Some strategies for offsider development

Offsider development is quite different from application development, and has its own unique difficulties and pitfalls. Over the last year or so, I have developed many offsiders and offsider-based systems. Based on this experience, I have come up with a few strategies that I have found useful. I share them with you here.

Please keep in mind that there are almost certainly other ways of approaching offsider development. This is just one way.

Development

Basic approach

The basic approach I take is that I always keep my development code completely seperate from my operational offsiders.

This inevitably leads to duplication of code (multiple copies of scripts, for example), but that is okay.

The advantage of keeping the development code seperate is that I can easily recreate the offsider if it somehow becomes corrupted. It also makes maintenance much simpler, especially in situations where the offsider is a clone of, or is otherwise based on, some other offsider(s).

This approach is in direct contrast to, for example, the Quick-start tutorial, where you develop an offsider by editing the methods directly.

How I organise my development code

The development code is in a completely seperate directory, usually in a completely different part of the file-system. You can decide for yourself how you are going to do backups and version control. I simply make time-stamped tar archives when I deem it appropriate.

Within this directory, I typically have the following:

`README`

This is for me. It is usually the first file I create. I write in it my thoughts about what the offsider is supposed to do, why I need it, and initial thoughts about how I intend to implement the functionality, and what messages the offsider will support (ie, its interface).

`INSTALL`

This is an executable, usually a shell script. It will create the offsider if neccessary, based on the development code in this directory. Otherwise, it will upgrade the offsider with the latest changes. If the offsider requires code from other offsiders, it will also refresh that code as well. This means that if those other offsiders change, then this one will reflect those changes. The bottom line is that this script should always do the right thing, no matter what the situation. Obviously, I need to think carefully about how this script is written, but once written, it becomes an invaluable tool for keeping the production offsider completely up-to-date in all circumstances.

`methods/`

This is a directory. Within this directory are all the methods that the offsider will implement. You will recall that *`foo`* `upgradeMe` will automatically copy methods from this directory. This is what the `INSTALL` script does. This is also what I do to just upgrade the latest method changes, without doing a complete INSTALL (which can be quite complex for really complex offsiders).

`keys/` and `pairs`

In my experience, these are less likely to be required. Some offsiders, however, make use of a very specific set of keys, and it is sometimes useful to pre-load them with default settings (even if they are empty).

`upgradeMe` can create keys, and place values in them. Two mechanisms are available, and both will happen by default.

You can place individual files in a subdirectory called `keys/`.

In addition, you can create a file called `pairs`. Within this file, put lines of the form:

`keyname``:` *`value for that key`*

(To help with documentation and maintenance, blank lines and comments, which are lines starting with the character `#` can be used, and will be ignored.)

Be aware, however, that `upgradeMe` will over-write the values for any keys upgraded in this manner. This can be a problem when upgrading existing offsiders that are being used in production.

The solution I use is to provide methods called `saveKeys` and `restoreKeys`.

`saveKeys` does:

```
this pairs | this attachment savedKeys
```

`restoreKeys` does

```
this getAttachment savedKeys | this cli noprompt noecho
```

The `INSTALL` script will run `saveKeys` before each installation if the offsider already exists, and will run `restoreKeys` at the end.

In this way, any new keys will get written to the offsider, whereas existing keys will be restored to their existing values.

In addition, I can run either of `saveKeys` or `restoreKeys` by hand as I see fit.

Other

This depends on how complex your offsider is, and how you want to work.

If my offsider has child offsiders, I generally create seperate subdirectories for each child offsider. Sometimes I will have all these subdirectories together in a subdirectory called `childOffsiders` or similar. The `INSTALL` script will `cd` into the appropriate subdirectory, and then do (for example) *`foo bah`* `upgradeMe` to upgrade child offsider *bah* of offsider *foo*.

Often I will provide each of the subdirectories with its own `INSTALL` script, which the main `INSTALL` script then calls. This means that I can run the `INSTALL` script that is in the subdirectory that I am currently in, and it will do the appropriate thing for the code I am currently working on. I can also use *`foo bah`* `upgradeMe` to provide running upgrades of methods that I am modifying.

