Chapter

19

The Dependency Mechanism

Perhaps the most magical thing about Smalltalk is the way it can automatically inform objects when other objects change their state. It does this by allowing one object to register an interest in getting change messages — by registering as a *dependent* of another object. The terminology is that the object registering an interest in being informed of changes is called the *dependent*. The object that it is dependent on is called the *parent*.

Being able to use the dependency mechanism allows you to write code where an object registers as a dependent then forgets about the parent completely. When the parent changes, the object will be informed and can do whatever it needs to do but until that time, it can forget the parent even exists. From the perspective of the parent, there is no need to programatically keep track of all the objects it needs to inform of changes. Instead, all you have to do is have the parent tell itself that it has changed, and all the dependents will be automatically notified. The dependency mechanism allows you to think of an application as a group of objects with wiring between them. A major part of the application now boils down to determining the wiring between the objects.

How is this useful? There are many applications where one object is interested in the state of another object, but the second object really doesn't care about the first object. It doesn't care whether there are many objects interested in it, or no objects interested in it. One example would be a stock object that holds onto its current trading price. There may be many applications interested in the price of the stock as it changes, but the stock really doesn't care about those applications. Another example would be a factory floor. There may be many monitors interested in the temperature and quantity of a liquid in a container, but the container has no interest in the monitors. However, in the examples, the stock and the container have to let other objects know about changes in their state.

There are two mechanisms for doing this. The interested objects could *poll* the stock and container objects, asking for their state every few seconds. However, this has disadvantages. It takes computer resources to constantly ask for information and this can add up if there are lots of objects polling. It also means that objects aren't informed of changes in real-time. If an object polls every five seconds, it could be five seconds before it discovers a state change, which could be disastrous on a factory floor. If it polls every hundredth of a second, huge amounts of computing bandwidth will be used if you have many objects polling.

The other approach is to use the dependency mechanism. The interested objects register an interest in some aspect of the stock or container's state, then wait until they are informed of changes. The stock or container

doesn't know or care how many interested objects there are. Let's go through an example and see how the dependency mechanism works.

We'll look at a simple example of a stock object and a watcher object that is monitoring the *stock price*. As a stock is traded, the stock object gets updated with the latest price and the number of shares in the trade. Since we don't know how many stock watchers will be monitoring any particular stock, and we don't know what they will do with the values, we don't want the stock to be responsible for who to tell, and what to tell them. Instead, we use the dependency mechanism. When the stock receives information about a new trade, it updates its instance variables. When it updates an instance variable, it tells itself that it has done so. This act of telling itself that it has changed triggers off the whole dependency notification mechanism.

We'll be showing several mechanisms and each example will be suffixed with a number. To execute the example, type the following into a workspace, highlight it, and evaluate it (don't do it yet since we haven't yet defined any classes and methods). Make sure that the number is appropriate for the example. The code for all these examples can be found in the file `depend.st`.

```
| stock |
stock := MyStock1 new.
MyStockWatcher1 watchStock: stock.
stock traded: 200 price: 20.
stock traded: 100 price: 30
```

Basic Dependency

Here are the class definition and the instance methods for the stock class. Notice that instance variables are changed through their accessors, and their accessors are responsible for telling the stock object what has changed, and the old value. This is the key part of the parent's role in the dependency mechanism. By sending `changed:with:`, a whole set of dependency code is invoked. The normal convention is to send the *old* value since the dependent can send a message to get the new value, and may otherwise not know the old value. Sometimes, however, you will see the new value sent. Whichever approach you take, be consistent.

