Interdisciplinary Programming Language Design

Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
{mcoblenz,jonathan.aldrich,bam,sunshine}@cs.cmu.edu

Abstract

Approaches for programming language design used commonly in the research community today 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 that examine how different designs might affect programmers. In this paper, we argue for an interdisciplinary approach that incorporates many different methods in the creation and evaluation of programming languages. We argue that the addition of user-oriented design techniques can be helpful at many different stages in the programming language design process.

CCS Concepts • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*;

Keywords programming language design, user-centered design, programming language evaluation

ACM Reference Format:
Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2018. Interdisciplinary Programming Language Design. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '18), November 7–8, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3276954.3276965

1 Introduction

Empirical studies suggest that the choice of programming language can significantly impact software quality and security [54], as well as performance and programmer productivity [48]. Understanding how to design languages better could clearly improve the way we engineer software.

Onward! '18, November 7–8, 2018, Boston, MA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6031-9/18/11...$15.00
https://doi.org/10.1145/3276954.3276965

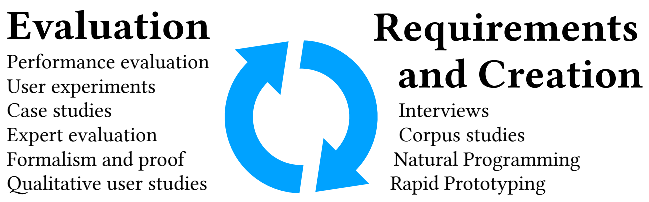

Figure 1. A typical design process

However, in many cases, language designers choose limited sets of techniques according to their skills and goals. For example, Stefik et al. advocate a randomized controlled trial-based approach [64, 65]. SIGPLAN created an evaluation checklist [5] that does not include methods of evaluating human factors-related aspects of languages, instead focusing on performance. The authors say "Because user studies are currently relatively infrequent in the papers we examined, we have not included them among the category examples." Neither of these two approaches is particularly helpful in the early stages of a language design, when a prototype is unavailable for evaluation.

We argue that these approaches are shortsighted and often insufficiently focused on understanding the relationship between design decisions and the impact of those decisions on programmers who use those languages. The commonly used approaches focus narrowly on a particular kind of evidence, which is often used to *evaluate* a particular design rather than to *guide* it. As an alternative, we propose a design process incorporating a wide variety of both formative and evaluative methods, integrating diverse kinds of evidence to guide the design of programming languages.

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

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 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, which may include mathematical foundations, performance characteristics, the ability of individual programmers to work efficiently (i.e. *usability*), and the ability of teams to construct large-scale software effectively. However, all these considerations may conflict. For example, mathematical modeling can be used to create a type-safe language. However, implementing the necessary checks at run time typically imposes performance overhead, while implementing those checks at compile time may make the language less usable by forcing the compiler to reject some safe programs.

We use *interdisciplinary* to emphasize that the process benefits from combining multiple techniques into a unified method, since multiple techniques are required in order to address the diverse set of goals that apply to each language design. 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 formal user-centered evaluations [64], resulting in designs that may fulfill theoretical and performance requirements but impose unnecessary burdens on their users. Other authors have argued for more user studies and focus on using randomized controlled trials (RCTs) [64]. As we will argue in the following, RCTs are an important evaluation technique, but should be complemented with other methods that are more effective at guiding designs.

We argue that the large, complex design space of programming languages justifies treating the design of production languages 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 methods to make programming languages 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.

The intent of this essay is to provoke thought regarding language design methodologies for language designers. We present some criticisms of historical language designs not to berate those designers for not using methods of which they were unaware, but instead to show how our methods could have been used to avoid making decisions that are now regarded as mistakes.

2 Context of Language Design

The goals of a particular language design depend on the intended set of programmers and their backgrounds as well as the target applications for programs written in the language. Below, we decompose the space of language goals into categories. For each category, we show in Table 1 several relevant *quality attributes* [36]. By using a diverse set of methods, a designer can obtain many different kinds of evidence regarding many different kinds of design questions.

