Exploring Language Support for Immutability

Michael Coblenz, Joshua Sunshine,
Jonathan Aldrich, Brad Myers
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA, USA
{mcoblenz, sunshine,
aldrich, bam}@cs.cmu.edu

Sam Weber,
Forrest Shull
Software Engineering Institute
4500 Fifth Avenue
Pittsburgh, PA, USA
samweber@cert.org,
fjshull@sei.cmu.edu

ABSTRACT

Programming languages can restrict state change by preventing it entirely (immutability) or by restricting which clients may modify state (read-only restrictions). The benefits of immutability and read-only restrictions in software structures have been long-argued by practicing software engineers, researchers, and programming language designers. However, there are many proposals for language mechanisms for restricting state change, with a remarkable diversity of techniques and goals, and there is little empirical data regarding what practicing software engineers want in their tools and what would benefit them. We systematized the large collection of techniques used by programming languages to help programmers prevent undesired changes in state. We interviewed expert software engineers to discover their expectations and requirements, and found that important requirements, such as expressing immutability constraints, were not reflected in features available in the languages participants used. The interview results informed our design of a new language extension for specifying immutability in Java. Through an iterative, participatory design process, we created a tool that reflects requirements from both our interviews and the research literature.

CCS Concepts

•**Software and its engineering** → **Language features;** *Software development techniques;*

Keywords

Programming language design, Programming language usability, Immutability, Mutability, Programmer productivity, Empirical studies of programmers

1. INTRODUCTION

Many designers of APIs and programming languages recommend using immutability in order to prevent bugs and security flaws. For example, Bloch devoted a section of his book *Effective Java* [6] to minimizing mutability. He cited the following benefits of immutability: simple state management; thread-safety; and safe and efficient sharing. Oracle's *Secure Coding Guidelines for Java SE* [31] states that immutability aids security. Microsoft's *Framework Design Guidelines* recommends against defining mutable value types in part because they are passed by copy, and programmers might write code that modifies copies but should instead modify the original structure [29]. Some programming languages, such as Rust [30], take these recommendations into account by carefully managing mutability. The functional programming language community is particularly concerned with state management, producing languages such as Haskell, which segregates code that manipulates state from code that does not. Proponents of functional languages argue that avoiding mutable data structures facilitates reasoning about behavior of programs because one can reason equationally about behavior rather than needing to know about the global or even local program state [1].

There are questions, however, about what immutability should mean and how to express immutability in programming languages, as evidenced by the myriad of different kinds of support that tools provide and the lack of empirical data to justify specific design choices. Some languages support programmer-provided specifications of immutability (expressing that certain data structures cannot be changed) or read-only restriction (expressing that certain references cannot be used to change a data structure). For example, `final` in Java can express that a particular field cannot be reassigned to refer to a different object, but the contents of the referenced object may still change. Furthermore, there is no way to express class-level immutability directly.

In C++, `const` data can still refer to non-`const` data, which can then be changed. Furthermore, `const` provides no guarantees regarding *other* references to the same object. This means that in addition to not providing the expected benefits of immutability to programmers, such as thread safety and simple state management, these annotations also do not provide the guarantees that would be needed for the many compiler optimizations that require all of an object's behavior to be guaranteed to be fixed.

This paper makes the following contributions:

- After reviewing the existing literature and implemented systems in this area, we develop a classification system for mutability restrictions in programming languages (§2), show where some existing systems fit in this classification (§3), and discuss possible usability

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884798

implications of existing design choices (§4).
- We interviewed eight expert software engineers to find out how practitioners feel about existing language features and tools, and what their perceived needs are. We show that existing programming language features and research tools do not adequately address the issues encountered by practitioners. We extract design recommendations from the results of our interviews (§5).
- IGJ is a Java extension that adds annotations specifying immutability and read-only restrictions [42]. We describe the design of *IGJ-T*, an extension to IGJ that enforces *transitive immutability*, which addresses problems that our interview participants described. We iteratively evaluated IGJ-T with three pilot users and refined our study based on the feedback (§6).

2. OVERVIEW OF CONCEPTS

In existing literature, the wide variety of approaches and subtle differences among the different goals can make it difficult to understand which problems each system addresses. Thus, it is important to distinguish among the many different meanings given to the term *immutability* and related concepts. In this section, we will give an overview of the various mechanisms and issues. Up to this point, we have been using the term *immutability* informally; here we synthesize definitions from existing literature to form a definition that will be used in the rest of the paper. A summary of the concepts appears in Table 2.

We use *object* here to mean any kind of state, such as a C `struct` or a functional language ref cell. *State* is data that is stored in memory. As an abbreviation, we will use *function* to refer to both functions and methods.

2.1 Type of Restriction

Immutability restricts modification through *all* references to a given object (see §2.5 for exceptions). In contrast, *read-only* restrictions disallow modification through *read-only* references to a given object, but not necessarily through all references. The distinction between read-only restrictions and immutability is critical to the correctness of programs. For example, an immutable object can be shared safely among threads without locks, but a function that has a read-only reference to an object has no guarantee that the object cannot be modified, so that function must assume the object may be mutable.

Assignability restrictions, such as `final` in Java, disallow assignment. In most imperative languages, variables are backed by storage, so assignability restrictions represent non-transitive immutability (see §2.3). Java's `final` keyword on fields is an *assignability* restriction. Although a `final` field can never point to a different object than the one it was initialized to point to, the referenced object's fields may themselves still be modified. In contrast, the C declaration `const int *x` provides *read-only* restrictions: `x` is a pointer to `int` and might later refer to some other address, but the reference `x` cannot be used to change the value at any memory location to which `x` points.

Ownership systems define a system-specific notion of ownership and a way of specifying which objects a given object owns. This enables enforcement of restrictions such as "an object may only directly modify objects it owns". *Ownership types* [10] use a notion of object context to partition the object store. This partitioning allows the systems to ensure that aliases to objects do not escape their owners, ensuring representation containment and defining a notion of abstract state, since the abstract state (§2.5) of an object includes only data it owns. Ownership is also useful for ensuring thread safety of mutable structures by ensuring that locks are acquired in a correct order according to the ownership structure before accessing objects [7].

2.2 Scope

Object-based restrictions apply to a particular object. *Class-based* restrictions apply to all of a class's instances. For example, `final` is object-based: it only applies to a specific reference and there is no way of specifying that all references to instances of a particular class are `final`.

Class restrictions, while commonly either only supported as syntactic sugar for object restrictions (e.g. IGJ) or unsupported entirely, are frequently needed in practice according to our interview participants. By necessity, many programmers who want class immutability must improvise (§3.2).

