Glacier: Transitive Class Immutability for Java

Michael Coblenz*, Whitney Nelson†, Jonathan Aldrich*, Brad Myers*, Joshua Sunshine*
*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA USA
Email: {mcoblenz, aldrich, bam, sunshine}@cs.cmu.edu
†Hampton University, Hampton, VA USA; Email: whitney.nelson@my.hamptonu.edu

***Abstract*—Though immutability has been long-proposed as a way to prevent bugs in software, little is known about how to make immutability support in programming languages effective for software engineers. We designed a new formalism that extends Java to support transitive class immutability, the form of immutability for which there is the strongest empirical support, and implemented that formalism in a tool called Glacier. We applied Glacier successfully to two real-world systems. We also compared Glacier to Java's `final` in a user study of twenty participants. We found that even after being given instructions on how to express immutability with `final`, participants who used `final` were unable to express immutability correctly, whereas almost all participants who used Glacier succeeded. We also asked participants to make specific changes to immutable classes and found that participants who used `final` all incorrectly mutated immutable state, whereas almost all of the participants who used Glacier succeeded. Glacier represents a promising approach to enforcing immutability in Java and provides a model for enforcement in other languages.**

***Keywords*-immutability, programming language usability, empirical studies of programmers**

I. Introduction

Mutability in software has been frequently cited as a source of bugs and security vulnerabilities [1], [2], [3], [4]. If a component depends on mutable data, the architecture typically must provide a facility for notifying the component when the data has been modified in order to maintain consistency. If mutable data is read and modified concurrently, there is a risk of a race condition unless synchronization is used correctly. These opportunities for bugs have led some experts, such as Bloch [5], to advise designing software so that as many structures as feasible are *immutable*: not modifiable through any reference. Other experts, such as Helland, have touted the benefits of immutability for distributed and database systems [6]. Likewise, programming languages have included features that facilitate formal specification of immutability. This offers two advantages over informal specification: enforcement, so that the compiler or runtime can inform the programmer when immutability is violated; and accurate documentation, so that a client of a component can know what immutability guarantees the component provides. Unfortunately, existing systems are either too hard to use or ineffective at preventing bugs [2].

The space of immutability is complex. Our prior work identified eight dimensions along which a language can support immutability [2]. If a programming language is to support the specification and enforcement of immutability, what kinds of immutability should the language support? Supporting as many different kinds of immutability as possible results in a complex system; to date, there are no usability studies published of immutability specification systems. We previously found that attempting to support many different kinds of immutability at once can result in a system that is very difficult to use effectively and correctly. Alternatively, a design that supports a small set of immutability-related features might be easy to understand and apply but fail to capture useful constraints. Such a system might fail to achieve the goals of immutability systems: preventing bugs and documenting and enforcing specifications. This motivates our research question: can we select a subset of immutability features and design a corresponding programming language such that:

1) Real users can use the immutability restrictions effectively with minimal training; and
2) Expressing immutability with the language actually prevents bugs in situations where software engineers have already decided on an immutable design?

To address these questions, we designed and implemented *Glacier*, a type annotation system for Java. *Type annotations* are an existing mechanism in Java that support extending the type system. Based on prior work that found that programmers would benefit from strong guarantees [2], we focused on *transitive class immutability*. *Transitivity* ensures that immutable objects can never refer to mutable objects; *class* immutability means that immutability of an object is specified in its class's declaration. Glacier, which stands for *Great Languages Allow Class Immutability Enforced Readily*, enforces immutability statically, with no effect on the runtime and therefore no performance cost on the compiled software, so that users can get strong guarantees at compile time.

We evaluated the practicality, applicability, usability, and usefulness of Glacier in two case studies on existing code and in a user study with 20 participants. In the case studies, we successfully applied Glacier to a spreadsheet model component and to a reusable immutable container class, observing that Glacier is applicable to these real-world, existing software systems. In applying Glacier, we also found two previously-unknown bugs in the spreadsheet implementation. In the user study, we compared Glacier with `final`, since `final` is the current state-of-the-practice mechanism for specifying immutability in Java. When given programming tasks, users in the condition where they only had `final` all made various errors that resulted in breaches of immutability, even after receiving explicit training in how to use `final` correctly; in contrast, although the participants who used Glacier had never seen it before, almost all of them succeeded in using it to specify immutability correctly. We also asked participants to complete programming tasks with immutable classes, and found that although most users were able to complete the tasks, all users of `final` wrote code that had bugs or security vulnerabilities due to improper mutation; Glacier prevented these problems at compile time.

This paper makes the following contributions:

1) A definition and formal model of transitive class immutability as an extension to Featherweight Java [7];
2) An implementation of that model in a tool called *Glacier*, which enforces transitive class immutability in Java. By enforcing only the kind of immutability for which there is the strongest empirical support, we have achieved significant simplifications relative to existing systems;
3) Evaluations of Glacier in two case studies on real software projects that showed that Glacier successfully captures a kind of immutability appropriate for those projects;
4) The first formal user study of any immutability system. We compared Glacier to `final` and found that all ten participants who used `final` wrote code that had bugs or security vulnerabilities, even after having been trained on correct `final` usage, in a situation in which Glacier statically detects those problems. Almost all the Glacier users were able to complete the tasks successfully.

II. Background

Although it might seem that immutability is a simple concept, designing an enforcement system requires making a collection of design choices regarding what immutability means and how it will be enforced. Prior work identified eight distinct dimensions of immutability [2], resulting in at least 512 different combinations of features. As such, any proposal should include a justification for its position in the design space. Some key dimensions of immutability include:

1) Restriction type: *assignability* restricts assignment to variables; *read-only restrictions* prevent writes through particular references to an object; *immutability* prevents writes through all references to an object.
2) Scope: *object-based* restrictions pertain to particular objects, while *class-based* restrictions pertain to all instances of a particular class.
3) Transitivity: *transitive* restrictions apply to all objects reachable from a given object via its fields; *non-transitive* restrictions apply only to the object's fields.
4) Polymorphism: *restriction polymorphism* allows one function to accept inputs with several different kinds of restrictions. In *parametric restriction polymorphism*, a parameter can represent a restriction instead of a literal restriction; then the actual restriction is according to the value of the parameter.
5) Enforcement: *static* enforcement occurs at compile-time; *dynamic* enforcement occurs at runtime.