Most of the methods of the stocks are defined in the superclass, *MyStock*. Where necessary we will override this behavior. The accessor methods look very similar for each variable, so we'll write them just for *price*. The other accessors will simply have different names for the variables.

```
Model subclass: #MyStock
   instanceVariableNames: 'price trade totalTraded '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'DependencyTests'

MyStock>>traded: aCount price: aPrice
   self price: aPrice.
   self trade: aCount.
   self totalTraded: self totalTraded + aCount

MyStock>>price
   ^price isNil
      ifTrue: [price := 0]
      ifFalse: [price]

MyStock>>price: aPrice
   | oldPrice |
```

```
oldPrice := self price.
price := aPrice.
self changed: #price with: oldPrice
```

The stock watcher adds itself as a dependent of the stock (the parent) when it is created. To do this it simply sends the stock an `addDependent:` message with itself as the parameter. Note that all the examples have the same `watchStock:` method so we will define it in the superclass, *MyStockWatcher*. The stock watcher has no instance variables so we are not showing the class definition.

```
MyStockWatcher class>>watchStock: aStock
   ^super new initialize: aStock

MyStockWatcher1>>initialize: aStock
   aStock addDependent: self

MyStockWatcher1>> update: aSymbol with: oldPrice from: aStock
   Transcript cr; print: thisContext; tab; print: aSymbol; tab.
   aSymbol == #price ifTrue:
      [Transcript
         nextPutAll: 'Price changed from: '; print: oldPrice;
         nextPutAll: ' to '; print: aStock price].
   Transcript flush.
```

We'll see the details of how this works in the next section, but briefly, the stock watcher is informed of all the changes and has to decide if the change is one that it is interested in. Since our stock watcher is only monitoring the price, it ignores all the other changes. If you run this code, you'll see that it is informed of several different changes, but only reports on the price change.

Underlying mechanism

There are some very sophisticated mechanisms that allow dependency to work but they all rely on the very basic mechanism of the parent sending itself a message from the `changed` family and the dependent being sent a message from the `update` family.

The dependency mechanism is implemented by Object, which means that any object can register itself as a dependent of any other object. When the parent changes, the most primitive mechanism for letting others know is to send itself `self changed`. This method is implemented by Object so let's see what it does.

```
Object>>changed
   self changed: nil

Object>>changed: anAspectSymbol
   self changed: anAspectSymbol with: nil

Object>>changed: anAspectSymbol with: aParameter
   self myDependents
      update: anAspectSymbol
      with: aParameter
      from: self
```

As you see, if your object simply sends the `changed` message, this becomes transformed into a `changed:with:` message with both parameters being nil. Usually you will provide more information by sending either the `changed:` or `changed:with:` message. If you want to inform your dependents what

changed, send `changed:` and pass a symbol as the parameter. When the dependent gets an update message, it will be able to look at the symbol and decide if it is interested in the change. If you want to go further, send the `changed:with:` message, passing the symbol and either the old or new value as parameters. This is what we did in our stock watcher example, sending the old value.

(There are two conventions worth knowing about. First, it's possible to send anything as the argument to `changed:`. However, by convention, most people send a symbol, the symbol being the selector that is used to retrieve the new value. Second, the usual convention is to send the old value in `changed:with:`, on the assumption that the dependents may no longer know the old value, but they can always get the new value by asking the parent. However, some people prefer to send the new value since there is less message sending involved. They take the approach that if any dependent cares about the old value they can store it, Whichever approach you take — sending the old value or the new value — make sure you do the same thing everywhere so that other programmers can rely on getting what they expect.)

Eventually the parent sends all its dependents the `update:with:from:` message, so the dependents must implement one of the `update` family of messages. What if you don't care about all the parameters of `update:with:from:`? Just as Object expanded `changed` up to `changed:with:`, it contracts `update:with:from:` down to `update`.

```
Object>>update: anAspectSymbol with: aParameter from: Sender
   ^self update: anAspectSymbol with: aParameter

Object>>update: anAspectSymbol with: aParameter
   ^self update: anAspectSymbol

Object>>update: anAspectSymbol
   ^self
```

Your dependent object thus has a choice of which of the three messages in the `update` family to override. Once you've implemented it, your method will be invoked rather than Object's method, since you are overriding Object's implementation. If you don't implement any `update` method, the results will be benign and you won't even notice that anything happened. In our example, we implemented `update:with:from:`. Figure 19-1 illustrates our example.

