The Decorator Pattern Revisited: Expressing Mixins in Languages with Single Inheritance

Extended Abstract for MPOOL 2001

Radha Jagadeesan Konstantin Läufer {radha,laufer}@cs.luc.edu
Mathematical and Computer Sciences, Loyola University Chicago, Chicago, IL 60626, USA

Introduction

Mixins, or *abstract subclasses*, allow the definition of a subclass to be parameterized with respect to its superclass [2]. Such parameterization allows reusing the same extension multiple times to create different subclasses from different superclasses. Notably, neither Java nor Smalltalk support mixins directly: these and other *single-inheritance* languages require every subclass to have a single, fixed superclass.

Existing research on mixins has focused on two approaches: (1) extending single-inheritance languages with a mixin mechanism (for example, [1, 3]); (2) expressing mixins in languages that already provide a more general mechanism, such as multiple inheritance (C++, CLOS, Eiffel) or genericity that allows type parameters as superclasses (C++, [5]).

In contrast, this paper explores ways to express the mixin paradigm within standard object-oriented languages with single inheritance, such as Java or Smalltalk, which lack genericity and multiple inheritance. We argue that mixin-style reuse can be achieved in such languages through a technique based on the Decorator design pattern [4]. We first describe shortcomings of the existing Decorator pattern. We then present our technique and assess its expressiveness; like the Decorator pattern, our technique requires only one level of subtyping (abstract data types with multiple implementations), but no code inheritance. We conclude by evaluating the run-time performance of our technique.

What's Missing from the Decorator Pattern?

The intent of the *Decorator* (*Wrapper*) pattern is to "provide a flexible alternative to subclassing for extending functionality". In particular, the Decorator pattern applies "when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. . . . " [4].

In the Decorator pattern, each reusable extension is implemented as a wrapper class that conforms to the interface of the original component and maintains a reference to the original component. The wrapper class implements those methods that are changed as part of the extension along with any needed instance variables, and forwards all unchanged methods to the original reference.

The need to reuse and compose extensions is precisely the problem addressed by mixins. However, the Decorator pattern solves the problem only in a limited form because it does not let extensions override methods. Even after applying an extension, methods that invoke other methods of the original component fail to invoke the new versions defined in the extension.

To illustrate this problem, we consider a simple collection interface, a stack implementation, and an extension.

```
interface Collection {
  void insert(Object item);
  Object remove();
  void clear();
  boolean isEmpty();
}

class Stack implements Collection {
  public Object remove() { ... }
  public void clear() {
    while (! this.isEmpty()) { this.remove(); }
  }
  // ...
}

class SizeOf implements Collection {
  private Collection zuper; // original component
  private int count;
  public SizeOf(Collection z) { zuper = z; }
  public Object remove() {
    Object result = zuper.remove(item);
    if (result != null) { count --; }
    return result;
  }
  public void clear() { zuper.clear(); }
  // ...
```

In the following client code, `s.clear` is forwarded to `Stack.clear`, which invokes `Stack.remove` instead of `SizeOf.remove`. Therefore, the size of the collection is not updated properly, and 2 is printed instead of 0.

```
SizeOf s = new SizeOf(new Stack());
s.insert("Hello"); s.insert("World"); s.clear();
System.out.println(s.size()); // prints 2
```

Solution

In the Decorator pattern, each extension represents a subclass, and the original component reference `zuper` corresponds to the embedded superclass instance. This models the part of inheritance that corresponds to composition, but not the part that corresponds to dynamic method binding and method overriding.

Our solution properly models method overriding via the following protocol:

- In addition to the original component reference, each extension maintains a reference `thiz` to the current outermost extension, which plays the role of the `this` reference that captures the actual class of the receiver object.
- Any overridable method invocation in the original component or in any extension is made through the reference `thiz`.

The protocol requires no changes on the client side.

Language support required to implement this protocol is the same as for the Decorator pattern: one level of subtyping (abstract data types with multiple implementations), but no code inheritance. In Java, this corresponds to a two-tier type hierarchy involving only interfaces and `final` classes (without further subclasses). It is also possible to implement the protocol in languages with simple record types, such as C, by making the receiver explicit.

