# Transformation for Class Immutability

Fredrik Kjolstad Danny Dig Gabriel Acevedo Marc Snir
University of Illinois at Urbana-Champaign
{kjolsta1, dig, acevedo7, snir}@illinois.edu

## ABSTRACT

It is common for object-oriented programs to have both mutable and immutable classes. Immutable classes simplify programing because the programmer does not have to reason about side-effects. Sometimes programmers write immutable classes from scratch, other times they transform mutable into immutable classes. To transform a mutable class, programmers must find all methods that mutate its transitive state and all objects that can enter or escape the state of the class. The analyses are non-trivial and the rewriting is tedious. Fortunately, this can be automated.

We present an algorithm and a tool, Immutator, that enables the programmer to safely transform a mutable class into an immutable class. Two case studies and one controlled experiment show that Immutator is useful. It (i) reduces the burden of making classes immutable, (ii) is fast enough to be used interactively, and (iii) is much safer than manual transformations.

**Categories and Subject Descriptors:** D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

**General Terms:** Design, Management

**Keywords:** Program transformation, immutability

## 1. INTRODUCTION

An immutable object is one whose state can not be mutated after the object has been initialized and returned to a client. By object state we mean the *transitively reachable state*. That is, the state of the object and all state reachable from that object by following references.

Immutability makes sequential programs simpler. An immutable object, sometimes known as a *value object* [17], is easier to reason about because there are no side-effects [7]. Applications that use immutable objects are therefore simpler to debug. Immutable objects facilitate persistent storage [2], they are good hash-table keys [12], they can be compared very efficiently by comparing identities [2], they can reduce memory footprint (through interning/memoization [12,14] or flyweight [8]). They also enable compiler optimizations such as reducing the number of dynamic reads [16]. In fact, some argue that we should always use immutable classes unless we explicitly need mutability [3].

In addition, immutability makes distributed programming simpler [2]. With current middleware technologies like Java RMI, EJB, and Corba, a client can send messages to a distributed object via a local proxy. The proxy implements an update protocol, so if the distributed object is immutable then there is no need for the proxy.

Moreover, as parallel programming becomes ubiquitous in the multicore era, immutability makes parallel programming simpler [9, 13]. Since threads can not change the state of an immutable object, they can share it without synchronization. An immutable object is *embarrassingly thread-safe*.

However, mainstream languages like Java, C#, and C++ do not support deep, transitive immutability. Instead, they only support shallow immutability through the `final`, `readonly`, and `const` keywords. This is not enough, as these keywords only make *references* immutable, not the objects referenced by them. Thus, the transitive state of the object can still be mutated.

To get the full benefits of immutability, deep immutability must therefore be *built into* the class. If a class is *class immutable*, none of its instances can be transitively mutated. Examples in Java include `String` and the classes in the `Number` class hierarchy.

It is common for OO programs to contain both mutable and immutable classes. For example, the JDigraph open-source library contains `MapBag` and `ImmutableBag`. `MapBag` is intended for cases where mutation is frequent, and `ImmutableBag` where mutations are rare.

Sometimes programmers write an immutable class from scratch, other times they refactor a mutable class into an immutable class. The refactoring can be viewed as two related technical problems:

1. The *conversion problem* consists of generating an immutable class from an existing mutable class.

2. The *usage problem* consists of modifying client code to use the new immutable class in an immutable fashion.

This paper solves the conversion problem. To create an immutable class from a mutable class (from here on referred as the *target class*), the programmer needs to perform several tasks. The programmer must search through the methods of the target class and find all the places where the transitive state is mutated. This task is further complicated


*ICSE* '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

by polymorphic methods and mutations nested deep inside call chains that may extend into third party code.

Moreover, the programmer must ensure that objects in the transitive state of the target class do not escape from it, otherwise they can mutated by client code. Such escapes can happen through return statements, parameters, or static fields. Finding objects that escape is non-trivial. For example, an object can be added to a `List` that is returned from a method, causing the object to escape along with the `List`.

Furthermore, once the programmer found all mutations, she must rewrite mutator methods, for example by converting them to factory methods. She must also handle objects that enter or escape the class, for example by cloning them.

In 346 cases we studied these code transformations required changing 45 lines of code per target class, which is tedious. Furthermore, it required analyzing 57 methods in the call graph of each target class to find mutators and entering/escaping objects. Because this analysis is interprocedural and requires reasoning about the heap, it is non-trivial and error-prone. In a controlled experiment where 6 experienced programmers converted JHotDraw classes to immutable counterparts, they took an average of 27 minutes, and introduced 6.37 bugs per class.

To alleviate the programmer's burden when creating an immutable class from a mutable class, we designed an algorithm and implemented a tool, Immutator, that works on Java classes. We developed Immutator on top of Eclipse's refactoring engine. Thus, it offers all the convenience of a modern refactoring tool: it enables the user to preview and undo changes and it preserves formating and comments. To use it the programmer selects a target class and chooses Generate Immutable Class from the refactoring menu. Immutator then verifies that the transformation is safe, and rewrites the code if the preconditions are met. However, if a precondition fails, it warns the programmer and provides useful information that helps the programmer fix the problem.

At the heart of Immutator are two inter-procedural analyses that determine the safety of the transformation. The first analysis determines which methods mutate the transitive state of the target class. The second analysis is a class escape analysis that detects whether objects in the transitive state of the target class state *may* escape. Although Immutator transforms the source code, the analyses work on bytecode and correctly account for the behavior of third-party Java libraries.

There is a large body of work [1,18–20] on detecting whether methods have side effects on program state. Previous analyses were designed to detect *any* side effect, including changes to objects reachable through method arguments and static variables. In contrast, our analysis intersects the mutated state with the objects reachable through the `this` reference. Therefore, it only reports methods that have a side effect on the current target object's state.

Similarly, previous escape analyses [4, 24] report any object that escapes a method, including locally created objects. Our analysis only reports those escaping objects that are also a part of the transitive state of the target class.