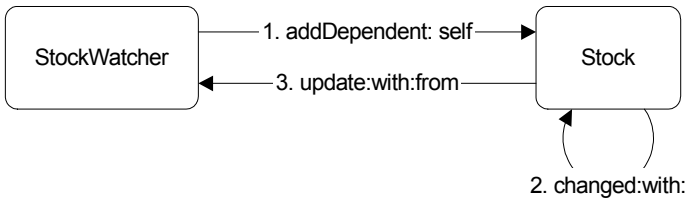

Figure 19-1.
`The addDependent:` mechanism.

Model

In the `changed:with:` method, Object sends the `myDependents` message to get a collection of dependents. Object has a class variable, *DependentsFields*, which is a Dictionary indexed by objects that have dependents. The general access mechanism is `DependentsFields at: objectWithDependents put: collectionOfDependents`.

The main problem with this scheme is that if an object doesn't remove itself as a dependent by sending its parent `removeDependent: self`, the dependents collection will never disappear from Object's class variable. (You can see if you are cluttering up this collection by inspecting Object's DependentsFields class variable.) Since the dependent objects continue to be referenced, they will not be garbage collected even though they are no longer needed. What we'd like is a mechanism that automatically gets rid of the dependency relationship when the parent is garbage collected, allowing the dependents to also be garbage collected when they are no longer needed.

This is where the class *Model* comes in. It is a subclass of Object, and has a single instance variable called *dependents*. An instance of Model keeps its dependents in a collection held by the *dependents* instance variable. This means that if the instance is garbage collected, the dependents collection will disappear even though dependents never send the `removeDependent:` message (however, it's good practice to always send `removeDependent:`). If you look back to example one, you'll notice that our stock class is subclassed off Model.

If you don't fully understand this, it doesn't matter. Just remember the basic rule: If you are going to use dependencies a lot, subclass your parent class off Model rather than off Object.

expressInterestIn:for:sendBack:

Notice that the `update:with:from:` method in Example One has to compare the symbol it receives with all the possible symbols it cares about (in effect doing a switch statement). The basic dependency mechanism suffers from the disadvantage that *all* dependents are sent *all* changed messages, even if they are not interested in most of them. Dependents must implement one of the `update` family of messages and must check to see if they are interested in the particular symbol that was sent.

We can eliminate the conditional checking that we saw in the update method by sending `expressInterestIn:for:sendBack:` instead of `addDependent:` to the parent object. In this approach, we tell the parent to express an interest in a particular change for us and send us a specified message when it sees the symbol associated with the change (i.e., the symbol sent by the parent when that particular thing changes). The syntax is confusing because you would expect the dependent to express interest in the parent. However, the syntax is such that the dependent tells the parent to express an interest in the dependent. Let's look at an example using this approach.

```
MyStockWatcher2>>initialize: aStock
   aStock
      expressInterestIn: #price
      for: self
      sendBack: #priceChanged:by:

MyStockWatcher2>>priceChanged: oldPrice by: aStock
   Transcript
      cr; print: thisContext; tab;
      nextPutAll: 'Price changed from: '; print: oldPrice;
      nextPutAll: ' to '; print: aStock price; flush
```

So how does this work? Rather than asking the parent to send us all changed messages and then filtering them out, we ask the parent to do the filtering, sending us a different message for each change we are interested in. If the message we asked to be sent has no parameters, we will just be sent the message. If the message has

one parameter (binary message or keyword message with one colon), we will be sent the old value as the parameter. If the message has two colons, as does our example, we will be sent the old value and the object that changed.

If you are interested in how this is done, the parent creates a *DependencyTransformer* and adds the DependencyTransformer as a dependent. The DependencyTransformer gets all the update messages and looks for the one we have expressed interest in. When it sees the change we care about, it sends us the message that we registered as the parameter to the `sendBack:` keyword. The DependencyTransformer is very clever when it comes to sending us the message. When we expressed interest in changes, the DependencyTransformer figured out how many parameters our message expects based on whether it is a binary message or how many colons are in the message. It implements `update:with:from:` so that it knows what changed, the old value, and the object that sent itself the changed message, and it sends us the appropriate number of parameters. Of course, if the parent didn't send `changed:with:`, one or more of the arguments will be *nil*. Figure 19-2 illustrates this mechanism.

