Cloning, Classes and Inheritance in the Offsider Technology

Cloning, Classes and Inheritance in the Offsider Technology

Object oriented programming, in its original and purest sense, means that style of programming where the only mechanism available to the programmer is to send messages to objects. You send a message to an object, the object responds to the message.

Despite what you might have read in *Introduction to Java* tutorials, or the like, classes and inheritance are not an essential part of this paradigm.

In fact, classes and inheritance are just one of a number of mechanisms that can be used to provide something called **code reuse**. From an object-oriented point of view, this provides a means of saying *"**ObjectA** is in some way similar to **ObjectB**"*. In practice, it enables us to create objects which behave in the same way when passed the same message.

Cloning

A number of object oriented programming languages (eg, **io**, **self**) do not have classes or inheritance at all. Instead, you are able to create an object by cloning an existing object.

This is the default behaviour provided by the **Offsider** framework. If you have an existing offsider, `foo` you can clone it to produce another object which is an exact, deep copy of `foo`.

```
foo clone bah
```

Any methods in `foo` are copied to `bah`.

The advantages of this approach are:

. **It is simple and straight forward.** It is easy to implement, and easy to understand.

. **Each offsider is completely self-contained.** You can move the offsider around within the filesystem without breaking it. You can create a tar file and copy it to another machine without breaking it. Much effort has been expended in ensuring that offsiders have this property. Cloning is part of this effort. (Note, we are talking here about the offsider infrastructure. The named executable will need to be rebuilt after each move or copy).

. **You can now modify both `foo` and `bah` without effecting the other.**

Disadvantages are:

. **You use up more disk space.** In fact, this is usually insignificant. Do the maths to work out what the cost is on your system

. **It can make code updates more complex.** If you discover a bug in a method, you will need to fix the bug in every object that has a copy of that method.

The comparative importance of the various advantages and disadvantages will depend on your system, your requirements, and your personal preferences. In my opinion, having self-contained offsiders is very important. On the other hand, the problem of code updates is also important, and significant.

Other methods for copying methods

Cloning an object will create an exact, deep copy of that object - including all its methods.

The Offsider framework provides a number of additional ways to reuse code by copying methods from other sources. All offsiders will respond to the following methods:

`method`
copy a file (or stdin) into this object, and make it into a method.

`allFrom`
copy all methods from another object into this one (overwriting any existing methods)

`moreFrom`
copy all methods from another object into this one (but don't overwrite any existing methods)

`upgradeMe`
copy files from the current directory into this object. Can be used to create or upgrade methods from a source directory.

The advantages and disadvantages are similar to those for cloning. On the other hand, methods like `allFrom` and `upgradeMe` can be used to propagate code updates to other objects that need them, thus addressing one of the drawbacks of cloning.

Taking advantage of the default Offsider method-search strategy. *faking Inheritance.*

By default, if an offsider does not have a particular method (say `foo`), then it will look for an executable called `Dictionary.foo` in its execution `$PATH`.

You can take advantage of this fact to implement code-reuse without cloning (or inheritance, or classes).

A good example of this is in the **Weave** technology. `Weave` is an Offsider which is a navigatible object database. Each object in the database is itself an Offsider. `Weave` provides dozens of methods which each of its database items can use. These are all stored in a special directory, within the Weave infrastructure. Whenever one of the database items is sent a message, the Weave adds this directory to

the `$PATH` environment variable, thus ensuring that the item will find that method. Specifically, look at `Weave`'s `event` method.

In general, you can create a directory somewhere. Into this directory, place executables with names like `Dictionary.`*`xxxx`* . Now, whenever you want an object to find methods from that directory, you need to make sure that this directory is in the `$PATH` variable. For example, you can set the `$PATH` variable from within that object's *named executable*.

During the normal creation of a new Offsider, the *named executable* is created by calling the method `makeExecutable`. You can override this method to make this setting of the `$PATH` variable automatic. Now you can use `makeExecutable` to rebuild the executable for that Offsider.

The advantage of doing it this way, is that now when you clone that offsider, the new offsider will get a copy of the new `makeExecutable` method, so when the *named executable* gets built, the `$PATH` variable will be set correctly within it. Therefor, the new offsider will recognise the methods in the special directory, even though it has not made copies of them. You thus get code reuse without copying any methods (except for the `makeExecutable` method).

Over-riding isMethod - *meta-inheritance*

The `isMethod` method is used to find the actual executable for a particular method name for a particular object. You can over-ride this method for an object (or a group of objects).