This paper makes the following contributions:

**Problem Description** While there are many approaches to specifying and checking immutability this is, to the best of our knowledge, the first paper that describes the problems and challenges of transforming a mutable class into an immutable class.

**Transformations** We present the transformations that convert a Java class to an immutable Java class.

**Algorithm** We have developed an algorithm to automatically convert a mutable class to an immutable class. The algorithm performs two inter-procedural analyses; one that determines the mutating methods, and one that detects objects that enter or escape the target class. Based on information retrieved from these and other analyses our algorithm checks preconditions and performs the mechanical transformations necessary to enforce immutability.

**Implementation** We have implemented the analyses and code transformations in a tool, Immutator, that is integrated with the Eclipse IDE.

**Evaluation** We ran Immutator on 346 classes from known open-source projects. We also studied how open-source developers create immutable classes manually. Additionally, we designed a controlled experiment with 6 programmers transforming JHotDraw classes manually. The results show that Immutator is useful. First, the transformation is widely applicable: in 33% of the cases Immutator was able to transform classes with no human intervention. Second, several of the manually-performed transformations are not correct: open-source developers introduced an average of 2.1 errors/class, while participants introduced 6.37 errors/class; in contrast, Immutator is safe. Third, on average, Immutator runs in 2.33 seconds and saves the programmer from analyzing 57 methods and changing 45 lines per transformed class. In contrast, participants took an average of 27 minutes per class. Thus, Immutator dramatically improves programmer productivity.

Immutator as well as the experimental data can be downloaded from: `http://refactoring.info/tools/Immutator`

## 2. MOTIVATING EXAMPLE

We describe the problems and challenges of transforming a mutable class into an immutable class using a running example. Class `Circle`, shown on the left-hand side of Fig. 1, has a center, stored in field `c`, and a radius, stored in field `r`. There are several methods to modify or retrieve the state. The programmer decides to transform this class into an immutable class, since it makes sense to treat mathematical objects as value objects.

Transforming even a simple class like `Circle` into an immutable class, as shown on the right-hand side of Fig. 1, is non-trivial. First, the programmer must find all the mutating methods. Method `setRadius` on line 19 is a direct mutator, and is easy to spot because it assigns directly to a field. Method `moveTo(int, int)` on line 27 is a mutator too. However, the code on line 30 does not change the value of `c` directly, but instead changes the object that `c` references. Therefore, this method mutates the transitive state of `Circle`. Method `moveBy` on line 34 is another mutator that does not mutate the object directly. Instead, it mutates state indirectly by calling `moveTo(Point)`. Finding all mutators (transitive and indirect) is complicated by long call chains, polymorphic methods, aliases, and third-party library code.

Furthermore, the programmer must locate all the places where an object enters or escapes the target class. Consider a client that creates a `Point` object and passes it to

```
1  public class Circle {
2    private Point c = new Point(0, 0);
3    private int   r = 1;
4
5
6
7
8
9
10
11
12
13
14
15   public int getRadius() {
16     return r;
17   }
18
19   public void setRadius(int r) {
20     this.r = r;
21   }
22
23   public void moveTo(Point p) {
24     this.c = p;
25   }
26
27   public void moveTo(int x, int y) {
28
29
30     c.setLocation(x, y);
31
32   }
33
34   public void moveBy(int dx, int dy) {
35     Point center = new Point(c.x+dx, c.y+dy);
36     moveTo(center);
37
38   }
39
40   public Point getLocation() {
41     return c;
42   }
43 }
```

```
1  public final class ImmutableCircle {
2    private final Point c;
3    private final int   r;
4
5    public ImmutableCircle() {
6      this.c = new Point(0, 0);
7      this.r = 1;
8    }
9
10   private ImmutableCircle(Point c, int r) {
11     this.c = c;
12     this.r = r;
13   }
14
15   public int getRadius() {
16     return r;
17   }
18
19   public ImmutableCircle setRadius(int r) {
20     return new ImmutableCircle(this.c, r);
21   }
22
23   public ImmutableCircle moveTo(Point p) {
24     return new ImmutableCircle(p.clone(), this.r);
25   }
26
27   public ImmutableCircle moveTo(int x, int y) {
28     ImmutableCircle _this =
29       new ImmutableCircle(this.c.clone(), this.r);
30     _this.c.setLocation(x, y);
31     return _this;
32   }
33
34   public ImmutableCircle moveBy(int dx, int dy) {
35     Point center = new Point(c.x+dx, c.y+dy);
36     ImmutableCircle _this = moveTo(center);
37     return _this;
38   }
39
40   public Point getLocation() {
41     return c.clone();
42   }
43 }
```

**Figure 1: Immutator converts a mutable `Circle` (left pane) into an immutable class (right pane).**

`moveTo(Point)`. Since the client holds a reference to the point, it can still mutate the object through the retained reference. The programmer may not have access to all existing and future client code so she must conservatively assume that the target class can be mutated through entering and escaping objects. Therefore, to enforce deep immutability, the programmer must find all the places where objects enter the target class (line 23–24) or escape (line 41), and clone them. However, the programmer should avoid excessive cloning and only clone where absolutely required.

Even for this simple example, the transformation requires inter-procedural analysis (line 30 and 36), which must take pointers into account (line 30). Our approach combines the strength of the programmer (the higher-level understanding of where immutability should be employed) with the strengths of a tool (analyzing many methods and making mechanical transformations).

Immutator automatically handles the rewriting (Section 4) and analysis (Section 5) required to make a class immutable.

## 3. IMMUTATOR

We implemented our algorithm for Generate Immutable Class as a plugin in the Eclipse IDE. To use Immutator, the programmer selects a class and then chooses the Generate Immutable Class option from the refactoring menu. Before applying the changes, Immutator gives the programmer the option to preview them in a before-and-after pane. Then Immutator makes the class *deeply* immutable.

Our algorithm transforms the target class in-place. However, the tool makes a copy of the target class and then transforms this copy. This provides the programmer with two variants of the same class: a mutable and an immutable one. The programmer decides where it makes sense to use one over the other.