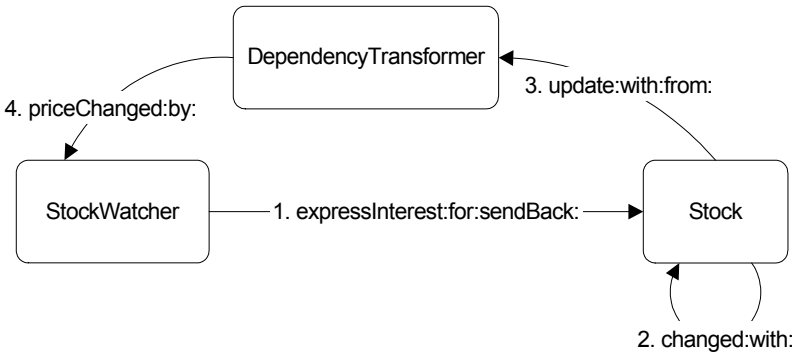

Figure 19-2. The `addDependent:` mechanism.

Finally, to stop being told about changes, the dependent has to retract interest in that particular change of the parent. It does so by sending the `retractInterestIn:for:` message, which should always be done. For example,

```
parent retractInterestIn: #number for: self
```

onChangeSend:to:

Hiding the complexity further is the `onChangeSend:to:` message, which is implemented by the abstract class *ValueModel*. It takes two parameters: the message that should be sent, and the object to send it to. Notice that we don't specify what change we are interested in. This is because a ValueModel handles both the expression of interest in changes, and the notification of changes, so it can make certain decisions for us. All that `onChangeSend:to:` does is express an interest in the symbol *#value*.

```
ValueModel>>onChangeSend: aSymbol to: anObject
   self
      expressInterestIn: #value
      for: anObject
      sendBack: aSymbol
```

So what is a ValueModel? It is basically a wrapper around a value — it holds the value. We can't change the value directly and can only change it through the ValueModel. Because the ValueModel has control of the change, it can notify dependents of the change. To set a new value, you send the ValueModel the `value:` message with the new value as a parameter, and `value:` simply sets the value and notifies any dependents by doing `self changed: #value`.

```
ValueModel>>value: newValue
   self setValue: newValue.
   self changed: #value
```

Because `onChangeSend:to:` translates into `expressInterestIn:for:sendBack:` and this latter message creates a DependencyTransformer, we now have a DependencyTransformer waiting for its parent to send `self changed: #value`. Once it receives this, it will send back the message we registered. ValueModel and its subclasses are widely used in the User Interface widgets.

Let's look at our third example, which uses instances of *ValueHolder*, the simplest subclass of ValueModel. In this example the *get* accessors return a ValueHolder (sending the `asValue` message to an object returns a ValueHolder holding the object). The *set* accessors go through a level of indirection because they can't set the value directly; they have to ask the ValueHolder to set it. So they send the `value:` message to the ValueHolder, which updates the value it is holding and informs itself about the change. Unlike the previous two examples, which inform their dependents of the old value, the ValueHolder mechanism provides no ability to do this. Figure 19-3 illustrates this mechanism.

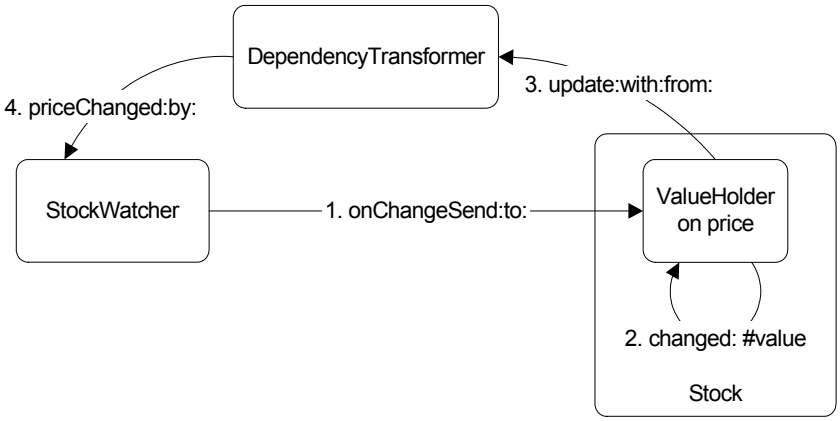

Figure 19-3. The `addDependent:` mechanism.