1. *Formal properties* concern the mathematical properties of the language, separate from its environment.
2. *Observational properties* characterize how a language affects programs as they are compiled and executed in the real world. These properties are affected by the language design as well as by the language implementation. For example, dynamic dispatch incurs some runtime costs relative to static dispatch; a language designer may avoid these costs by requiring static dispatch (although then the programmer must likely manually replicate some of avoided dynamic dispatch work).
3. *Effects on programmers*: although much of the impact of a language is on its direct users, the science of designing for programmers is much less developed than the science of reasoning about programs directly (in terms of their behavior and performance).

The context of language design also includes the historical and computational environment in which the language is to be used. Some of the methods we propose in this paper might not have been appropriate to apply early in the history of computing. For example, consider the early development of LISP [38]. In that environment, there were few users from whom one might try to gather data, and few machines on which to experiment. Although this may reflect the context of some domain-specific languages today, many languages are now targeted at larger audiences of programmers, whom the designer may not understand well.

Table 1. Common language design objectives

Property	Cat.	Summary
Type safety	1	A type-safe language guarantees that all programs written in the language will behave only according to the language specification, rather than exhibiting undefined, potentially dangerous behavior.
Correctness guarantees	1	Absence of particular bugs or adherence to formal specifications
Computational power	1	Model of computation, e.g. Turing-completeness
Security properties	1	Formal properties pertaining to security, e.g. information flow
Efficiency	2	Execution cost
Portability	2	Execution on different platforms
Compilation time	2	Time to compile individual files and whole systems
Learnability	3	How hard is it for people of particular backgrounds to learn the language?
Error-proneness	3	To what extent does the language make it easy for programmers to write buggy code?
Expressiveness	3	To what extent can users specify their intent using the formal mechanisms of the language?
Understandability	3	How easy is it for readers of code to answer their questions about the program?
Ease of reasoning	3	How easy or hard is it for readers to draw inferences about properties of programs?
Modifiability	3	How easy or hard is it to adapt software to changing requirements?
Local reasoning	3	To what extent is it possible to make inferences about software by understanding small pieces of larger systems?
Coordination	3	How does the language facilitate coordination of activities among multiple developers?

Some approaches that were available early in the history of language design seem infeasible for general-purpose computers in 2018. For example, LISP was designed in a tightly coupled way with the machines on which it would run, and inspired custom hardware according to the language semantics [38]. Now, tight coupling with significant hardware architectural modifications for the purpose of running a novel programming language is unlikely. On the other hand, this approach may serve as a model for designing languages for special-purpose hardware, such as GPUs or embedded systems.

Many languages are designed or maintained by committees of experts. We argue in this paper that these committees would do well to take data from users into account – not as a replacement for their experience, but as a supplement to it. An analogous approach is used in the design of user interfaces: companies employ expert designers to make the best decisions they can, but even so, they gather data from users throughout the design process.

3 Integrating Design Perspectives

3.1 Stereotypes

For comparison and discussion purposes, we describe several stereotypical approaches to language design. Our intent here is to draw contrasts between important language design styles and observe that although designers from a different perspectives may have different priorities, they can each benefit from using more diverse methods than are currently in use.

A **logician** is primarily concerned with the relationship between programming and mathematics. A logical approach is useful for quickly eliminating from consideration many designs that are not internally consistent. Viewing programming as the practice of writing *correct* programs — that is, programs that meet particular mathematical specifications — the logician is focused on the concise, convenient, and 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. In the logician's view, the best programming languages are those that are *discovered*, rather than designed [77]; these languages' constructs follow inevitably from the Curry-Howard correspondence between programs and proofs in mathematical logic.

