OOPSLA'92
Vancouver, British Columbia, Canada
5 – 10 October 1992

Addendum
to the
Proceedings

Poster Submission— Interfacing Different Object-Oriented Programming Languages

Report by:

Scott Danforth
IBM Object Technology Products

Abstract

This poster presentation illustrates the use of SOM (the IBM System Object Model) for interfacing, different object-oriented programming (OOP) languages. Our approach allows classes defined in one OOP language to be used by different (possibly non-OOP) languages—both for subclassing, and for object creation. This extends the utility of OOP class libraries and makes it possible to define "multi-language" objects, whose supporting methods and instance variables are provided by different languages.

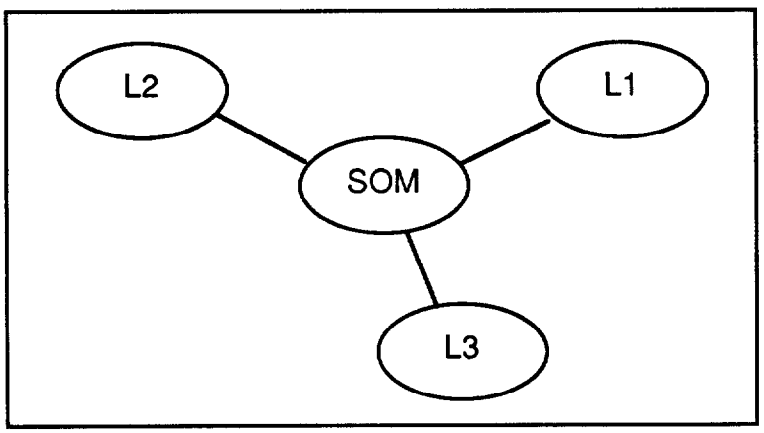

Figure 1 — SOM provides a Common Object Model for use by different languages

Summary

SOM [1] can be used to integrate classes provided by different OOP languages, making these classes available across language boundaries—even to non-OOP languages. This is illustrated in Figure 1, which shows SOM as a central hub integrating, different languages and object models. This figure illustrates two important aspects of our approach. First, SOM is used as a central component, available for use by a variety of different languages regardless of their object models. Second, because of this, the order N^2 problem of interfacing, different languages is reduced to an order N problem.

SOM is not a programming language; SOM classes are made available in a language-neutral fashion via an API accessible to both OOP and non-OOP languages. This is an initial step in the direction of interfacing different OOP languages, and is illustrated in Figure 2. In this figure, a SOM class named A defines the method *foo*, and *usage bindings* for this class are provided to the client languages *L1* and *L2* (the language used to implement the SOM class is not important - the SOM API is used to define the class and register a procedure that supports the *foo* method on objects that are instances of this class).

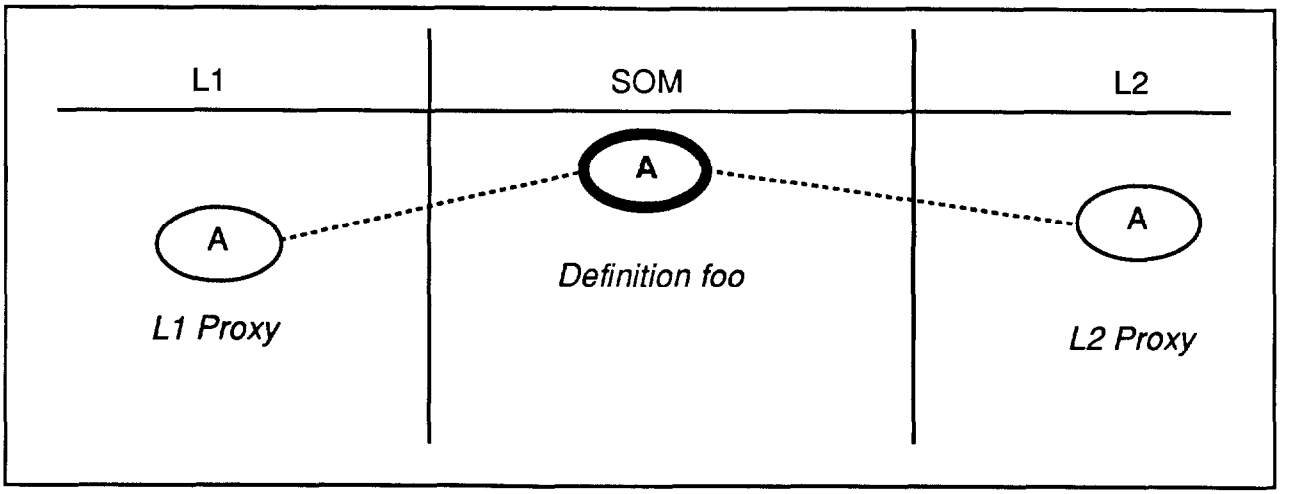

Figure 2 — Defining and exporting SOM classes to different languages using Proxies