2.3 Transitivity

Transitive restrictions pertain to all objects reachable from a given object, including objects captured by closures or methods. *Non-transitive* restrictions pertain only to the immediate fields in a given object.

Non-transitive restrictions provide weak guarantees because they say little about the *behavior* of the abstraction that the object's interface provides. For example, if a list object is non-transitively immutable, then the number of items in the list is fixed, and the list always refers to the same objects, but the contents of those objects can still be changed. A function that needed to know that a list of integers only contained positive integers would need to re-check all elements after every possible opportunity for mutation of list elements. However, in order to ensure that an object is immutable, one must verify that all objects in the transitive closure of references from that object are immutable.

Though the assignability and read-only features provided in many popularly-used languages are non-transitive, including Java's `final` and C++'s `const`, researchers have proposed transitive restrictions in a variety of different systems. Because of the important implications of this design decision and the discrepancy between research and practice, we will focus on this question in §6.1.

2.4 Initialization

Systems might *relax* restrictions during initialization in order to facilitate initialization. A common method for creating a cyclic data structure involves modifying an element after it is created. The cyclic data structure may be mutable during initialization but immutable afterward. Alternatively, systems can *enforce* restrictions during initialization.

2.5 Abstract vs. Concrete State

The *abstract state* of an object refers to the portion of the state of an object that is conceptually a part of that object. For example, in a splay tree, sometimes during a read operation the internal tree structure will be rotated in order to move frequently accessed elements closer to the root. Even though this is a change to the internal data structure, it is not exposed to the caller and therefore read operations do not change the splay tree's abstract state. In contrast, the *concrete state* includes the object's entire representation.

Table 1: Summary of Some Existing Systems (abbreviations are from Table 2)

System	Type	Scope	Trans.	Init.	Abstr.	Compat.	Enforcement	Polymorph.
Java `final`	a	o	n	e	N/A	c	s	n
C++ `const`	r	o	n	e	a	c	s[1]	n
Obj-C immutable collections	i	o	n	e	N/A	c	s[1]	n
.NET Freezable [22]	i	o	n	e	N/A	o	d	n
Java `unmodifiableList`	r	o	n	e	N/A	o	d	n
Guava `ImmutableCollection`	i	o	n	e	N/A	o	s, d[2]	n
IGJ [42]	i, r	c, o	n	e	a	c	s	p
JAC [19]	r	o	t	e	c	c	s	n
Javari [38]	r	c, o	t	e	a	c	s	p
OIGJ [43]	i, r, o	c, o	n	r	a	c	s	p
`immutable` [16]	i, r, o	c, o	t	r	a	o	s	p
C# isolation extension [13]	i, r, o	c, o	t	r	a	c	s	p
JavaScript `Object.freeze`	i	o	n	e	c	o	d	n

[1] These approaches provide static enforcement to the extent possible in these languages.
[2] Static deprecation warning, runtime exception

Table 2: Summary of Dimensions

Dimension	Possible choices
Type	**i**mmutability, **r**ead-only restriction, **a**ssignability, **o**wnership
Scope	**o**bject-based, **c**lass-based
Transitivity	**t**ransitive, **n**on-transitive
Initialization	**r**elaxed, **e**nforced
Abstraction	**a**bstract, **c**oncrete
Backward compat.	**o**pen-world, **c**losed-world
Enforcement	**s**tatic, **d**ynamic
Polymorphism	**p**olymorphic, **n**on-polymorphic

Caches are often excluded from the abstract state because the contents of the cache duplicate information available elsewhere, and other than causing performance differences, the contents of the cache are invisible to clients. The abstract state might also exclude state kept only for debugging purposes. By excluding the cache from the abstract state of the object, writing to the cache can be considered a *benign* operation and can therefore be done by clients without write access. However, this allows the possibility of race conditions if the cache is not thread-safe but a programmer assumes it is because the object appears to be immutable.

Logical restrictions pertain to an object's abstract state. *Bitwise* restrictions govern an object's concrete state; this term arose in C, where programmers consider how structures are arranged in memory. Some languages provide features that let programmers differentiate these: in C++, for example, a `mutable` member variable can be modified even through a `const` reference. Unfortunately, it may be tempting for a programmer to assume that if all references to an object are `const`, the object is thread-safe, but the `mutable` members may be mutated in a thread-unsafe way. Thus, an object that is only logically-immutable may not be thread-safe.

2.6 Backward Compatibility

When facilities for restrictions are added to a programming language after the language is created, code that uses the extended features may need to interface with code that does not. If this is to be permitted, then there is a question of what guarantees are made. For example, if a reference to an immutable object is passed as input to an function whose interface does not specify restrictions, then the called function might mutate the object, violating the guarantee. Systems that make a *closed world* assumption assume that all code that uses code with restrictions itself has any restrictions declared. In contrast, the *open world* assumption is that there may be interfaces with un-restricted code and that the system must provide guarantees for this code too, either by making conservative assumptions about these APIs or by checking conditions dynamically. Open-world systems that support class immutability must ensure that instances of immutable objects encapsulate their representations because otherwise clients may mutate the representation objects directly. Furthermore, open-world systems that support object immutability must ensure that immutable objects are not exposed to unchecked clients [16].

The *closed-world* assumption can be a significant impediment to adoption for language extensions, since in a system that makes a closed-world assumption, adoption in new code requires that all clients of that code also adopt the system.

2.7 Enforcement

Static restrictions are enforced at compile time. *Dynamic* restrictions are enforced at runtime.

Static enforcement typically has a problem of *virality*: in order to call a method on an object that has an immutability or read-only restriction, the method typically must guarantee that it will not modify state, but if the method itself calls methods, then those methods must also be so guaranteed, and so on for the transitive closure of methods called by the first method. This can be burdensome if the guarantees must be annotated by programmers. In addition, the static analysis must be conservative, and therefore may give errors on code that is actually safe. C++'s `const` is viral, and programmers complain that "const-correctness" is therefore difficult to achieve.

Anders Hejlsberg, the lead C# architect, when asked to explain why C# did not offer C++'s `const` feature, stated the problem quite bluntly:

> We hear that complaint all the time too: "Why

> don't you have const?" Implicit in the question is, "Why don't you have const that is enforced by the runtime?" ...
>
> The reason that `const` works in C++ is because you can cast it away ... [41]

2.8 Polymorphism

Restriction polymorphism would mean that the same function can operate on inputs with different restrictions. Parametric polymorphism in restrictions is the most relevant example: restrictions can be specified in a parameter rather than explicitly so that one implementation can operate on inputs with different restrictions, while still obeying those restrictions. *Non-polymorphic* restrictions require programmers to write implementations that statically assume particular restrictions.