Design recommendations. Our prior work included interviews with professional software engineers and concluded that the *transitive immutability* subspace seemed to reflect the needs of our interviewees. Immutability can provide particularly useful guarantees: immutability provides guarantees regarding state change, rather than guarantees regarding access (as in the case of read-only restrictions). Relative to non-transitive immutability, *transitive* immutability is more useful: the entire state of an immutable object is immutable, rather than just a part that depends on the object's implementation. For example, if a *transitively* immutable `Person` object has a reference to an `Address` object, `Address` must be immutable as well. As a result, objects that are transitively immutable can be shared safely among threads without synchronization, and invariants that are established regarding objects' state are always maintained. Our interviews also found evidence in support of *class* immutability, with some engineers observing that most classes serve a particular architectural role, and that role typically either requires mutability or not.

III. The Design of Glacier

A. Evidence-based design

We designed Glacier using an evidence-based approach. Based on our prior findings showing that transitive immutability provides particularly useful guarantees, we concluded that Glacier would support transitive immutability. In order to facilitate practical usage of Glacier, since Witschey et al. found that simplicity and ease of use are predictive of adoption [8], we designed Glacier to be as simple as possible while still enforcing immutability. Glacier is a static typechecker, so it provides strong, compile-time guarantees and imparts no runtime cost on programs. When invoking the existing Java compiler on the command line, users can pass a command-line argument that causes the compiler to invoke the Glacier annotation processor; users of a build system can arrange to always pass this argument by default. This approach has practical advantages, since teams can adopt Glacier without changing compilers and individual programmers can choose when to invoke the checker, for example skipping checking to temporarily use unsafe debugging code. However, it is possible to circumvent these checks by not running the annotation processor. For example, from a class that is compiled without Glacier, one could modify a public field in an `@Immutable` class defined in an external .jar file.

B. Syntax and context

We were interested in evaluating our tool in the context of an existing corpus of code and with programmers who might be able to use it. As such, we implemented Glacier in the context of Java, which has a large and active user base. Java is also representative of a broad class of object-oriented languages. Implementing Glacier as a type annotation processor has several benefits over a from-scratch approach: by using Java type annotations, Glacier uses only existing Java syntax and can be parsed by the standard Java parser. Glacier is implemented within the Checker Framework [9], which facilitated Glacier's development. In Glacier, types can be annotated with `@Immutable` to indicate that they are immutable. Types that are not annotated are not guaranteed to be immutable. Glacier represents this with an implicit annotation of `@MaybeMutable`.

C. Class immutability

If an object-oriented language is to provide transitive immutability, should it provide class immutability, object immutability, or both? Some systems, such as IGJ [10], provide both. However, we seek to design a system that is as simple as possible and yet still reflects users' needs. We found, perhaps surprisingly, that supporting only class immutability and not object immutability resulted in significant simplifications to the system. For example, suppose `@Immutable`, could be applied to objects and classes. Consider an identity method:

```
interface DateUtilities {
    public Date identity(Date d);
}
```

The declaration of `identity` does not specify whether its argument is `@Immutable`. A caller of `identity` may require that the annotation on the returned object is the same as the annotation on the passed object, but the interface does not provide that guarantee. Polymorphism addresses this problem:

```
interface PolymorphicDateUtilities {
    public  @I Date identity(@I Date d);
}
```

This notation means that the `Date` input to `identity` has some annotation, `@I`, and the returned object has the same annotation. Though polymorphism increases flexibility, adding this feature increases the complexity of the language.

Another problem with object immutability pertains to the subtyping relationship between mutable and immutable instances of a particular class. Consider a method that took an `@Immutable` object as a parameter. Passing a `@MaybeMutable` object would be unsafe because the method might assume that no state in the object will change in the future, for example sharing it among threads. Likewise, a method that expects a mutable object cannot take an immutable object as an argument because the immutable object lacks mutating methods. Therefore, in Glacier there is no direct subtyping relationship between the mutable and immutable types. By supporting only class immutability, the user can decide whether each class should be mutable or immutable, and then there is no problem of subtyping because there's only one class to discuss. An alternative design would involve a common supertype of both the immutable and mutable subclasses. This requires introducing a third type, again resulting in more complexity. Either the user has to manually implement all three classes, or there must be a system by which the user may specify how to generate them.

Supporting only class immutability also simplifies error messages: when a user sees an error message pertaining to a type that includes a particular annotation, the user can always know that the annotation came from the class's declaration or a conflicting local annotation, rather than via type inference (which would otherwise be important to avoid the proliferation of annotations on all types). When a user sees a type name, the annotation is implicit; if there is a declaration of `@MaybeMutable class Date`, then there is no need to annotate any other usage of the `Date` type because every `Date` is `@MaybeMutable`.

For convenience, any mention of `@Immutable` objects refers to objects that are instances of `@Immutable` classes.

D. Restrictions of immutability

Glacier enforces two restrictions on the fields of `@Immutable` classes: all fields must be `@Immutable`, and fields cannot be assigned outside the class's constructors. Note that the former requirement implements *transitive* immutability: an `@Immutable` class's fields must all be `@Immutable`, so the referenced objects cannot have their fields reassigned or refer to mutable objects, etc. `final` is permitted on fields but is redundant with `@Immutable` on the containing class.

When a reference to an object is of `@Immutable` type, Glacier guarantees that the referenced object is immutable. However, if a reference type is not `@Immutable`, Glacier provides no immutability guarantees. In particular, the referenced object may dynamically be `@Immutable`. As a result, subclasses of `@Immutable` classes must be `@Immutable`, but subclasses of `@MaybeMutable` classes can be either `@MaybeMutable` or `@Immutable`. Importantly, a subclass of a `@MaybeMutable` class can only be `@Immutable` if no superclass has a non-final field or field of `@MaybeMutable` type. Likewise, if an interface is declared `@Immutable`, then all implementing classes must be `@Immutable`, but a `@MaybeMutable` interface can be implemented by an `@Immutable` class. All subinterfaces of `@Immutable` interfaces must also be `@Immutable`. This ensures that all subtypes of an `@Immutable` type are `@Immutable`.

