Situated Objects

Patrick Dubroy
Y Combinator Research, USA
pat.dubroy@ycr.org

Abstract

We explore a form of object-oriented messaging in the context of an ownership tree, where owners can respond to messages on behalf of their transitively-owned objects. If all access to an object is mediated by its owner (*owners-as-accessors*), then we call such an object a *situated object*. We discuss how situated objects can offer another perspective on the perennial separation-of-concerns problem in object-oriented programs.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-Oriented Programming

1. Introduction

Since Harrison and Ossher's landmark "critique of pure objects" [3], many different solutions for modelling the *extrinsic* properties of objects have been proposed [5, 4, 7]. We explore another solution to this problem in the context of a dynamic ownership system.

1.1 Motivation

As an example of the need to model extrinsic properties of objects, we use a vector drawing application. In such an application, we might have a Document object which contains a ShapeList, which in turn contains Shapes. In the user interface, a shape may be selected or not. Selection is an inherently contextual property; a "pure" Rectangle object shouldn't need to deal with such an application-level concern.

However, if another component has a reference to a shape, it is convenient to be able to directly ask the shape if it is selected. A typical solution is to implement an `isSelected` method for shapes which simply forwards the message to the shape's container. But if there are many kinds of container (e.g., Group, Layer, etc.), they may all need to be modified to handle `isSelected`.

If the system is built using third-party classes, it may not be possible to modify them directly, but subclassing can be used instead. Unfortunately, in most languages this requires changing or wrapping any existing methods that instantiate these classes, to ensure that they instantiate the appropriate subclass instead.

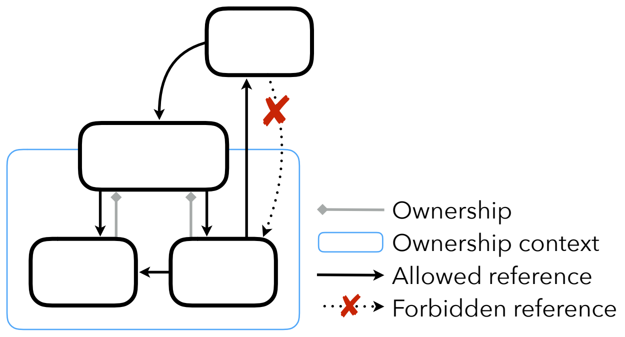

Figure 1. Owners-as-dominators

1.2 Ownership Systems

The model described above is an example of a *part-whole hierarchy* — a document is composed of a shape list, a shape list is composed of shapes, etc. In a part-whole hierarchy, an object can be said to "own" the objects it is composed of.

Various *ownership systems* have been proposed [1] to make it possible to formalize and enforce the notion of ownership. In these systems, an object has at most one owner; together, the objects form an *ownership tree*. The details and implications of ownership vary considerably across systems; in an *owners-as-dominators* model [2] (Figure 1), the type system ensures that all paths to an object from the root of the system pass through that object's owner. By contrast, in an *owners-as-accessors* discipline [6], there are no restrictions on references between objects, but all accesses (i.e., message sends) to an object must be made via its owner.

2. Situated Objects

Building on the notion of owners-as-accessors, we propose the following: instead of outright forbidding message sends that bypass an object's owner, what if such messages are allowed, but transparently dispatched down the receiver's ownership chain?

A naive implementation would be similar to the hierarchical event dispatch found in many UI toolkits: beginning at the receiver, we walk the `owner` links, recording each one in a list. When the root owner is reached, we use the list to dispatch the message back down the ownership chain, beginning at the root owner.

At any point in the dispatching process, we allow an owner to abort the hierarchical dispatch and respond directly on behalf of the receiver, much like how a standard OO method can choose to whether or not to invoke the superclass implementation. If all access to an owned object goes through this dispatch mechanism, we call such an object a *situated object*, because its observed behaviour comes from its intrinsic behaviour combined with its position in the ownership tree.

2.1 Prototype

To evaluate this idea, we have been experimenting with a JavaScript library for defining and instantiating situated objects. The library manages object ownership and message dispatch, but does not enforce any restrictions based on the ownership tree. (Our primary interest is in ownership-directed messaging — not the details of a dynamic ownership system.)

2.1.1 Implementing the Selection Protocol

With Situated Objects, we only need to respond to `isSelected` and `setSelected` in the object that manages selection (in our case, the ShapeList). An external component can still send the `isSelected` message to a shape, but — without the knowledge of either the sender or the shape object — the method implementation comes from the ShapeList.

Here is how the ShapeList might be implemented[1]:

```
var ShapeList = situated.createClass({
  parts: {
    selIdx: -1,
    shapes: []
  },
  'shapes/:idx/isSelected'(idx) {
    return this.selIdx === idx;
  },
  'shapes/:idx/setSelected'(idx, val) {
    if (val) {
      this.selIdx = idx;
    } else {
      var shape = this.at(`shapes/${idx}`);
      if (shape.isSelected()) {
        this.selIdx = -1;
      }
    }
  },
  ...
});
```

[1] This example uses some features of the ECMAScript® 2015 (also known as *ES6*) syntax — specifically, *method definitions* and *template literals*.

To create a new situated object class, the programmer passes in an object which maps from *message patterns* to *methods*. When a situated object is dispatching a message, it tries to match the message against each of its patterns. If one matches, it executes that method. If not, the message is dispatched down to the appropriate child, with the first path component stripped off.

In this manner, any shape in the ShapeList is automatically becomes a "situated" shape that can respond to `isSelected` and `setSelected`. For example:

```
var shape = shapeList.at('shapes/0');
shape.setSelected(true);
```

Since all messages to the shape will go through the ShapeList (even messages from the shape itself) this message routing effectively changes the interface of the shapes.

3. Discussion

This is very early work that we think could greatly benefit from the feedback of NOOL attendees. Among the questions we have yet to answer are:

- How does owner-based dispatch interact with traditional inheritance-based method dispatch? Can we combine them in a sensible way?
- What kinds of message patterns should we support?
- Are there useful connections to the message patterns used in pub/sub architectures?

References

[1] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 15–58. Springer, 2013.

[2] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proc. OOPSLA*, pages 48–64. ACM, 1998.

[3] W. Harrison and H. Ossher. *Subject-oriented Programming: A Critique of Pure Objects*. ACM, 1993.

[4] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008. URL `http://www.jot.fm/issues/issue_2008_03/article4/`.

[5] G. Kiczales et al. Aspect-oriented programming. In *Proc. ECOOP*, pages 220–242. Springer, 1997.

[6] J. Noble and A. Potanin. On owners-as-accessors. In *Proc. IWACO*, 2014.

[7] A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with expanders. In *Proc. OOPSLA*, pages 37–56. ACM, 2006.