In C++, `const` restrictions are always expressed without polymorphism in `const`: each method's parameters and result are either `const` or not. The result is a common pattern: the programmer must write `const`- and non-`const` versions of many methods so that those methods can be invoked with both `const` and non-`const` inputs. For example, in C++, it is impossible to write a single "identity" method that takes a `const` or non-`const` array and returns the same array with the access restriction preserved; one must instead write two (typically overloaded) different methods. In contrast, IGJ [42] supports immutability parameters `@I` so that the restrictions can be expressed as a parameter of a type.

3. A SURVEY OF EXISTING SYSTEMS

Table 1 contains a summary of some related systems.

3.1 Historical and Research Systems

The functional programming community was one of the first proponents of *immutability*: language features that ensure that once a data structure is created it will always have the same value. In 1984 Abelson and Sussman [1] promoted immutability on the grounds that it supports formal reasoning and makes concurrent programming easier. Programs that use only immutable structures are called *pure.*

In contrast to functional languages, imperative languages emphasize mutation of program state. By adding features that allowed restrictions on what data structures could be modified in each context, language designers hoped to facilitate reasoning about programs. At first, languages provided features restricting what functions could do to their parameters. In Pascal [4], `var` parameters were passed in by reference, but non-`var` parameters were passed by value, preventing modification of the passed parameter. Modula-3 [9] provided a `READONLY` parameter annotation, which prevented a function from using the formal parameter in a context that would require an l-value, such as the left-side of an assignment statement. In Ada [3] parameters to functions could be declared `in`, `out` or `in out` to indicate whether the function could read but not modify, modify but not read, or both read and modify each parameter.

Later, support for modularity led to features that controlled mutation in entire modules. In Turing [18], a variable defined in module A could be imported non-`var` into module B which would prevent the variable from being changed by B, although A could still mutate it. A common attribute of these features was that data structure mutation would be permitted through some references but not others, i.e. these represented read-only restrictions. Notice that these features are all weaker than immutability: immutability guarantees that a data structure will never change, whereas read-only restrictions only state that changes will not occur via certain mechanisms or locations.

IGJ [42] provides Java annotations that implement immutability. For example, `@Immutable Date d` is a reference to an immutable date. No fields can be modified on an `@Immutable` object; IGJ verifies that no non-`@Immutable` references to an `@Immutable` object can be obtained. `@Immutable` specifies non-transitive immutability: if an `@Immutable` object's fields are not `@Immutable`, then those objects' fields may still be assigned to. `@Mutable`, the default annotation for unannotated fields, also grants exceptions permitting modification of fields in `@Immutable` objects.

`@Readonly` in IGJ specifies a read-only reference. The holder of a `@Readonly` reference cannot use that reference to modify the referenced object, but there may be other non-`@Readonly` references to the same object (*aliases*). IGJ also supports a form of class immutability, in which specifying `@Immutable` on a class serves as syntactic sugar in place of adding annotations in a variety of other places. Finally, IGJ supports an immutability parameter `@I`, which takes the value of another annotation. For example, if `@I` is the immutability parameter for a class and a field in that class is annotated with `@I`, then that field in a particular instance of the class will have an immutablity annotation according to the annotation of that instance. This is the sense in which IGJ supports transitive immutability: if all fields are annotated with the class's immutability parameter for all fields transitively included in that class, then immutability specified by a reference to an object will be transitive.

Unkel and Lam [39] generalized `final` to define *stationary* fields, for which all writes occur before all reads, and provided an algorithm to find them. Of fields in their corpus, 44-59% were stationary, but only 11-17% were `final`.

Haack et al. designed a Java-like language with a class modifier, `immutable`, which specifies that all class instances are immutable, and uses ownership types (with ownership annotations) to enforce encapsulation [16]. Later work by Haack and Poll [15] avoided the need for ownership types and supported object immutability, read-only references, and class immutability. They also permitted temporary modification of read-only structures after initialization by using stack-local memory regions to isolate new immutable objects.

Skoglund and Wrigstad [36] proposed a transitive read-only restriction system for Java, which includes the ability to check for read-only restrictions at runtime. Zibin et al. [42] argued that allowing runtime checks of restrictions hampers program understanding but there are no user studies confirming or refuting this claim. JAC [19] provides `readonly` (a transitive read-only restriction) as well as `readimmutable` (a transitive read-only restriction that also disallows reading of transitively mutable state) and `readnothing` (providing no access to state at all; suitable for pure functions). JAC extracts a readonly portion of each class by restricting return types: methods return read-only references when called on read-only objects. As a result, a programmer need only provide one implementation of a class, and JAC can generate restricted versions of the class. However, this approach has not been adopted by the Java community. In contrast,

Javari [38] provides a simpler access rights system (for example, JAC includes three different levels of read-only access, but Javari only has one).

One impediment to understanding and reasoning about programs is the general alias problem of determining what references exist to a given object. With a precise alias analysis, one could find out whether a given object might be modified by a particular call. Unfortunately, a precise *may-alias* analysis is undecidable [20]. Therefore, various approaches have been developed that facilitate reasoning about aliases by restricting which aliases can exist.

Noble, Vitek, and Potter [27] presented a system for *flexible alias protection* that provides several aliasing mode declarations that allow aliasing invariants to be checked statically. For example, one mode ensures that "a container's representation objects may be read and written, but must not be exposed outside their enclosing container...". Other approaches for alias protection include balloon types [2] and islands [17], which prevent external references to objects that are encapsulated by other objects' interfaces.

Servetto et al. proposed *placeholders* as a technique for safely initializing circular immutable

Ownership types, introduced by Clarke [10], use a notion of object context to partition the object store. Many of these, including Clarke's original approach, restrict aliasing. Other approaches, such as universe types [11], restrict mutation but not access to owned objects. In contrast, Rust [30] expresses ownership without explicit object contexts; instead, it works at the level of variable bindings to ensure that no more than one binding exists to a given resource. Rust provides facilities for borrowing ownership at function call boundaries in order to make function calls more convenient. OIGJ [43], another ownership system, extends IGJ with a notion of ownership so that objects cannot be leaked outside their owners.