However, the programmer can not use the deeply immutable version if the class is to be used in client code that relies on structural sharing of mutable state. Consider a `Graph` that contains mutable `Node` objects. The semantics of the `Graph` class ensure that several nodes can share the same successor node. If the programmer made `Graph` immutable, Immutator would change mutator methods like `addEdge(n1,n2)` to clone the entering nodes, thus transforming the graph into a tree. On the other hand, structural sharing of immutable objects does not contradict with deep-copy immutable semantics. If the `Graph` contained immutable `Node` objects, then Immutator would not clone `Node` objects, thus preserving the sharing semantics of the original class.

Before transforming the target class, Immutator checks that it meets four preconditions, and reports failed preconditions to the programmer. The programmer can decide to ignore the warnings and proceed, or cancel the operation, fix the root cause of the warnings and then re-run Immutator.

## 3.1 Transformation Preconditions

Immutator checks the following preconditions:

**Precondition #1** The target class can only have superclasses that do not have any *mutable* state.

**Precondition #2** The target class can not have subclasses as these can add mutable state to the target objects.

**Precondition #3** Mutator methods in the target class must have a `void` return type and must not override methods in superclasses. This is because Immutator rewrites mutator methods to return new instances of the target class and must use the return type for this. Methods in Java can only return one value and it is not allowed to change the return type when overriding a method.

**Precondition #4** Objects that enter or escape the transitive state of the target class must either already implement `clone`, or the source code of their classes must be available so that a `clone` method can be added.

While these preconditions may seem restrictive, we believe that value classes are likely to meet them. For example, software that follows the command-query separation principle (methods either perform an operation, or return a value) will not have mutators with non-void return types, thus meeting precondition 3. Furthermore, preconditions 1 and 2 are limitations of the current implementation, and not inherent to the approach. We leave for future work to refactor a whole class hierarchy.

# 4. TRANSFORMATIONS

This section describes the transformations that Immutator applies to the target class. We will use the motivating example introduced in Fig. 1 to illustrate the transformations.

**Make fields and class final** First, Immutator makes all the fields of the class `final`. Final fields in Java can only be initialized once, in constructors or field initializers. Immutator also makes the target class final. This prevents it from being extended with subclasses that add mutable state.

**Generate constructors** Immutator adds two new constructors (line 5 and 10). The first constructor is the default constructor and it does not take any arguments. This constructor initializes each field to their initializer value in the original class or to the default value if they had none. The second constructor is a *full* constructor. It takes one initialization argument for each field, and is private as it is only used internally to create instances.

## 4.1 Convert Mutators into Factory Methods

Since the fields are `final`, methods can not assign to them. Immutator converts mutator methods into factory methods that create and return new objects with updated state.

We call a method a *mutator* if it (i) assigns to a field in the transitive state of a target class instance, or (ii) invokes a method that is a mutator method.

**Convert direct mutators** Setters are a common type of mutator in object-oriented programs. Lines 19–21 on the right-hand side of Fig. 1 show the transformation of `setRadius` to a factory method. Immutator changes (i) the return type to the type of the target class, and (ii) the method body to construct and return a new object, created using the full constructor. The constructor argument that is assigned to the `r` field is set to the right-hand-side of the assignment expression. The arguments for the other fields (e.g., `c`) are copied from the current object. Thus, the factory method returns a new object where the `r` field has the new value, while the other fields remain unchanged.

However, not all mutators are simple setters. Some contain multiple statements, while others mutate fields indirectly by calling other mutators. `moveBy`, on line 34–38, demonstrates both of these traits. It contains two statements, and it mutates `c` indirectly by calling `moveTo`.

The right-hand side shows how Immutator transforms `moveBy` into a factory method. It introduces a new local reference, called `_this`, to act as a placeholder for Java's built-in `this` reference. After `_this` is defined at the first mutation, Immutator replaces every explicit and implicit `this` with `_this`.

Furthermore, for every statement that calls a mutator, Immutator assigns the return value of the method (which is now a factory method) back to `_this`. Thus, the rest of the method sees and operates on the object constructed by the factory method. Finally, the `_this` reference is returned.

An interesting property of this technique is that it shifts the mutations from the target object to the `_this` reference. That is, instead of mutating the object pointed to by `this`, it mutates the state of `_this`. Ideally, Immutator would reassign back to `this`, but in Java the built-in `this` reference can not be assigned to. Therefore, Immutator replaces it with the mutable place-holder `_this`.

**Convert transitive mutators** Consider the `moveTo(int, int)` method on line 27–32. Although this method never assigns to the `c` field, it still mutates `c`'s transitive state through the `setLocation` method. Immutator notices that the method `setLocation` does not belong to the target class, but to `java.awt.Point` in the GUI library. Therefore, Immutator can not rewrite `setLocation` into a factory method.

As before, Immutator creates the `_this` reference, and returns it at the end of the method. Furthermore, Immutator clones `c`, so that the mutation does not affect the original object referenced by `this`. The cloned `c` is passed as an argument to the new `Circle`, which is assigned to `_this`. Since `_this.c` now refers to a clone of the original `this.c`, we can allow the mutation through `setLocation`.

## 4.2 Clone the Entering and Escaping state

Another way the transitive state of the target object can be mutated is if client code gets hold of a reference to an object in its internal state, and then mutates it outside of the target class. This can happen in two ways: (i) through objects that are entering the target class (e.g., `Point p` on line 24), or (ii) through objects that are escaping the target class (e.g., `c` on line 41).

An object *enters the target class* if it is visible from client code, and is assigned to a field in the transitive state of the target class. For example, the client code could call `moveTo(Point)` and then mutate the point through the retained reference.

We define a *target class escape* as an escape from any of its methods, including constructors. An escape from a method means that an object that is transitively reachable through a field of the target class is visible to the client code after the method returns. For example, on line 41, the object pointed to by `c` escapes through the return statement, and can then be mutated by client code. Escapes can also occur through parameters and static fields

If an object enters or escapes then current or future client code may perform any operations on it, and Immutator must conservatively assume that it will be mutated.