A **pragmatist** is interested in designing languages that are effective for software systems in order to achieve various pragmatic or commercial goals. Some pragmatists think in terms of productivity [39], but others think in terms of exploration [31, 59]. Another kind of pragmatist is interested in using programs to understand some phenomenon of interest. The common thread among pragmatists is that the language is a tool for doing some other kind of activity, and the language must be evaluated against that activity. We can think of *effectiveness* as pertaining to programmers' abilities to achieve their goals. As such, performance and adoption (which depends on many different attributes, including learnability [40] and interoperability) are often priorities. In many cases, a pragmatic approach is community-oriented, as in the Java Community Process and the Python Enhancement Proposals mechanism. In other cases, a pragmatist may be focused on their own needs, creating a language for a particular domain or industry. Of course, design of a completely new language is relatively infrequent; instead, designers make incremental changes to languages in order to make them more effective for users.

Some pragmatists design languages in the context of a project that would benefit from a new language. For example, C was designed in the context of the UNIX operating system [57]. Designers of domain-specific languages may be embedded in the domain rather than focusing on language design per se. For example, SQL was developed by database researchers [13]. Lua was designed in part for Petrobras, a Brazilian petroleum company [28].

A pragmatic approach is useful for designing serviceable languages. Risk aversion frequently results in the selection of well-proven techniques, such as object-oriented and imperative programming. It is not necessary for designers 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 quickly and cost-effectively finishing a useful design. 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.

An **industrialist** is interested in designing languages that are effective for writing large software systems in order to achieve various pragmatic or commercial goals. As such, performance and adoption (which depends on many different attributes, including learnability [40] and interoperability) are often priorities. In many cases, an industrial approach is community-oriented, as in the Java Community Process and the Python Enhancement Proposals mechanism. These approaches codify methods used to evolve programming languages that are in use. Of course, design of a completely new language is relatively infrequent; instead, practitioners evolve existing languages in order to make them more effective.

An **empiricist** views programming languages as critical tools for programmer productivity. The methodological focus is on using carefully designed experiments to demonstrate 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 programmers; after gathering sufficient data, designers will be able to make a large portion of their design choices on the basis of experimental evidence. The Quorum programming language claims to be the first evidence-oriented programming language [67]. Quorum incorporates empirically validated results obtained to date. For example, empirical methods have been used to show, for example, that certain static type systems have particular benefits over dynamic type systems in specific situations [20].

An **educator** focuses on pedagogical benefits of programming languages: to what extent will learning and using a particular language achieve particular educational objectives? 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 [29, 55]. Logo uses a graphical environment to make abstract concepts more concrete and fun [49]. Some educators prioritize real-world applicability, preferring to teach languages that are in current industrial use.

3.2 Interdisciplinary Design: Calls to Action

Each of the stereotypes is useful for language design and yet individually too limited, focusing on particular design goals but not others.

While the *logician's* formal methods can link language constructs to program properties, they cannot directly tell us which program properties are the most important. Logicians sometimes argue that programming constructs derived from mathematics may also relate closely to the way the logician is thinking, as in the *closeness of mapping* usability heuristic [23]. Unfortunately, this has yet to be shown empirically; the logical community has not yet been convinced that human factors-related methods are relevant. There are promising exceptions, though. Hudak et al. conducted a study comparing Haskell to several other languages [27]. The authors refer to an experiment, but we think of this as a case study because each language was used once by one or two (non-randomly-assigned) programmers each, so it is not clear how to generalize the results. The case study resulted in Haskell implementations that appeared to be shorter than those in other languages, but no statistics were computed (or would have been appropriate to compute) for the study. The authors also report several subjective quality metrics that were assessed by a panel of experts, but it is not clear whether these metrics correlate with any kind of real-world programmer performance.

We challenge the Haskell community (and other related communities) to provide direct empirical evidence of the benefits of their approaches via randomized controlled trials. The question of purity is a particularly interesting one: the language design centers around the supposed benefits of purity, but we lack evidence regarding whether, overall, the tradeoff is a good one. If purity is helpful, then for which users and which applications – and for which users and applications is it harmful? Does hiding side effects inside monads actually help programmers, or is the net effect that the program is *more difficult* to write and maintain?