It is not obvious that it should be permitted to declare an immutable subclass of a mutable class. It might seem that if the superclass has a guarantee of mutability, the subclass should adhere to that guarantee. However, that is precisely why the alternative to `@Immutable` is `@MaybeMutable`: a `@MaybeMutable` class is *not guaranteed to be mutable*. A significant disadvantage of this design decision is that adding a non-final or `@MaybeMutable` field to a `@MaybeMutable` class is a breaking change for `@Immutable` subclasses; this disadvantage is compounded by the fact that subclasses may not even be in the same package as the superclass and the implementor of the superclass may not be aware of the existence of subclasses. However, the problem of changes in superclasses unexpectedly breaking subclasses is long-standing in object-oriented systems and is well-known as the *fragile base class problem* [11]. Enabling immutable classes to subclass certain mutable classes enables existing, commonly-used design patterns to be compatible with Glacier. For example, Google's Guava libraries [12] provide an ImmutableList class that (indirectly) extends java.util.AbstractCollection, which cannot be @Immutable because it has mutable subclasses. However, ImmutableList itself does not support mutation and can be annotated @Immutable. Because practicality is a design objective of Glacier, we considered allowing @Immutable subclasses of @MaybeMutable classes a good tradeoff to make. It might be possible to address the fragile base class problem by supporting an additional annotation for classes that must not refer to any mutable state but which may have @MaybeMutable subclasses.

Java primitives, such as int and boolean, are @Immutable. Conceptually, assignment of a new value to a primitive-type variable reflects binding the variable to a different primitive, not a mutation of an existing value. Glacier includes a list of JDK classes, such as String and Integer, that are @Immutable, but that list does not currently include all immutable classes in the JDK.

It is an error in Glacier to give an annotation to a type use that is different from annotation given at the type's declaration. If no annotation is provided in the type's declaration, then the annotation @MaybeMutable is implicit. As a special exception, however, both @Immutable Object and @MaybeMutable Object are permitted, so that all @Immutable types have a common supertype that specifies immutability. For example, one can specify a container that can hold any immutable object. @Immutable Object is a subtype of @MaybeMutable Object: a @MaybeMutable Object can refer to any object at all.

The default behavior in the Checker framework is that receiver annotations are contravariant with respect to overrides. Glacier overrides this behavior to permit covariant annotation overriding in the receiver permission. This is important to allow methods of Object to be overridden in @Immutable subclasses, and it is safe because dispatch to the method of an @Immutable subclass implies that the @Immutable annotation on the receiver is correct.

E. Additional annotations for arrays

Arrays are an older Java feature and their design has various inconsistencies with other aspects of Java, so they pose some special problems. For example, occasionally it is desirable to write a method that can take both mutable and immutable arrays; this is safe if the method can be statically guaranteed to never reassign any of the array elements. Note that this case does not arise with other kinds of objects because with other objects, the types dictate the immutability annotation. To address this case, Glacier includes an additional annotation, @ReadOnly, which is used on array parameters to methods. A @ReadOnly array can also be referenced by a field that has a @ReadOnly array type.

The empty array poses a special problem: is it mutable or immutable? It is fundamentally immutable because it has no indices that can be modified, but it is also possible to declare a mutable array and reference an empty array with it. One workaround might be to declare two different empty arrays: one mutable and one immutable. Instead, the Glacier type hierarchy includes a *bottom element*, so named because it is a subtype of all other types. The bottom element, @GlacierBottom, applies to objects that have all properties of mutable and immutable objects, and can therefore be used when one wants either kind of object. One can declare an empty array of Object as follows: static final Object @GlacierBottom [] EMPTY_ARRAY = new Object @GlacierBottom [0];. null also has annotation @GlacierBottom because it can be assigned to references with any annotation.

When any new object is allocated, it is guaranteed to not be aliased directly. For example, when the clone() method is called on an array, there are no aliases to that array (though there may be aliases to its elements). As a result, the result of a clone() call may be assigned to an @Immutable array or to a @MaybeMutable array; Glacier achieves this by annotating the return type of clone() with @GlacierBottom. Certain JDK methods also return @GlacierBottom arrays, such as Arrays.copyOf.

Though the above complexity is necessary due to the idiosyncrasies of arrays in Java, we expect that most users do not use arrays regularly, instead preferring collection classes, and so these details will not be exposed to most users.

F. Typecasts

Normally, Java permits unsafe downcasts at compile time and checks for safety at runtime. Glacier has no runtime component, so unsafe casts are forbidden. For example, if u is of type @Immutable C, then Glacier reports a compile error on this cast: ((@MaybeMutable C)u).

G. Type parameters

Suppose an @Immutable class has a type parameter:

```
@Immutable class Box<T> {
    T obj;
}
```

If Box is instantiated with a mutable type for T, then Box contains a mutable object, which is a violation of transitive immutability. As a result, Glacier restricts type parameter instantiations on @Immutable classes to @Immutable types. This is a conservative approximation, since the type parameter may never be used in a field. However, checking whether a type parameter is used as a field depends on the implementation of the referenced class, which might result in confusing errors and which would violate modularity: if an immutable class was changed from not using its type parameter in a field to doing so, that would be a breaking change for clients that instantiated the class with a mutable type parameter. Furthermore, it is our experience that most generic classes use their type parameters in fields, so the conservative nature of this restriction is unlikely to be important in many use cases.

H. Robustness to future changes

One of the problems with final is that although it restricts assignability on fields to which it is applied, there is no way to specify that all fields of a class are final. When adding a new field to an immutable class, the author may neglect to mark the field `final`. Likewise, `final` cannot specify restrictions at the *usage* of a type, so clients of a class cannot ensure that it is `final`. Glacier solves this problem by permitting users to annotate any type use with an annotation. If that annotation is inconsistent with the annotation used in the type's declaration, Glacier will report an error. This lets programmers specify that they depend on the immutability of a particular class they are using so that the compiler will report an error if that class is ever edited to make it mutable in the future. Although this offers an opportunity for authors of APIs to break clients, we think of this as exposing an *existing* mechanism of client breakage, which already exists but cannot be identified by the compiler.

I. *Glacier formalization*

To help inform the design of Glacier, we created a formal model (shown in Figure 1). Our formalism is an extension of Featherweight Java [7], which is a commonly-used minimal core calculus for Java. Gray boxes show changes in Glacier. For conciseness, not all rules from Featherweight Java are presented; those not presented are still part of the system.