Immutator handles entering and escaping objects by inserting a call to the `clone` method to perform a deep copy of the object in question, as seen on the right-hand side of line 24 and 41. However, if the entering or escaping object is itself immutable, Immutator does not clone it. The current implementation considers the following classes to be immutable: the target class, `String`, Java primitive wrapper classes (e.g., Integer), and classes annotated with `@Immutable`.

When Immutator needs to use a `clone` method that does not exist, it generates a `clone` stub and reports this to the user, who must implement the stub.

Immutator avoids excessive cloning. For example, it could have inserted a `clone` call in the private constructor on line 11, but this would have caused unnecessary cloning. Instead, Immutator calls `clone` sparsely, at the location where objects enter or escape, or where the target class is transitively mutated (e.g., on line 30).

Moreover, Immutator ensures some structural sharing [11], by not adding calls to clone objects that enter from another instance of the same class. For example, when the transformed `setRadius` method is called, a new instance of `ImmutableCircle` is created (line 20 on the right-hand side). However, only the `r` field is mutated, while the `c` field (the center) remains the same. Since the old circle will not mutate the center, and since the center is not visible from the outside, the new circle does not have to clone it. The result is that the two circles share a part of their state.

# 5. PROGRAM ANALYSIS

In the previous section we discussed the transformations to make an existing class immutable. In order to perform these transformations Immutator first analyzes the source code to establish preconditions and to collect information for the transformation phase.

Immutator does not perform a whole-program analysis, but only analyses the target class and methods invoked from it. Thus, the analysis is fast and can be used interactively.

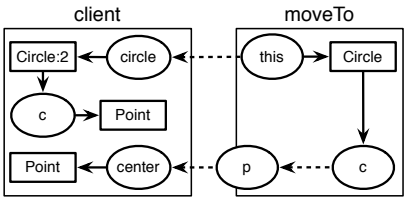
At the heart of Immutator are two analyses. The first detects mutating methods so that these can be converted to factory methods. The second detects objects that enter or escape the target class so that they can be cloned. Both analyses work on a representation generated from byte code, and can therefore analyze third-party library code.

## 5.1 Analysis Data Structures

Immutator creates several data structures that are necessary for the program analyses. It constructs both of these data structures using the WALA analysis library [23] as a starting point.

The first data structure is a *call graph* (CG) starting from every non-private method of the target class. The call graph is used to find mutators as well as entering/escaping objects. For each node in the callgraph Immutator also constructs a *control flow graph* (CFG) that is used later to find transitive mutations and to build a points-to graph.

In addition to the control-flow structures, Immutator builds a *points-to graph* (PTG). Points-to analysis establishes which pointers (or *references* in Java terminology) point to which *storage locations*. We model the heap storage locations as object allocation sites.




```
1 public void client() {
2   Circle circle = new Circle();
3   Point center = new Point(5, 5);
4   circle.moveTo(center);
5 }

6 public class Circle {
7   // ...
8   public void moveTo(Point p) {
9     this.c = p;
10  }
11 }
```

**Figure 2: An example points-to graph**

An example of the points-to graphs that Immutator create is illustrated using a simple client program in Fig. 2. The graph contains two types of nodes: references, depicted graphically as ellipses, and heap-allocated objects depicted as rectangles. The explicit formal arguments of a method are placed on the border of its bounding box. Directed edges connect references to the objects they point to. For example, the object created on line 2 is represented by the rectangle `Circle:2`, and the reference it is assigned to on the same line is represented by the `circle` ellipse. This object has a field `c`, which is constructed in the field initializer of class `Circle`. References are connected to their objects by directed edges. The points-to graph only captures relations between references and objects, and does not include scalar primitives.

Notice that the assignment on line 9 creates an alias between the references `c` and `p`. This is represented in the points-to graph as a dashed arrow, and is called a *deferred edge*. A deferred edge means that `c` can point to any objects that `p` can point to. We also use deferred edges to represent the relations between formal and actual arguments since Java is a pass-by-value language where actuals are copied into the formals.

Immutator constructs this points-to graph using an inclusion-based (Andersen-style) points-to analysis. The analysis is partly *flow-sensitive* with respect to local variables as it is computed from an SSA representation of the source code. It is also *context-insensitive* since it does not take the calling context into account.

Note that Immutator constructs additional nodes that do not exist in the program when they are needed to complete a method summary. One such example is the `Circle` allocation site and its `c` field in the `moveTo` method. When Immutator creates the summary for `moveTo`, the `this` reference is not connected to any allocation sites. Therefore, Immutator constructs additional object and field nodes in order to add the deferred edge that represents the assignment of `p` to `c`.

## 5.2 Detecting Transitive Mutators

The goal of this analysis is to find the methods that are mutating the transitive state of the target object, either directly or indirectly by calling another mutator method.

Fig. 3 shows the pseudocode of the algorithm for detecting mutator methods. The algorithm takes as input the set $M$ of

```
Input: M ← Set of Methods in CG,
       MTC ← Methods in Target Class,
       PTG ← Points-to Graph
Output: MUT // Set of mutator methods

  // Step 1: Find the transitive state of the target class
  TARG ← ∪_{m∈MTC}(transitiveClosure(this, PTG))

  // Step 2: Find transitive mutators
  for each m in M do
    for each fieldAssignments <o.f = expr> do
      if o can reach TARG through deferred edges in
      PTG then
        MUT ← MUT ∪ m

  // Step 3: Find indirect mutators
  for each m in M, in reverse topological order do
    for each m' in callees(m) do
      if m' ∈ MUT then
        MUT ← MUT ∪ m
```

**Figure 3: Detecting transitive and indirect mutators**

methods in the call graph, the set $MTC$ of methods declared in the target class, and the points-to graph presented in Section 5.1. The output of the algorithm is a set $MUT$ of mutator methods.