After doing lots of changes, I will generally run the top level `INSTALL` script again just to make sure, in case I have missed anything.

If the functionality of my offsider has more than one well-defined area, I will sometimes use seperate subdirectories for the different functional areas. For example, I might seperate data retrieval from reporting from cgi-interfaces. This is a matter of personal taste.

Useful tools and methods

The offsider framework provides a number of useful methods that can be used in the `INSTALL` script, (or by hand). Understand these methods, and set up your development directory structure to make use of them. The methods most likely are:

`createStandard`

Will make sure that an offsider exists in a standard location, creating it if neccessary.

`create`, `clone`

Will create an offsider from scratch (`create`), or make a clone of an existing one (`clone`). Unlike `createStandard`, both of these methods will fail if the offsider already exists. Put the command inside an `if` test. (`toBaseDirectory` can be useful for testing whether an offsider exists. Pipe `stderr` to `/dev/null`).

`upgradeMe`, and friends

These methods are specifically designed for copying executables and data from the current directory to an offsider. In particular, `upgradeMe` is very easy to use, but quite flexible. By default, it will copy methods and keys if they are present. This method will also recognise (and run) `pre.upgrade` and `post.upgrade` scripts, if they are present in the current directory.

`allFrom`

This method will copy all the methods (or other material, like keys or attachments), from one offsider to another. This is useful if your offsider is based on another one, and you want your `INSTALL` script to always upgrade the latest changes. Obviously, you should upgrade from other offsiders before you upgrade any methods specific to this offsider, in case methods for this offsider override methods from the other ones.

There is also a method called `moreFrom`. This is not of any particular use for doing code upgrades.

Testing and Debugging

One of the advantages of having seperate development and production code is that you can do quick and dirty changes to your production code (to put in traces and so on), knowing that you can undo those changes simply by running the `INSTALL` script. It may sound counter-intuitive (or even dangerous!)

to be putting traces into so-called "production" code, but it works. (If there truly is a production system, you would be upgrading from your development code into a test offsider anyway.)

Obviously, the development directory is also a good place to put test suites and so on.

Templates

Templates are a concept that crops up in offsiders, and can influence how you set up your development work.

The offsider framework doesn't have a default concept of class or inheritance. Instead, we use the concept of cloning. Thus you sometimes create offsiders which are never intended for production use, but are essential for setting up production offsiders. A good example is `Weave`, which can be found at its **SourceForge Project Page** `(http://weavedb.sourceforge.net/)` . `Weave` implements a navigatable object database, and so can be used to store data. However, you would never actually use `Weave` to store data. Instead, you use `Weave` to create clones, and then use those clones to store the data.

Therefor, from a usage point of view, templates have a similar purpose to classes in a typical object-oriented programming language. (However, from the framework's point of view, there is absolutely no difference between an offsider that is intended to be used as a template, and an offsider that is not.)

The reason that templates can influence your development is that sometimes you will develop a template, which is then used to create your production offsiders. Thus your development code will contain material for you template, and your `INSTALL` script may well upgrade your production offsiders *in addition to* your template. This will become obvious as you become more familiar with the framework in more complex situations.

To use object-oriented application development as a comparison, the way it is often done is that you define a class, and then use the class to create objects that derive functionality from the class. There is, however, one big difference between offsiders and object-oriented applications. With application code, the objects are all create from scratch whenever the application is run. With offsiders, objects stay created, and will even survive rebooting the machine. Thus you need to put more thought into what happens to the offsiders as development of the templates progresses (a problem that doesn't even arise for object-oriented application development).

Documentation

Last (ironically enough!), but not least, is all that user and system documentation, web pages and so on. Obviously, the development directory is a good place to put all that stuff.

In conclusion

Since offsiders persist permanently, you need to think differently about the development process. I have elaborated some of my thoughts, based on experience, about how you might approach offsider development.

You might have different thoughts, or experiences. If so, I would like to hear from you. Please contact me if you have anything to add, your input is valued and may well help other developers.

Cheers, Glen. `glenelg.smith@gmail.com` Feb 2010