ported from Java [58].

While run-time performance was not a driving motivation for our work on typestate, our initial implementation of Plaid was very slow. Therefore, two of us advised a student whose thesis demonstrated that Plaid could be implemented with a modest slowdown compared to previous dynamic languages [14].

Determining directly whether Plaid helps programmers is difficult because of confounding effects: Plaid is different from Java in many ways, not just in its support for typestate. However, support for typestate (either in Plaid or in Java) affects not only the language, but the surrounding set of tools. We modified the `javadoc` tool to produce documentation that included an ASCII-art state machine, listed state pre- and post-conditions for each method, and grouped methods by state. In a controlled experiment, we found that programmers were able to answer state-related questions 2.2 times faster and were 7.9 times less likely to make errors [56]. This experiment offers the most direct evidence for the benefit of our approach, but one of the major threats to external validity is that the experiment was done in a controlled setting; how do we know that the results will transfer to the real world? Fortunately, the qualitative protocol study described earlier addresses this threat: in the experiment, we chose questions that were asked by programmers doing real

[2] We present the work in a logical order; the actual research was done in an order that reflected the interests of students as well as our group's ongoing exploration of different research methods.

StackOverflow tasks. This example shows that a properly-designed pairing of experiment and formative study can be much more convincing than either study in isolation.

Glacier is an extension to Java that supports transitive class immutability [15, 16]. We started with the question "What kinds of immutability should a programming language support, and how should it support them?" We began with a literature review to understand existing approaches. We found a progression of increasingly complex research systems [11, 24, 25, 27, 59] supporting increasing numbers of kinds of immutability, but little evidence regarding which of these were actually needed in practice. However, immutability is a frequently-discussed topic in the software industry so it is an area where experts are like to have well-formed opinions. Therefore, we conducted *interviews* of professional software engineers to see what kinds of evidence we could gather regarding the utility of different kinds of immutability approaches. These interviews suggested, among other things, that developers would like immutability to help in developing concurrent systems. Language theory tells us that a transitive immutability system could be effective for this and other identified goals, an observation also supported by the interviews themselves. Interested in evaluating the effect of supporting transitive immutability, we built a *prototype* (informed by a formal model) and conducted a small *qualitative study* comparing an existing research tool, IGJ [64], with our prototype, IGJ-T. We noted that our participants had difficulty understanding the error messages in IGJ, which resulted in part from the wide variety of scenarios that IGJ was designed to support, such as both class and object immutability.

To improve our chances of obtaining a system that people could use effectively, we focused on a simpler system, which supported only transitive, class-based immutability. We hoped that this point in the design space would result in a simple, usable system that expressed constraints that were relevant to real programs. We evaluated this hypothesis in a *randomized, controlled experiment*. We assigned ten participants to use plain Java with `final` and ten others to use our extension, Glacier, on code writing tasks. We found that most of the participants who only had `final` accidentally modified state in immutable objects, resulting in bugs. We also asked participants to use their assigned tools to specify immutability in a small codebase. We found that most of the participants who had Glacier could specify immutability correctly; in contrast, every `final` user made mistakes when attempting to enforce immutability. Both of these results were statistically significant.

Finally, we conducted a *case study* applying Glacier to real-world systems. We were able to express the kinds of immutability used in a real Java spreadsheet implementation and in a Guava collections class (with one caveat for caching). On this basis, we argue that Glacier is likely to be applicable to a variety of real-world systems; its simplicity does not limit its utility to tasks in artificial lab studies. In contrast, we argue that its simplicity increases its value by providing usability so that programmers are able to use it effectively.

Glacier shows one example of how researchers can inform their research with *qualitative* methods, including interviews and qualitative lab studies, and then show benefit of their tools in a *quantitative* lab study afterward. We were able to show a successful quantitative result after significant iterations and qualitative human-centered evaluations, arguably because our design had been informed by other research methods.

8 Conclusions

Every design method has limitations. We argue for the application of *many* different methods in the programming language design process: theoretical methods as well as quantitative and qualitative user-oriented methods. These user-oriented methods have been shown to be useful in the process of creating and evaluating programming languages and programming language extensions.

An individual researcher or language designer may not be familiar with the entire breadth of methods we promote. Instead, we recommend collaborative efforts, where researchers work together to apply theoretical, formative, and summative techniques in order to prove relevant properties, explore fruitful portions of the design space, and show that their designs benefit users in specific, quantifiable ways.

Acknowledgments

This material is based upon work supported by NSF grant CNS-1734138, NSF grant CNS-1423054, by NSA lablet contract H98230-14-C-0140, by the Software Engineering Institute, and by AFRL and DARPA under agreement #FA8750-16-2-0042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, New York, NY, USA, 187–197. DOI: http://dx.doi.org/10.1145/581339.581365

[2] Eran Avidan and Dror G. Feitelson. 2017. Effects of Variable Names on Comprehension an Empirical Study. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 55–65. DOI: http://dx.doi.org/10.1109/ICPC.2017.27

[3] 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 *PLATEAU '17 Workshop on Evaluation and Usability of Programming Languages and Tools*.