In Step 1, the algorithm finds the objects and fields that represent the transitive state of the target class. To do so, the algorithm computes the transitive closure of the `this` references of the target class, i.e, all nodes in the points-to graph reachable from `this`. These nodes, called $TARG$, are the set union of all nodes reachable from the `this` reference in target class methods.

In Step 2, the algorithm finds all transitive mutating methods. These include mutators inside and outside (e.g., `setLocation()`, called on line 30) the target class. The algorithm visits every field assignment instruction in all the target class methods, as well as methods invoked from the target class. For each assignment it checks whether the left-hand side of the assignment is a reference node that may point to one of the objects in the transitive state of the target class. If it can, this means that the instruction assigns to the transitive state of the target class, and the algorithm marks the method as a direct mutator.

In Step 3, the algorithm propagates mutation summaries from direct mutators backwards through the call graph. If method $m$ calls $m'$ and $m'$ is a mutator, then $m$ becomes a mutator too. To do this, the analysis visits, in reverse topological order (post-order), the methods in the call graph and merges the mutation summaries of the callees with the summaries of the callers.

## 5.3 Detecting Escaping and Entering Objects

The goal of this analysis is to find mutable objects that enter or escape the target class. These objects can be mutated by a client, thus mutating the transitive state of the target class. Therefore, the analysis finds and clones them.

The algorithm detects entering/escaping objects that are mutable *and* assigned/fetched to/from the transitive state of the target class.

Fig. 4 shows the pseudocode of the algorithm for detecting entering or escaping objects. The algorithm takes as input

```
Input: API ← Set of non-private methods in Target Class
       MTC ← Methods in Target Class,
       PTG ← Points-to Graph,
Output: ESC // Set of Escaping Objects
        ENT // Set of Entering Objects

  // Step 1: Find the transitive state of the target class
  TARG ← ∪_{m∈MTC}(transitiveClosure(this, PTG))

  // Step 2: Find the transitive closure of the outside nodes
  OUT ← ∪_{m∈API}(transitiveClosure(actuals, PTG)
                ∪ transitiveClosure(returns, PTG)
                ∪ transitiveClosure(statics, PTG))

  // Step 3: Find the escaping objects
  for each deferred edge e ∈ PTG do
    if (e.source ∈ OUT) && (e.sink ∈ TARG) then
      ESC ← ESC ∪ e.sink

  // Step 4: Find the entering objects
  for each deferred edge e ∈ PTG do
    if (e.source ∈ TARG) && (e.sink ∈ OUT) then
      ENT ← ENT ∪ e.source
```

**Figure 4: Detecting entering and escaping objects**

the points-to graph presented in Section 5.1. The output of the algorithm are two sets: $ENT$ containing objects that enter the target class, and $ESC$ containing objects that escape the target class.

In Step 1, the algorithm finds the nodes that form the transitive state of the target class. The transitive state, denoted by the $TARG$ set, is the transitive closure of the *this* reference of every method in the target class.

In Step 2, the algorithm finds the nodes that are outside of the target class, but that interface with it. Since these are the nodes through which client code interacts with the target class, they are also the nodes that objects can enter or escape through. We call these nodes the *boundary nodes*, as they are at the boundary of the target class.

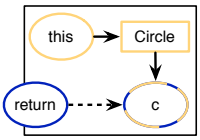
The boundary nodes are:

- actual arguments passed to non-private (API) methods
- references returned from non-private methods
- static reference fields.

The algorithm computes the transitive closure of boundary nodes, and labels the resulting set $OUT$.

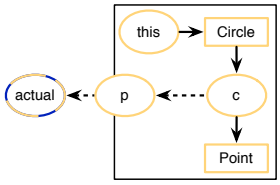
In Step 3, the algorithm finds the escaping objects. Escaping objects are the objects in the transitive state of the target class that can be *seen from methods outside the target class*. To find these objects, the algorithm visits all the deferred edges that start in $OUT$ and end in $TARG$. We are only interested in the edges that end in $TARG$, because we only care about escaping objects in the transitive state of the target class. For such edges, the algorithm adds the sink target node to the $ESC$ set.

Fig. 5(a) shows an example of an escaping object. It shows the points-to graph for the `getLocation` method, with an additional node representing the return statement. We color the transitive state of the target class (which is the transitive closure of `this`) with orange. We then color the outside nodes with blue. In this example, the only boundary node is




```
public Point getLocation() {
  return c;
}
```

(a) Example of an object escaping from `getLocation`




```
public void moveTo(Point p) {
  this.c = p;
}
```

(b) Example of an object entering through `moveTo`

**Figure 5: Detecting escaping and entering objects**

the return node, and its transitive closure includes `c`. Notice that `c` is colored with both blue and orange. This means `c` escapes because it can be seen from the outside (it is blue), and it is part of the transitive state of the class (it is orange).

In Step 4, the algorithm finds the entering objects. Entering objects are objects that are visible outside the target class methods, and that can be *seen from the target class*. To find these objects, the algorithm visits all the deferred edges that start in $TARG$ and end in $OUT$. We only visit edges that start in $TARG$, because we only care about the entering objects that are assigned to a field in the transitive state of the target class. For such edges, the algorithm adds the sink outside node to the $ENT$ set.

Fig. 5(b) shows an example of an entering object. It shows the points-to graph for the `moveTo` method. As before, the transitive state of the target class is colored orange, and the transitive closure of the boundary nodes (i.e., the actual argument) are blue. The actual parameter is a part of the transitive state of the target class (it is orange), and it can be seen from the outside (it is blue). Therefore, objects may enter through it.

For pedagogical reasons, we chose simple examples to illustrate escaping and entering objects. In the codes illustrated in Fig. 5(a) and 5(b), it is very easy to spot the entering/escaping objects. However, in many cases they are more difficult to find, especially if objects enter or escape through containers, or escape through parameters. Section 7 shows an example of a state object escaping through an iterator container. The open-source developer overlooked this escaping object, but Immutator correctly finds it.