The *empirical* approach focuses on summative evaluations. 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 enable 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.

Some researchers have argued that the programming languages community might look to the field of medicine for insight regarding appropriate evidence in scientific fields [64, 66]. We observe, however, that 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 [19, 58, 63]. Notably, controlled trials form only one component of the three; the medical community considers other relevant aspects of a clinical situation when recommending treatment. In addition, an epidemiological approach considers the population-level effects of decisions rather than just the effects on individuals.

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.

When comparing to a medical approach, is important to consider that medical trials are done to evaluate treatments, not to design them. Before starting a drug trial, drug designers use separate methods to design new drugs; evaluation of efficacy is done much later [21]. Furthermore, fortunately for programming language designers, the risks of testing a bad design are substantially lower than in drug designs, which suggests that a less risk-averse process is likely appropriate.

We challenge the empiricists, then, to develop ways of validating *formative design techniques*: methods that can be used to elicit evidence from users in the absence of a working system that can be evaluated. How can we know whether insights obtained from qualitative work with programmers are likely to generalize?

The perspective of the *educator* 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. This approach was incorporated into the design of C++: "If in doubt, pick the variant of a feature that is easiest to teach" [69]. However, languages that focus on pedagogical goals may not be ideal for creating large, complex systems. Educators must choose whether to prioritize teaching particular aspects of programming so that students can be effective when using other languages, or to prioritize practical application. There is a tradeoff of *authenticity*: students who learn languages that are not used for "real" development may feel they are not learning authentic programming methods.

Our challenge for educators is twofold. First, they should ensure that their perspective reaches past the question of how to teach students how to use *specific programming constructs* (e.g. `for` loops) and into the question of how to design languages that *facilitate reasoning about computation*. By doing so, they may uncover new approaches to programming that make all programmers more effective — not just novices.

3.3 Use of Multiple Methods

The number of design decisions involved in designing a particular programming language is immense; we hope that future work will analyze this space comprehensively, but our experience suggests that there are at least thousands of decisions that are made in the design of any given language. These include 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 [52, 61], it is our experience that many language design *decisions* are not orthogonal. For example, in a language we are working on now, the design of an alias control mechanism interacts with the design of a mechanism that facilitates static reasoning about state. We argue, then, that it is risky to combine the results of individual experiments without performing an additional, more holistic evaluation: one 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 using 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 practically and applied to a wide variety of different language designs.

An important aspect of an interdisciplinary approach is that it allows us to collect detailed qualitative results regarding tradeoffs among different designs. Rather than focusing on *whether* a particular design promotes faster task completion times compared to another, we seek to learn *why* [34]: 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 later 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 [43] rather than limiting the use of user-oriented methods to 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 [11]. 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 [78]. 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.

4 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 or a finished design). Each method is used to obtain data, but the validity of the data depends on the method and how it is applied. Importantly, validity is not a binary concept. One cannot say that a use of a method was valid; instead, one must enumerate the threats to validity and discuss how those threats were mitigated in the study. Key kinds of validity that trade off include:

- **External validity** considers to what extent the data generalize to other situations. For example, the results of a study involving undergraduates may not generalize to professional software engineers.
- **Internal validity** asks to what extent the results may be confounded by variables that were not considered. For example, although participants were randomly assigned to the experimental conditions, the experimenter might (unwisely) run all of the participants in one condition before all of the participants in the other condition, risking a confounding variable of time (perhaps a major world event occurred later in the study, impacting the later participants' ability to focus on the experiment).

The methods, which are described below, are also summarized in Table 2.

4.1 Methods for Requirements and Creation

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 extension to Java, 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 would be 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.

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 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 [74], 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).