In the example we'll just show the accessors for *price* since the other accessors have the same structure, just different variable names. Additionally, the `traded:price:` message knows that we are dealing with a ValueHolder, so it sends the `value` message to get the current value of *totalTraded*.

```
MyStock3>>traded: aCount price: aPrice
   self totalTraded: self totalTraded value + aCount.
   self price: aPrice

MyStock3>>price
   ^price isNil
      ifTrue: [price := 0 asValue]
      ifFalse: [price]
```

```
MyStock3>>price: aValue
   self price value: aValue
```

If you look at the following stock watcher methods, you'll notice that we have an instance variable, *stock*, which we need to add to the class definition for MyStockWatcher3. Because ValueHolders do `self changed: #value` rather than sending the `changed:with:` message, they don't pass back a value. When our StockWatcher discovers that the price has changed, it needs to send the stock a message to find out the new price. To do this, it needs to keep hold of the stock in an instance variable. Also, because no value is passed, we don't have a way to track the old value unless we were to keep it in an instance variable.

```
MyStockWatcher3>>initialize: aStock
   stock := aStock.
   stock price onChangeSend: #priceChanged to: self

MyStockWatcher3>>priceChanged
   Transcript
      cr; print: thisContext; tab;
      nextPutAll: 'New price='; print: stock price value;
      flush
```

We could dispense with the instance variable if we are prepared to specify a two keyword message where we know the parameter to the first keyword will be *nil*. So we might write the above two methods as follows.

```
MyStockWatcher3>>initialize: aStock
   aStock price onChangeSend: #priceChanged:by: to: self

MyStockWatcher3>>priceChanged: aPlaceHolder by: aStock
   Transcript
      cr; print: thisContext; tab;
      nextPutAll: 'New price='; print: aStock value;
      flush
```

To cancel the dependency we set up with `onChangeSend:to:`, we send the `retractInterestIn:for:` message. For example,

```
parent retractInterestIn: #priceChanged for: self
```

Which mechanism should you use?

Some people use `onChangeSend:to:` for basic dependencies since it is an elegant message. However, for general dependencies the ValueHolder mechanism suffers from some disadvantages. To register as a dependent, an object must send `onChangeSend:to:` to a ValueHolder. This means that the parent object, which may be an instance variable, must hold a wrapper rather than real data, which is conceptually inelegant. A more serious disadvantage is that for objects to register an interest in a variable, you have to open up the details of the object. The data is no longer fully encapsulated in the sense that other objects now know how the data is stored.

With either the `expressInterestIn:for:sendBack:` or the `addDependent:` mechanism, the parents are free to change the way they store and manipulate their data, as long as they still notify themselves of changes with the same symbol.

ValueHolders and other subclasses of ValueModel are widely used in the User Interface widgets and you'll see a lot more of them in Chapter 23, Model-View-Controller, and Chapter 24, MVC Dependencies. However,

for basic dependencies, I prefer the other mechanisms: `expressInterestIn:for:sendBack:` and `addDependent:`. Of the two, my preference is for `expressInterestIn:for:sendBack:`. With the `addDependent:` mechanism you have a single method that receives notification of all changes to all parents. It's now up to your method to make sense of all the notifications, which can result in a very procedural-looking set of `booleanCondition ifTrue:` statements. On the other hand, `expressInterestIn:for:sendBack:` allows you a narrower focus on the specific problem. When a particular aspect of the parent changes, send me a particular message. When another aspect in a different parent changes, send me a different message. It's a nice object-oriented approach.

Broadcasting

VisualWorks provides the ability for a parent to broadcast a message to all its dependents. In our stock example, there may be cause for concern if a stock price changes more than thirty percent in a single trade. Who's responsibility is it to care about this? A good argument can be made that the stock itself doesn't care and that it's the responsibility of the stock watchers to handle the situation. For the sake of this discussion, however, let's make the case the stock cares that its price has risen more than twenty percent, and that it needs to inform its dependents.

