Formulating Update Messages

Jon. Hallett*

Abstract

This paper presents a method with which we can generate update messages for use with Smalltalk's dependency mechanism. The basic idea is that any messages which cause an object to change are forwarded to the object's dependants. The method is perfectly general and future-proofs objects against changes in their dependants.

Keywords: Dependencies, Object-Oriented Programming, Smalltalk, Update Messages

1 Introduction

Smalltalk [1] provides a number of features that help us structure our programs. One of the most interesting of these features is the dependency mechanism, which invites us to structure our programs in terms of objects that depend on one another's state. Using these dependencies, we can write programs in which objects react to the state changes of other objects.

This kind of program structure is useful in a variety of contexts, but it is especially useful in programs that have a graphical user interface. Smalltalk's own user interface, with its model-view-controller (MVC) [2] architecture, is held together by dependencies. MVC uses dependencies to ensure that the graphics on screen present an up-to-date view of the objects they depict.

In Smalltalk, the implementation of the dependency mechanism is very simple: each object maintains a list of its dependants and, when it changes, it sends a message to every object in its list.

Despite this simplicity there are some subtleties to consider. In particular, the exact form of the message an object sends to its dependants is left open. There are no conventions for this, so when programmers use dependencies, they must decide, for each object individually, what it needs to send to its dependants. This is not always easy; it can be difficult for the programmer to tell what the dependants need to know in order to react appropriately. It is certainly impossible for the programmer to predict what future dependants, added as a system grows, will need to know.

To help programmers decide what messages objects should send to their dependants, this paper presents a method for formulating update messages for a given object. Essentially the trick is to forward to the dependants the message that caused the object they depend on to change. This method is surprisingly flexible, and perfectly general. It also has other benefits: for example, it makes it easier for us to work out where a given update message has come from, a significant problem in large programs.

The rest of the paper is divided into four sections: Section 2 describes Smalltalk's dependency mechanism; Section 3 presents the method for formulating update messages; Section 4 discusses the generality of the method; and Section 5 presents the conclusions.

2 Smalltalk's Dependency Mechanism

Smalltalk's dependency mechanism is implemented by three messages: `addDependent:`, `removeDependent:` and `broadcast:`. All three messages are defined in the root class, Object, and are inherited by all other classes.

These three messages are not the only ones provided by the dependency mechanism: there are others, two of which (`changed:` and `update:`) are discussed below. The essence of dependencies, though, is implemented by the messages `addDependent:`, `removeDependent:` and `broadcast:`, so our discussion will focus on them.

Let us start by examining `addDependent:` and `removeDependent:`. These two messages make and break dependencies between objects. Dependencies between objects are stored explicitly: conceptually, though not in fact, every object in Smalltalk contains a list of its dependants. The contents of this list are controlled by `addDependent:` and `removeDependent:`. When sent to a object, `addDependent:` adds its argument to the object's list of dependants. Similarly, `removeDependent:` removes its argument from a object's list of dependants.

*Department of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, UK, e-mail: j.hallett@ecs.soton.ac.uk

At this point it is appropriate to introduce a new piece of terminology. Although we have a word for describing objects that depend on other objects—they are *dependants*—we do not have a word for objects that are depended upon. In the rest of this paper such objects will be referred to as *models*.

While `addDependent:` and `removeDependent:` control how objects depend on one another, the real work of dependencies is done by `broadcast:`. It is using `broadcast:` that models notify their dependants that they have changed state. When a model finishes executing a method that might have caused it to change state, the programmer will arrange for it to send itself a `broadcast:` with, as the argument, a message selector.

When a model receives a `broadcast:` message, it sends every object in its list of dependants a unary message. The precise unary message sent is determined by `broadcast:`'s argument, which must be a unary message selector symbol. This message sent to the dependants is known as an *update message*, and the form of these update messages is the central concern of this paper.

Actually, `broadcast:` is only one of a family of messages. The others are of the form `broadcast:with:`, `broadcast:with:with:`, etc. These messages are just like `broadcast:` except that, instead sending a unary message to each dependant, they send a keyword message with one or more actual arguments. These more general forms are used more often than simple `broadcast:`. For example, most objects that use dependencies choose to broadcast `update:` and one argument to their dependants, using `broadcast:with:`. In fact, this is done so often that class Object provides a method, `changed:`, as a shorthand for a `broadcast:with:`. The implementation of `changed:` is:

```
changed: anObject
    self broadcast: #update:
         with: anObject.
```

But, whether programmers choose to use `broadcast:` or to use `changed:`, Smalltalk gives no guidance as to the contents of their arguments. These arguments are actually very important: dependants rely on these arguments as their only clue to how their models have changed. How dependants behave on finding that their model has changed is often governed entirely by the contents of these arguments.

Traditionally, programmers decide what information a `changed:` or a `broadcast:` needs to convey by inspecting the methods of the dependant objects that will receive them. They find out what information a dependant actually needs to perform its tasks, and then arrange for the model to provide this, and only this, information.