Gordon et al. [13] focused on providing safe parallelism by combining immutable and isolated types, with support for polymorphism over type qualifiers. They provided `writable`, `readable`, `immutable`, and `isolated` qualifiers. An `isolated` reference ensures uniqueness: "all paths to non-immutable objects reachable from the isolated reference pass through the isolated reference."

Another approach is to use *capabilities*, which pair pointers with policy information that specifies what access rights accompany those pointers [8]. This changes the default on references from "no restrictions" to "all restrictions" but specifies all attributes positively, for example permitting writing. This approach is in contrast to all of the approaches described above, which use language features to express restrictions relative to a default of no restrictions.

3.2 Popular Languages and Libraries

We show through examples below that despite the fact that some popular languages offer read-only restrictions, programmers still desire immutability guarantees and attempt to implement them using the features that they have available. This often leads to misunderstandings, if not by the original programmer, then by other programmers that have to read and maintain the code.

For example, the CERT C Secure Coding Standard [34] says "Immutable objects should be `const`-qualified. Enforcing object immutability using `const` qualification helps ensure the correctness and security of applications." C does not have any features that guarantee immutability in general, though `const` does provide some guarantees for fields. In particular, if one has a pointer allowing `const` access to an object, there may be other non-`const` pointers to that same object. For example, consider this C code, which uses `const` to express constraints in an interface:

```
void threadUnsafePrintIfPositive(const int *x) {
  if (*x > 0) {
    printf("%d", *x);
  }
}
```

Even though `x` is a pointer to a `const int`, this only means that `threadUnsafePrintIfPositive` cannot modify the referenced `int`. This function is not thread safe because it dereferences `x` twice and `*x` may change between the dereferences. If `*x` were immutable rather than just read-only, then the above code would be safe, but such immutability cannot be expressed in C.

As we described in §2.1, the requirements for making a Java class immutable are complex. If a method returns a reference to an internal data structure and cannot ensure that it cannot be modified by a caller, the structure must be copied before being returned to prevent a caller from being able to mutate it. This is often easy for programmers to ignore or forget, and the results in security-sensitive code can be serious. For example, Java's "Magic Coat" security bug was caused by the `getSigners()` method returning a reference to a mutable array holding the signers of a class [21].

Rather than supporting immutability or read-only restrictions in programming languages, some authors have added limited support in libraries. These typically convey these restrictions in names and documentation, resulting in ad hoc (i.e. expressed informally, rather than formally in language constructs) class-based restrictions. The result is that a client for whom immutability of a data structure is required cannot rely on the compiler to give an error if the data structure is later made mutable.

For example, the Foundation framework [28] provided with Objective-C separates immutable and mutable classes in many cases by making the mutable classes subclasses of the immutable classes: the subclasses have additional methods that expose mutation. This ensures that mutating methods cannot be invoked on immutable objects (except that Objective-C is not type-safe). Likewise, the Java JDK provides a collection of immutable classes, such as String and Number, but these are not specified as being immutable in any formal way. In Java, to make a class immutable, Bloch [6] recommends doing all of the following: provide no methods that modify an object's state; prevent subclassing; make all fields final and private; and ensure exclusive access to mutable components. If a programmer wants to know whether a class is immutable, complicated manual (but mechanizable) verification is necessary, but even this does not guarantee that the class will be immutable in the future, since a *new version of the code can easily make the class mutable* in a way that does not cause the compiler to produce any error messages for clients.

The Java JDK provides `Collections.unmodifiableList`, which wraps a given collection object with a wrapper that throws exceptions on modification. However, modifications to the original list are still permitted, so this approach is an example of a read-only restriction, not im-

mutability. Furthermore, the objects in the list may still be modified, so the restriction is non-transitive. In contrast, Google's Guava library copies all the elements when constructing an `ImmutableCollection` from a `Collection`. However, `ImmutableCollection` retains the mutating methods from its superclass, so though the compiler gives a deprecation warning, code that attempts to modify an `ImmutableCollection` compiles and raises exceptions at runtime. Furthermore, `ImmutableCollection` can contain mutable objects, so the immutability is non-transitive.

Microsoft's .NET framework provides a Freezable class [22], which is a superclass for objects that can have a state in which they are immutable. The author of a class that descends from Freezable must add calls to specific APIs before and after modifying state, so enforcement is dynamic and semantics depend on the placement of those calls. JavaScript includes `Object.freeze`, which dynamically enforces shallow immutability [23].

3.3 Empirical Evaluation

Though empirical studies of the effects of mutability on programmer productivity and program comprehension have been conducted, they have been quite limited. For example, Dolado et al. [12] provided initial empirical data regarding the influence of side effects, which mutate data structures, on program comprehension. They compared pre- and post-increment to explicit assignment in C, finding that programmers answered questions about program behavior more accurately with explicit assignment than with pre- and post-increment (e.g. `x=y+1; y=y+1` vs. `x=++y`). However, this comparison was limited to properties of syntax and does not provide recommendations for other design questions, such as on how to enforce encapsulation.

Stylos and Clarke [37] examined the process by which programmers write code that instantiates objects in C++, C#, and VB.NET and found that users prefer and are more effective at instantiating *mutable* classes. When instantiating immutable classes, the programmer must supply all parameters to the constructor at initialization time (the *required construtor* pattern); in contrast, with mutable classes, it is possible to supply none or only some of the parameters at initialization time and defer setting the rest until later (the *create-set-call* pattern). When figuring out how to call the constructor of an immutable class, programmers must interrupt their work to figure out how to instantiate the arguments, which may themselves have arguments, and so on. The create-set-call pattern, in contrast to the required constructor pattern, lets participants defer understanding how to create the required arguments until after they had finished calling the first constructor. The authors concluded that in contrast with the common advice to prefer immutability over mutability, immutability sometimes interferes with usability.

The paucity of empirical work in this area motivated us to conduct our own interviews with programmers. Our aim was to find out how immutability-related language constructs affect programmers (see §5 below). However, significantly more work will be required in this area if we are to base language design decisions on empirical data.

4. USABILITY OF FEATURES

The large collection of different possible language features supporting state change restrictions presents an interesting design problem. Many of the features are compatible with each other and, indeed, more recent systems have included a collection of different features so that programmers can choose which features to use. However, including all of the possible features at the same time makes a system more complex, and more complex systems are harder to use. For example, one of our interview subjects mentioned that in C++, it is common practice to only use `const` for pointers to `const` objects and never for `const` pointers to mutable objects because it's too hard to keep the distinction straight. `const int * x`, `int const * x`, `int * const x`, and `int const * const x` are all legal C declarations; the first two denote pointers to const integers, the third denotes a const pointer to a variable integer, and the fourth denotes a constant pointer to a const integer.

