Three Approaches to Object Evolution

Tal Cohen* Joseph (Yossi) Gil†
Google Israel Engineering Center‡

ABSTRACT

Dynamic object reclassification allows changes to the type of an object at runtime. This paper makes the case for *object evolution*, a restriction of general reclassification by which an object may gain, but never lose properties. We argue that evolution is an expressive and useful language construct and can be implemented efficiently. Further, the monotonicity property of evolution promotes static type-safety better than general reclassification. We describe three concrete variants of evolution, relying on *inheritance*, *mixins* and *shakeins*, and explain how any combination of these can be integrated into a concrete programming language. We chart the language design space, mention our implementation, and introduce the notion of *evolvers*, a critical mechanism for maintaining class invariants in the course of reclassification.

1. INTRODUCTION

A frog may turn into a prince, if kissed. The modeling of such a phenomenon in the object-oriented world is known as (dynamic) *object reclassification*. As indicated by the literature, starting at least as early as 1993 [26], and as we shall reiterate here, the need for reclassification arises frequently in the software world, and cases such as the frog and prince example are not rarities. Some languages provide built-in support for reclassification. For example, SMALLTALK offers the "`becomes:`" method [18, p. 246], and in PYTHON, an object can be reclassified by assigning a new value to its `__class__` attribute. Yet, these mechanisms are notoriously unsafe and difficult to use, which probably explains why strongly-typed languages, like JAVA, include no such support.

State of the art research on object reclassification (see e.g., [11–13, 15–17, 25]) battles with the challenge of extending JAVA with a type-safe alternative to `becomes`. In this paper, we are interested in the tradeoff between expressive power and type-safety offered by a particular kind of reclassification, what we call *object evolution* (*OE*), by which dynamic changes to an object's class are *monotonic*—an object may gain, but never lose, capabilities. Once an object evolves, it cannot retract its steps and be reclassified into its previous class.

Evolution is not as general as reclassification, and may not allow changes as drastic as a frog turning into a prince. Our main interest is not so much with the theoretical foundation of object evolution, which previously received attention in the literature, but rather in the practical issues raised by the introduction of evolution into strongly-typed languages like JAVA.

We argue that there are many applications of monotonic evolution in practical systems. Most examples used in previous work about reclassification are monotonic: A suvery of the motivational examples of numerous papers about reclassification [6, 11–13, 17, 25], found a total of ten distinct examples in such diverse domains as banking, GUI development, games, and more. Of these, only three examples are in fact non-monotonic; the other seven could all be implemented using object evolution.[1]

The monotonicity property makes it easier to maintain static type safety with object evolution than in general object reclassification. Note that the monotonicity property may make evolution irreversible. This restriction is ameliorated by separating the notion of class from that of type, and with the help of shakeins [8] we find that object evolution can support repeated state changes, and even undo changes, under certain limitations.

We shall also see that OE requires less changes to the host language and collects a reduced performance toll, mostly because all descends in the inheritance hierarchy are necessarily monotonic.

The contributions of this paper include:

1. *The Case for Object Evolution.* We argue that object evolution is in line with object-oriented thinking and accepted design paradigms. For example, the STATE design pattern is naturally expressed with evolution. Object evolution is also thread-safe and integrates well with other useful programming techniques, such as lazy data structures.
2. *Concrete Language Extension.* We discovered an interesting problem of proper initialization in the course of reclassification: the object must maintain the state of existing fields, which may have changed after their initialization, yet its newly acquired fields must also be properly initialized. The class invariants of the new class must likewise be satisfied.
   We present *evolvers* as a complementary mechanism to constructors, containing the additional initialization code that separates an object of one class from an object of another. Like constructors, evolvers can accept parameters, indicating that an object cannot be evolved into a new class without some additional required information. Conversely, we also show that in many cases, *default evolvers* can be automati-

PPPJ '09, August 27–28, 2009, Calgary, Alberta, Canada.

---

[*] Corresponding author, `talcohen@google.com`

[†] Research supported in part by the IBM faculty award.

[‡] Research done in part while the authors were at The Technion.

[1] For details see http://ssdl-wiki.cs.technion.ac.il/wiki/index.php/Reclass_Examples . For a general survey of related work, see Sec. 6.

cally derived from the constructors of a class.

3. *Chart of the Language Design Space.* This paper presents three flavors of the object evolution mechanism, tagged *I-Evolution*, *M-Evolution* and *S-Evolution*, relying on inheritance, mixins and shakeins, respectively. The flavors are *independent*, meaning that a language designer can choose to implement any of the seven possible combinations, ranging from choosing a single approach to integrating all three.
4. *Analysis of Runtime Failures.* Just as an object construction operation can fail (e.g., when the constructor throws an exception), so can object evolution. We study and compare the relative merits of the three approaches by the kinds of runtime failures they may generate.
5. *Implementation Strategies and Prototype.* Finally, we turn to dealing with implementation. We briefly outline several alternatives, each appropriate for different usage scenarios, and present a prototype implementation as a JAVA extension.

1.1 Three Approaches to Object Evolution

We present three theoretical approaches to OE, each with its unique power of expression and underlying metaphor. These approaches are not mutually exclusive; all three, or any subset thereof, can co-exist in the same programming language.

The first approach, *I-Evolution*, is based on standard inheritance. Here, an object can evolve into any subclass of its own class. This change is necessarily monotonic, since a subclass may only extend its base class. Evolution is expressed using the syntax $v \to C(\cdots)$, meaning the object referenced by variable v is evolved (using the $\to$ operator) to an instance of class C. The parenthesis will often be empty, i.e., $v \to C()$; however, the evolution process may accept parameters.

The evolution target C must be a subclass of v's type. The set of possible reclassification targets is therefore defined by the inheritance tree. The similarity of this tree to taxonomy trees used in biology to describe evolution inspired the process's name.

The second approach, *M-Evolution*, is based on mixin inheritance [5]. With M-Evolution an object can only evolve into a subclass defined using *mixins*. Recall that given a class C and a mixin M, the application of M to C, denoted $M\left\langle C\right\rangle$, is a subclass of C. Class $M\left\langle C\right\rangle$ is an ordinary class, and can therefore serve as the target of an I-Evolution operation. With M-Evolution, however, the evolution target is selected—and possibly generated—at runtime, based on the object's actual type at the time of evolution. The M-Evolution operation $v \to M\left\langle v\right\rangle(\cdots)$ selects $M\left\langle V\right\rangle$ as its target, where V is v's runtime type. Thus, an M-Evolution can be thought of as an application of a mixin to an instance rather than to a class. Because a mixin can only extend its operand, M-Evolution is also guaranteed to be monotonic.

Finally, *S-Evolution* is limited to shakein inheritance. *Shakeins* [8] are a programming construct that, like mixins, generates a new class from a given class parameter. Unlike mixins, a shakein does not generate a new *type*. Given a shakein S and a class C, the shakein application $S\left\langle C\right\rangle$ represents a new class but not a new type; it is an *implementation class* [9]. (See Sec. 4.3 below for a more detailed overview of shakeins.)

S-Evolution can be thought of as an application of a shakein to an instance rather than to a class. Such an application, by definition, does not change the object's type (in contrast to its class); in particular, the shaked object cannot understand any new messages. S-Evolution is therefore *trivially* monotonic, and resembles instance-specific behavior facilities in SMALLTALK [3]. However, unlike instance-specific behavior, the behavior itself is described in an organized manner (in the shakein's definition) rather than relying on ad-hoc changes to an object's message handlers.

A unique feature of S-Evolution is that it can be temporary, i.e., in certain circumstances, the object may later re-evolve into a different shakein-based class, *undoing* (or "de-evolving") the effect of the first shakein. Whereas shakeins can be used as enhanced aspects [7,8], S-Evolution introduces the possibility of using shakeins as dynamic aspects [20,23,24].