# 6. DISCUSSION

There are cases when the programmer wants only partial immutability. For example, the programmer wants some fields to be excluded from the immutable state of the class (e.g., a `Logger` field), or some fields to be shallowly immutable. Or the programmer does not want to clone the entering/escaping objects (e.g., for performance reasons), but rather to document contracts. These are trivial extensions to Immutator and require no additional analysis.

Currently, Immutator handles most of the complexities of an object-oriented language like Java: arrays, aliases, polymorphic methods, and generics. It models arrays as an allocation site with just one field, which represents all the array elements. Although this abstraction does not allow Immutator to distinguish between array elements, it allows Immutator to detect objects that enter or escape through arrays. Immutator disambiguates polymorphic method calls by computing the dynamic type of the receiver object using the results of the points-to analysis described in Section 5.1. Immutator also preserves the generic types during the rewriting.

**Limitations** Since Immutator analyzes bytecode, it correctly handles calls to third-party libraries. However, if the program invokes native code, Immutator can not analyze it. Also, like any practical refactoring tool, Immutator does not handle uses of dynamic class loaders or reflection.

**Future work** We plan to solve the *usage problem*, i.e., updating the client code to use the transformed class in an immutable fashion.

Additionally, we will relax some of the constrains imposed by the current preconditions, to allow Immutator to transform more classes. For example, we could completely eliminate the requirement that the target class has no superclass/subclass (P1/P2), by allowing Immutator to transform a whole class inheritance hierarchy at once. Similarly, we could eliminate the requirement that mutators have a void return type (P3). Immutator could, for example, return a `Pair` object which encapsulates both the old return type, and the newly created object. Immutator would then have to change the callers of such methods to fetch the appropriate fields.

# 7. EVALUATION

To evaluate the usefulness of Immutator we answer the following research questions:

Q1: How applicable is Immutator?

Q2: Is Immutator safer than manual transformations?

Q3: Does it make the programmer more productive?

All these questions address the higher level question "Is Immutator useful?" from different angles. Applicability measures how many classes in real-world programs can be directly transformed, i.e., they meet the preconditions. Correctness ensures that the runtime behavior is not modified by the transformation. Productivity measures whether automation saves programmer time.

## 7.1 Methodology

We use a combination of three empirical methods, one controlled experiment and two case studies, that complement each other. The experiment allows us to quantify the programmer time and programmer errors, while the case studies give more confidence that the proposed algorithm and experiment findings generalize to real-world situations.

**Case Study #1 (CS1)** We ran Immutator on *all* classes in 3 open-source projects, a total of 346 concrete classes. Table 1 shows the projects that we used: Jutil Coal 0.3, jpaul 2.5.1 and Apache Commons Collections 3.2.1.

We do not suggest that every class in a project should be immutable. That is not for a tool to decide. Rather, we evaluate how well the transformation works over all classes

| proj. | SLOC | tests | classes | analyzed methods | edits/ class[2] | time/ class | passed preconditons | | | | failed preconditons | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | classes | mutator | enter | escape | classes | P1 | P2 | P3 | P4 |
| **jutil** | 4,605 | 70 | 70 | 3,397 | 43 | 2.09 s | 32 | 39 | 12 | 19 | 34 | 3 | 16 | 10 | 24 |
| **jpaul** | 5,661 | 42 | 54 | 2,471 | 33 | 2.16 s | 21 | 25 | 11 | 2 | 26 | 4 | 11 | 9 | 9 |
| **apache** | 26,323 | 13,009 | 222 | 12,857 | 50 | 2.44 s | 57 | 29 | 16 | 13 | 156 | 24 | 90 | 64 | 63 |
| **Total** | 36,589 | 13,122 | 346 | 18,725 | 45 | 2.33 s | 110 | 93 | 39 | 34 | 216 | 31 | 117 | 83 | 96 |

**Table 1: Results of applying Immutator to 3 open-source projects**

without imposing a selection criteria that could limit the generalization of the findings.

**Case Study #2 (CS2)** We also conducted case studies of how open-source programmers implement immutability. To find existing immutable classes in real-world projects we used two code search engines: krugle (`www.krugle.org`), and Google (`www.google.com/codesearch`). We searched for Java classes whose name contains the word 'Immutable' and classes whose documentation contained the word 'Immutable'. These are classes that are likely to be immutable, and the documentation of these classes confirmed that the developers intended them to be immutable. We also searched for classes implementing an `Immutable` interface, a convention used in some open-source projects. In cases when we found errors in their immutable classes, we contacted the developers to ask for clarification.

**Controlled Experiment** We asked 6 experienced programmers (with an average of 7 years of Java programming) to manually transform for immutability 8 classes from the JHotDraw 5.3 framework. JHotDraw is an open-source 2D graphics framework for structured drawing editors.

We gave each programmer a 1-hour tutorial on making classes immutable, and then we asked them to transform one or two JHotDraw classes and report the time. We used classes from the `Figure` class hierarchy that made sense to become immutable. Since the `Figure` classes are part of a deep class inheritance hierarchy, we told the participants to treat the target class as if it was the only class in the hierarchy, i.e., to change only the target class. No programmer got a class larger than 400 LOC. We also used Immutator to transform the same classes (we relaxed the first two preconditions), and we compared the results against a golden-standard.

To answer the applicability question, we wrote a statistics tool that applied the transformation to all classes in each project from CS1. For classes that did not pass all preconditions, the tool collected the failed preconditions. Since we ran Immutator in automatic mode, it only applied the transformation to classes that passed all preconditions. In interactive mode, Immutator could have transformed more classes, after the programmer addressed failed preconditions.

To answer the correctness question, we ran extensive test suites before and after all transformations from CS1. We only used projects that had extensive tests to help us confirm that the transformation did not break the systems. We also carefully inspected a few classes that we chose randomly.