It might seem that a more expressive language — one that allows the programmer to be more precise about what invariants should be checked — would always be better than a less-expressive language. However, this is not necessarily the case. The Cognitive Dimensions of Notations framework [14] provides guidelines for evaluating and comparing usability of different notations. The *error-proneness* dimension refers to the probability of making mistakes, particularly ones where the consequences are hard to find. Language features for restricting state change exist primarily to *prevent* bugs and *clarify* meanings. However, in many cases, using the wrong restriction results in a weaker guarantee than intended but no obvious immediate problems. For example, in a situation that requires object immutability, the programmer might specify a read-only restriction instead by mistake. This may typecheck but not provide the needed guarantee. Likewise, a programmer might annotate an interface as returning a read-only object when in fact the returned object is immutable. This might lead clients of the interface to go to extra trouble and degrade performance, such as by adding locks to ensure thread-safety, when the object was already immutable.

In contrast to the disadvantages of complexity, the *hidden dependencies* dimension of Cognitive Dimensions, which refers to problems that occur when dependencies are not obvious, confirms a positive aspect of state change restrictions: when a reference to mutable state exists, a programmer may write code that mutates that state without being aware of the reference. This situation reflects a hidden dependency, so this dimension suggests not only that immutability is likely to be helpful, but also that transitive immutability is likely to be better than non-transitive immutability.

Nielsen's heuristic evaluation technique [26] offers a collection of heuristics that can be applied when evaluating user interface designs. These heuristics are particularly useful when user studies with real users cannot be conducted, perhaps due to an incomplete implementation or cost constraints. The "be consistent" heuristic, which is also included in the Cognitive Dimensions framework, suggests that features that correspond with different concepts should have different names. When different features are given similar names, users may be confused; this problem can be seen in C's `const` syntax, where the position of `const` is significant but it is not obvious which position has which meaning. The differences between proposed features can be subtle on the surface even though the features represent significantly different meanings. For example, the system by Haack et al. includes qualifiers `RdWr`, `Rd`, and `Any` [15]. One might guess that `Any` is equivalent to `RdWr` because read and write would seem to be the only available kinds of access, but in fact

`RdWr` corresponds to a mutable object, `Rd` corresponds to an immutable object, and `Any` corresponds to a read-only restricted reference. `Any` means that the actual immutability invariant is either `RdWr` or `Rd` and therefore a holder of a reference to an `Any`-object cannot write to it because it might be immutable (`Rd`). It is easy to imagine users confusing these different terms, based on the heuristic evaluation evidence; we lack stronger evidence since Haack and Poll did not publish a usability study of their system.

This problem of features with potentially confusing names is not limited to just one system. IGJ [42] supports an object annotation `@Immutable`. However, in order to make immutability transitive, one must use the immutability parameter on fields in the transitive closure of objects that are referenced by the `@Immutable` object. This means that when a programmer has a reference to an `@Immutable` object, it is necessary to locate and read an arbitrary amount of class implementation code to determine what immutability means for this particular object. Furthermore, a future code edit may invalidate the programmer's reasoning, resulting in the programmer making an immutability assumption that is later violated by a programmer who was unaware of the previous reasoning. This is a *hidden dependency* and may cause bugs. Systems that support removing fields from the abstract state, such as C++'s `mutable`, have a similar problem. However, the real-world implications of these design choices have not yet been tested.

JAC [19] provides a larger collection of features: `writable`, `readonly`, `readimmutable`, and `readnothing`. `Readimmutable` methods can only read the parts of an object's state that are immutable after initialization. The authors say that `readimmutable` may be useful for objects used as keys in hash tables, since the parts of those objects that contribute to the hash code must not change while the objects are used as keys. However, it is unclear whether this complexity is warranted; perhaps it would be better to simply require that the entire object be immutable. We view this question as a tradeoff between flexibility and simplicity rather than assuming that more flexibility is always better.

5. INTERVIEWS WITH DEVELOPERS

5.1 Methodology

Given the large collection of immutability- and read-only restriction-related design choices that must be made in order to construct a concrete programming language or language extension, we wanted to find out how practicing software engineers think about state when writing software. We obtained IRB approval and conducted semi-structured interviews with a convenience sample (N=8) of software engineers at several US- and Europe-based organizations. We focused on software engineers who work on large software projects, with the assumption that these projects would be the most likely (relative to smaller projects) to encounter interesting problems with state management. Likewise, due to the participants' expertise, any problems raised are likely to come up in a wide variety of situations and be relevant problems for consideration in a language design. Our participants had spent a long time as software engineers, with a mean professional experience of fifteen years and a minimum of seven years. They typically had worked on projects with millions of lines of code and hundreds of people. Participants reported significant usage of immutable and access-restricted interfaces and they had a multitude of strong opinions about state management in large software systems. Their experience was primarily in C++, Java, and Objective-C. Additional information about methodology can be found in an auxiliary document in the ACM Digital Library.

5.2 Results

Relevance. Asked about bugs caused by state changing when it should not have, one participant exclaimed,

> "Oh God, like, most of them!... my favorite is where you have data that is supposed to be immutable and is only settable once in theory but that's not well enforced and so it ends up getting re-set later either because it gets re-initialized or because someone is doing something clever and re-using objects or you have aliasing where two objects reference the same other object by pointer and you make changes...".

Another participant cited library boundaries as a problem: this engineer's module depended on data that got changed by a third-party library, resulting in half-updated or not-updated state. This resulted in mistrust among the groups because it was frequently unclear which team was responsible for any given bug. All the participants who worked on software with significant amounts of state said that incorrect state change was a major source of bugs.

General techniques. All of the participants reported using various techniques to ensure that state remained valid. Techniques included ensuring that lifecycle and ownership are well-defined using conventions such as C++'s RAII; restricting access with private variables and interfaces; using consistency checkers and writing repair code; unit and manual testing; and assertions. According to one participant, good design is better than using `const`: "if you simply do not depend on an object, there is no way you could possibly modify it directly." Another participant uses immutability as a key part of the architecture: "By design, we've made the key data structures immutable. We never update or delete any data objects. This ensures that an error or problem can never put the system into an undesirable state."

C++ `const`. Several participants used `const` in C++ to make sure state does not change, but it does not meet their needs. First, there is no way to make special cases for fields on a per-method basis: either fields are part of an object's abstract state or not. One participant reported removing `const` from a collection of methods rather than making a particular field mutable. Second, the viral nature of `const` makes using it a significant burden, since declaring a method `const` requires transitively annotating all methods that the first method calls. One participant wished for tools that automatically added `const` as needed in such cases; for Java, Vakilian et al. proposed a tool that would help programmers add similar annotations [40].