Corpus studies can show the prevalence of particular patterns in existing code, including patterns of bugs in bug databases. For example, Callaú et al. [12] investigated the use of dynamic features in Smalltalk programs, Malayeri et al. [37] investigated whether programs might benefit from a language that supported structural subtyping, and we studied how Java programmers used exception handling features [30]. 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 [76]. 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 [43] 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 [47], a programming environment for children, as well as professionally targeted languages, such as blockchain programming languages [2]. 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 human-computer interaction research, and can be used for language design as well [42]. 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, which threatens validity.

Participatory design [9, 46] invites domain experts to help explore the design space and analyze tradeoffs. The assumption is that their specific expertise is likely to complement the general language design expertise of the language designer.

Programming language and software 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 transitive immutability for the Glacier type system [16]. Similarly, an understanding of how modularity affects modifiability from the software engineering literature [50] motivates the module systems present in many languages, and more recent theories about how software architecture [62] 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.

4.2 Methods for Evaluation

Qualitative user studies have been used to evaluate many different kinds of tools, including programming languages [16, 47], APIs [44], and development environments [33]. 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.

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 [71]. 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 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 [45], 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 [5] for empirical evaluations of programming languages; although is title is "Empirical Evaluation Checklist," it describes only performance evaluations. The checklist hints at limitations of this approach, such as a 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 [75], static type systems [20], and typechecking [53]. In some ways, RCTs 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 about C++ lambdas might apply [75], but not how one might improve 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 [51]. 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. Recently, Misailovic et al. have argued that in many cases, approximating the language semantics suffices [41].

Formal verification via tools such as Dafny [35] or Coq [6] 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 unwritten the specification that *the program shall not overflow the stack for input within the expressible range of machine-size integers*.

5 Examples

In this section, we show how combinations of the above methods have been helpful in particular examples of programming language designs. We also relate cautionary tales showing how specific language design mistakes may have been prevented if the designers had applied the methods we suggest in this essay.

5.1 Exemplars

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* [68]. Two of us were involved in a decade-long interdisciplinary research project

Table 2. Summary of methods

Method	Phases supported	Key benefits	Challenges and limitations
Interviews	Requirements, Creation	Gathers open-ended qualitative data from experts	Depends on skill of interviewer and selection of participants; results may not generalize
Surveys	Requirements, Creation	Assesses opinions among a broad audience; can generalize interview results	Output is subjective; may not reflect reality
Corpus studies	Requirements, Creation	Assesses incidence of problems or applicability of solutions in a large dataset	Depends on appropriate datasets and efficient methods of analysis
Natural programming	Requirements, Creation	Obtains insights from people without biasing them toward preferred solutions	Data may be biased toward participants' prior experiences
Rapid prototyping	Requirements, Creation	Facilitates efficient design space exploration	Lack of fidelity in prototypes may hide faults
Programming language theory	Requirements, Creation, Evaluation	Ensures sound designs	High cost; applying formal methods too early may limit ability to iterate, but applying too late can waste time on unsound approaches
Software engineering theory	Requirements, Creation, Evaluation	Improves practicality of designs	Unclear how to prioritize recommendations when they conflict
Qualitative user studies	Requirements, Creation, Evaluation	High-bandwidth method to obtain insight on user behavior when using systems	Results may not generalize; Results depend on skills of experimenter and participants
Case studies	Evaluation	Tests applicability of systems to real-world cases; allows in-depth explorations of real-world difficulties	Requires finding appropriate cases; generalizability may be limited
Expert evaluation	Evaluation	Benefit from experience acquired by experts	Biased by opinions of experts, which may not reflect real-world implications of the design
Performance evaluation	Evaluation	Reproducible way of assessing performance	Results depend heavily on selection of test suite
User experiments	Evaluation	Quantitative comparison of human performance across systems	Results may not generalize to non-trivial tasks, other kinds of participants, expert users, long-term use, or use on large systems
Formalism and proof	Requirements, Creation, Evaluation	Provides definitive evidence of safety	Results are limited to the specific theorems proven

that illustrates how different research methods complement one another in exploring language and type system support for typestate.[1]

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 [4]. 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 [71]. We found that when performing these tasks, developers spent 71% of

[1]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.

