Object Visibility: Making the Necessary Connections

Reprinted from the October 1991 issue of *The Smalltalk Report*
Vol. 2, No. 2
By: Rebecca J. Wirfs-Brock

An exploratory design is by no means complete. It is a rough conceptual sketch of the key objects of a design, what their roles are, and a partial list of their responsibilities and collaborations. A lot of detail needs to be added before this relatively high level design description can turned into code. I want to focus on just one of those details: how to turn an imprecise list of collaborations into a more rigorous design description, and finally into Smalltalk code. It is a relatively straightforward process. One that can be tackled systematically. Following a few general principles during this translation process results in classes that are more reusable, and easier to change or enhance.

Turning an architectural drawing into a detailed set of blueprints shares a few similarities with the software construction process. When developing detailed blueprints, an architect translates a rough architectural drawing into a specific list of materials to be used, and a fairly explicit map of how those materials should be composed in the finished product. This still leaves a lot of latitude for decision making and creativity during construction, just ask anyone who has had a house built. However, the structure of the finished product is readily apparent from these drawings. You don't start construction expecting a barn and end up with a skyscraper! The same principles apply to constructing software.

Before turning an object-oriented design description into a detailed set of software blueprints, you must consider the tools and environment you will use during construction. Mapping an object-oriented design into Smalltalk code requires matching up object-oriented design concepts with the *appropriate* Smalltalk language and programming constructs. It's essential when constructing a solution to have a good understanding of pre-existing components. Proficient Smalltalk programmers know their Smalltalk class library. They don't construct a new class when a readily available one will do the job, even if it isn't perfect. Architects don't invent new kinds of fasteners or building material for each construction project.

Before systematically adding more rigor to our collaborations, let's examine our Smalltalk construction environment. How many different ways are there is Smalltalk for one object to have visibility of another? Objects can't collaborate unless they can send each other messages. The client, or sender of a message, first needs to have visibility of the server, or receiver of the

message. Message sending is all done within the context of a method. Anyone with a modest amount of Smalltalk programming experience should be able to come up with most of these techniques fairly quickly. For new Smalltalkers, this is a good exercise. These are fundamental implementation constructs. *You will use these constructs (and other techniques) when you translate designs into executable program code.*

Here's a list, in no particular order, of ways that an object may be visible within a method:

- an object always has visibility of itself (sending messages to self is fundamental to Smalltalk programming)
- those objects passed in as arguments
- an object's Class (by sending the message self class)
- values of instance variables—objects that are part of the object's encapsulated state
- any object returned as a result of sending a message to an object that is already visible
- objects assigned to temporaries
- any class whose name is known
- you can create an object whenever you need it (assuming you know the name of its class)
- an object that is a value of a global variable (for example, Smalltalk)
- class variables of the object's class or any of its superclasses
- variables in pools specified by the object's class
- constant objects known to the language (for example nil, true, and false)
- literals (including integers and floating point objects, strings, literal arrays, a literal block)

Enough! I asked my colleagues for additions and got several that were far too obscure to include for the purpose of this column. I am not trying to develop an exhaustive list. It's long enough that it needs some organization. Let's organize these objects into four categories:

1. Globals of varying scope. We can include globals, pools and pool variables, even class variables in this category. These global spaces typically contain objects visible to many other objects. If you can name an object in one of these global spaces, it's yours for the accessing.

2. Objects that *dynamically* become known within the context of a method. These include objects passed in as arguments and any object returned from a message. An object that becomes visible in this way can always be retained for later reference, or discarded, as needed.

3. Objects that are part of an object's encapsulated state, ie. instance variables.

4. Basic programming constructs. It's difficult to write any significant code without using nil, true or false. Literals are also in this category, and are just as ubiquitous.

Examining the Exploratory Design

What kinds of collaborations typically are recorded? Most collaborations are recorded between objects at the same or next layer of detail. If a designer has figured out the details of an algorithm, quite a number of collaborators at very different conceptual levels may be listed. This is an exception rather than the rule. It is more common to have just a vague idea some kind of collaborative effort is required. Most often, collaborators that are listed are objects that will dynamically become known, and not those that are permanently visible.