One participant complained that `const` applies to methods and fields but not to whole classes. In contrast, another participant said that marking methods `const` is very helpful, as is having two interfaces to every object: a mutable interface and a const interface. The former approach corresponds to class immutability if all methods are marked `const`; the latter approach implements read-only restrictions if all methods in one interface are const.

Participants said that there were many situations in which

they wanted to use `const` but it did not express the invariant they needed. One participant wanted to restrict access in a more fine-grained way than just mutability: for example, by disallowing access to particular methods even though those methods did not change state, or permitting some kinds of state changes but not others. Another participant cited the discrepancy between abstract and concrete state: though one can mark particular fields as `mutable` in C++, it would be preferable to be able to show that the value of a field has no effect on views of the object.

Thread safety. We were particularly interested in how participants used immutability to manage state that was shared across threads, since free sharing across threads is a frequently-cited benefit of immutable data structures. We asked participants what techniques they use to ensure state is safe when accessed concurrently. Two participants described architectural techniques that hide concurrency from users, avoiding this problem. Immutability did not seem to be a commonly used technique; participants cited traditional methods, such as serial queues and mutexes. One participant, a framework designer, mentioned a focus on minimizing dependencies on mutable state across threads. A major theme that arose was about reuse: when reusing existing structures, participants had to assume that structures were mutable because they had no guarantees otherwise. One participant mentioned taking advantage of immutability if it was already present. Another participant pointed out that it is rare to be able to design a component from the ground up, so there is usually some synchronization needed. This suggests that when designing components for reuse, it is helpful to be able to specify to consumers of the components which aspects are immutable. We conclude that existing techniques for specifying immutability in the languages these participants used are insufficient for facilitating reuse in concurrent systems.

Immutable classes. We asked participants about their use of immutable classes and found that immutable classes are used very frequently but that languages the participants use do not provide any explicit support for them. In fact, one participant pointed out that some languages make immutable classes difficult to write and use. In C++, copying objects is difficult and error-prone, so one company's style guide recommends disabling copy and assignment operations. As a result of the difficulty of copying objects, objects tend to be mutable.

Participants mentioned using immutable classes for copy-on-write situations; for objects that manage relationships between other objects; and for wrapping an existing mutable class in order to enforce a particular invariant. One participant mentioned avoiding changing classes from immutable to mutable because then anything holding a reference to the object would suddenly depend on whatever could now cause mutations. This again is a notion of class immutability, not object immutability. That is, instances of a class serve a particular role in an architecture, and the class defines the role; making all instances mutable would violate invariants of many of the clients of that class.

Security. Since many recommendations in favor of immutability come from the security community [35] [31], we asked participants about their experiences with security considerations in state management. One participant talked about a security-related object that the team wanted to make immutable but which needed to be modified as part of the teardown process, so it had to be made mutable instead. None of our participants had a focus on security in their jobs, though most of them worked on software that could potentially have security problems. Their perspective was generally that security was important in security-related components, such as authentication, but otherwise not a primary concern. One participant, who worked in part on cloud-based software, mentioned that privacy is a more difficult issue than security because the requirements are less clear and because privacy issues come up in more contexts.

5.3 Implications on language design

From the above interview findings, we extracted the following observations that are relevant for the design of programming languages and tools. Since the interviews were of programmers working on large, object-oriented systems, our interview findings are most applicable to object-oriented languages that are intended to be used for large systems.

- Both read-only restrictions and immutability are used in practice for different purposes, but the languages that our participants used do not reflect their needs.
- State management is a core issue that developers consider when designing architectures and APIs, and developers do use read-only references and immutability (when available) to enforce encapsulation.
- Existing mechanisms do not solve the problems that developers have when building concurrent systems, in part because they frequently re-use existing code that does not provide immutability guarantees. Lack of transitivity of existing mechanisms results in guarantees that are too weak to be useful.
- Incorrect state change is a frequent cause of bugs. As a result, programming language features that help developers manage state have a good chance of preventing many bugs and so research about such features is a worthwhile endeavor.
- Even limited read-only restriction mechanisms, such as `const`, can be too hard for programmers to use effectively. Designs must emphasize simplicity and usability, or the features will not get used.
- Facilitating immutability is a core issue in programming language design; languages that make copying objects onerous or expensive discourage programmers from using immutability-related features.

5.4 Limitations

Our small sample size resulted in several limitations to our findings. We do not know whether programming languages used for small projects or for short-lived projects would have the same design considerations, since many of the justifications for the recommendations came from experiences with large, long-lived projects. It is unclear to what extent the participants' backgrounds biased our findings, since nearly all of them had formal computer science training, whereas across the industry, software engineers come from a variety of different backgrounds. Furthermore, our participants were very experienced; novices may benefit from different language design choices than experts do.

6. ITERATIVE DESIGN OF FEATURES

Following techniques from the human-computer interaction community [25], we are conducting a user-centered, iterative design process to create language features with which

users can express state change constraints in a way that is both usable and useful. Rather than assuming that users need all possible features, we are starting from the assumption that the language should be as simple as possible while still providing what users need to address their problems. There is a tradeoff between simplicity and completeness: adding keywords for all possible features that users might need might make addressing more problems *feasible* but also make the system so complex that users cannot successfully use those features correctly. By designing our system according to the needs that practitioners have expressed and iteratively testing specific designs with users, we hope to arrive at a practical solution that addresses many, but likely not all, of the problems that practitioners face in this space.

We must distinguish between requests from practitioners and their actual needs. Users frequently do not know what features would benefit them most [5], so asking them what features they want is only the beginning of the process. We grounded our interviews by asking primarily about their experiences and only secondarily about their feature requests; by basing our designs on the requirements of the systems the users were building, we increase the chance that our system is effective on real-life problems.

6.1 Transitivity

Based on our finding that some programmers want guarantees that can only be provided by transitive immutability and the disparity in transitivity support between research languages and commonly-used languages, we focused on the question of transitivity. We piloted a user study comparing a non-transitive subset of IGJ [42] with a modified version that always enforces transitivity, which we call IGJ-T. We started with IGJ due to its availability and practicality: it can be used by experienced Java programmers and requires learning only how IGJ annotations are processed, not a whole new programming language. Initially IGJ was an extension to Java using generics [42], but we used a revised version based on the Java annotation system [32].