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 have gotten with either study in isolation.

We designed typestate support both as a set of annotations and an analysis on top of Java, and as a separate language, Plaid [72]. Formal models of a typestate checking system in each setting were proved sound [8, 22]. 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 [3, 7], and in Plaid, verifying that the language could express complicated state machines in real examples ported from Java [72].

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 [70]. 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 StackOverflow tasks. This example shows that a properly designed pairing of experiments and formative studies can be much more convincing than either study in isolation.

Glacier [15] is an extension to Java that supports transitive class immutability [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 [10, 24, 25, 32, 73] 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 [79], 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 an RCT. 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.

Does Glacier's simplicity restrict its utility to contrived tasks in lab studies? We conducted a *case study* applying Glacier to real-world systems [15]. 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. In fact, 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.

AppleScript is a scripting language that was designed in 1991. Unlike the above examples, which were academic research projects, AppleScript was designed in a commercial environment. The designers had a practical goal: allow users, many of whom would not be trained as programmers, to write scripts that automate tasks involving GUI applications. Although it was based on an existing language, HyperTalk, the designers wanted it to be a more general way of interfacing with applications via Apple Events, the platform's mechanism for inter-process communication.

The AppleScript team collected requests for features in *interviews* and in a focus group; they also collected early feedback from key developers [18]. In addition, they used user studies to assess the usability of their language. For example, they asked novice users what behavior they expected of particular code. Users were also asked for their preferences: for example, between `window named "fred"` and `window "fred"`. This approach is one example of earlier work that resembles the *natural programming* technique [43]. For example, an early version used the syntax `put x into y`, but participants thought that after executing that statement, `x` would no longer be defined. The designers changed the language to use `copy x into y` instead.

The work on AppleScript also included controlled user experiments, but Cook reported that they were done too late to affect the design of the language [18]. Perhaps surprisingly, although the team included a member with a background in formal semantics, formal methods were not a significant part of the development process – in part because the other members were unfamiliar with those techniques, making them ineffective for communication, in part because that work would not have yielded interesting results for the traditional parts of the language, and in part because the required time was not available for the novel parts of the language. Most of the language design insight came from user studies and informal case studies.

5.2 Cautionary Tales and Missed Opportunities

Dennis Ritchie wrote about the design of the C language [56]. Interestingly, some of the design decisions in C came out of the experience of convenience of the language designers. For example, part of the justification for using terminators at the ends of strings rather than storing the size of strings separately was convenience. Unfortunately, this decision has led to decades of security vulnerabilities [60]. The decision may have been convenient for programmers, but it disregarded the bug-prone nature of this design decision. Perhaps user studies would have revealed the risk of this kind of bug, motivating a design change. Although security was not a design priority at the time, the designers surely knew of the risk of bugs in general.

Ritchie also described the process by which the operator precedences were established [56], based on historical precedent from the B language and the B-language example, `if (a == b & c)`. This code compares `a` to `b` and then checks whether `c` is nonzero. The precedence of `&&` was inherited from `&`, which binds *less* tightly than `==`. Then, Ritchie commented: "Today, it seems that it would have been preferable to move the relative precedences of `&` and `==`, and thereby simplify a common C idiom: to test a masked value against another value, one must write `if ((a & mask) == b)` where the inner parentheses are required but easily forgotten." This represents a missed opportunity for empirical studies: perhaps they could have revealed the error-prone nature of this language decision and motivated a change to prevent a large number of bugs.

Learners of C sometimes ask about the distinction between arrays declared with empty brackets and pointers, e.g. the function `void f(int a[])` vs. `void f(int *a)`. This was a remnant from a predecessor of C called NB, with the idea that it would communicate to readers that `a` should be interpreted as an array rather than a pointer to a scalar, but Ritchie later viewed this as confusing [56]. Perhaps empirical studies would have revealed this language design flaw too.