The following example illustrates our approach. We define an interface for extensible collections and, to avoid code duplication, an abstract support class for base implementations and extensions.

```
interface Extensible extends Collection {
  Extensible getThis();
  void setThis(Extensible thiz);
  Extensible getSuper();
  void setSuper(Extensible zuper);
}

abstract class Mixin implements Extensible {
  private Extensible thiz = this; // cur receiver
  private Extensible zuper; // original component
  public Mixin(Extensible z) { setSuper(z); }
  public Extensible getThis() { return thiz; }
  public void setThis(Extensible thiz) {
    this.thiz = thiz; // propagate up the chain
    if (zuper != null) { zuper.setThis(thiz); }
  }
  public Extensible getSuper() { return zuper; }
  public void setSuper(Extensible z) { zuper = z; }
  public void insert(Object item) {
    getSuper().insert(item);
  }
  // remaining forwarding methods to getSuper()
}
```

The `Stack` class is still implemented as before, except that it now extends `Mixin` and replaces each use of `this` by `thiz` (via the accessor `getThis()`) to enable method overriding by extensions applied later.

```
class Stack extends Mixin {
  public Stack() { super(null); }
  public void clear() {
    while (! getThis().isEmpty()) {
      getThis().remove();
    }
  }
  // ...
}
```

The `SizeOf` extension is also implemented as before, except that it now extends `Mixin` for proper maintenance of the `thiz` reference and uses the accessor `getSuper()`.

```
class SizeOf extends Mixin {
  private int count;
  public SizeOf(Extensible z) { super(z); }
  public Object remove() {
    Object result = getSuper().remove(item);
    if (result != null) { count --; }
    return result;
  }
  public void clear() { getSuper().clear(); }
  // ...
}
```

We have used this technique successfully in the context of language interpreters. In the Interpreter pattern, programs are represented as trees (using the Composite pattern), and interpreters are represented as operations on the trees (using the Visitor pattern) [4]. By expressing instrumentation of program evaluation (such as dynamic checkers, debuggers, etc.) as reusable extensions, different kinds of instrumentation can be combined flexibly and dynamically.

Legacy base classes can be reused unchanged through adapters that replace each method in the original component by one that invokes the method through an invocation of `getThis()`. For lack of space, the details of this mechanism will be provided in the full paper.

Performance

There is a valid concern about the run-time performance of what is essentially a re-implementation of the built-in inheritance mechanism at the object level. We have conducted tests intended to compare the performance of both mechanisms in the worst case: a linear class hierarchy in which each overriding method invokes the method it overrides, upward along the entire inheritance chain. Specific results are given in the appendix.

The performance of our implementation varies widely compared with that of built-in inheritance in both Java and C++, ranging from over twice as fast in Java on Windows to about 18 times slower in C++, asymptotically.

Appendix

The table below indicates the running times in seconds of different versions of the aforementioned performance test. The leftmost column ("depth") represents the depth of the inheritance hierarchy or number of extensions applied. For each category, the left column ("inh") represents built-in inheritance and the right column ("mix") dynamic mixins. For C++, an additional column ("C") represents a C-style implementation using `structs` and explicit receivers.

Two platforms were used for testing: Windows 2000/Pentium II and Solaris 2.7/UltraSPARC. For Java, the Sun Java 2 SDK v1.3 (HotSpot VM) was used on both platforms. For C++, Microsoft Visual C++ v6.0 was used on Windows, and GNU C++ v2.95.2 on Solaris.

References

[1] G. Bracha and D. Griswold. Extending Smalltalk with mixins. In OOPSLA'96 Workshop on Extending the Smalltalk Language, April 1996. Electronic note available at http://www.javasoft.com/people/gbracha/mwp.html.

[2] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[3] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.

[4] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[5] Yannis Smaragdakis and Don Batory. Mixin-based programming in C++. Technical Report CS-TR-98-27, University of Texas at Austin, January 1998.

	Windows 2000					Solaris 2.7				
	Java		C++			Java		C++		
depth	inh	mix	inh	mix	C	inh	mix	inh	mix	C
0	9	9	0	0	0	9	9	3	0	1
100	17	13	0	2	4	46	47	4	5	4
200	36	17	1	4	6	97	100	3	9	9
300	59	21	1	8	10	156	155	3	14	14
400	83	25	2	12	21	202	231	3	20	19
500	107	37	2	33	40	256	259	3	24	24
600	134	52	2	48	58	308	308	3	30	29
700	157	70	3	62	69	361	359	3	34	33
800	184	83	4	73	80	413	408	3	39	38
900	210	94	5	83	90	465	461	4	44	43
1000	233	110	5	92	102	557	512	3	49	48