Outline. Sec. 2 makes the case for object evolution using real-world motivating examples. Sec. 3 presents the concept of object evolution in greater detail, and also explains where a simple evolution operation might fail. Sec. 4 provides details about each of the three kinds of object evolution. An overview of possible implementation strategies, and a discussion of our own prototype implementation, are discussed in Sec. 5. Sec. 6 discusses previous work and outlines some directions for further research.

2. THE CASE FOR OBJECT EVOLUTION

As early as 1993, Taivalsaari [26] argued that design often needs objects that change their behavior at runtime. (Taivalsaari's own proposed solution, *modes*, can be nicely implemented using S-Evolution.) This need for reclassification motivated much subsequent research, including [6, 9–13, 15–17, 25].

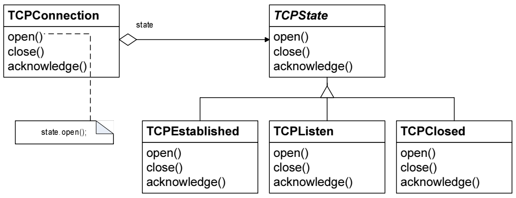
An important demonstration of this need is provided by the programming language ***e***, manufactured and sold by Cadence, and used widely in the hardware verification industry. What is called *when-inheritance* [19] in ***e*** is in fact a mechanism, similar to S-Evolution, by which an object reclassifies itself.

This section emphasizes the case for object evolution showing several cases where object evolution can be used to improve upon program design. Sec. 2.1 explains how the STATE design pattern maps naturally to evolution. In Sec. 2.2 we show how program design of lazy data structures can benefit from evolution. Two examples are used there for concreteness: The DOM representation of HTML data structures, and the evolution of the Abstract Syntax Tree in the different stages of the compilation process.

2.1 Implementing the STATE Design Pattern

In their presentation of the STATE design pattern, the Gang of Four use a TCP connection class as an example [14, p.305]. Fig. 2.1 shows the class structure realizing this example.

Figure 2.1 The state-changing `TCPConnection` class (from [14]). The object can change its behavior, seemingly changing its type, by replacing the internal reference (`state`) to a different implementations of `TCPState`.



The connection object is required to respond differently to messages (such as `open`) based on its current state, which can be either of "established", "listen", and "closed". Rather than represent the state as an **`int`** data member (or an **`enum`**), the design pattern suggests representation using a data member s of a dedicated *state* type S, to which all requests are delegated.

The abstract state class (here, `TCPState`) has a concrete subclass for each possible state. Each such subclass responds differently to messages; for example, the `close` message changes the object's state (to "closed") if it is in either the "established" or "listen" states, but throws an exception if it is already in the "closed" state.

To change its state, the object simply replaces the instance to which the state variable s refers.

The intent of the pattern is to "*[allow] an object to alter its behavior when its internal state changes.* ***The object will appear to change its class***" [14, p.305; emphasis added]. But, as this description suggests, the same effect can be better achieved by literally allowing the object to change its class at runtime.

Fig. 2.2 outlines the code for an implementation of the same `TCPConnection` class, which relies on object evolution. Here, the state-changing operations use object evolution (lines 3, 4 and 10) to change the object's state by advancing its class. Since evolution is transparent to aliasing, any reference to the connection will now use the newly-classified object, and thus any method invocation will be affected by the new state.

Figure 2.2 Implementing `TCPConnection` and its state-changes using object evolution. Operations that change the object's state do so by evolving **`this`** to a different subclass (lines 3, 4 and 10).

```
1  public class TCPConnection {
2    // This class represents the initial state, ''listen''.
3    public void open() { ...; this→TCP_Established(); }
4    public void close() { this→TCP_Closed(); }
5    public void acknowledge() { ... }
6  }

8  class TCP_Established extends TCPConnection {
9    public void open() { /* ignore */ }
10   public void close() { ...; this→TCP_Closed(); }
11   public void acknowledge() { ... }
12 }

14 class TCP_Closed extends TCP_Established {
15   public void open() { throw new IllegalStateEx(); }
16   public void close() { throw new IllegalStateEx(); }
17   public void acknowledge() { ... }
18 }
```

Several benefits of the approach should be immediately apparent:

- *Fewer classes.* Whereas the STATE design pattern solved this particular problem using five classes (a wrapper, an abstract state class, and three concrete state classes), the evolution-based solution requires only three (one class per state).
- *No code duplication.* In the STATE pattern, the state class, `TCPState`, copies the interface of the wrapper class. Such fragile code duplication is not needed with object evolution.

Additionally, for more complex scenarios:

- *No need to transfer state data.* Because the state is always represented by the same object, there is no need to copy data from the old state object to the new one with each state transition.

The only limitation of this solution is that connection object cannot be reused (see Sec. 4.1). A better solution, using shakeins and state-groups, is presented in Sec. 4.3.3.

2.2 Lazy Data Structures

Since object evolution moves objects down the inheritance tree, it can be used to evolve instances of general, top-level classes into more specific sub-classes. Such changes can be useful as more information about the object is obtained (see example in Sec. 2.2.1 below), or for lazy evaluation of data structures. In the latter case, nodes in the data structure are first represented as general "node" objects, to be replaced by specific nodes on a per-need basis.

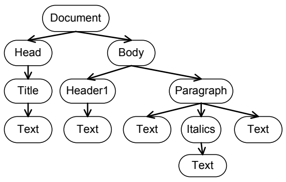
Consider, for example, the hierarchical in-memory representation of HTML files (or XML files, etc.), and in particular, the common DOM (Document Object Module) tree representation.

Fig. 2.3(a) shows a simple HTML file; in Fig. 2.3(b) we see its DOM representation. We see that every opening tag is represented as a tree node, while the HTML content that occurs from this tag to its matching closing tag is represented as the subtree rooted at this node. A sequence of plain text, with no tags, is represented as a leaf node of type `Text`; e.g., the fragment `<i>nice</i>` (line 7) is represented as an `Italics` node with a `Text` subnode.

Figure 2.3 A sample HTML document and its Document Object Model (DOM) tree. Nodes in the tree are instances of classes shown in the hierarchy of Fig. 2.4.

```
1 <html>
2  <head>
3   <title>Welcome!</title>
4  </head>
5  <body>
6   <h1>This is my page.</h1>
7   <p>How <i>nice</i>!</p>
8  </body>
9 </html>
```

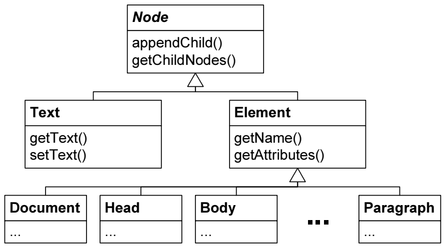
(a) An HTML document.



(b) The document's DOM tree.

Fig. 2.4 is a UML class diagram for the classes used in Fig. 2.3(b). We see that abstract class `Node` is at the hierarchy's root, that `Text` is a final class, and that different classes offer different services.

Fig. 2.4. Classes used for representing DOM trees.



Since programs often end up using only part of the tree, a common optimization technique is lazy evaluation, where one object represents an entire subtree, to be expanded on a per-need basis.

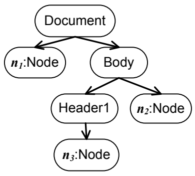
Here, lazy evaluation means that class `Node` is not abstract. Its instances denote yet-unparsed HTML fragments.

Fig. 2.5 shows a possible intermediate state of Fig. 2.3(b)'s tree. The left-hand child of the root node, marked n_1 in the figure, represents the subtree contained inside the `<head>...</head>` tag pair. Should the program code delve into this subtree, this node must be expanded, with new nodes created to represent its children.