When a given client language has an object model, it is possible to provide usage bindings for a SOM class via a *proxy class*, expressed in terms of the given client language. Users of the client language then see SOM classes in terms of the client language and the client language implementation can provide type checking support for SOM class and SOM object usage. When a client language does not have an object model, usage bindings cannot be expressed in terms of classes. Then, some other language-specific approach is used to hide the details of using the SOM API, and the proxy-based techniques we illustrate here are modified accordingly.

Stated in terms of Figure 2, an example of interfacing different OOP language would be to (1) allow *L1* to subclass its proxy for *A*, creating a new class *B* that overrides *foo*, (2) create a SOM class proxy corresponding to this new class, and (3) make the new class available (by SOM proxy) to *L2* as well. Figure 3 illustrates the logical form of one possible solution to this problem, using arrows to represent proxy classes' deferral to appropriate definition of *foo*.

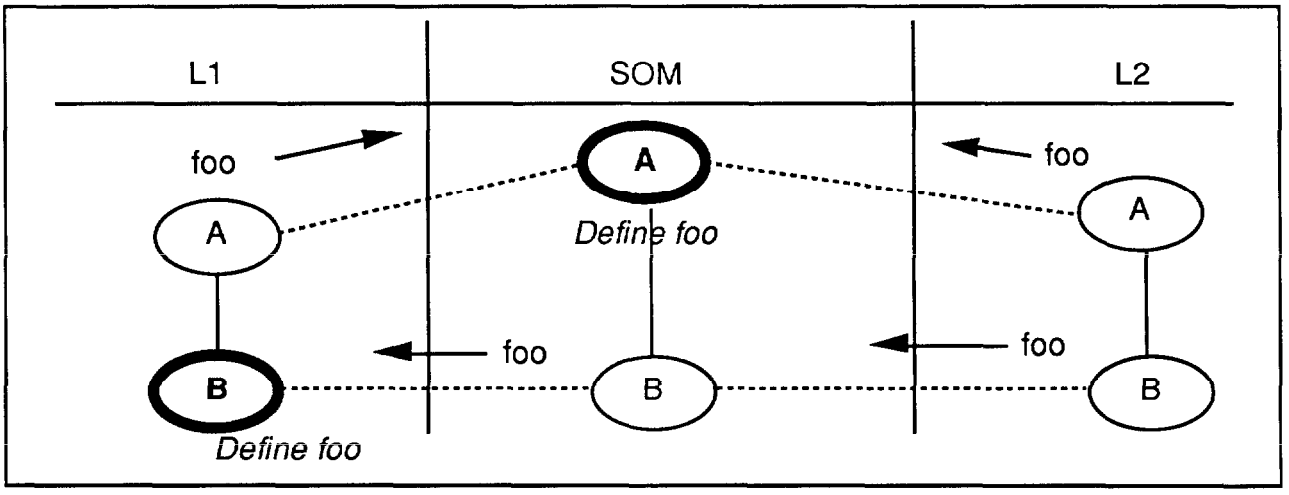

Figure 3 — Interfacing different object-oriented languages

In general, three requirements of a solution to the overall problem of interfacing different OOP languages can be identified:

1. Method calls (i.e., virtual functions) must be correctly dispatched. This is arranged by appropriate definition of classes. As illustrated in the poster presentation, SOM's capability for dynamic class definition is extremely useful in this regard.
2. When the calling language is different from the language in which the method is implemented, it is necessary to provide argument conversion (i.e., from a SOM object to a client language object, or vice versa). This requirement embodies deceptively subtle complications when the client language itself does argument conversion when calling methods (e.g., as in C++).
3. To provide the full benefits of polymorphism, proxy classes should be related in their class hierarchy in the same way as the corresponding SOM classes are related.

Many details of importance in light of the above three requirements are not illustrated by Figure 3. These are considered in the poster presentation.

References

[1] OS/2 2.0 Technical Library *System Object Model Guide and Reference*, IBM document 10G6309, 1992.

Contact information:

Scott Danforth
IBM Object Technology Products
IBM Zip 9370
11400 Burnet Road
Austin, TX 78758
Office: 512-838-8074
Fax: 512-838-1032
shd@ot.austin.ibm.com