To be able to run existing test suites, we wrote a tool that generates a mutable adapter between the immutable classes and the tests. The adapter has the same interface as the original class, but contains a reference to an instance of the immutable class. When a test calls a mutator, the adapter invokes the corresponding factory method of the immutable instance, and assigns the returned object to the reference. Our generated adapters were not adequate for 9% of the case study classes, due to not supporting static instance fields. Additionally, due to exceptions raised by our current implementation, we failed to analyze 20 of the classes in CS1. These were excluded from the reported data.

Furthermore, to compare correctness of manual versus tool-assisted transformation, we carefully analyzed the immutable classes that were produced manually in the second case study (CS2) and in the controlled experiment.

To answer the productivity question, we used Immutator to transform all the classes in Table 1 that met the preconditions. For each class, we report the number of methods that Immutator analyzed, as well as the number of source changes. We further broke this down into the total number of lines that had edits, the number of mutators that had to be converted to factory methods, and the number of entering or escaping objects that had to be cloned. We also report the time Immutator spent analyzing and transforming the code. For the controlled experiment, we asked each programmer to report the time spent to analyze and transform a class.

## 7.2 Results

To be useful, Immutator must be applicable, correct, and must increase programmer productivity.

### 7.2.1 Applicability

Table 1 shows that 33.74% of the classes in CS1 meet the preconditions without requiring any modification from the programmer. Out of the classes that failed preconditions, most are due to superclasses containing mutable state (P2), entering/escaping objects (P4), and mutators with non-void return values (P3).

However, keep in mind that a programer would not select all classes, but rather the ones that provide benefit. We hypothesize that such classes are more likely to meet the preconditions. Even in cases when classes do not meet all preconditions, Immutator enables the programmer to identify issues with the push of a button.

### 7.2.2 Correctness

For each project in CS1, we ran the full test suite before and after the transformations. The transformations did not cause any new failures.

Table 2 shows that even expert programmers make errors when creating immutable classes. The last set of three columns show how many entering or escaping objects the open-source programmers forgot to clone, and how many mutating methods they still left in the immutable class.

We confirmed with the open-source developers that our findings indicate genuine immutability errors in their code, and that developers meant those classes to be deeply immutable. Most agreed that their implementation choice was an incorrect design decision or was made for the sake of performance. Furthermore, the JDigraph developers took our patch and fixed the errors.

Table 3 shows the data for the controlled experiment. Pro-

[2]Does not include the adapter class

| project | immutable class | programmer errors | | |
|---|---|---|---|---|
| | | mutator | enter | escape |
| JDigraph | ImmutableBag | - | 1 | - |
| | FastNodeDigraph | - | 2 | - |
| | HashDigraph | - | 2 | - |
| | ArrayGrid2D | - | 2 | - |
| | MapGrid2D | - | 2 | - |
| WALA | ImmutableByteArray | - | 1 | - |
| | ImmutableStack | - | 2 | 3 |
| j.u.c.[3] | ImmutableEntry | - | 2 | 2 |
| Guava | ImmutableEntry | - | - | 2 |
| peaberry | ImmutableAttribute | - | - | 1 |
| Spring | ImmutableFlow-AttributeMapper | 2 | 2 | - |

**Table 2: Immutability errors in open-source projects**

grammers made errors similar with the ones in CS2. However, the density of errors was higher: 6.37 errors/class. The manual inspection of the immutable classes generated by our prototype implementation revealed 4 bugs. None of these were inherent to the algorithm.

### 7.2.3 Productivity

Table 1 shows that Immutator saved the programmer from editing 45 lines of code per target class on average. More important, many of these changes are non-trivial: they require analyzing 57 methods in context to find transitive mutations, entering and escaping objects. In contrast, when using Immutator, the programmer only has to initiate the transformation. On average, Immutator analyzes and transforms a class in 2.33 seconds using a Macbook Pro 4.1 with a 2.4 GHz Core 2 Duo CPU. Compared to the time taken to manually transform a class in the controlled experiment, 27 minutes, this is an improvement of almost 700x.

## 8. RELATED WORK

**Specifying and checking immutability** There is a large body of work in the area of *specifying* or *checking* immutability [16, 22, 26].

Pechtchanski and Sarkar [16] present a framework for specifying immutability constraints along three dimensions: lifetime (e.g., the whole lifetime of an object, or only during a method call), reachability (e.g., shallow or deep immutability), and context. Immutator enforces deep immutability for the whole lifetime of an object, on all method contexts.

Tschantz and Ernst [22] present Javari, a type-system extension to Java for specifying *reference immutability*. Reference immutability means that an object can not be mutated through a particular reference, though the object could be mutated through other references. In contrast, *object immutability* specifies that an object can not be mutated through any reference, even if other instances of the same class can be. Zibin et al. [26] build upon the Javari work and present IGJ that allows both reference and object immutability to be specified. *Class immutability* specifies that no instance of an immutable class may be mutated. Reference immutability is more flexible, but weaker than object immutability, which in turn is weaker than class immutability. Immutator enforces class immutability.

[3]java.util.collections

| JHotDraw class | SLOC | time [min] | programmer errors | | |
|---|---|---|---|---|---|
| | | | mutator | escape | enter |
| EllipseFigure | 104 | 17 | 1 | - | 5 |
| ArrowTip | 145 | 15 | - | - | - |
| ColorEntry | 97 | 16 | - | - | 1 |
| ImageFigure | 154 | 20 | 2 | - | 4 |
| LineConnection | 344 | 53 | 2 | 1 | 2 |
| FigureAttributes | 204 | 24 | 1 | 1 | 1 |
| TextFigure | 381 | 45 | 7 | 2 | 6 |
| PertFigure | 311 | 30 | 10 | - | 5 |
| Total | 1740 | 220 | 23 | 4 | 24 |

**Table 3: Results of the controlled experiment**

These systems are very useful to document the intended usage and to detect violations of the immutability constraints. But they leave to the programmer the tedious task of removing the mutable access. In contrast, Immutator performs the tedious task of getting rid of mutable access, by converting mutators into factory method, and cloning the state that would otherwise escape.