In a lazy DOM parser which does not use object-evolution, the expansion step must either (a) replace the node object n_1 with a specific node (i.e., create a new object and discard the old one), or else, (b) change the state of this object, so that it now represents a specific node.

The first solution requires that the parser must not allow references to node n_1 to leak, since the existing object must be replaced with a new one, and the old one must cease to exist. This complicates the implementation, and in particular requires an expansion of the subtree whenever n_1 is requested by any client, even if that client will not eventually access any child of n_1. (Things are further complicated in other data structures, such as directed graphs, where there are multiple references to the object.)

Fig. 2.5. A possible stage in the lazy creation of the tree from Fig. 2.3(b). Nodes n_1, n_2 and n_3 were not yet expanded.



The second solution implies that the `Node` class must have two operational states, pre- and post-expansion. After the expansion, it must be able to act as any of its subclasses; in this example, n_1

must be able to act as an instance of class `Head` after its expansion, whereas n_2 must be able to act as an instance of `Paragraph` and n_3 as `Text`. The STATE pattern can be used here: maintain a field of type `Node` in each un-expanded node (e.g., n_1), and, upon expansion, assign a new instance of a specific subclass (e.g., `Head`) to this field. Any message received by the node will now be delegated to the more specific `Node`-typed field. An implementation of a lazy DOM tree with the STATE design pattern is inefficient, since it requires delegation. Such an implementation is also cumbersome, complicating both the design and the implementation of classes: `Node`'s API must include the union of all methods found in all subclasses, and some of these methods might fail at runtime (e.g., the method `getText` from class `Text` must be processed in the expanded n_3, but rejected by the expanded n_1 and n_2).

Now consider the OE-based solution. Whenever the subtree represented by object n_1 must be expanded, we can *evolve* this object from its current class (`Node`) to any of its subclasses, and in particular `Head`. The evolution operation $n_1 \rightarrow$`Head(···)` affects the object itself, so all references to it are immediately affected; no need to update each reference. The object's new class is a subclass of its old, so that the object can still accept and process any message it could previously accept; and it can now also accept and process messages added by the interface of its specific new class.

This solution requires no delegation, and no new object is introduced into the system. There is also no need for an awkward inflation of the interface of the superclass `Node`, and type safety is maintained; e.g., if a `Node` is evolved into a `Head`, it has no `getText` method, and any attempt to use such a method will fail at compile-time.

2.2.1 Representing Knowledge Refinement

An important special case of lazy data structures are systems in which knowledge increases over time, and the increase in knowledge allows us to replace a general class with a specific subclass. As a concrete example, consider the classes used to represent the abstract syntax tree (AST) data structure in a compiler implementation. A top-level class, `MethodInvocation`, can be used to represent the general notion of an invocation expression, whereas its subclasses represent specific invocation types, e.g., `StaticMethodInvocation` for static method calls, `DynamicMethodInvocation` for ordinary calls, `InterfaceMethodInvocation`, etc. Each of these subclasses is a specific, *refined* version of the superclass.

In many compiler designs, the parser generates an AST from the source code; the back-end then processes this tree. Often, the parser does not have the knowledge required for classifying a given AST node at its most refined representation level; e.g., given the source fragment "`x.m()`" in a JAVA program, the parser will generate a `MethodInvocation` node. The back-end will then replace this node with a more specific node, such as `InterfaceMethodInvocation`, based on data obtained from the symbol table regarding `x`'s type and the declaration of method `m` in that type. The change is a *refinement* based on gathered knowledge.

Just as with lazy data structures, a refinement entails either (a) the creation of a new node object to replace the old one, or (b) representing all possible options in the top-level class (`MethodInvocation` in this example). As in the case of DOM tree nodes, the first option implies that the AST data structure must prevent the reference to the raw type from leaking; all references must be meticulously tracked, and replaced when the object is refined. The second option implies that the top-level class must contain knowledge about all possible refinement options. This contradicts modular design and complicates future expansions.

With object evolution, refinement is represented as the object sliding down the inheritance tree to a state that represents our new, refined knowledge about it. All references are immediately updated, while the program design remains completely modular.

3. OBJECT EVOLUTION

An object evolution operation replaces, at runtime, the type of an object with the type of a selected subclass. As the target type is always a subclass of the current type, the set of class members is either unchanged or enlarged, i.e., the change is monotonic. Since no member is removed by the operation, we have a guarantee that *any message understood by the object prior to the evolution operation is understood after the operation as well*, thereby ensuring type safety after the evolution occurred.

The action of object evolution is executed on a particular reference to the object, but it affects *the object itself.* All references to the object, including fields, local variables, etc. now reference the evolved object. Evolution is therefore *transparent to aliasing*.

Evolution is written using the syntax $v \rightarrow C(\cdots)$, meaning the object referenced by variable v is evolved (using the → operator) to an instance of class C. (The → operator can be written as "−>".) The parenthesis will often be empty, i.e., $v \rightarrow C()$; however, the evolution process may accept parameters, as described below.

For example, consider the lazy tree evaluation scenario discussed above, and in particular the class hierarchy presented in Fig. 2.4. Given the variable definition and initializations

```
Node n1 = new Node(···); Node alias1 = n1;
```

we can now evolve `n1` into any subclass of `Node`; e.g., `n1→Head()` will evolve the object referenced by both `n1` and `alias1` from an instance of class `Node` to an instance of its indirect subclass `Head`.

The evolution expression, `n1→Head()`, returns a value; this value is a reference to the same object (i.e., it equals `n1`), but its static type is the target class, `head`. We can thus store the result in a new variable that will allow us to access defined in class `Head`, or its superclass `Element`, as in:

```
Head h1 = n1→Head();
Attributes attr = h1.getAttributes();
```

Variables `h1` and `n1` now refer to the same object, so the test `h1 == n1` will yield true. However, their static type is different, so `getAttribute()` cannot be applied directly to `n1` (see Fig. 3.1).

Figure 3.1 An example of the effect of evolution on references to the object. The static type of references does not change; however, they can be downcasted to the new type.

```
Node n = new Node(···);
Node alias = n;

if (someCondition()) {
   n.getAttributes(); // Compile-time error
   ((Head)n).getAttributes(); // Run-time error
   Head h = n→Head();
   if (h == n) { ... } // Condition holds
   n.getAttributes(); // Compile-time error
   h.getAttributes(); // Will succeed
   ((Head)n).getAttributes(); // Will succeed
   ((Head)alias).getAttributes(); // Will succeed
}

((Head)n).getAttributes(); // Downcast might fail at runtime
```

3.1 Evolvers: Maintaining Class Invariants

The object evolution operation takes an instance of one class and mutates it into an instance of another. Yet simply adding new fields and methods is not sufficient. Consider an object v of type C_0 that undergoes an evolution process, $v \rightarrow C(\cdots)$. The object state, which initially satisfies the class invariants of C_0, must now satisfy C's invariants.[2]

Standard objects of class C go through an orderly construction process, which ensures that class invariants are satisfied. In particular, the constructor begins by invoking an inherited constructor (using the keyword **`super`** in JAVA[3]); after the inherited constructor returns, the invariants of the superclass C_0 are satisfied, and the rest of the constructor body must ensure that the additional invariants introduced in C are also satisfied. But object v had only gone through the C_0 construction process. We therefore conclude that object evolution must allow the mutating object to execute any required code in order to meet the invariants of its target class.