Syntax:

Mod ::= `assignable` | `final`

CL ::= [`@Immutable`] `class C extends C` `implements` $\bar{\text{I}}$ { $\overline{\text{Mod}}$ $\bar{\texttt{C}}$ $\bar{\texttt{f}}$; K $\bar{\texttt{M}}$}

IF ::= [`@Immutable`] `interface` I `extends` $\bar{\text{I}}$ {M-Decl}

K ::= C($\bar{\texttt{C}}$ $\bar{\texttt{f}}$) { `super` ($\bar{\texttt{f}}$); `this.`$\bar{\texttt{f}}$= $\bar{\texttt{f}}$; }

M-Decl::= C m($\bar{\texttt{C}}$ $\bar{\texttt{x}}$)

M ::= M-Decl { `return` t; }

t ::= x | t.f | t.m($\bar{\text{t}}$) | `new C(`$\bar{\text{t}}$`)` | (C) t | t.f = t |v ::= `new` C($\bar{\text{v}}$)

Subtyping:

$$\frac{}{\texttt{@Immutable Object} <: \texttt{Object}}$$

$$\frac{[\texttt{@Immutable}]\ \texttt{interface}\ I\ \texttt{extends}\ \overline{J}\{\ldots\}}{I <: J_i}$$

$$\frac{[\texttt{@Immutable}]\ \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\{\ldots\}}{C <: D}$$

$$\frac{[\texttt{@Immutable}]\ \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\{\ldots\}}{C <: I_i}$$

Syntactic MUTABLE and IMMUTABLE judgements: If an `@Immutable` class includes mutable fields, it will be judged IMMUTABLE but fail to typecheck.

$$\frac{}{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\ \{\overline{\text{Mod}}\ \bar{\texttt{C}}\ \bar{\texttt{f}},\ K\ \overline{M}\}\ \text{MUTABLE}}$$

$$\frac{}{\texttt{@Immutable class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\ \{\overline{\text{Mod}}\ \bar{\texttt{C}}\ \bar{\texttt{f}},\ K\ \overline{M}\}\ \text{IMMUTABLE}}$$

$$\frac{}{\texttt{interface}\ I\ \texttt{extends}\ \overline{I}\{\text{M-Decl}\}\ \text{MUTABLE}}$$

$$\frac{}{\texttt{@Immutable interface}\ I\ \texttt{extends}\ \overline{I}\{\text{M-Decl}\}\ \text{IMMUTABLE}}$$

MUT-FREE judgement: If a class's fields, including all fields introduced by superclasses, are all final and immutable, then the class may be used as a superclass of an immutable class.

$$\frac{}{\text{Object MUT-FREE}}$$

$$\frac{}{\texttt{@Immutable class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\ \{\ldots\}\text{MUT-FREE}}$$

$$\frac{D\ \text{MUT-FREE} \qquad \forall i.A_i = \texttt{final} \wedge C_i\ \text{IMMUTABLE}}{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\ \{\overline{\text{Mod}}\ \bar{\texttt{C}}\ \bar{\texttt{f}},\ K\ \overline{M}\}\ \text{MUT-FREE}}$$

Static Semantics: The T-mutable-class rule is the same as in FJ except with the additional condition that D is mutable.

$$\frac{\begin{array}{c}\texttt{fields}(D) = \bar{\texttt{D}}\ \bar{\texttt{g}} \\ \bar{\texttt{M}}\ \text{OK in}\ \texttt{C} \\ K = \texttt{C}(\bar{\texttt{D}}\ \bar{\texttt{g}}, \bar{\texttt{C}}\ \bar{\texttt{f}})\ \{\text{super}(\bar{\texttt{g}}); \text{this}.\bar{\texttt{f}} = \bar{\texttt{f}};\} \\ D\ \text{MUT-FREE} \\ \forall i.C_i\ \text{IMMUTABLE} \\ \text{methods}(I) \subset (\text{decl}(\overline{M}) \cup \text{methods}(D))\end{array}}{\texttt{@Immutable class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\ \{\overline{\text{Mod}}\ \bar{\texttt{C}}\ \bar{\texttt{f}},\ K\ \overline{M}\}\ \text{OK}}$$

$$\frac{\begin{array}{c}\texttt{fields}(D) = \bar{\texttt{D}}\ \bar{\texttt{g}} \\ \bar{\texttt{M}}\ \text{OK in}\ \texttt{C} \\ K = \texttt{C}(\overline{\texttt{Dg}}, \overline{\texttt{Cf}})\ \{\text{super}(\bar{\texttt{g}}); \text{this}.\bar{\texttt{f}} = \bar{\texttt{f}};\} \\ \{\text{super}(\bar{\texttt{g}}); \text{this}.\bar{\texttt{f}} = \bar{\texttt{f}};\} \\ D\ \text{MUTABLE} \\ \forall i.I_i\ \text{MUTABLE} \\ \text{methods}(I) \subset (\text{decl}(\overline{M}) \cup \text{methods}(D))\end{array}}{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ \overline{I}\ \{\overline{\text{Mod}}\ \bar{\texttt{C}}\ \bar{\texttt{f}},\ K\ \overline{M}\}\ \text{OK}}$$

$$\frac{\Gamma \vdash t_0 : C_0 \qquad \texttt{fields}(C_0) = \overline{\texttt{Cf}} \qquad \Gamma \vdash t_2 : C_i' \qquad C_i' <: C_i \qquad C_0\ \text{MUTABLE}}{\Gamma \vdash t_0.f_i = t_2 : \text{Unit}}$$

Casting: Casting is as in FJ.

Fig. 1. Formalization of Glacier; gray boxes show differences with Featherweight Java.

IV. EVALUATION: CASE STUDIES

A. *Objectives*

The restrictions that Glacier enforces were justified by prior work [2], our goals of simplicity, and the recommendations of experts [5], but do those restrictions reflect situations that arise in real software? Can Glacier work in software systems that are large and complex? Though we cannot infer from case studies that Glacier is applicable to *all* systems (indeed, it likely is not), the goal of case studies was to gain an understanding of situations to which Glacier does apply and to refine the design of Glacier itself. For example, we found in the second case study that some immutable classes derive from classes that also have mutable subclasses; a previous formulation of Glacier did not reflect that use case. The case studies also drew our attention to the problems of overriding methods defined in `Object`. Finally, the case study systems provided a source of interesting test cases and helped us make Glacier more robust, particularly in the area of type parameters.