**Supporting program analyses** Components of our program analyses have previously been published: detecting side-effect free methods [1, 18–20] and escape analysis [4, 24]. Our analyses detect side effects and escapes *only* on state that is reachable from the target class.

Side-effect analysis [1, 18–20] uses inter-procedural alias analysis and dataflow propagation algorithms to compute the side effects of functions. There are two major differences between these algorithms and Immutator's analysis for detecting mutators. First, the search scope is different. Our algorithm detects side-effects to variables that are part of the transitive state of the target class, whereas previous work determines all side-effects (including side effects to method arguments that do not belong to the transitive state). Consider the method `drawFrame` from `TextFigure` in JHotDraw:

```
public void drawFrame(Graphics g) {
  g.setFont(fFont);
  g.setColor((Color)getAttribute("TextColor"));
  g.drawString(fText, ...);
}
```

The previous algorithms would determine that `drawFrame` is a mutator method, because it has side effects on the graphics device argument, `g`.

However, if Immutator transforms `TextFigure` then `drawFrame` will not mutate the transitive state of the target class, thus eliminating the need to clone the graphics device.

Second, our algorithm distinguishes between (i) methods in the target class that directly or indirectly assign to the fields of the target class and (ii) methods outside the target class (potentially in libraries) that do not assign to target class' fields, but mutate these fields transitively. Immutator converts the former mutators into factory methods, and rewrites the calls to the latter methods into calls dispatched to a copy of `this` (e.g., see the `_this` receiver in Fig. 1, lines 28–29). This enables Immutator to correctly transform code that invokes library methods.

Escape analysis [4, 24] determines if an object escapes the current context. So far, the primary applications of this analysis has been to determine whether (i) an object allocated inside a function does not escape and thus can be allocated on the stack, and (ii) an object is only accessed

by a single thread, thus any synchronizations on that object can be removed. There are three major differences between these algorithms and Immutator's escape analysis. First, our algorithm detects escaped objects that belong to the transitive state of the target class. Second, our algorithm is designed to be used in an interactive environment. Thus, it does not perform an expensive whole program analysis, but only analyzes the boundary methods of the target class. Third, in addition to escaping objects, our algorithm also detects entering objects.

**Refactoring** The earliest refactoring research focused on achieving behavior-preservation through the use of pre- and post-conditions [15] and program dependence graphs [10]. Traditionally, refactoring tools have been used to improve the design of sequential programs. The more recent work has expanded the area with new usages. We have used refactoring [5, 6] to retrofit parallelism into sequential applications via concurrent libraries. In the same spirit, Wloka et al. [25] present a refactoring for replacing global state with thread local state. Schäfer et al. [21] present Relocker, a refactoring tool that lets programmers replace usages of Java built-in locks with more flexible locks. Our transformations for class immutability makes code easier to reason about and enables parallelism by prohibiting changes to shared state.

# 9. CONCLUSIONS

Programmers use immutability to simplify sequential, parallel, and distributed programming. Although some classes are designed from the beginning to be immutable, other classes are retrofitted with immutability. Transforming mutable to immutable classes is tedious and error-prone.

Our tool, Immutator, automates the analysis and transformations required to make a class immutable. Experiments and case studies of manual transformations, as well as running Immutator on 346 open-source classes, show that Immutator is useful. It is applicable in more than 33% of the studied classes. It is safer than manual transformations which introduced between 2 and 6 errors/class. It can save the programmer significant work (analyzing 57 methods and editing 45 lines) and time (27 minutes) per transformed class.

## Acknowledgments

This research was partially funded by Intel and Microsoft through the UPCRC Center at Illinois, and partially supported by NSF grant 0833128. The authors would like to thank Vikram Adve, John Brant, Nick Chen, Ralph Johnson, Darko Marinov, Edgar Pek, Cosmin Radoi, Manu Sridharan, and anonymous reviewers for providing helpful feedback. Danny thanks Monika Dig, his greatest supporter.

# 10. REFERENCES

[1] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL*, pages 29–41, 1979.

[2] D. Bäumer, D. Riehle, W. Siberski, C. Lilienthal, D. Megert, K.-H. Sylla, and H. Züllighoven. Want value objects in java? Technical report, 1998.

[3] J. Bloch. *Effective Java: Programming Language Guide*. Addison-Wesley, 2001.

[4] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *OOPSLA*, pages 1–19, 1999.

[5] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *ICSE*, pages 397–407, 2009.

[6] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson. Relooper: refactoring for loop parallelism in Java. In *OOPSLA*, pages 793–794, 2009.

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

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[10] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM TOSEM*, 2(3):228–269, 1993.

[11] R. Hickey. The clojure programming language. In *DLS*, 2008.

[12] Java SE 6 API Specification. `http://java.sun.com/javase/6/docs/api`.

[13] D. Lea. *Concurrent Programming In Java*. Addison-Wesley, second edition, 2000.

[14] D. Marinov and R. O'Callahan. Object equality profiling. In *OOPSLA*, pages 313–325, 2003.

[15] W. F. Opdyke and R. E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA*, pages 145–160, 1990.

[16] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *Java Grande*, pages 202–211, 2002.

[17] D. Riehle. Value object. In *PLOP*, 2006.

[18] A. Rountev. Precise identification of side-effect-free methods in Java. In *ICSM*, pages 82–91, 2004.

[19] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM TOPLAS*, 23(2):105–186, 2001.

[20] A. Salcianu and M. C. Rinard. Purity and side effect analysis for java programs. In *VMCAI*, pages 199–215, 2005.

[21] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Refactoring java programs for flexible locking. *To appear in ICSE*, 2011.

[22] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, 2005.

[23] T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/wiki/index.php`.

[24] J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.

[25] J. Wloka, M. Sridharan, and F. Tip. Refactoring for Reentrancy. In *FSE*, pages 173–182, 2009.

[26] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, and M. D. Ernst. Object and Reference Immutability using Java Generics. In *FSE*, pages 75–84, 2007.