To this end, we define *evolvers*, which are constructor-like class members executed upon evolution. Syntactically, an evolver for class C is named $\rightarrow C$ (whereas a constructor is named C). For example, an evolver defined in class `Head` must be called `→Head`. Like constructors, evolvers have no return type. Also like constructors, evolvers can be overloaded; the parameters $(\cdots)$ passed to the evolution operation $v \rightarrow C(\cdots)$ dictate which evolver will be used.

Unlike constructors, evolvers do not begin by calling an inherited version. When the evolver begins its execution, the current object (**`this`**) is an instance of the class C, which does not yet satisfy its class invariants; it only satisfies the invariants of its superclass C_0. This is similar to the state of the object in the constructor, right after the call to `super(···)` is completed. It is therefore the evolver's role to initialize the newly acquired fields, possibly based on the values of the inherited fields, so that all invariants are satisfied.

For a concrete example, consider Fig. 3.2, showing an implementation of class `Head` which contains a field named `title`.

Figure 3.2 A class with an explicit evolver (line 8). This evolver will be used when an object is evolved from the superclass `Entity` to class `Head`, ensuring that the added fields are properly initialized.

```
1  class Head extends Entity {
2     private String title = null;

4     public Head() { // Constructor
5        super(); // Call superclass constructor (could also be implicit)
6        initTitle(); }

8     public →Head() { initTitle(); } // Evolver

10    private void initTitle() {
11       // Parse the node's content and set title accordingly
12       Node t = findSubElementByName("title");
13       if (t != null) title = t.getTextContent(); }

15    // ... rest of the class not shown
```

The implicit invariant of this class is that field `title` must be set by the **`<title>`** sub-element of **`<head>`**, or **`null`** if no **`<title>`** node exists. To preserve this invariant, both the constructor (lines 4–6) and the evolver (line 8) use the private method `initTitle` to initialize the field `title`. As we shall see, such code duplication can be avoided using default evolvers.

The body of an evolver could be different from that of the constructor. In particular, the constructor can make certain assumptions about the state of fields inherited from the superclass; it knows for certain that the superclass was only just constructed itself. An evolver, however, can execute long after the superclass instance was created, and the state of the inherited fields can vary from what it was after construction. The only valid assumption for the evolver is that the superclass fields maintain the superclass's class invariants.

Classes in JAVA that define no constructor obtain a default constructor, generated by the compiler; this constructor merely invokes `super()`. In a similar manner, classes that define no evolver obtain one or more *default evolvers*. For every constructor that begins with a parameter-less call to `super()`, (directly or, by a chain of `this(···)` calls, indirectly), a default evolver is generated. Each default evolver accepts the same parameters as the constructor that triggered its synthesis, and shares the same body, except the call to `super()`. The visibility level (**`private`**, **`public`**, etc.) is also shared.

Default evolvers makes it possible to remove line 8 (the evolver definition) from Fig. 3.2; an identical default evolver would be automatically generated. This also means that, in the same figure, the `title` initialization code could be inlined as part of the constructor itself, rather than presented as a private method.[4]

If no default evolvers can be generated (because all constructors call `super(···)` with one or more parameters), then the class must define explicit evolvers if it is to serve as an evolution target.

3.1.1 Evolution Steps

In the DOM tree example, class `Head` extends class `Element`, which extends `Node`, which in turn extends `Object` (Fig. 2.4). Therefore, whenever a new instance of `Head` is created, the constructor first invokes the constructor of `Element`, which first invokes that of `Node`, etc. We have that the construction process always begins at the topmost level (`Object`) and progresses down in the inheritance tree towards the actual type (e.g., `Head`), with each step initializing its own fields and ensuring that its own invariants are maintained.

When an object is evolved, some nonempty prefix of this initialization chain had already occurred (at the very least, the `Object` constructor was executed). The evolution process must now ensure that the remaining tail is executed. Therefore, given the inheritance chain $C_n \prec C_{n-1} \prec \ldots \prec C_1 \prec C_0$, when object v is evolved from class C_0 to class C_n, it is not only the evolver of C_n that executes; the evolver of every class residing between the two in the inheritance tree runs first: $\rightarrow C_1$, followed by $\rightarrow C_2$, etc. These are *implicit evolution steps*. Because v's position in the inheritance chain (its dynamic type) is known only at runtime, the required implicit evolution steps are also known only at runtime. Only the final, explicitly named evolver $\rightarrow C_n$ is guaranteed to take place when an object is successfully evolved to type C_n.

For example, when an instance of `Node` is evolved into an instance of `Head`, the evolver `→Element` runs first (an implicit step), followed by `→Head`. This completes the initialization chain for a proper instance of `Head`.

We have seen that the evolution step might accept parameters. When the statement $v \rightarrow C_n(p_1, \ldots, p_k)$ is executed (assuming v's current type is C_0), the parameters $p_1, \ldots, p_k$ are passed to the evolver $\rightarrow C_n$. For other evolvers in the chain between C_0 and C_n, a parameter-less evolver is used.

If an interim step in the evolution chain, $\rightarrow C_i$ for some $i \in \{1 \ldots n-1\}$, has no zero-parameters evolver, the parameter-requiring steps cannot be implicit, and v may not be directly evolved to C_n. It must first be evolved to the interim step C_i, passing parameter(s) to one of C_i's evolvers; only then can it be evolved to C_n. If there are multiple such parameter-requiring steps in the chain between C_0 and C_n, then multiple explicit steps must be used.

Consider for example the trio of classes defined in Fig. 3.3. Given the variable declaration and initialization `A v = new A(0)`, the evolution statement `v→C(2)` will fail to compile, since `v` must first be

---

[2] The invariants of a subclass are always additions to those of the superclass; see the *invariant inheritance rule*, [22, p.465]. Note that while JAVA classes do not have invariants specified explicitly in code, they almost always have implicit *conceptual* invariants, often made explicit in the documentation.

[3] Or the keyword **`this`**, which delegates to a different constructor in the same class. Still, at the end of the delegation chain there must reside a constructor that begins with a call to `super(···)`.

[4] In rare situations, where the cosntructor initialization sequence depends on work done by the inherited constructors in non-obvious ways, subtle bugs may ensue. This concern may be addressed by limiting default evolver generator to specifically-tagged (annotated) constructors.

evolved into an instance of `B` before it can become an instance of `C`, and this interim stage requires its own parameter. We must therefore use two explicit stages, as in `v→B(1)→C(2);`. (This isn't a special syntax; we're simply taking advantage of the evolution expression's return value.)

Figure 3.3 An inheritance chain where each step requires an additional construction/evolution argument. Because the evolution from `A` to `B` cannot be implicit, an object cannot be directly evolved from `A` to `C` (an interim step must be explicitly used).

```
class A {
  int a;
  public A(int a) { this.a = a; } }

class B extends A {
  int b;
  public B(int a,int b) { super(a); this.b=b; }
  public →B(int b) { this.b = b; } }

class C extends B {
  int c;
  public C(int a,int b,int c) { super(a,b); this.c=c; }
  public →C(int c) { this.c=c; } }
```

3.2 Evolution Failures

All three approaches presented above integrate with the static type system. Once an object has evolved, it assumes a new class, and it will never be the case that an object receives a message it cannot deal with.

However, in certain circumstances that cannot be statically determined, the evolution operation itself might fail. Such cases are called *evolution failures*.

Thus, with regard to type safety, object evolution can be likened to a downcast operation: The operation itself might fail, but once completed successfully, the reference or object can be safely accessed using its newly-assumed class.

Two possible failures are common to all approaches. The most trivial failure occurs when the reference to the object to be evolved happens to be **`null`** at runtime. The other common possible cause for failure is when the evolver throws an exception (just as a constructor may throw an exception, making **`new`** fail).[5]

Beyond these shared causes, each of the three approaches entails its own set of possible causes for failure.