B. *Case study: ZK Spreadsheet Model*

ZK Spreadsheet is a commercial, partly open-source, Java spreadsheet implementation [13]. It supports importing documents from Excel and provides a server-based spreadsheet component that can be inserted into web pages via an Ajax client-side component. As a case study of Glacier, we refactored the model portion of ZK Spreadsheet 3.8.3 (comprising about 36 KLOC) so that cell styles were immutable (cell styles record information required for correct visual rendering of cells, such as background color, font, etc.). We also updated the rest of the spreadsheet implementation (comprising about 21 KLOC) to use the new model, accordingly. We added annotations so that Glacier could enforce immutability statically. The refactoring took approximately 20 hours, not counting time spent fixing bugs in Glacier; this would likely have been less if we had already been familiar with the ZK codebase. In the process, we identified two existing bugs in the spreadsheet implementation, one of which was due to incorrect copying code; in our revised version, no copying was necessary because immutable objects can be shared safely. The other bug related to font cache misses when changing fonts in cells.

Before starting, we asked the authors of ZK Spreadsheet whether they had any immutable structures in their software, and they explained that they did not because they were wary of the performance cost of copying that would be likely if objects were immutable. However, cell styles can be shared among many cells, and ZK Spreadsheet has no data structure tracking which cells use a given style. This means that when a user wants to modify a cell's style, the system must make a fresh style, since modifying the existing style might incorrectly affect other cells. As a result, though the cell class was mutable, it was copied on nearly every modification. We believe, therefore, that the performance cost of the change to use immutable styles is minimal.

Our refactoring primarily used three strategies to convert mutable classes to immutable ones. In most cases, clients that mutated classes changed a small number of parameters at a time; in these cases, we added a new constructor that took the previous instance and the new value of the parameter. This approach was similar to that used by Kjolstad et al in their automatic refactoring tool [14]. Other classes had many attributes that typically needed to be modified at once; if those constituted most of the state of the object, the client called a constructor; otherwise, we used a mutable Builder object [15] to represent the collection of changes. This approach prevented overly verbose, inefficient implementations that would have resulted from using the first approach alone.

From this case study, we conclude that transitive class immutability, as implemented in Glacier, can be adopted in some real-world systems with a practical amount of effort and that Glacier can enforce transitive class immutability in some complex, real systems.

C. Case study: Guava ImmutableList

After our initial case study on application software, we wanted to see how Glacier might be used on a system that has very different characteristics. Google's Guava project includes several immutable collection classes, including `ImmutableList`; though relatively small, this library is designed to be used in a very wide range of projects. We annotated `ImmutableList` and its superclass, `ImmutableCollection`, with Glacier's `@Immutable` and made the appropriate changes necessary to make them compile. As a result of the use of generics in `ImmutableList`, when using Glacier, it was necessary to specify annotations for the bounds of the type parameters. For example, the original declaration of `ImmutableList` included: `public abstract class ImmutableList<E> extends ImmutableCollection<E>`. With Glacier, however, E is restricted to immutable objects, so the new declaration reads `@Immutable public abstract class ImmutableList<E extends @Immutable Object> extends ImmutableCollection<E>`. This constrains `E` to descending from `@Immutable Object`, expressing an *upper bound* on the type parameter. One might expect to write `@Immutable E` rather than `E extends @Immutable Object`, but Java specifies that `@Immutable E` expresses a *lower* bound on `E` rather than an upper bound; that is, it specifies that `E` must be a *supertype* of `@Immutable Object` rather than a *subtype*.

ImmutableList included this method: `static Object[] checkElementsNotNull(Object... array)`. `checkElementsNotNull` took and returned a mutable array, but callers passed an immutable array to `checkElementsNotNull`, which was an error. Because Java methods cannot be overloaded with different annotations, we were unable to provide an alternative method with the same name that takes and returns an immutable array. This is one case in which polymorphism might be desirable. However, because `checkElementsNotNull` never modifies the input array, it is not actually necessary to return an array. We addressed this problem by refactoring this method to only do the checking and not return the input array.

The only aspect of ImmutableList that we were unable to represent in Glacier is a cache in ImmutableCollection, which caches an ImmutableList representation of the collection. Some languages, such as C++, permit exclusion of specific fields from enforcement of immutability. Glacier has no provision for allowing mutable fields in immutable objects so that Glacier can provide strong guarantees. A workaround would be to populate the cache inside the ImmutableCollection constructor, but this would have a performance cost if the list representation is never needed. We hope to extend Glacier in the future to permit lazy initialization of fields in immutable objects; such initialization could be done safely if it is based only on state that was available at initialization time.

V. User study

We conducted a user study of Glacier and found that Java programmers could use Glacier effectively with little training. In contrast, Java programmers without Glacier were unable to use `final` to express immutability correctly even after receiving appropriate training. We also found that Java programmers without Glacier mutated immutable state, creating bugs and security flaws; Glacier detects these errors statically. Though there is a wide variety of proposals in the literature for systems that support immutability, we have not found any others that have been evaluated in a formal user study.

A. Methodology

We recruited 20 experienced Java programmers to participate in our study, which was approved by our IRB. For each sequential pair of participants, we randomly assigned one to a control condition, in which the participant used `final`, and the other to a treatment condition, in which the participant used

Glacier, resulting in ten participants in each condition. After obtaining informed consent, we gave participants a pre-study questionnaire regarding their programming experience, including an assessment of their prior understanding of `final`. Participants assigned to the `final` condition were asked to read three pages of documentation on `final`; participants in the Glacier condition completed a two-page tutorial on how to use Glacier. Participants were permitted to ask questions during this phase of the study. The remainder of the study consisted of four programming tasks in three different Java packages. Participants used the IntelliJ IDEA Community 2016.2 Integrated Development Environment (IDE) with Java 1.8 on a 15" MacBook Pro; we recorded audio and a video of the screen for analysis. During the study, we helped participants as needed with issues related to the computer system and IDE they were using, such as how to find a web browser and how to copy/paste, but did not answer questions about Glacier or `final`.

A study replication package is available [16], including all materials that were used in the study.

Task 1: making Person immutable. The Person package only included two classes: Person and Address. We asked participants: "Please make any necessary changes so that 'Person' in the 'person' package is immutable. After you're done, there should be no way to change an instance of a class after it is created." Participants had 22 minutes to complete this task.

```
public class Person {
    String name;
    Address address;
    ...
}
```

We expected that some participants in the `final` condition would neglect to mark Address as `final`.

Task 2: making Accounts immutable. The Accounts package represents all of the user accounts on a computer system. We asked participants: "Please make any necessary changes so that 'Accounts' in the 'useraccounts' package is immutable. After you're done, there should be no way to change an instance of a class after it is created." Participants had 20 minutes to complete this task.

```
public class Accounts {
    User [] users;
    ...
}
```

We expected that some participants in the `final` condition would neglect to modify the `User` class; in addition, making this class immutable required defensively copying the `users` array because there is no way in Java to make array elements `final`, and we expected that some participants would forget.

