Extending Smalltalk with Mixins

Gilad Bracha David Griswold

August 5, 1996

1 Introduction

This document describes the extension of Smalltalk with mixins. The motivation for this proposal is the desire for increased code sharing. Smalltalk has a clean and simple single inheritance semantics. However, this can sometimes result in undesirable code replication.

Mixin based inheritance [BC90, Bra92] does not introduce the complications associated with multiple inheritance . It preserves the simple semantics of Smalltalk subclassing and message lookup, while allowing a large degree of code sharing. It is therefore desirable to try and incorporate mixins into Smalltalk.

2 Specifics of the Extension

2.1 What is a Mixin

A mixin specifies a set of modifications (overrides and/or extensions) to be applied to a superclass parameter. A mixin differs from an ordinary subclass definition in that it abstracts over the identity of its superclass. Mixins may be viewed as functions that take superclasses and produce subclasses. Consider the following pseudo-code :

```
Point subclass ColorPoint  {
instance variables:
   red green blue
methods:
  red: r green: g blue: b
  hue: h saturation: s value: v
  red
  green
  blue
  hue
  saturation
  value
}
```

The code in ColorPoint has little to do with points. It applies to a variety of colored graphical entities. Replicating the color specific code in a variety of places is undesirable. An easy and natural way to achieve this to incorporate the code into a mixin:

```
Mixin Color(G) {
G subclass {
instance variables:
   red green blue
```

```
methods:
  red: r green: g blue: b
  hue: h saturation: s value: v
  red
  green
  blue
  hue
  saturation
  value
}}
```

The mixin Color abstracts over an unspecified superclass G. Color can then be used at different points in the subclass hierarchy, by *invoking* it on different actual superclasses, much as one invokes a function at multiple points in a computation. This is accomplished using |>, the mixin invocation operator :

```
ColorPoint = Color |> Point
ColorCircle = Color |> Circle
```

The |> operator takes a mixin M and a class S, and produces a new class that modifies its superclass S with the code defined in M.

The mixin can be applied to various superclasses to derive different classes. However, the source code of the mixin is shared among all these mixin applications, promoting modularity.

The analogy between mixins and functions is useful in understanding the properties of mixins. Here are some key points:

- One must distinguish between mixins and classes, just as one distinguishes between functions and the values they produce. A mixin is not a class; it must be invoked on an actual superclass parameter to produce a class.
- Mixins do not affect the semantics of subclass construction or method lookup. This is analogous to the fact that placing an operation like + within a function's body does not change the semantics of +.
- A mixin may not be applicable to all classes, much as a function is not necessarily well defined for all inputs. A function may place specific requirements on its actual parameters. Likewise, a mixin may place various requirements on the superclass. For example, a mixin may contain calls to super methods. These must be defined for any actual superclass. Similarly, if a mixin defines an instance variable, it must not be defined in the actual superclass.
- A mixin can be composed with another mixin, to produce a composite mixin, in a manner completely analogous to function composition. We write $M1 * M2$ to denote the composition of mixin M_1 with mixin M_2. By definition, $(M_1 * M_2) \triangleright S = M_1 \triangleright (M_2 \triangleright S)$.

See [BC90, Bra92] for a more extensive discussion of mixins.

2.2 Identity Issues

In Smalltalk, every class is itself a first class object with its own identity, and potentially with its own class and instance variables. It follows that every mixin invocation produces a distinct class, with a distinct identity, as its result. This is essential to prevent syntactically identical invocations (possibly in different libraries) accidentally sharing state with unforseen consequences.

2.3 Extracting Mixins from Classes

It is often desirable to treat an existing class as a mixin. S mixin returns a mixin that contributes the definitions given locally by the class S.

Consider the case of a class in a library of graphical widgets:

Widget subclass AgregateWidget{...}

Such a class groups widgets together. It is useful to think of this class a collection of widgets. However, one cannot usually inherit functionality from Collection since graphical widgets belong to a separate hierarchy. By revising the definition to

Collection mixin |> Widget subclass AgregateWidget{...}

one can reuse all of the functionality in the Collection class.

3 Interaction with Existing Language Features

Mixins do not change the semantics of subclassing or of message lookup in Smalltalk. As a result, mixins do not influence the behavior of existing Smalltalk programs. The extension is therefore fully upward-compatible. Furthermore, mixins can easily be eliminated from a program by automatically creating a class for each mixin invocation, and duplicating the mixin's code in it.

4 Implementation Issues

In principle, mixins are easily implemented using Smalltalk's reflective capabilities. One can simply enclose a class constructor within a method that takes a superclass and a name as arguments. The method must also reflectively add the required methods to the new class. Of course, this solution is too awkward to be useful in practice. A usable implementation must provide programming environment support for creating and incrementally modifying mixins. Whenever a change is made to the mixin, it must be propagated to all invocations, and checked for validity. More ambitious implementations may seek to share code among a mixin's invocations.

5 Implementation status and Experience

A working implementation of mixins in Smalltalk has been constructed. Mixins have been used in the construction of real applications, and the code sharing benefits have been significant. In particular, the ability to extract mixins from existing classes has proven very valuable. Further experience is necessary, however, to to determine how extensively one should use mixins, and to establish a methodology of working with mixins.

6 Conclusion

Mixin based inheritance can easily be added to Smalltalk without changing the existing semantics of Smalltalk subclassing and method lookup. Preliminary experience with a working implementation indicates that the extension fullfills its goal of increasing code sharing, without the complications inherent in multiple inheritance.

References

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, October 1990.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, 1992.