In the I-Evolution operation $v \to C(\cdots)$, the evolution target C must be a subclass of v's type. Herein lies a risk of evolution failure, since while C can be verified to be a subclass of v's *static* type, we cannot verify in advance that it is also a subclass of v's *dynamic* type. For example, we may try to evolve an object of static type `Pet` to type `Dog`, but if the object's runtime type is `Cat` (a different subclass of `Pet`), this evolution attempt will fail.

The target of the M-Evolution operation $v \to M\langle v \rangle(\cdots)$ is $M\langle V \rangle$, where V is v's runtime type. The operation's target is therefore necessarily a subclass of v's dynamic type, avoiding the risk presented by I-Evolution operations. The risk is further reduced by defining the concept of *idempotent mixins*, i.e., mixins that can be repeatedly applied to a class with no adverse effect. However, M-Evolution can still fail if mixin M cannot be applied to V for one of two reasons: If V is a **`final`** class, or if the application results in accidental overriding [2] (i.e., a mixin which introduces a new method `m()` is applied to a class that happens to have a method `m()` of its own, which the mixin is not meant to override).

Finally, because it only offers trivial monotonicity, S-Evolution is the least susceptible to failure. Like M-Evolution, S-Evolution selects the target class based on the evolving object's dynamic type, thereby avoiding the risk faced by I-Evolution.

Unlike mixins, shakeins are immune from accidental overriding, because they can only override existing methods or introduce **`private`** ones. Thus, S-Evolution can only fail when a shakein is applied to a an object whose dynamic type is **`final`**.

4. THE THREE KINDS OF EVOLUTION

4.1 I-Evolution: Using the Inheritance Tree

The most straightforward of the three approaches, I-Evolution allows an object v of static type C to evolve into any subclass of C. If C is an **`interface`**, then v can be evolved into any class that implements it.

The examples presented so far were all based on I-Evolution. I-Evolution's main limitation lies with its simplicity: change must be *down a pre-determined path*, i.e., it can only propagate down the statically-defined inheritance tree. In the TCP connection example (Sec. 2.1), once a connection object reaches the closed state, it is in what we may metaphorically term "an evolutionary dead-end"; it can no longer change its state. To represent a fresh connection, a new `TCPConnection` object must be created. We shall later use S-Evolution to overcome this limitation in this case and others.

4.1.1 Evolution to Mixin-Generated Classes

The target of an I-Evolution operation can be any class; in particular, it can be a class generated using a mixin. As an example, consider the mixin `Blocked` (Fig. 4.1).[6]

Figure 4.1 A mixin for creating immutable `List` classes.

```
mixin Blocked {
   inherited public void add(Object o);
   inherited public void remove(int index);

   @Override public final void add(Object s) {
      throw new UnsupportedOperationException(); }

   @Override public final void remove(int index) {
      throw new UnsupportedOperationException(); }

   // ... etc. — List has many more methods to override...
}
```

This mixin can be applied to classes that implement JAVA's standard interface `List`. The result is a list that cannot be modified, since any attempt to add or remove objects will yield an exception.

Using OE, list objects can be evolved into blocked-list objects at any stage of their life. For example, the following code can be used:

```
List myList = new Vector();
myList.add(···); // add numerous data items
myList→Blocked<Vector>();
```

Here, applying the mixin to class `Vector` generates a new class that refuses to add or remove items. Once the evolution completes, no client that holds a reference to this list object will be able to alter its content. There are many uses to this capability, including security considerations and improved performance for defensive programming [4, #39] (since there is no need to create a copy of the list).[7]

4.2 M-Evolution: Better Use of Mixins

M-Evolution is a variant of object evolution, where the target of any evolution statement is the result of applying a mixin to the *runtime* type of an object. An M-Evolution statement for variable v

---

[5] It is possible to prevent evolution into some specific class by providing an evolver that unconditionally throws an exception. However, a simpler solution is declaring a **`private`** evolver.

[6] We use the syntax of JAM for defining mixins in our JAVA-like language; yet for consistency, the application of mixins is expressed using a generics-like syntax.

[7] It is for these security considerations that the methods in mixin `Blocked` were defined as **`final`**—to prevent the application of a reverse mixin, "`Unblock`".

uses the syntax $v \rightarrow M \langle v \rangle (\cdots)$, where M is a mixin. The operation selects $M \langle V \rangle$ as its target, where V is v's runtime type. If class $M \langle V \rangle$ did not previously exist, the evolution operation will cause it to be generated, at runtime. M-Evolution therefore avoids the "evolutionary dead-end" limitation of I-Evolution by dynamically extending the inheritance tree.

To understand the usefulness of the concept, consider `Blocked` (Fig. 4.1) again. While it can be used to generate a subclass of any class that implements `List`, it is hardly useful in a context where all we have is an instance whose static type is `List`, and its dynamic type unknown. This is a common case, e.g., with methods that accept a `List` reference as a parameter. Should such a method wish to evolve its parameter to an immutable object using `Blocked`, it can try to evolve it into `Blocked<ArrayList>`, `Blocked<Vector>`, or any of numerous other combinations—the code would look like this:

```
public void blockParam(List lst) {
   if (lst instanceof Vector)
      lst→Blocked<Vector>(); //Attempt I−Evolution
   else if (lst instanceof ArrayList)
      lst→Blocked<ArrayList>(); //Another I−Evolution attempt
   else //... etc.
   else throw new Exception("Unknown List impl.");
}
```

However, no branch in the code is guaranteed to succeed, since the total number of classes that implement `List` is unbounded. The solution is to apply a mixin to the runtime type of the object, using M-Evolution:

```
public void blockParam(List lst) { lst→Blocked<lst>(); }
```

Here, mixin `Blocked` accepts as a parameter not a type, but a *variable*; it generates a new class, at runtime, based on that variable's dynamic type. The resulting type of the variable after the evolution statement can be e.g., `Blocked<Stack>`, `Blocked<Vector>`, etc.

4.2.1 M-Evolution and Idempotent Mixins

A moment's reflection will reveal that the M-Evolution statement in the code above can never fail, except in certain rare scenarios (when the object's runtime type is a **`final`** class, or if accidental overloading ensues; see Sec. 3.2). In most cases evolution will succeed since no matter where in the inheritance tree does the variable's runtime class reside, it can evolve downwards. There is no dead-end to reach, as the inheritance tree can be expanded at runtime. In particular, even if the type is already the result of applying the `Blocked` mixin, it can further evolve; the type can change, e.g., from class `Blocked<Vector>` to class `Blocked<Blocked<Vector>>`. No complication is introduced by the repeated application of the mixin, since it is *idempotent*.

Our next example is based on the classic mixin `Undo` (Fig. 4.2).

Figure 4.2 A sample mixin, which adds an `undo` method to classes that have a `getText` and `setText` methods. (Based on [2, Fig.1]).

```
@Idempotent public mixin Undo {
  inherited public String getText();
  inherited public void setText(String s);

  private String lastText;

  @Override public void setText(String s) {
    lastText = getText(); super.setText(s); }

  public void undo() { setText(lastText); }
}
```

Mixin `Undo` can be applied to any class that features the two methods `getText` and `setText`, such as the standard-library class `JButton`. The ability to repeatedly apply `Undo` (generating, e.g., `Undo<Undo<JButton>>`) with no adverse effect is less obvious, since every such application alters the memory footprint of each instance (by adding a new `lastText` field). Also, every repeated application will add a new invocation to the chain of operations that implement `setText`. However, other than by means of performance measurement, external clients have no way to tell an instance of `Undo<Undo<JButton>>` from an instance of `Undo<JButton>`; the behavior remains identical. We therefore maintain that this mixin is also idempotent, and mark this in the source code using the `@Idempotent` annotation.