This *ad hoc* approach is not really satisfactory. It introduces an unhealthy dependency between the code of the model and the code of its dependants: if a dependant is rewritten or if a new kind of dependant is added to the program, then we might have to modify a model's code. We don't want to have to do this. After all, a model shouldn't depend on its dependants at any level.

But we don't have to work like this. As we shall see in the next section, we can design general-purpose arguments to `broadcast:` and `update:` that can support any conceivable dependant and can save us from having to modify models if we ever modify their dependants.

3 Formulating Update Messages

It is actually surprisingly easy to arrange for a given model to generate general-purpose update messages. The trick is to have the model forward the messages it receives to it dependants.

There are several ways of forwarding messages in Smalltalk. One possibility uses `changed:`; another uses `broadcast:`.

Taking the first possibility, we can forward messages to dependants by calling `changed:` with an argument that contains the message received by the model. The argument must, of course, fully describe the message: this means that the argument must include not only the message's selector but also all its arguments, including the implicit argument `self`.

It is convenient to bundle the elements of the message into an `Array`. This array can then be used as the argument to `changed:`.

Of course, not every message a model receives has to be forwarded to its dependants. Only those that might have caused the model to change have to be forwarded. Methods that cannot possibly change the model's state, such as query methods, need not be forwarded.

As an example, consider this trivial implementation of a stack:

```
instance variables: anArray

class methods

new
    ^super new initialize.

instance methods

initialize
```

```
    anArray <- Array new.

push: anObject
    anArray addLast: anObject.

pop
    anArray removeLast.

top
    ^anArray last.

isEmpty
    ^anArray isEmpty.
```

Only two messages can cause stacks to change state—`push:` and `pop`—so only these messages need to be forwarded.

To forward `push:`s and `pop`s, we modify their methods like so:

```
push: anObject
    anArray addLast: anObject.
    self changed: (Array with: #push:
                         with: self
                         with: anObject).

pop
    anArray removeLast.
    self changed: (Array with: #pop
                         with: self).
```

Here we can see the calls to `changed:` with the arrays containing the messages.

As mentioned above, there is another obvious way for models to forward their messages, this time using `broadcast:`. Instead of sending each dependant the message `changed:` with an argument containing the forwarded message, we could arrange for a model to send the dependants the forwarded message itself, using `broadcast:`. Actually, we can't just send the forwarded message directly, because, as mentioned before, we need to make sure the dependants get the implicit argument `self` as well as the ordinary arguments. What we do instead is extend the message we want to forward with an extra keyword, providing another argument place with which we can pass `self` to the dependants.

A convenient way of adding the extra keyword, with a pleasant incidental benefit, is for us to extend the message selector that we want to forward with the class name of the model. Using this scheme on the stack example, we would extend `push:` to `stack:push:`. Then `push:`'s method would become:

```
push: anObject
    anArray addLast: anObject.
    self broadcast: #stack:push:
         with: self
         with: anObject.
```

The other message that needs to be forwarded, `pop`, is slightly trickier to extend. This is because `pop` is a unary message and so, rather than having an extra keyword grafted on, it actually needs to be made into a keyword message, say `stackPop:`. Then, `pop`'s method becomes:

```
pop
    anArray removeLast.
    self broadcast: #stackPop:
         with: self.
```

The incidental benefit of this way of extending the message selector is that it makes finding the source of update messages very easy. With the name of the originating class on the front of every update message, it is obvious where they are coming from. In even a medium-size system, this is a good thing.

4 Generality of the Method

Though simple, this method is perfectly general in the sense that it provides us with a way of adding to any model a set of update message capable of precisely describing any possible change in any model to any dependant.

This method and its generality are based on two basic properties of objects. First, an object can only change state because it has been sent a message asking it to change; this is because only an object's methods can change it, and these methods are invoked only by messages. Second, an object accepts only a limited number of messages, as defined by its protocol; this means we can list all the messages that a given object can accept.

Using these to basic properties of objects, we can see that it is possible to list all the messages that can cause a given object to change. We will know that this list is complete and that it completely characterises the changes which the object may undergo.

The method presented in this paper uses this list to determine which messages to forward to dependants. In using this list we guarantee that the messages forwarded to dependants completely describe the changes of the models.

In fact, in this scheme, so much information is provided to the dependants that they could, if they wished to, duplicate the behaviour of their models. They of course rarely need to, but the possibility is reassuringly still there.

5 Conclusions

In this paper, we have seen a method for formulating the update messages used in Smalltalk's dependency mechanism. The method produces update messages that are completely general, and are suitable for use with any dependant.

Of course, this technique is not limited to use only in Smalltalk. The basic insight—that a model should forward the messages it receives to it dependants—is applicable in any environment. Smalltalk does, though, make this scheme particularly easy to implement.

References

[1] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[2] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, August 1988.