Participants in the Glacier condition who did not complete tasks 1 and 2 in the allotted time were told how to finish because otherwise the resulting compiler errors would interfere with the next tasks.

Revision with advice. After they completed tasks 1 and 2, participants in the `final` condition were given a copy of page 73 from *Effective Java* [5], which outlines how to make a class immutable:

1) Don't provide any methods that modify the object's state.
2) Ensure that the class can't be extended.
3) Make all fields `final`.
4) Make all fields `private`.
5) Ensure exclusive access to any mutable components.

Participants could ask any questions for clarification; then, they were told they could revise their work from the previous tasks if they liked.

Task 3: FileRequest.execute(). We were interested in whether using Glacier would prevent programmers from creating security vulnerabilities in their software. The Java getSigners() bug [17] involved a private array being returned from an accessor, enabling any client to modify the contents of the array. We replicated the structure of the getSigners() bug in the context of the code from the previous task. Participants were told: "A FileRequest represents a request for a particular file from a web server, represented by a WebServer object. Normally, third-party clients implement their own types of requests, so it is important that the Accounts object that a Request gets access to is secure. As a test of the Accounts system, please implement FileRequest.execute() so that it does the appropriate access checks before granting access. In the process, you will need to implement User.getAuthorizedFiles()." Participants had 20 minutes to complete this task.

Although implementing User.getAuthorizedFiles() was stated in the description as an incidental task, we were primarily interested in whether participants who used `final` remembered to copy the private array, `authorizedFiles`. Neglecting to do so would result in a security vulnerability similar to the `getSigners()` bug, since then any client of User could change which files a User was authorized to access. In the Glacier condition, participants could either copy the private array before returning it, or change the return type to return an `@Immutable` array; by enforcing transitive immutability of User, Glacier would identify all unsafe handling of the array. Participants in the `final` condition who did not copy the array but told the experimenter they were done with the task were given a sample of exploit code and then given an opportunity to revise their solution.

Task 4: HashBucket.put(). We wanted to know whether Glacier could prevent users from accidentally inserting mutation into existing immutable classes in real-world-like situations; is this an error that many programmers make without Glacier? We based our task on bug #1297 [18] in BaseX, which is an open-source XML database [19]. In that bug, the `delete` method on an implementation of an immutable hash map incorrectly modified the old hash map's data structures. In order to replicate this in a small user study, we simplified the hash map implementation to use a much simpler data structure while leaving the external API and comments in place as much as possible. The result was code that should be substantially easier to read and understand than the original and included many hints that the class was immutable, such as the fact

	final	Glacier
Users who made errors enforcing immutability (after all tasks)	10/10	0/10
Completed FileRequest.execute() tasks with security vulnerabilities	4/8	0/7
Completed HashBucket.put() tasks with bugs	7/10	0/7

TABLE I
SUMMARY OF USER STUDY RESULTS

Error	# users
Provided mutating methods	0
`Person` not `final`	6
`Address` not `final`	10
`Accounts` not `final`	2
`User` not `final`	9
Fields of `Person` not `final`	2
Fields of `Address` not `final`	6
`Accounts.users` not `final`	1
Fields of `User` not `final`	4
Fields of `Person` not `private`	4
Fields of `Address` not `private`	8
`Accounts.users` not `private`	2
Fields of `User` not `private`	7
Omitted copying `users` in `Accounts` constructor	4
Omitted copying `users` in `Accounts.getUsers()`	2
Omitted copying `authorizedFiles` in `User` constructor	8

TABLE II
ERRORS MADE BY PARTICIPANTS USING `FINAL` FOR IMMUTABILITY THAT REMAINED AFTER REVISION. ERRORS CONSIST OF FAILURES TO FOLLOW THE ADVICE IN *Effective Java* [5].

that all modification methods returned a new object and the fact that the implementations of the provided methods made extensive use of copying.

We gave participants our hash map, which was implemented with an array of buckets, each of which contained lists of keys and values. The instructions to participants included: "HashBucket.put() is only partially implemented. Please finish the implementation by replacing the placeholder 'return this' with the right code." Since this task was last, we gave participants as much time as they needed to complete this task except when the total study period was exhausted. Users in the `final` condition who erroneously modified the old object's data structures and declared that they were done, if time allowed, were given an additional test case that exhibited the problem and the opportunity to fix their implementations.

B. Participants

We solicited participation from several different degree program mailing lists at Carnegie Mellon and from our acquaintances. Most of the participants were either Master's or Ph.D. students. We recruited twenty Java programmers, six of whom were female, and paid them $15 after they completed our study, which took about an hour and a half. Their programming experience ranged from four to nineteen years, with a mean of 9.5 years. Everyone had at least a year of Java experience; the mean was three years. They had an average of two years of professional experience writing software. We also asked participants to self-report their level of Java expertise, selecting from "novice," "intermediate," and "expert." Three participants identified themselves as experts; the rest considered themselves intermediate-level. Fifteen (75%) had used Java annotations before; eighteen (90%) had used `final` before.

We asked participants five questions about the behavior of `final`; the average score was 3.45/5. 11 participants knew what `final` meant when specified on a class, and 11 knew what `final` meant when specified on a method declaration. 17 participants knew that `final` does not forbid assignment in a constructor; 17 knew that it forbids assignment in setters; 13 knew that it does not forbid calling setters on final fields. We found no relationship between experience using Java and the number of these questions participants answered correctly.

C. Results

Results are summarized in Table I. The denominators vary because some participants did not complete all tasks.

After participants revised their code according to the advice from *Effective Java*, we counted errors (shown in Table II) participants made with respect to that advice. Despite having the recommendations available while editing, every participant using `final` made mistakes. Two users made no non-transitive mistakes (i.e. mistakes directly in the `Person` and `Accounts` classes). No users remembered to make `Address` final, even though an instance of `Address` was used in `Person`. We conclude that enforcing immutability with Java as it currently exists is too complicated and error-prone for Java programmers to do effectively.

We stopped one user in each task at the time limits (22 minutes and 20 minutes, respectively). The average initial (pre-revision) time for Person and Accounts were 4 and 6 minutes, respectively. Participants spent an average of 6 minutes revising after receiving the *Effective Java* page. Among participants who said they were done with both tasks, the total average time, including revisions, was 15 minutes.