Again, an example of this technique is found in the **Weave** technology. Every item (event) in the Weave database can subscribe to one or more *templates*. A *template* is an object, internal to the Weave infrastructure, which will supply methods for any Weave event that subscribes to it. The mechanism for doing this is coded into the `isMethod` method that is supplied by the Weave for its events.

In general, you can use `isMethod` to provide alternative strategies for an object to locate methods. In particular, you can use it to implement *inheritance*.

Let's suppose you have created an offsider, and provided it with some form of inheritance, by overriding `isMethod`. If you clone this offsider, the new offsider will use this version of `isMethod`. Therefor, it will exhibit the same method-lookup behaviour. We might call this *meta-inheritance*, since the new offsider will inherit the original offsider's mechanism for inheritance.

Classes and Inheritance in general

Classes and inheritance are often interconnected in object oriented languages, but they are really unrelated concepts. A *class* is something that build objects. *Inheritance* is a mechanism for implementing code-reusage in a heirarchical manner. Often, classes are used to supply inheritance, but you can implement classes without inheritance (think **JavaScript** constructors), or inheritance without classes (see above, where we describe *fake inheritance*, and *meta-inheritance*).

Neither classes nor inheritance are implemented by default in the Offsider framework. There are a number of points that lead to this decision..

. there are many different models around for both classes and inheritance. which one do you use? How do you choose?

. the solutions that make sense for a programming language generally don't make sense for the Offsider framework.

. problems that a programming language faces, which are solved by inheritance, are not problems that the Offsider framework faces.

. Offsiders are designed to provide maximum flexibility to the application programmer. Why restrict this flexibility? On the other hand, given that flexibility, how can we ensure that any given class/inheritance strategy will work in every case?

I guess what I am saying is that Offsiders provide all of the neccessary tools (methods) for implementing classes or inheritance in whatever way makes sense to you, but have chosen not to impose any solutions by default. Another way of saying this is the Offsider framework provides *"mechanism, not policy"* with respect to classes and inheritance. Another way of saying this is that ineritance and classes are an **application** consideration, not a **framework** consideration.

Given that Offsiders represent a new and different paradigm, (and in particular, different from using an object oriented programming language), it made sense to understand the paradigm more before jumping to arbitrary solutions.

Having said that, there is already an application solution that provides both classes and inheritance. The Object/Class implementation (described below) is a very elegant and simple solution which provides Classes that work very much as expected. Unfortunately, there are serious long-term problems with them that are not immediately obvious.

The `Object` Offsider *(inheritance)*

One demonstration of the power and flexibility of the Offsider framework is an offsider which goes under the name of `Object`. `Object` implements one form of inheritance.

`Object` provides two simple methods.

> `isMethod`
> **which overrides the default isMethod, and implements inheritance.**
>
> `addInheritance`
> **which allows you to manipulate the inheritance pathways.**

In addition, `ObjectHelper` is an offsider that provides a couple of extra methods that are somewhat similar to `super()` in **Java**.

It is instructive to look at both of these offsiders. Not only will it show you how the Offsider framework can be used to provide novel solutions, it will also highlight the way in which this framework is very different from a programming language, and how what is straight-forward in the context of a language is not at all straight-forward for Offsiders.

The `Class` Offsider

`Class` is an offsider that is based on the `Object` offsider. It implements classes which

. can be used to instantiate Objects,

. provide both class methods and instance methods,

. can be subclassed,

. provide inheritance properties similar to those you might find in a class-based object oriented programming language.

Again, it is instructive to look at how this offsider is put together. Starting with the `Object` offsider, the entire `Class` framework is provided by a 7-line installation script, and two simple methods, consisting of 44 lines of shell-script (including comments - removing comments and blank lines you are left with 12 lines of actual code).

The `Class` Offsider give you the ability to create new classes that work very much as classes in familiar object oriented programming languages, but that convenience comes at a very real cost.

Because the inheritance linkage between the various objects and classes is hard-wired, it means that none of the offsiders can be moved from one place in the filesystem to another without breaking that inheritance. In addition, since we are now forcing offsiders to rely on other offsiders, we can no longer copy individual offsiders to another machine - we need to copy the entire collection (and we are still faced with the problem of hard-wired inheritance).

This may not seem like such a problem, (because it is not a problem that you would encounter with a programming language), but when you consider that offsiders are persistent (and therefor permanent), you might find that there are long-term implications that you have not foreseen.

In summary

*Beware the seduction of **familiarity**! It is but an illusion.*