The broadcast mechanism lets it do that. There are two messages, `broadcast: aMessage` and `broadcast: aMessage with: aParameter`. Since the stock watchers need to know which stock is sending the message, it makes more sense to use the second message. So, we might see code in the `traded:price:` method in Stock to do something like the following:

```
MyStock traded: aCount price: aPrice
   ... code to set values ...
   price notNil ifTrue:
      [(aPrice - price / price) abs > 0.3
         ifTrue: [self broadcast: #bigChange: with: self]].
```

This unilaterally sends the `bigChange:` message to all dependents passing the stock object as the parameter. If the dependents haven't implemented `bigChange:`, an exception will be raised. So there's a responsibility on the dependents to implement this message, even if they don't care about large changes in price. There's also an argument against specifically broadcasting a large change in price since one might argue that the responsibility of stock is simply to inform its dependents about changes in price. It's the responsibility of the dependents to decide if a given change is too high.

In the dependency scheme, the general philosophy is that dependents know about and care about the parent, but the parent doesn't know or care about the dependents. This philosophy breaks down with broadcasting because if the parent decides to send a new message, some programmer has to figure out all the objects that register as dependents, then add the new message to each of them. An alternative and better approach would be to modify the broadcast messages to do safe performs (look for implementors of `safelyPerform:` to see how this is done).

If you modify `traded:price:` to include the broadcast code without adding the `bigChange:` method to the stock watchers, then run the three examples, you will see some interesting results. In Example One, an exception will be raised saying that StockWatcher1 doesn't understand the message. In Example Two, it will be a DependencyTransformer that doesn't understand the message. In Example Three, no exception will be raised

because the Stock3 object has no dependencies — it's the instance variable ValueHolders that have the dependencies. So, to use broadcasting, you'll have to use the `addDependent:` mechanism. Note that none of the system classes send either `broadcast:` or `broadcast:with:`, and personally, I'd recommend against using them for the above reasons.

Dependency Considerations

There are two things to be aware of when using the dependency mechanism. First, parents should not send explicit messages to their dependents. Second, and most importantly, it's worth getting into the habit of always breaking dependencies when they are no longer needed.

Dependency violation

The dependency mechanism is supposed to be a one-way mechanism. The parent or model doesn't know anything about its dependents. It doesn't even know if it has any dependents. However, sometimes people write code so that the parent, in Jim Dutton's phrase, gropes its dependents. Here's an example.

```
self myDependents do:
   [ :each | (each isKindOf: SomeClass)
      ifTrue: [ each grope ] ]
```

This code violates the basic rule of dependencies — that a parent should have no knowledge of its dependents.

Breaking dependencies

When the product is almost complete, most organizations try to run it as they think customers will use it. Usually they notice that the product has memory leaks, and that it slowly consumes the available memory. Memory leaks occur because objects are not being garbage collected, and objects are not garbage collected if one or more other objects reference them. When dependencies are not broken, dependent objects are referenced by their parents. If the parent object is held in a class variable or a global, or is refererenced by another object in a class variable or a global, or is referenced by an object that is referenced by an object in a class variable or a global, etc., the dependents will never be garbage collected. Sometimes it is extremely difficult to track down exactly why an object is not being garbage collected because the reference path can be very long. In fact it's more difficult to track down memory leaks when the parent is a subclass of Model because the dependencies of non-Models are tracked in one place, in Object's class variable DependentsFields.

To reduce the chances of memory leak, you should get into the habit of *always* breaking dependencies when the dependent is no longer needed or the dependent no longer needs the dependency. If the dependency was set up using `addDependent:`, break it with `removeDependent:`. If it was set up with `expressInterestIn:for:sendBack:` or `onChangeSend:to:`, break it with `retractInterestIn:for:`. The breaking of dependencies is usually done in a `release` method such as shown below. In the example, the MyClass object removes itself as a dependent of another object, then tells one if its instance variables to break any dependencies it may have, and finally gives its superclass a chance to do any releasing.

```
MyClass>>release
   someObject removeDependent: self.
   self someVariable release.
   super release.
```