[4] Nels E. Beckman. 2010. *Types for Correct Concurrent API Usage*. Ph.D. Dissertation. Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, USA. CMU-ISR-10-131.

[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*.

[6] E. D. Berger, S. M. Blackburn, M. Hauswirth, and M. Hicks. 2018. Empirical Evaluation Checklist (beta). (January 2018).

[7] Yves Bertot and Pierre Castran. 2010. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions* (1st ed.). Springer Publishing Company, Incorporated.

[8] Kevin Bierhoff. 2009. *API Protocol Compliance in Object-Oriented Software*. Ph.D. Dissertation. Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, USA. CMU-ISR-09-108.

[9] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular Typestate Checking of Aliased Objects. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 301–320. DOI: http://dx.doi.org/10.1145/1297027.1297050

[10] Susanne Bødker and Ole Sejer Iversen. 2002. Staging a Professional Participatory Design Practice: Moving PD Beyond the Initial Fascination of User Involvement. In *Proceedings of the Second Nordic Conference on Human-computer Interaction (NordiCHI '02)*. ACM, New York, NY, USA, 11–18. DOI: http://dx.doi.org/10.1145/572020.572023

[11] John Boyland, James Noble, and William Retert. 2001. Capabilities for Aliasing: A Generalisation of Uniqueness and Read-Only. In *European Conference on Object-Oriented Programming*, Jørgen Lindskov Knudsen (Ed.).

[12] Bill Buxton. 2007. *Sketching user experiences: getting the design right and the right design*. Morgan Kaufmann.

[13] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. 2011. How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*. ACM, New York, NY, USA, 23–32. DOI: http://dx.doi.org/10.1145/1985441.1985448

[14] Sarah Chasins. 2012. An Efficient Implementation of the Plaid Language. (2012).

[15] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2017. Glacier: Transitive Class Immutability for Java. In *Proceedings of the 39th International Conference on Software Engineering - ICSE '17*.

[16] 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*.

[17] Wikipedia contributors. 2018. List of educational programming languages. (2018). https://en.wikipedia.org/wiki/List_of_educational_programming_languages

[18] David M. Eddy. 2005. Evidence-Based Medicine: A Unified Approach. *Health Affairs* 24 (2005). Issue 1.

[19] 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*. ACM, New York, NY, USA, 632–642. DOI: http://dx.doi.org/10.1145/2568225.2568299

[20] Matthias Felleisen. 1990. On the Expressive Power of Programming Languages. In *Science of Computer Programming*. Springer-Verlag, 134–151.

[21] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 36, 4, Article 12 (Oct. 2014), 44 pages. DOI: http://dx.doi.org/10.1145/2629609

[22] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 57–76. DOI: http://dx.doi.org/10.1145/1297027.1297033

[23] 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.

[24] Christian Haack and Erik Poll. 2009. Type-based Object Immutability with Flexible Initialization. In *European Conference on Object-Oriented Programming*. DOI: http://dx.doi.org/10.1007/978-3-642-03013-0

[25] C. Haack, E. Poll, J. Schäfer, and A. Schubert. 2007. Immutable objects for a java-like language. In *European Symposium on Programming*.

[26] Mary Beth Kery, Claire Le Goues, and Brad A Myers. 2016. Examining programmer practices for locally handling exceptions. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 484–487.

[27] Gunter Kniesel and Dirk Theisen. 2001. JAC—Access right based encapsulation for Java. *Journal of Software Practice & Experience - Special issue on aliasing in object-oriented systems* 31, 6 (2001), 555–576. http://dl.acm.org/citation.cfm?id=377334

[28] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering* 32, 12 (2006).

[29] Thomas D LaToza and Brad A Myers. 2010. On the importance of understanding the strategies that developers use. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 72–75.

[30] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370. http://dl.acm.org/citation.cfm?id=1939141.1939161

[31] Bass Len, Clements Paul, and Kazman Rick. 2003. Software architecture in practice. *Boston, Massachusetts Addison* (2003).

[32] Donna Malayeri and Jonathan Aldrich. 2009. Is Structural Subtyping Useful? An Empirical Study. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (ESOP '09)*. Springer-Verlag, Berlin, Heidelberg, 95–111. DOI: http://dx.doi.org/10.1007/978-3-642-00590-9_8

[33] Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[34] Leo A. Meyerovich and Ariel S. Rabkin. 2012. Socio-PLT: Principles for Programming Language Adoption. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2012)*. ACM, New York, NY, USA, 39–54. DOI: http://dx.doi.org/10.1145/2384592.2384597

[35] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7 (July 2016), 44–52. DOI: http://dx.doi.org/10.1109/MC.2016.200

[36] Brad A. Myers, John F. Pane, and Andy Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47 (2004), 47–52. Issue 9.