Because we wanted to compare always-non-transitive immutability to always-transitive immutability in our study, we only told our participants about the `@Immutable` annotation, not any of the others. This forced a non-transitive approach to immutability for the participants in the control condition. For the transitive case, which we wanted to be identical to IGJ except for transitivity, we created a version of IGJ, *IGJ-T*. In IGJ-T, if a class has a constructor that returns an `@Immutable` result, then all fields of that class must be transitively `@Immutable`. The implementation of IGJ-T works by visiting all methods when typechecking; for methods that are constructors that are annotated `@Immutable`, it recursively visits the types of the class's fields and verifies that all of them are marked `@Immutable`.

The design of IGJ greatly facilitated verification of transitive immutability relative to writing an analyzer from scratch. IGJ is implemented as part of the Checker framework [33], which provides a collection of related facilities for checking properties with annotations in Java. In particular, the Checker framework made it very easy to extend IGJ with a new verification on constructors. IGJ-T gives error messages like the following example:

```
error: [transitivity.invalid] Cannot declare
Simple() with annotation @Immutable because Simple
transitively contains @Mutable Date d on path
```

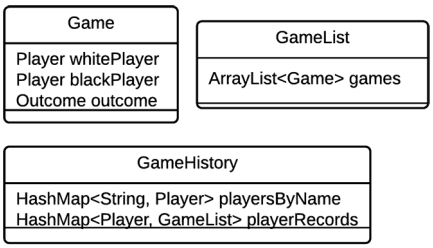

Figure 1: Class diagram of starter code

```
@Immutable AContainer a -> @Mutable Date d
    @Immutable Simple () {
               ^
```

IGJ-T gives the full field path to the erroneous field so that programmers can easily find the cause of the problem.

6.2 Study design and pilot results

We designed four tasks, intended to take 90 minutes total, in the context of a program to track outcomes of chess games (Figure 1 summarizes the architecture). After obtaining IRB approval, we recruited a convenience sample (N=3) of PhD students experienced in Java to pilot our study. Two participants completed a pre-test regarding their programming experience and understanding of `final`. Of these, one participant with over a year of Java experience, on hearing an explanation of the fact that `final` only applies to assignment but not to the referenced objects, exclaimed, "no one ever highlighted that key thing [before]!" The tasks are summarized below:

1. Suppose your language has a feature that lets you specify that once a Game has happened, it should not be changed. Please make any changes necessary to the existing program to express this using whatever language feature you think would be best.
2. Now, we will show you the feature we have designed, which we call `@Immutable`. (Participants were then given a description of IGJ or IGJ-T depending on the experimental condition to which they were assigned). For the following sample code, please write which lines would give compile errors to test your understanding of `@Immutable`. (The sample code included a Person class with fields for eye color and name; the eye color was supposed to be fixed and the name could be changed. The instructions showed how `@Immutable` could be used to make the eye color immutable and compared `@Immutable` with `final`, which would not suffice when used only on the eye color field because the eye color was a reference to a Color object.)
3. Please implement the `updatePlayerName()` method to change the name of a player.
4. Please implement the `changeGameOutcome()` method to update the history for a corrected game outcome. Game must remain `@Immutable`.

The first task was designed to elicit how the participants would want to express the immutability concepts, if given free reign. This is a form of participatory design that we call the *natural programming* elicitation approach [24], since it tries to reveal how users would *naturally* express these concepts. Note that the prompt (and the instructions beforehand) did not use the word *immutable* in order to try to

avoid biasing the participants' vocabulary choice.

The second task introduced the design of the immutability construct that participants would be using and verified that they understood it in a small test. All of the participants understood their version of `@Immutable`.

In the third task, we expected participants in the IGJ condition to erroneously mutate an object that was used as a key in a hash table, resulting in a bug; we expected participants in the IGJ-T condition to spend longer on the task but avoid the bug. In the fourth task, we expected participants in the IGJ-T condition to spend extra time solving the problem but reap little benefit.

Though we have only begun our user study — so far we have piloted it with three participants and the first was only given the first two tasks — we have learned helpful insights about Java programmers' expectations. In the first task, one participant used `immutable` as a qualifier on the Game class declaration; the other added a `freeze` qualifier on a Game parameter to one of the constructors of GameList (neither participant used the Java annotation syntax).

The third task required participants in the IGJ-T case to essentially rewrite the entire game history; in the IGJ case, a simpler rewriting was possible. In both cases, however, the participants were surprised that they had to rewrite data structures rather than modifying them in place: "It seems complicated because even if you want to change the name you have to reconstruct everything." Some Java programmers appear to find immutable data structures confusing, and encountering them forces a difficult problem-solving process. By completing the study, we hope to better understand the tradeoffs of immutability: in what situations is it beneficial to make structures immutable, and in what situations do the costs of immutability outweigh the benefits?

One participant in the non-transitive case had recently been doing more functional programming, so we expected that this experience might make immutability more intuitive. However, this participant modified Person in place, not realizing that this would break various aspects of the data structures, in part because Person was being used as a hash table key. When this participant discovered the problem while testing and debugging, the participant exclaimed, "this is what happens when you mutate [stuff] in place!" and commented that maybe upon switching back to an imperative programming context, it was hard to remember all the problems that are inherent in imperative programming.

Due to the time spent on the third task, one participant did not start the fourth task; the other two found it similar to the third task. We plan to adjust the tasks to reduce the time required for the third task and make the fourth task more meaningful.

Though we restricted our study to a small subset of IGJ, some participants had difficulty understanding its error messages while trying to fix their bugs. For example, one participant spent over two minutes debugging this error message:

```
error: [method.invocation.invalid] call to
setBlackPlayer((@org.checkerframework.checker.igj.
qual.Immutable :: t_starter.Player)) not allowed
on the given receiver.
    game.setBlackPlayer(newPlayer);
                        ^
found   : @Immutable Game
required: @Mutable Game
```

The participant kept checking the argument to `setBlackPlayer`, but the problem was that `game` was `@Immutable` and the solution was to create a new `Game` object instead of modifying the existing one. Note that this error was an existing part of IGJ. In our limited pilot, only one user used IGJ-T, and that user did not encounter any transitivity error messages.

The pilot user studies we have completed so far are only the beginning; we are still refining the study. However, threats to validity include the small set of tasks we used in comparison to the wide variety of tasks that real software engineers perform; the small codebase and short timeframe of the study; and the relative inexperience of our participants, who are mostly graduate students with small amounts of professional experience. Eventually, however, we hope to compare effectiveness of the two language extensions across the different tasks and find out whether transitive immutability prevents bugs and whether it imposes a significant time cost on programmers due to its complexity.