Making Person and Accounts immutable with Glacier. All of the Glacier participants successfully annotated Person with Glacier. Three of them did not finish modifying Accounts within 20 minutes, though one was given additional time and succeeded after 6 extra minutes. A common obstacle in the Accounts task was initializing an immutable array. The starter code included `String[] files = {"RootUserBankAccounts.txt"};`. Unfortunately, Java forbids annotations in an obvious place for array constants: `String @Immutable [] files = @Immutable {"RootUserBankAccounts.txt"};`. Participants needed to write `String @Immutable [] files = new String @Immutable [] {"RootUserBankAccounts.txt"};`. Most users solved this with an Internet search, but the time required to do this was very variable. If we ignored this time, then two additional participants (9/10) would have succeeded within 20 minutes.

Several of the earlier participants did not annotate one of the classes until the next task due to a problem with the build system we were using, in which it failed to rebuild all files that depended on the changed files; we later revised the instructions to avoid this problem. Correcting for this and deducting the time spent on array initialization resulted in an average total annotation time (across both tasks) of 11 minutes; applying these corrections to `final` users results in an average of 14 minutes for those users. The difference compared to the average time for `final` users is significant with $p < 0.1$ (Wilcoxon rank sum test).

FileRequest.execute(). Seven of the Glacier users successfully completed this task. One participant encountered an unrelated build system bug; two did not finish within 20 minutes. One of these two misinterpreted the starter code and got stuck while implementing `getAuthorizedFiles()` as a much more complex method than the accessor we intended it to be. Eight of the `final` users successfully completed this task within 20 minutes. Though Glacier prevented any security problems for the Glacier users, four of the `final` users (half of those who finished) failed to copy the private authorizedFiles array, causing a security vulnerability similar to the Java `getSigners()` bug.

HashBucket.put(). All of the `final` users said they completed the task within 27 minutes, but they required up to an additional 11 minutes to fix their bugs after we showed them the additional test case. The average total time for `final` users was 18 minutes; the average total time for Glacier users who finished was 14 minutes. The difference in times was not significant. Seven of the `final` users incorrectly modified the HashBucket's internal data structures, resulting in a bug. In addition, six `final` participants returned the existing HashBucket instance rather than creating a new one. One Glacier user gave up after 29 minutes, having gotten an error from Glacier after trying to modify an immutable array and could not figure out another way to solve the problem. In addition, two Glacier users had already exhausted the overall study period and could not be given enough time to finish. Overall, 3 of 10 `final` users completed the task correctly; Glacier detected the problem statically, and 7 of 9 Glacier users who started the task completed it successfully.

D. Discussion

We had hypothesized that participants would spend significantly less time specifying immutability with Glacier than with `final` because using Glacier required only adding annotations, whereas using `final` required making several kinds of changes, such as copying arrays in constructors. We did not see a significant difference in times at $\alpha = .05$, but we believe we would have if we had more participants: variance was high in both tasks, as is typical in studies of programmers [20]. For example, some users (particularly in the `final` condition) wrote test code to see if they could cause data to be mutated; others wrote no tests (time spent writing and executing tests was included in the task times above). In addition, `final` users would have spent more time if they had completed all of the work required to do the tasks correctly. However, even if Glacier did not save users any time in specifying immutability relative to `final`, it likely took users the same amount of time to enforce a much stronger property while avoiding mistakes.

One might have expected `final` participants to make fewer errors than they did, considering that all of the participants in that condition had used `final` before. However, some of the participants had never attempted to use `final` to enforce immutability, as one participant remarked when starting to read the documentation on `final`: "I've only used `final` on integers before, so this will be instructive." Many of the participants vocalized considering and rejecting Bloch's advice, for example reasoning that since a class had no setters, it was immutable, and therefore other changes (such as making the class `final`) were unnecessary. When using `final` for immutability, then, it is not sufficient to say that a class should be immutable; one must say exactly what kinds of future changes the class should be robust to.

In the `final` condition, the requirement to defensively copy arrays in constructors and accessors was particularly problematic. For example, 80% of the participants omitted a defensive copy of the `authorizedFiles` array in the `User` constructor. Some participants complained about the performance impact of this strategy: after implementing defensive copying in `getUsers()`, one participant remarked, "I'm not really happy. If there's a lot of users and `getUsers` is called frequently. . . that will slow down the performance." Two participants, after reading the *Effective Java* page, asked for an explanation of why defensive copies were necessary, but even after hearing the explanation, one of these two users omitted required defensive copying. We conclude that although following the advice would result in certain protections against mutation, few users can successfully apply the advice to even a simple programming project, even when given the advice immediately before needing to use it, and even with access to the recommendations while programming; we believe the problem is one of complexity, lack of enforcement, and lack of understanding that the recommendations are relevant.

Some Glacier users reported that the two annotations on arrays—one on the array itself and one on the component type—was confusing. Though this a fundamental aspect of containers, we believe that one of the reasons users faced difficulty is that the design of arrays (a relatively old feature) is inconsistent with the approach used in generic classes, in which the component type is specified in angle brackets rather than next to the container name. However, the difficulties with arrays may not represent a significant problem for most classes, which use containers such as `ArrayList` rather than arrays. The user study did not include tasks involving annotated type parameters; our experience suggests that *using* them is straightforward, but writing parametrized classes can be difficult due to the need to specify type parameter bounds.

E. Limitations

The main threats to validity are of our study are due to the simplicity and limited nature of our tasks and the relative inexperience of our participants. Likewise, our participants came from a relatively narrow set of backgrounds. It is possible that more-expert Java programmers would have been able to use `final` more successfully and that the training we provided for `final` was insufficient. We think it is unlikely that programmers would be *less* likely to incorrectly mutate immutable structures in a more complex codebase than the one we provided, but perhaps more-expert programmers would be better at identifying the implicit immutability requirements. We selected our tasks to expose opportunities to mutate structures incorrectly; though we have shown that these tasks do result in incorrect mutation, the fraction of real-world programming tasks that are similar is unknown.

VI. Related Work

Usability analysis of programming languages has been pursued by a variety of different researchers when considering different aspects of language designs [21]. Endrikat et al., for example, found that static typing improves software maintainability [22]. However, Uesbeck et al. found that although C++ lambdas are a highly-touted feature of C++, they actually impose costs to programmers [23]. Together, these results suggest that although static typechecking can be beneficial, it is important to do usability studies to assess the impact of any proposed language feature.

