Chapter

32

Changes to System Classes

Sometimes it is useful to modify system classes. To be able to add a method to Object that everyone can inherit can make life much easier. In fact, in several chapters in this book we've seen new methods added to system classes. There are several kinds of change you can make and we'll look at them in this chapter.

Some people feel it is inappropriate to modify system classes for various reasons. The arguments listed below are all valid arguments, and after reading this chapter you will have to decide where your own beliefs lie.

- If you change a system class then get a new release of the system software, your change will not be there. It may be difficult to put back in because of possible restructuring of system classes. You may simply forget to reimplement the change and this may not be discovered until a customer uses the software.
- The system classes are highly reliable. Any change you make could detract from that reliability.
- Programmers won't know that certain behaviors were added and will expect to find the changes in other projects they work on.
- Programmers may expect a method to behave a particular way, then discover too late that the method was modified to behave a different way.

Subclass where possible

It's better to subclass off a system class than to add a method or modify a method. It's not always possible to subclass, as we'll see, but the failure to subclass is often a result of not attempting to subclass. For example, when I originally extended the VisualLauncher with a few changes, I simply added new methods. Only later did it dawn on me to subclass from VisualLauncher. Besides the virtues of not modifying system classes, subclassing has the additional virtue that it is obvious when programmers are using a non-standard class.

If you want to add behavior to Object that you want inherited by all your application objects, consider creating a new class, *MyApplicationObject*, that is subclassed off Object. If you subclass all your application objects from MyApplicationObject, you can now add application specific behavior that all your objects will inherit, without having to modify Object in any way. We saw an example of this in Chapter 5, Instance Creation.

Types of changes to system classes

There are two facets to consider when talking about changing system classes. First is that some changes will be to classes used in the development environment, while others will be to more fundamental classes that are used in your deployed application. The other facet is that there is a difference between adding methods to system classes and modifying existing methods.

Development environment vs. deployed classes

There are changes that you can make to your development environment that will never be executed in a deployed application. More importantly, there are changes that you can make to your development environment that don't have to be incorporated into the production application. For example, you might make changes to remap your keyboard for development, to allow resizing of Browser panes, or to add more sophisticated breakpoints. None of these changes have to be included when building the application for testing and deployment.

Added vs Modified methods

Adding methods to system classes is a lot less risky than modifying existing methods. When you modify a method, you have to be very careful that your change will not cause harm to any of the senders of the message. Sometimes this is easy to determine, sometimes difficult. Avoid modifying a method in a way that will confuse programmers who expect certain results from message sends. To use an extreme example, if someone modified `add:` to return *self* rather than the object added, the result is likely to be confusion and code that no longer works.

Adding functionality by subclassing is the right thing to do when possible, but it's not always possible or easy. Suppose we want to add `trimWhiteSpace` to remove leading and trailing white space from a string. It's not easy to add a subclass of String because `String new` actually returns a subclass of String, and creating a literal string, such as 'hi there', will not create an instance of our new subclass. So, it's much easier to add `trimWhiteSpace` to String.

In fact, I'm all in favor of adding methods as long as they are well thought through and well tested. I'm also okay with modifying methods to extend functionality as long as the old expected behavior is preserved. For example, in Chapter 16, Processes, we looked at the idea of subclassing off Process. Unfortunately, the method `Process>>anyProcessesAbove:` assumes that there are no subclasses of Process, so in order to subclass off Process we had to modify this method to operate on instances of Process and its subclasses.

If you do have to modify a system method, try to put the smallest possible change in the modified method, and most of the processing in another method (on one of your objects if possible). You might end up with something like the following.

```
SystemClass>>someModifiedMethod
   ...
   self doNewThing.
   anInstanceOfMyClass doAnotherNewThing.
   ...

SystemClass>>doNewThing
   "Here's where most of the new code might be"
```

```
MyClass>>doAnotherNewThing
   "Here's another place where most of the new code might be"
```

Summary

- Subclass where possible.
- Be comfortable adding well-tested methods to system classes.
- Don't worry too much about modifications to system classes for development environment changes if you don't need those changes in the deployed image.
- Selectively modify fundamental system classes as long as you are extending functionality rather than modifying functionality.

Distinguishing your additions and modifications

For two reasons, it's important to be able to distinguish system methods you've added and modified from base system methods. First, it's only fair to give programmers information that they are using non-standard methods. Second, when you get a new release, you need to be able to identify the changes you've made so that you can add them to the new image.

There are two mechanisms I use. The first is to group all added methods into protocols called `(additions)` and to move all modified methods to protocols called `(modifications)`. Yes, the parentheses are part of the name — it makes the name stand out. Chapter 29, Meta-Programming, tells you how to find all classes with protocols named `(additions)` or `(modifications)`.

However, I don't do this for development environment changes that I get from the Smalltalk archives or similar places. Some of these fileIns make such extensive changes to system classes that it's more important to make sure they are only in development environments and not in production applications.

The second mechanism I use to distinguish my changes from the standard classes is to place one of the following statements as the first executable line in the methods (the method definitions are shown later in the chapter). By noting new methods and changed methods in this way, it's easy to find all the new or changed methods by browsing references to either `newMethod:` or `changedMethod:`.

```
self newMethod: 'Date. Author. Why'.

self changedMethod: 'When. Who. Why'.
```

Why use both the message sends and the protocol name? In part, to increase my chances of tracking all the changes that I have to carry forward to new releases. However, the two mechanisms do serve different purposes. When I want to find a method I wrote but can't remember the details of, looking for message references will get me there quickly. If I'm browsing a class, the special protocol names tell me immediately that there are extensions or modifications to the class. By grouping added and modified methods, I can easily file out all the changes to system classes by choosing the *file out* menu option from the protocol window.

Some useful additions

This next section shows a few minor additions to *Object* that I've found useful. We've seen additions and changes to system classes in other chapters, but we are not repeating them here.

The following methods make it much easier to print information to the Transcript. Some people find it easier conceptually to use the `echoYourself` methods, but once you get used to the idea of objects printing themselves to the Transcript, it's less typing to use the `echo` methods. The code is based Ernest Micklei's echo.st contribution to the Smalltalk archives.

```
Object>>echo
   Transcript cr; nextPutAll: self displayString; flush

Object>>echo: anObject
   Transcript
      cr; nextPutAll: self displayString;
      space; nextPutAll: anObject displayString;
      flush

Object>>echoYourself
   self echo

Object echoYourselfAnd: anObject
   self echo: anObject
```

The next few methods don't do anything. They exist simply so that you can find all references to them.

```
Object>>newMethod: aString
   "Does nothing. Used to find methods added to system classes"

Object>>changedMethod: aString
   "Does nothing. Used to find system class methods that have been
changed"

Object>>obsoleteMethod: aString
   "Does nothing. Used to find obsolete methods"

Object>>optimizePoint: aString
   "Does nothing. Used to find methods that need additional work"
```