We say that a mixin is idempotent if:

1. It is annotated using `@Idempotent` (e.g., `Undo`), or
2. It meets both of the following criteria (e.g., `Blocked`):
   (a) All (if any) introduced members (fields or methods) are **`private`**.
   (b) Any method that it overrides is replaced rather than refined (i.e., the new method body does not call the inherited version using **`super`**).

Given an idempotent mixin M_I and arbitrary type T, the runtime system will always provide $M_I \langle T \rangle$ when $M_I \langle M_I \langle T \rangle \rangle$ is requested. This approach prevents the creation of unnecessarily long "threads" in the inheritance tree, that might result from the repeated application of a single idempotent mixin to the same object.

4.3 S-Evolution: Evolving with Shakeins

4.3.1 A Brief Overview of Shakeins

Shakeins [8] are a programming construct that, like mixins, generates a new class from a given class parameter. However, unlike a mixin, a shakein does not generate a new *type*. Given shakein S and class C, the shakein application $S \langle C \rangle$ represents a new class but not a new type.

Shakeins can thus be viewed as *type re-implementors*: A shakein can be used in object construction expressions, but one cannot define variables of type $S \langle C \rangle$. Since they share the same type, the set of externally-accessible members of $S \langle C \rangle$ is identical to that of C.

Shakeins can use *pointcut expressions* and *advice* [21] to selectively generate new implementations of methods in the original class. They can therefore be used much like aspects for addressing the problem of scattered and tangled code.

Figure 4.3 A shakein that generates a transactional implementation of a class.

```
 1 @Idempotent public shakein Transactional {
 2  pointcut publicMethod := public ?*(*);

 4  around: publicMethod {
 5    Transaction tx = Session.getTransaction();
 6    Object result;
 7    try {
 8      tx.begin();
 9      result = proceed; // Invoke the original impl. of the current method
10      tx.commit();
11    } catch (Exception e) { tx.rollback();  }

13    return result;
14  }
15 }
```

For example, applying the shakein `Transactional` (Fig. 4.3) to class C generates a new version of C, in which every public method is enveloped in a database transaction. If the original method invocation (line 9) is successful, the transaction is committed (line 10); otherwise, it is aborted (line 11).

Another example of a shakein is `ReadOnly`, from Fig. 4.4. When applied to any class, this shakein silently blocks all calls to setter methods—**`void`** methods with names that begin with `set`, followed by an uppercase letter—that accept a single parameter. Setters are matched by the pointcut expression in line 2. The **`around`** advice (line 4) then re-implements each setter as a no-action method.

Figure 4.4 The `ReadOnly` shakein blocks all setter methods.

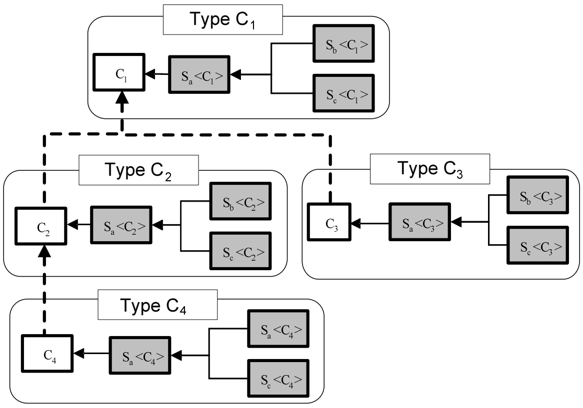
```
@Idempotent public shakein ReadOnly {
  pointcut setter := void set[A-Z]?*(_);

  around: setter { return; }
                    // Block silently; do not invoke original version
}
```

By writing `ReadOnly<JButton>`, we obtain a read-only version of the `JButton` class, where methods `setText`, `setIcon`, etc. will all be replaced. While a mixin could be used to reach the same effect, the mixin will necessarily be longer, explicitly overriding each setter method. Also, the mixin will be highly specific to the `JButton` class. To create a similar subclass of `JCheckBox`, a new mixin will be required, specific to that class; whereas with the `ReadOnly` shakein, we can simply write `ReadOnly<JCheckBox>`.

Thus shakeins, unlike mixins, are more flexible, since they are sensitive to their parameter. Unlike a mixin, however, a shakein may not introduce new members to its argument (except for **`private`** members). For example, a mixin like `Undo` (Fig. 4.2), which introduces the public method `undo`, cannot be created using a shakein.[8]

For all classes C_1 and C_2 so that $C_2 \prec C_1$ (i.e., the type of C_2 is a subtype of C_1), and for arbitrary shakeins S and S', we have that $S' \langle C_2 \rangle \prec S \langle C_1 \rangle$. For example, if class `LimitedAccount` is a subclass of `Account`, then the type of `Transactional<LimitedAccount>` is a subtype of `Transactional<Account>`'s type. Fig. 4.5 makes a graphical illustration of this fact. It depicts a simple base class hierarchy consisting of classes $C_1 \ldots C_4$, where $C_4 \prec C_2 \prec C_1$, and $C_3 \prec C_1$. There are also three shakeins, S_a, S_b and S_c, where shakeins S_b and S_c are implemented using S_a, and each of the shakeins is applied to each of the classes. We see that the type of class C_i ($i = 1, \ldots, 4$), is the same as its three re-implementations $S_a \langle C_i \rangle$, $S_b \langle C_i \rangle$ and $S_c \langle C_i \rangle$. This common type is denoted by a round-cornered box labeled "Type C_i". As shown in the figure, the subtyping relationship is not changed by re-implementations; e.g., the type of class $S_a \langle C_4 \rangle$ is a subtype of $S_b \langle C_2 \rangle$'s type.

Figure 4.5 A class hierarchy subjected to shakeins (from [8]). Each round-cornered box represents a single type; internal boxes represent different implementations of each type.



4.3.2 Shakeins and Object Evolution

S-Evolution is a variant of object evolution, where the target of any evolution statement is the result of applying a *shakein* to the *runtime* type of an object. An S-Evolution statement for variable v uses the syntax $v \rightarrow S \langle v \rangle (\cdots)$, where S is a shakein.

Much like M-Evolution, S-Evolution extends the inheritance tree as needed at runtime, and therefore cannot fail due to inheritance dead-ends. Also like M-Evolution, shakeins can be marked idempotent (as was the shakein `ReadOnly`), making their repeated application a "fail-safe" operation. And, because shakeins cannot introduce new non-**`private`** class members, they are not susceptible to failure by accidental overriding.

4.3.3 Shakein State-Groups

Shakeins and S-Evolution can substitute the STATE design pattern, since in this pattern, all state classes implement the same interface. For example, state classes `TCPListen`, `TCPEstablished`, and `TCPClosed` all implement the interface defined by the abstract class `TCPState` (Fig. 2.1); we have a set of classes that share the same type. Such sets can also be generated by applying different shakeins to the same base class; each round-cornered box labeled "Type C_i" in Fig. 4.5 presents an example.

We define a *state-group* of shakeins as a set of shakeins that share the `@StateGroup` annotation, with the same string parameter; different, independent state-groups can be created using different string parameters. For example, the three shakeins in Fig. 4.6 form the state-group `"Connection"`.

Figure 4.6 A shakeins state-group for generating the various state classes of `TCPConnection`. This provides a full replacement to the STATE pattern, because there is no limit to the number and type of state changes (*cf.* the limited I-Evolution version in Fig. 2.2).

```
 1 @StateGroup("Connection") public shakein Listen {
 2   public void open() { ...; this→Established<this>(); }
 3   public void close() { this→Closed<this>(); }
 4   public void acknowledge() { ... }
 5 }

 7 @StateGroup("Connection") public shakein Established {
 8   public void open() { /* ignore */ }
 9   public void close() { this→Closed<this>(); }
10   public void acknowledge() { ... }
11 }

13 @StateGroup("Connection") public shakein Closed {
14   public void open() { throw new IllegalStateEx(); }
15   public void close() { throw new IllegalStateEx(); }
16   public void acknowledge() { ... }
17 }
```

The compiler enforces the limitation that all shakeins in a given state-group must define the same set of **`private`** class members.[9] In the example, this requirement is met vacuously.