Lists of collaborators certainly aren't exhaustive, and they aren't very precise. This doesn't mean we have a bad design, only a preliminary one. During the early stages in design, we determine when we need to use the services of some key collaborators, but we don't yet need to determine precisely how we will use them. First, we develop a model of what an object should do along with a vague idea of some of its key collaborators. Next we need to try out a number of alternatives.

To add precision, we need to determine whether an ongoing dialog or a single message will do. We need to construct a model of how each responsibility will be accomplished. This requires experimentation. There's no one right way to decompose a solution. However, when working out these details, there are a number of principles worth following to make your implementation cleaner.

Limit Visibility

One guiding principle is *make objects visible of each other on a need to know basis.* An even stronger statement: don't retain visibility of any object if you absolutely don't have to. In general, design objects so they know about as few other objects for as short a time as possible. If an object only needs to know about another for the duration of a method, pass it in as an argument. Let the client supply necessary information. Of course, this can be carried to extremes, resulting in objects with poorly designed, complicated to use interfaces.

Simplify Collaboration Sequences

Complex message protocols that have lots of arguments, or require exacting sequences of messages between client and server, make objects difficult to use and understand. A balance must be achieved between exposing too much complexity and giving enough controls to the client. This indeed is a matter of style. Simple interfaces are worth striving for.

I drive a car with a manual transmission. I prefer to drive a car with a manual transmission. I enjoyed learning to drive a car with a manual transmission because of the extra control I have. It is so second nature to me now that it doesn't seem very difficult at all. For years, my mother has driven a car with an automatic transmission. When automatic transmissions became popular, she switched from driving a manual to an automatic. She preferred the simplicity. It certainly is much easier to accelerate a car by sending the single message myCar accelerate. I go through this sequence whenever I need to shift gears before accelerating:

```
myCar depressClutch
myCar shiftGear:  a GearValue
myCar releaseClutch
myCar accelerate.
```

Most people prefer a simpler interface, provided the necessary services are offered. Too many software engineers, on the other hand, are used to offering a manual transmission interface to drive their objects when their clients would appreciate a simpler pattern of collaboration.

Store Facts in One Place

If the same objects are used in a number of methods, hold on to this shared information in the object's class. Class methods can easily be designed to yield this default information. It is a matter of style whether these objects should returned from class methods or stored in class variables. From an instance's perspective, maintenance of this constant information is an appropriate responsibility of its class, however that may be accomplished. This eliminates sprinkling the same literal objects over a number of instance methods. If a literal value needs to be modified, the programmer only has to make the change in one place.

Work at reducing the number of objects that a class depends on. Direct reference to any global objects is considered harmful by many Smalltalkers. Code with 'hard-wired' references to other objects is fragile. It is highly dependent on the correct context being established before it can run. It is hard to reuse code containing global references in another context. To be reused, code must either by reworked to remove direct global references, or scaffolding code must be executed to set up the necessary global context.

Limit Dependencies on Object Structure

Sending messages to self is a valuable implementation technique for two reasons. It allows the programmer to separate the detailed steps from the main parts of an algorithm. This allows a maintenance programmer to see the forest for the tress. It clearly identifies steps in an algorithm that can be performed differently by a subclass method.

Just as importantly, sending accessing messages to self allows code to be insulated from changes in instance variable structure. It also allows subclass developers to override those accessing methods and provide the necessary information in another way.

Develop a Sense of Style

Don't try to use every language construct when translating design level collaborations into a Smalltalk implementation. Current Smalltalk environments have too many ways to make objects visible, for my tastes. Multi-person teams should develop and stick to a style guide that addresses when and how to use particular Smalltalk constructs, and how to simplify collaborations patterns. Smalltalk programming style is an evolving art form, and different organizations quite naturally develop their own unique sense of style. It is important to cultivate a sense of style (it will evolve) and create some coding guidelines before translating a design into code.