Though there is a large collection of immutability-related systems proposed in the literature, we have not found any usability studies of these systems. A more comprehensive review of these can be found in prior work [2], [1]. IGJ [10] implemented class- and object- immutability as well as read-only references and supported polymorphism, but did not enforce transitivity; this work included case studies but no user studies, so it is unknown whether other Java programmers can use IGJ effectively. Haack and Poll [24] proposed a type system for object immutability, read-only references, and class immutability that supported initialization outside constructors, but the only evaluation of this system was theoretical. Skoglund and Wrigstad [25] proposed a type system for read-only references in Java, but it only supported a subset of Java and it does not appear that they implemented their system. Java `final` and C/C++ `const` do not express transitive class immutability. .NET Freezable and JavaScript `Object.freeze` are enforced dynamically, not statically.

Pure4j is an annotation processor for Java that provides `@ImmutableValue` [26], and, like Glacier, enforces that annotated classes are transitively immutable. However, it provides no solution for arrays, assuming that all arrays are mutable. It requires that all fields be declared `final`, which is redundant in Glacier. Unlike Glacier, it requires that public methods of immutable classes take only immutable parameters and that instance methods that are not inherited from a base class be pure. This is a stronger restriction than immutability and forbids access to global state, whereas immutability in Glacier pertains only to state reachable specifically via fields of objects. Though method purity can be helpful in certain cases, such as that of thread safety, it also restricts applicability. For example, a pure method cannot read or write files or the network. Our focus on immutability rather than purity reflects the evidence we have of what developers need [2].

Immutables is another annotation processor for Java [27], but rather than enforcing immutability, it generates immutable classes from abstract value classes. It can also automatically generate builders and factories. Kjolstad et al. proposed a tool, Immutator [14], to automatically make immutable versions of classes and conducted an experiment showing that programmers make errors when refactoring classes to be immutable; our focus here was on enforcing immutability, not refactoring.

Some functional languages, such as Haskell, emphasize immutability. Isolating code that has side effects is a core part of Haskell, so it is unclear what the usability impact of this design decision is. Other functional languages, such as SML, promote immutable value types, but do not prevent mutable state or provide any way of forbidding it inside modules.

VII. Future Work

Future work should expand the range of situations to which Glacier applies by adding support for delayed initialization of fields (for example, caches) in immutable objects. In addition, Glacier does not consider external sources of mutability, such as the filesystem or network; future work should analyze to what extent these kinds of hidden mutability compromise the guarantees that Glacier provides. A future corpus study could analyze to what extent the system applies to existing code. A refactoring tool could help software engineers adopt Glacier more easily and also be used in a corpus study of applicability.

We have not found data regarding to what extent (and in what situations) designing components to be immutable is beneficial. Understanding when to make components immutable is a critical step in using immutability systems effectively.

VIII. Conclusion

We have designed, formalized, and implemented Glacier, which enforces transitive class immutability in a Java annotation processor. We conducted a user study and found that Java programmers could generally specify immutability effectively with it; in contrast, Java programmers in our study could not use `final` to specify immutability even though they had advice on how to do so. We also found that programmers incorrectly mutate immutable data structures when they only have `final`, whereas Glacier detects those errors statically. Glacier represents a promising approach to enforcing immutability in real-world Java software and implements a model that could be extended to other languages as well.

Acknowledgment

We would like to thank Michael Ernst, Werner Dietl, and Suzanne Millstein for their help with the Checker Framework, and Alex Potanin for helpful discussions. We are also grateful for the help of our experiment participants and the team at ZKoss. This material is supported in part by NSA lablet contract #H98230-14-C-0140 and by NSF grant CNS-1423054. 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.

REFERENCES

[1] A. Potanin, J. Östlund, Y. Zibin, and M. D. Ernst, "Immutability," in *Aliasing in Object-Oriented Programming*, 2013, pp. 233–269.

[2] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull, "Exploring language support for immutability," in *International Conference on Software Engineering*, 2016.

[3] 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.

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

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

[6] P. Helland, "Immutability changes everything," *Communications of the ACM*, vol. 59, no. 1, pp. 64–70, dec 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2859829.2844112

[7] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight java: A minimal core calculus for java and gj," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 3, pp. 396–450, May 2001.

[8] J. Witschey, O. Zielinska, A. Welk, E. Murphy-Hill, C. Mayhorn, and T. Zimmermann, "Quantifying Developers' Adoption of Security Tools," in *Proceedings of Foundations of Software Engineering*, 2015.

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

[10] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kielun, and M. D. Ernst, "Object and reference immutability using Java generics," in *Foundations of Software Engineering*. ACM, 2007, pp. 75–84.

[11] L. Mikhajlov and E. Sekerinski, "A study of the fragile base class problem," in *European Conference on Object-Oriented Programming*. London, UK, UK: Springer-Verlag, 1998, pp. 355–382.

[12] Google, "Guava: Google core libraries for java," https://github.com/google/guava. Accessed Aug. 9, 2016.

[13] Potix Corportation, "Zk spreadsheet," https://www.zkoss.org/product/zkspreadsheet. Accessed Aug. 9, 2016.

[14] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir, "Transformation for class immutability," in *International Conference on Software Engineering*. New York, New York, USA: ACM Press, May 2011, p. 61.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[16] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine, "Glacier," http://mcoblenz.github.io/Glacier/. Accessed Aug. 12, 2016.

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

[18] BaseX Team, "Map: remove and check values," https://github.com/BaseXdb/basex/issues/1297. Accessed Aug. 10, 2016.

[19] ——, "Basex," http://basex.org. Accessed Aug. 10, 2016.

[20] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1335–1382, oct 2014.

[21] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon, "Programmers are users too: Human-centered methods for improving programming tools," *Computer*, vol. 49, no. 7, pp. 44–52, July 2016.

[22] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik, "How do api documentation and static typing affect api usability?" in *International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 632–642.

[23] P. M. Uesbeck, A. Stefik, S. Hanenberg, J. Pedersen, and P. Daleiden, "An empirical study on the impact of c++ lambdas and programmer experience," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 760–771. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884849

[24] C. Haack and E. Poll, "Type-based Object Immutability with Flexible Initialization," in *European Conference on Object-Oriented Programming*, Jul. 2009.

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

[26] R. Moffat, "Pure4j @immutablevalue specification," http://pure4j.org/concordion/org/pure4j/test/checker/spec/ImmutableValue.html. Accessed Aug. 24, 2016.

[27] E. Lukash, "Immutables," https://immutables.github.io/. Accessed Aug. 24, 2016.