Shakeins in the same state-group are *mutually exclusive*, meaning that if C is a class, and shakeins S_1 and S_2 are in the same state-group, the application of S_1 to $S_2 \langle C \rangle$ yields $S_1 \langle C \rangle$, rather than $S_1 \langle S_2 \langle C \rangle \rangle$. Such an application is called a *state transition*.[10]

When applied to class `TCPConnection` from Fig. 2.2, the three shakeins from Fig. 4.6, generate the state subclasses. For example, `Established<TCPConnection>` is equivalent to class `TCPConnectionEstablished` (from Fig. 2.2), etc. These shakeins capture the increment between `TCPConnection` and each of its subclasses, but use S-Evolution statements (lines 2, 3 and 9).

This state-group can overcome the inability of the I-Evolution-based solution to retract its steps. Given a connection in the "closed" state, we can now change its state back to "listen" by applying the `Listen` shakein to its dynamic type. Doing so will change the

---

[8] The private members introduced by a shakin may include data fields. See [8] for a detailed discussion of the implications of repeatedly applying shakeins in such cases.

[9] Shakeins can never introduce non-**`private`** class members, since they may not change the base class's type.

[10] While state transition is not, strictly speaking, a move down the inheritance tree, it is still a form of object evolution, because conceptually the new state *could* be defined as a subclass (but not a subtype) of the old one. The fact that it is not a subclass is a means for avoiding needlessly long inheritance "threads". We use the term "transition" only for this special form of S-Evolution.

object's type from `Closed<TCPConnection>` to `Listen<TCPConnection>`. There is no limit on the number of times the state can be changed by re-applying the appropriate shakein; and these changes do not generate an inheritance tree of unbounded depth.

Transitions within a state-group are type safe, because the type of a shakein-applied object, $S_1 \langle C \rangle$, is *identical* to that of $S_2 \langle C \rangle$, and to that of C itself. Shakein $S_2 \langle C \rangle$ recognizes all messages that $S_1 \langle C \rangle$ recognized (and vice versa). The only fine point is the type of **`this`** *in methods overridden by the shakein application*. Such methods may call **`private`** methods, or access **`private`** data members, defined in S_1. Hence the requirement that all shakeins in a state-group include the exact same set of private members.

The trivial case of a state-group with only a single shakein (or mixin) is in fact equivalent to an idempotent shakein; given such a shakein S_I, applying it to $S_I \langle C \rangle$ yields $S_I \langle C \rangle$. However, state-groups with more than one member can only be defined for shakeins, and not for mixins, because the mixin type itself can be used to define variables. For example, had the three shakeins from Fig. 4.6 been created as a set of mixins, then some code could have defined a variable of these types, as in:

```
Established e = new Established<TCPConnection>();
```

Now, by the definition of evolution within state-groups, evolving the object to mark a closed connection would change its type to `Closed<TCPConnection>`; yet the variable `e` cannot refer to such an object. The problem is never encountered with shakeins, since no variables (or fields, etc.) of shakein types are possible.

4.3.4 Objects with Multiple States

Multiple shakeins can be applied to a single object. E.g., given state-group $\mathbf{S}$ with states S_1 and S_2, and state-group $\mathbf{R}$ with R_1 and R_2, one can create classes such as $S_1 \langle R_1 \langle C \rangle\rangle$, $R_2 \langle S_1 \langle C \rangle\rangle$, etc., depending on the shakeins used and the order of application.

What happens when we apply R_2 to an instance of $S_1 \langle R_1 \langle C \rangle\rangle$? There are four possible semantics for handling such cases:

1. *Error semantics.* Applying R_2 results in a runtime error.
2. *Accumulation semantics.* The result is $R_2 \langle S_1 \langle R_1 C \rangle\rangle$, i.e., the new shakein is added to the object and does not replace the old shakein from its group (no state transition takes place).
3. *Order-preserving semantics.* The result is $S_1 \langle R_2 \langle C \rangle\rangle$, i.e., the state-transition preserves the order in which shakeins from the different state-groups were originally applied.
4. *Re-ordering semantics.* The result is $R_2 \langle S_1 \langle C \rangle\rangle$, i.e., the state-transition re-orders the shakeins applied to the object.

Accumulation semantics implies that the application results in an object which is simultaneously in states R_2 and R_1; this situation is not desired since R_2 and R_1 belong to the same group, which usually implies that they are mutually exclusive. We therefore eliminate this option.

Assuming we're not interested in error semantics, *the principle of least surprise* dictates that re-ordering semantics is preferable. Having a previously-applied shakein take precedence over the most recently-applied one (as in the case of order-preserving semantics) can lead to awkward situations. E.g., if shakein R_2 is `ReadOnly` (Fig. 4.4), failure to take precedence implies that applying `ReadOnly` to an object might yield a *non*-read-only object. Conversely, it *does* make sense that applying some shakein S to a read-only object might make that object mutable, depending on the nature of S.

4.3.5 Shakeins as Dynamic Aspects

Previous work [7, 8] introduced shakeins as a more organized, parameterized alternative to aspects. As noted before, a shakein can apply advice (**`before`**, **`after`** or **`around`**) to methods of the inherited class, yielding a new implementation of the base class's type without changing the class itself. One example is the `Transactional` shakein presented above (Fig. 4.3). Another example is the shakein `Log` from Fig. 4.7, which can be used as a logging aspect.

Figure 4.7 A logging aspect (shakein) and it's canceling counterpart. These can be applied to objects alternatively, and so they act as a dynamic aspect.

```
@StateGroup("Log") public shakein Log[String filename] {
  pointcut publicMethod = public (*);

  before: publicMethod {
    FileOutputStream fos = new FileOutputStream(filename);
    // ... log the operation to the file ...
  }
}

@StateGroup("Log") public shakein NoLog { /* empty */ }
```

`Log` accepts a string parameter, which is the filename to which the log will be written. Logging objects can be created using statements like: `List verboseList = new Log["my.log"]<Vector>();`. Following this, any access to a public method of the `verboseList` object will be logged in the specified file.

With object evolution, we can dynamically apply this aspect to existing objects. E.g., a method that accepts some parameter `lst` of type `List` can issue the statement `lst→Log["system.log"]<lst>()` and evolve the list into a logging list.

The use of state-groups allows the dynamic aspect to be removed, given an empty shakein from the same group: `lst→NoLog<lst>()` (shakein `NoLog` is an empty shakein, defined in Fig. 4.7).

5. IMPLEMENTING OBJECT EVOLUTION

The key problem with implementing object evolution is the need to increase the memory footprint of an object already on the heap, as new data members are added. Possible solutions include:

1. *Using object handles.* Certain GC-supporting systems (e.g., GILGUL's VM [9]) store, in each reference variable, a pointer to a *forwarding pointer*, or *handle*, rather than a direct pointer. The handle itself is never moved, but it can be modified if the object is re-located in memory. This can be used to implement evolution—if the target class introduces new fields, the object can be moved to a newly-allocated address.
2. *Compacting evolution.* With this solution, every evolution operation forces a complete GC run, including memory compaction. Since compaction supports moving objects around memory, the evolving object can be assigned a new location with sufficient space for its new class.
3. *Old objects as proxies.* Here, we introduce a new field to `java.lang.Object`. This field, denoted `newRef`, is **`null`** by default; any other value indicates that the object had evolved, and v.`newRef` is a reference to v's new address. Any attempt to operate on an object with a non-**`null`** `newRef` is *forwarded*; i.e., the old object acts as a proxy to the new one.