Tony Hoare included NULL references in ALGOL. He later argued that the inclusion of NULL references was a *billion-dollar mistake* [26]: "But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years." In this essay, we argue that language design should be done with respect to the behavior of the users of the language, not with respect to the convenience of the language designer. Hoare now seems to agree [26]: "[Language design] is a serious activity; not one that should be given to programmers with 9 months experience with assembly; they should have a strong scientific basis, a good deal of ingenuity and invention and control of detail, and a clear objective that the programs written by people using the language would be correct."

Hoare hypothesized: "By investigating the logical properties of your programming language and finding out how difficult it would be to prove correctness if you wanted to, you would get an objective measurement of how easy the language was to use correctly. If the proof of program correctness requires a lot of proof rules and each rule requires a lot of side conditions... then you know that you've done a bad job as a language designer. You do not have to get your customers to tell you that." We challenge the community to evaluate this hypothesis scientifically rather than either ignoring it or taking it as an assumption.

6 Conclusions

We summarize the lessons of this essay as follows:

- Language design should be interdisciplinary, applying a wide variety of methods
- Designers should use more human-oriented, qualitative, and formative methods
- Designers should draw more on empirically based software engineering principles
- The application of one method should be guided by results from complementary methods
- Methods should be chosen to mitigate critical risks to achieving the language design goals

Our experience applying the ideas above indicates that, while every design method has limitations, an interdisciplinary approach to combining theoretical methods with quantitative and qualitative user-oriented methods is effective 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; indeed, none of us is an expert in all the methods mentioned in this paper, nor even all the ones used in our collective research. Instead, we recommend collaborative efforts, where designers work together to apply theoretical, formative, and summative techniques in order to provide evidence of relevant properties, explore fruitful portions of the design space, and show that their designs benefit users in specific, quantifiable ways.

Acknowledgments

We appreciate the insightful comments from James Noble, Mary Shaw, Richard Gabriel, the anonymous reviewers, and the participants of Dagstuhl Seminar 18061. This material is based upon work supported by the US Department of Defense, by NSF grants CNS-1734138 and 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. Michael Coblenz is supported by an IBM PhD fellowship. 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. https://doi.org/10.1145/581339.581365

[2] 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*.

[3] 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.

[4] 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*.

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

[6] 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.

[7] 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.

[8] 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. https://doi.org/10.1145/1297027.1297050

[9] 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. https://doi.org/10.1145/572020.572023

[10] 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.). https://doi.org/10.1007/3-540-45337-7_2

[11] Bill Buxton. 2007. *Sketching user experiences: getting the design right and the right design*. Morgan Kaufmann. https://doi.org/10.1016/b978-0-12-374037-3.x5043-3

[12] 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. https://doi.org/10.1145/1985441.1985448

[13] Don Chamberlin. 2012. Early History of SQL. *IEEE Ann. Hist. Comput.* 34, 4 (Oct. 2012), 78–82. https://doi.org/10.1109/MAHC.2012.61

[14] Sarah Chasins. 2012. *An Efficient Implementation of the Plaid Language*. Honors thesis. Swarthmore College Honors Thesis.

[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*. https://doi.org/10.1109/ICSE.2017.52

[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*. https://doi.org/10.1145/2884781.2884798

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

[18] William R. Cook. 2007. AppleScript. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 1–1–1–21. https://doi.org/10.1145/1238844.1238845

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

[20] 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. https://doi.org/10.1145/2568225.2568299

[21] FDA. 2018. The Drug Development Process > Step 3: Clinical Research. https://www.fda.gov/ForPatients/Approvals/Drugs/ucm405622.htm

[22] 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. https://doi.org/10.1145/2629609

[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. https://doi.org/10.1006/jvlc.1996.0009

[24] Christian Haack and Erik Poll. 2009. Type-based Object Immutability with Flexible Initialization. In *European Conference on Object-Oriented Programming*. https://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] Tony Hoare. 2009. Null references: The billion dollar mistake. *Presentation at QCon London* 298 (2009).