[37] Brad A. Myers and Jeffrey Stylos. 2016. Improving API Usability. *Commun. ACM* 59, 6 (May 2016), 62–69. DOI: http://dx.doi.org/10.1145/2896587

[38] Jakob Nielsen and Rolf Molich. 1990. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 249–256.

[39] Fatih Kursat Ozenc, Miso Kim, John Zimmerman, Stephen Oney, and Brad Myers. 2010. How to Support Designers in Getting Hold of the Immaterial Material of Software. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 2513–2522. DOI: http://dx.doi.org/10.1145/1753326.1753707

[40] J. F. Pane, B. A. Myers, and L. B. Miller. 2002. Using HCI techniques to design a more usable programming system. In *Proceedings IEEE 2002*

Symposia on Human Centric Computing Languages and Environments. 198–206. DOI: http://dx.doi.org/10.1109/HCC.2002.1046372

[41] Victor Pankratius, Felix Schmidt, and Gilda Garretón. 2012. Combining Functional and Imperative Programming for Multicore Software: An Empirical Study Evaluating Scala and Java. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 123–133. http://dl.acm.org/citation.cfm?id=2337223.2337238

[42] D. L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. DOI: http://dx.doi.org/10.1145/361598.361623

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

[44] Rob Pike. 2012. Go at Google: Language Design in the Service of Software Engineering. (2012).

[45] Terrence W. Pratt and Marvin V. Zelkowitz. 1996. *Programming Languages: Design and Implementation*.

[46] L. Prechelt and W.F. Tichy. 1998. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Transactions on Software Engineering* 24, 4 (apr 1998), 302–312. DOI: http://dx.doi.org/10.1109/32.677186

[47] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. 2017. A Large-scale Study of Programming Languages and Code Quality in GitHub. *Commun. ACM* 60, 10 (Sept. 2017), 91–100. DOI: http://dx.doi.org/10.1145/3126905

[48] David L Sackett, William MC Rosenberg, JA Muir Gray, R Brian Haynes, and W Scott Richardson. 1996. Evidence based medicine: what it is and what it isn't. (1996).

[49] Robert W. Sebesta. 2006. *Concepts of Programming Languages, Seventh Edition*.

[50] Mary Shaw and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[51] B. Spring. 2007. Evidence-based practice in clinical psychology: what it is, why it matters; what you need to know. *Journal of Clinical Psychology* 63 (2007). Issue 7. DOI: http://dx.doi.org/10.1002/jclp.20373

[52] Andreas Stefik and Stefan Hanenberg. 2014. The Programming Language Wars: Questions and Responsibilities for the Programming Language Community. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 283–299. DOI: http://dx.doi.org/10.1145/2661136.2661156

[53] Andreas Stefik, Stefan Hanenberg, Mark McKenney, Anneliese Andrews, Srinivas Kalyan Yellanki, and Susanna Siebert. 2014. What is the Foundation of Evidence of Human Factors Decisions in Language Design? An Empirical Study on Programming Language Workshops. In *Proceedings of the 22Nd International Conference on Program Comprehension (ICPC 2014)*. ACM, New York, NY, USA, 223–231. DOI: http://dx.doi.org/10.1145/2597008.2597154

[54] Andreas Stefik, Melissa Stefik, and Evan Pierzina. 2018. The Quorum Programming Language. (2018). https://quorumlanguage.com

[55] Robert E Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering* 12, 1 (1986), 157–171.

[56] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. 2014. Structuring Documentation to Support State Search: A Laboratory Experiment about Protocol Programming. In *European Conference on Object-Oriented Programming (ECOOP)*.

[57] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. 2015. Searching the State Space: A Qualitative Study of API Protocol Usability. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC '15)*. IEEE Press, Piscataway, NJ, USA, 82–93. http://dl.acm.org/citation.cfm?id=2820282.2820295

[58] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. 2011. First-class state change in Plaid. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 713–732.

[59] Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: Adding Reference Immutability to Java. In *Object-oriented programming, systems, languages, and applications*. DOI: http://dx.doi.org/10.1145/1094811.1094828

[60] Preston Tunnell Wilson, Justin Pombrio, and Shriram Krishnamurthi. 2017. Can We Crowdsource Language Design?. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. ACM, New York, NY, USA, 1–17. DOI: http://dx.doi.org/10.1145/3133850.3133863

[61] 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 *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 760–771. DOI: http://dx.doi.org/10.1145/2884781.2884849

[62] Christopher Unkel and Monica S. Lam. 2008. Automatic inference of stationary fields. *ACM SIGPLAN Notices* 43, 1 (jan 2008), 183. DOI: http://dx.doi.org/10.1145/1328897.1328463

[63] Michelle Yeh, Young Jin Jo, Colleen Donovan, and Scott Gabree. 2013. *Human Factors Considerations in the Design and Evaluation of Flight Deck Displays and Controls*. Technical Report. Federal Aviation Administration.

[64] 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*. ACM, 75–84.