Each solution represents different performance tradeoffs. For example, compacting evolution would be ideal if evolution operations are rare: each evolution is costly, but there is zero overhead to all other operations. Choosing the optimal solution requires empirical data about the frequency and type of evolution operations that are common in programs. Such data will also enable fine-tuning of solutions (e.g., for evolution operations that take place inside loops).

Our own implementation[11] uses proxy objects. It is based on a JAVA agent which patches classes at load-time; neither the JVM nor

---

[11] http://ssdl-wiki.cs.technion.ac.il/wiki/index.php/Object_Evolution

the library were modified. The code demonstrates a few possible optimizations for proxy-based implementations.

6. CONCLUSIONS AND RELATED WORK

This paper propounded the inclusion of object evolution mechanisms into mainstream programming languages, and in particular JAVA. We showed how OE integrates well with inheritance, and with the less popular mechanisms of mixins and shakeins. It is also thread safe.[12] Our language design decisions favored type-safety, but in respect of practical concerns, not with as much zeal as other approaches to reclassification. Specifically, our proposal does not provide a compile time guarantee that all evolution operations are successful. To our knowledge, this is the first attempt to reconcile in this manner the conflicting just purposes of type-safety, reclassification flexibility and practical concerns. In this section we compare our approach with previous work.

Monotonic Reclassification. The idea of improving the type safety of reclassification by making the changes monotonic dates back to Beck's *Scriptable Objects* [3]. Scriptable Objects are (trivially) monotonic since, like shakeins, they do not change the object's interface. But they are not as general as shakeins, in that a shakein is applicable to multiple classes.

Object extension [15,17] includes non-trivial monotonic changes, (*self-inflicted extension*). With no template serving as the object's new class, only the object's own methods can access any newly introduced method or data member (if we use a static type system).

Ghelli [15] suggested a calculus in which "incompatible changes" cannot occur, by letting the same object assume different *roles* in different contexts. His work is done in the context of *Fibonacci* [1], a DB language. Roles are more natural to OODBs than to OO programming languages, since the notion of a dynamic type of an object may conflict with the "role" imposed on it.

Systems that do offer non-monotonic, type-safe object reclassification restrict the choice of objects that can be reclassified, and the range of reclassification target classes. In Serrano's *wide classes* system [25], objects can only be *widened* to instances of wide subclasses of their current class. In other words, what we call evolution is restricted to a pre-designated set of subclasses. Conversely, only wide objects can be *shrunk*. The ability to shrink objects implies that the system cannot be used in statically-typed languages.

The Work on FICKLE. The FICKLE language (and its newer versions) are more general than evolution, since they allow non-monotonic reclassification. However, reclassification in FICKLE is restricted in the sense that only instances of designated *root* classes can be reclassified, and the destination class must be a pre-designated *state* subclass of the root class. To maintain static typing, the FICKLE type system must track the *effect* of each method, namely the list of classes whose instances may be reclassified (directly or indirectly) by the method; the programmer is requested to annotate each method with the list of reclassification changes it may make. Additionally, state classes cannot be used to define variables, although this restriction was somewhat eased in FICKLE$_{\text{II}}$.

The set of classes composed of a root class and its state subclasses in FICKLE can be compared to a class and the set of re-implementations that can be derived from it using a shakein state group. In both cases, objects can be reclassified freely inside the group. Also, to maintain type safety, in both cases there are restrictions on using the non-root classes as types. In FICKLE$_{\text{II}}$, these restrictions are relaxed and pertain to fields only, whereas in our system they are absolute. Most importantly, FICKLE is strongly-typed, i.e., the reclassification operation cannot fail. However, shakeins integrate into the existing type system and do not require tracking each method's effect or marking the root class as such. In FICKLE, any subclass (direct or indirect) of a root class must be a state class, whereas with shakeins, regular classes, that can have regular subclasses, are used as roots. Thus, shakeins can be used to create a state group from existing classes within a pre-defined hierarchy, such as the standard library.

In FICKLE$_3$, any object may change its class to any other class. The implementation, in which any object may be reclassified into any class, should deal with the need to wrap all objects.

Object Replacement. Related to reclassification and evolution but different is *object replacement*, as offered by the GILGUL language. GILGUL [9] extends JAVA in allowing a global change of all references to a certain object, redirecting them to another object. Static type safety is maintained by the requirement that the type of the new object must be the same as that of the type of the original object, or a subtype thereof. Thus, replacement must deal with the same kind of runtime failures that might occur with I-Evolution.

7. REFERENCES

[1] A. Albano *et al.* Fibonacci: A programming language for object databases. *The VLDB J.*, 4(3):403–444, 1995.

[2] D. Ancona *et al.* Jam–designing a Java extension with mixins. *ACM Trans. on Prog. Lang. Syst.*, 25(5), 2003.

[3] K. Beck. Instance specific behavior: how and why. *The Smalltalk Report*, 2(6):13–15, March-April 1993.

[4] J. Bloch. *Effective Java, 2nd Edition*. Addison-Wesley, 2008.

[5] G. Bracha and W. R. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP'90.*

[6] C. Chambers. Predictable classes. In *ECOOP'93.*

[7] T. Cohen and J. Gil. AspectJ2EE = AOP+J2EE. In *ECOOP'04.*

[8] T. Cohen and J. Gil. Shakeins: Non-intrusive aspects for middleware frameworks. In *Trans. AOSD*, Nov. 2006.

[9] P. Costanza. Dynamic replacement of active objects in the Gilgul programming language. In *IFIP/ACM Working Conf.*

[10] F. Damiani *et al.* Re-classification and multi-threading: Fickle$_{\text{MT}}$. In *SAC'04.*

[11] F. Damiani *et al.* Refined effects for unanticipated object re-classification: Fickle$_3$ (extended abstract). In *ICTCS'03.*

[12] S. Drossopoulou *et al.* Fickle: Dynamic object re-classification. In *ECOOP'01.*

[13] S. Drossopoulou *et al.* More dynamic object re-classification: Fickle$_{\text{II}}$. *ACM TOPLAS*, March 2002.

[14] E. Gamma *et al.* *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[15] G. Ghelli. Foundations for extensible objects with roles. *Inf. Comput.*, 175(1):50–75, 2002.

[16] G. Ghelli and D. Palmerini. Foundations of extensible objects with roles (extended abstract). In *FOOL'6, 1999.*

[17] P. D. Gianantonio *et al.* A lambda calculus of objects with self-inflicted extension. In *OOPSLA'98.*

[18] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[19] Y. Hollander, M. Morley, and A. Noy. The e language: A fresh separation of concerns. In *TOOLS'01 Europe*.

[20] P. Kenens *el al.* An AOP case with static and dynamic aspects. In *ECOOP'98 Workshop Reader*.

[21] G. Kiczales *et al.* An overview of AspectJ. In *ECOOP'01.*

[22] B. Meyer. *Object Oriented Software Construction, 2nd ed.* Prentice-Hall, 1997.

[23] R. Pawlak *et al.* JAC: An aspect-based distributed dynamic framework. *Soft. - Pract. and Exper.,* 34(12), 2004.

[24] A. Popovici, T. R. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD'02*.

[25] M. Serrano. Wide classes. In *ECOOP'99.*

[26] A. Taivalsaari. Object-oriented programming with modes. *J. of OO Prog.*, 6(3):25–32, June 1993.

---

[12] See http://ssdl-wiki.cs.technion.ac.il/wiki/index.php/OE_Threads for details.