7. FUTURE WORK

The studies we are investigating are focused on comparing transitive to non-transitive immutability in Java, but as can be seen from §2, there is large design space of features for immutability. Because of the potential large impact of immutability on programmer productivity and software quality, it will be important in the future to develop an empirical basis for designing and using immutability features in programming languages. We are also planning to refine the design of IGJ-T to more easily facilitate class immutability. Finally, to mitigate the threats to validity and improve external validity, we hope to follow our lab study with a more longitudinal study to evaluate to what extent our findings generalize to real-world software projects.

8. CONCLUSIONS

Despite the vast design space for language features supporting immutability in programming languages and plentiful advice regarding how programmers should use them, there is only scarce empirical evidence supporting these recommendations. We presented a classification of immutability features and an analysis of their tradeoffs, challenging the notion that providing a very flexible feature set is best. We have begun experiments with users to find out how users interact with immutability features along the transitivity design dimension, but future work will be necessary to understand what features and usage patterns best benefit users.

9. ACKNOWLEDGMENTS

We appreciate assistance from Michael Ernst and Werner Dietl in understanding the details of IGJ. We are grateful for the help of our experiment participants; our anonymous reviewers; and Matt Fredrikson, Jan Hoffman, James Noble, Cyrus Omar, and Alex Potanin for feedback on this paper. This material is supported in part by NSA lablet contract #H98230-14-C-0140, by NSF grant CNS-1423054, and by Contract No. FA8721-05-C-0003 with CMU for the operation of the SEI, a federally funded research and development center sponsored by the US DoD. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of any of the sponsors.

10. REFERENCES

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1984.

[2] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *European Conference on Object-Oriented Programming*. 1997.

[3] American National Standards Institute. *Military Standard Ada Programming Language*, 1983. Also MIL-STD-1815A.

[4] American National Standards Institute. *The Pascal Programming Language. ANSI/IEEE 770X3.97-1983*, 1983.

[5] H. Beyer and K. Holtzblatt. *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann Publishers, Inc., 1997.

[6] J. Bloch. *Effective Java, Second Edition*. Addison-Wesley, 2008.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. *Object-oriented programming, systems, languages, and applications*, 2002.

[8] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and Read-Only. In *European Conference on Object-Oriented Programming*, 2001.

[9] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, Aug. 1992.

[10] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *Object-oriented programming, systems, languages, and applications*, 1998.

[11] W. Dietl, S. Drossopoulou, and P. Müller. Generic universe types. In *European Conference on Object-Oriented Programming*. Springer, 2007.

[12] J. Dolado, M. Harman, M. Otero, and L. Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Transactions on Software Engineering*, 29(7):665–670, July 2003.

[13] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. *Object-oriented programming, systems, languages, and applications*, 2012.

[14] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[15] C. Haack and E. Poll. Type-based Object Immutability with Flexible Initialization. In *European Conference on Object-Oriented Programming*, July 2009.

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

[17] J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. *Object-oriented programming systems, languages, and applications*, 1991.

[18] R. C. Holt and J. R. Cordy. The turing programming language. *Communications of the ACM*, 31(12):1410–1423, Dec. 1988.

[19] G. Kniesel and D. Theisen. JAC—Access right based encapsulation for Java. *Journal of Software Practice & Experience - Special issue on aliasing in object-oriented systems*, 31(6):555–576, 2001.

[20] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, Dec. 1992.

[21] G. McGraw and E. Felton. *Securing Java*. Wiley, 1999.

[22] Microsoft, Inc. Freezable objects overview. https://msdn.microsoft.com/en-us/library/vstudio/ms750509(v=vs.100).aspx. Accessed Feb. 8, 2016.

[23] Mozilla Developer Network. Object.freeze(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze. Accessed Feb. 9, 2016.

[24] B. A. Myers, J. F. Pane, and A. Ko. Natural programming languages and environments. *Communications of the ACM*, 47:47–52, 2004.

[25] J. Nielsen. *Usability engineering*. Academic Press, Boston, 1993.

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

[27] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *European Conference on Object-Oriented Programming*, 1998.

[28] Apple, Inc. The Foundation Framework. https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/ObjC_classic/. Accessed Feb. 8, 2016.

[29] Microsoft Corp. Framework design guidelines. https://msdn.microsoft.com/en-us/library/ms229031(v=vs.110).aspx. Accessed Feb. 8, 2016.

[30] Mozilla Research. The Rust programming language. https://www.rust-lang.org. Accessed Feb. 8, 2016.

[31] Oracle Corp. Secure coding guidelines for the Java SE, version 4.0. http://www.oracle.com/technetwork/java/seccodeguide-139067.html#6. Accessed Feb. 8, 2016.

[32] University of Washington. IGJ immutability checker. http://types.cs.washington.edu/checker-framework/current/checker-framework-manual.html#igj-checker. Accessed Feb. 8, 2016.

[33] University of Washington. The Checker Framework. http://types.cs.washington.edu/checker-framework/. Accessed Feb. 8, 2016.

[34] R. C. Seacord. *The Cert C Secure Coding Standard*. Pearson Education, Inc., 2009.

[35] R. C. Seacord. *Secure Coding in C and C++*. Addison-Wesley, 2013.

[36] M. Skoglund and T. Wrigstad. A mode system for read-only references in Java. In *3rd Workshop on Formal Techniques for Java Programs*, 2001.

[37] J. Stylos and S. Clarke. Usability Implications of Requiring Parameters in Objects' Constructors. In *International Conference on Software Engineering*, 2007.

[38] M. S. Tschantz and M. D. Ernst. Javari: Adding

Reference Immutability to Java. In *Object-oriented programming, systems, languages, and applications*, 2005.

[39] C. Unkel and M. S. Lam. Automatic inference of stationary fields: A generalization of java's final fields. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 183–195, New York, NY, USA, 2008. ACM.

[40] M. Vakilian, A. Phaosawasdi, M. D. Ernst, and R. E. Johnson. Cascade: A universal programmer-assisted type qualifier inference tool. In *International Conference on Software Engineering*, 2015.

[41] B. Venners and B. Eckel. A conversation with anders hejlsberg, part viii. Feb. 2004.

[42] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kielun, and M. D. Ernst. Object and reference immutability using Java generics. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 75–84. ACM, 2007.

[43] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and immutability in generic Java. *ACM SIGPLAN Notices*, 45(10):598, oct 2010.