[27] Paul Hudak and Mark P Jones. 1994. Haskell vs. Ada vs. C++ vs. awk vs.... an experiment in software prototyping productivity. *Contract* 14, 92-C (1994), 0153.

[28] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2007. The Evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 2–1–2–26. https://doi.org/10.1145/1238844.1238846

[29] Caitlin Kelleher and Randy Pausch. 2005. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Comput. Surv.* 37, 2 (June 2005), 83–137. https://doi.org/10.1145/1089733.1089734

[30] 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. https://doi.org/10.1145/2901739.2903497

[31] Mary Beth Kery and Brad A Myers. 2017. Exploring exploratory programming. In *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*. IEEE, 25–29. https://doi.org/10.1109/VLHCC.2017.8103446

[32] 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

[33] 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). https://doi.org/10.1109/TSE.2006.116

[34] 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. https://doi.org/10.1145/1833310.1833322

[35] 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

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

[37] 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. https://doi.org/10.1007/978-3-642-00590-9_8

[38] John McCarthy. 1981. History of Programming Languages I. ACM, New York, NY, USA, Chapter History of LISP, 173–185. https://doi.org/10.1145/800025.1198360

[39] André N. Meyer, Thomas Fritz, Gail C. Murphy, and Thomas Zimmermann. 2014. Software Developers' Perceptions of Productivity. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 19–29. https://doi.org/10.1145/2635868.2635892

[40] 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. https://doi.org/10.1145/2384592.2384597

[41] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 309–328. https://doi.org/10.1145/2660193.2660231

[42] 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. https://doi.org/10.1109/MC.2016.200

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

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

[45] 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. https://doi.org/10.1145/97243.97281

[46] 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. https://doi.org/10.1145/1753326.1753707

[47] 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. https://doi.org/10.1109/HCC.2002.1046372

[48] 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

[49] Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, Inc., New York, NY, USA.

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

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

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

[53] 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. https://doi.org/10.1109/32.677186

[54] 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. https://doi.org/10.1145/3126905

[55] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch:

Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[56] Dennis M. Ritchie. 1993. The Development of the C Language. In *The Second ACM SIGPLAN Conference on History of Programming Languages (HOPL-II)*. ACM, New York, NY, USA, 201–208. https://doi.org/10.1145/154766.155580

[57] Dennis M. Ritchie. 1996. History of Programming languages—II. ACM, New York, NY, USA, Chapter The Development of the C Programming Language, 671–698. https://doi.org/10.1145/234286.1057834

[58] 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.

[59] D. W. Sandberg. 1988. Smalltalk and Exploratory Programming. *SIGPLAN Not.* 23, 10 (Oct. 1988), 85–92. https://doi.org/10.1145/51607.51614

[60] Robert C Seacord. 2013. *Secure Coding in C and C++*. Addison-Wesley.

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

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

[63] 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. https://doi.org/10.1002/jclp.20373

[64] 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. https://doi.org/10.1145/2661136.2661156

[65] A. Stefik and S. Hanenberg. 2017. Methodological Irregularities in Programming-Language Research. *Computer* 50, 8 (2017), 60–63. https://doi.org/10.1109/MC.2017.3001257

[66] 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. https://doi.org/10.1145/2597008.2597154

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

[68] 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. https://doi.org/10.1109/TSE.1986.6312929

[69] Bjarne Stroustrup. 2007. Evolving a Language in and for the Real World: C++ 1991-2006. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 4–1–4–59. https://doi.org/10.1145/1238844.1238848

[70] 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)*. https://doi.org/10.1007/978-3-662-44202-9_7

[71] 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

[72] 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.

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

[74] 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. https://doi.org/10.1145/3133850.3133863

[75] 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. https://doi.org/10.1145/2884781.2884849

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

[77] Philip Wadler. 2015. Propositions As Types. *Commun. ACM* 58, 12 (Nov. 2015), 75–84. https://doi.org/10.1145/2699407

[78] 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.

[79] 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. https://doi.org/10.1145/1287624.1287637