What is an offsider really? Object, System or Database?

The *Offsider* technology is a totally new concept, paradigm and way of working. As far as I can tell, no-one currently works this way.

This immediately leads to the problem of how to explain the concept. There are currently no accepted words or phrases to describe this new concept, and furthermore, there are no examples that I can draw on to adequately describe or illustrate the concept.

This rant is an attempt to address that problem by comparing offsiders to some existing concepts, and elaborating how they are similar, and how they are different.

Hopefully this will get you a better idea of what offsiders are really. Keep in mind, however, that the only way you can understand offsiders truly is to use them for some practical purpose.

Offsiders as objects

Alan Kay, the person who invented the term *object-oriented programming*, and who designed and implemented the first intentionally object-oriented programming language, has said that the two most important concepts in object-oriented programming is the concept of an `object`, and the concept of a `message`.

He is quoted as saying (**definition of object oriented** `(http://c2.com/cgi/wiki?AlanKaysDefinitionOfObjectOriented)` :

OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme LateBinding of all things. (**Alan Kay** 2003)

Much of that quotation applies directly to offsiders (although protection and hiding of state-process cannot be enforced since everything is exposed through the file system).

In particular, note the qualities of *only messaging*, *local retention* and *extreme LateBinding of all things*. These are definitely properties of offsiders.

There are a number of problems that arise from saying that offsiders are persistant objects.

What are objects?

The first is that there are numerous programming languages that claim to be object-oriented (and even some, like `JavaScript` which don't claim to be, but effectively are), and yet very little common ground between them. Fundamental concepts like *messaging* are foreign to languages like `Java`. Some languages (`io`) do not implement classes, others (`JavaScript`) don't implement inheritance. Amongst languages that do implement classes, each has a different idea of what a class is and what its properties are.

So the first problem is that each programmer will be thinking different things upon hearing the term *object*. More than that, most programmers will be thinking specifically about `Java`, which is probably the worst possible match to how offsiders work (no messaging, exposure of state-process, extreme early binding of all things).

Nevertheless, offsiders are most definitely objects, and your only interaction with an offsider is by means of messaging. It's just that they may or may not behave like your own particular language of choice.

not application programming

The second difficulty is that all programming languages are specifically about application programming. In particular it means that your focus is on a single executable running as a single process. (`unix` programmers will be aware of `fork`, of course, but even so the application code is focused on a particular process. This becomes explicit when using `fork`, because the code has to determine exactly which process it is. Similarly, inter-process communication is just that - communication between running processes.)

Therefor, all objects exist together within a single process. They are not visible to, or available to any other process (without a lot of trouble), and they do not persist when the process dies (unless you put in the code to make that happen).

Offsiders, on the other hand are by their nature persistent and therefor permanent. They are not tied to any particular process. In fact, they exist completely external to any process (offsiders NEVER get instantiated into memory).

not a programming language

The third point is that the offsider framework is not a programming language. There is no grammer, no enforcable default syntax or semantics, no control structures, integer arithmatic, and any of a hundred other features that any language would provide. There is absolutely not any form of garbage collection, and it is hard to see how there ever could be.

There is precious little scope for optimisation (automatic or otherwise) at the framework level.

Also, you can't create a parallel implementation in the language of your choice. What I mean by that is that you can't implement a `Java` object *`foo`* with method *`bah`* and have it be an alternative way of interacting with an `offsider` object *`foo`* with method *`bah`*. (Actually, you can, but only if every method in the `Java` object is nothing but a very thin wrapper to the external `offsider` object).

The offsider framework is not a library per se, so you can't bind it to a language either. It might be possible to go some way down this track, but you soon run out of options, and are forced to just use the raw executables - which isn't of any particular advantage.

The point about a programming language is that it generally uses a language engine. Most compiled languages embed common code into every executable, `Java` has its own runtime, scripting languages use an explicit engine to interpret the code. This engine is in complete control, and so can manage things like pointers, garbage collection, type safety (for those of you who still believe in that) and so on. This in turn means that things like inheritance are managable because the engine sets up all the objects and knowns exactly where all of them are at all times. It can do garbage collection because it knows when an object is no longer being pointed to. None of this is possible for offsiders. First because there is no central authoritative engine controlling and keeping track of everything. Second because there is no way of knowning when an offsider is no longer needed.

On the other hand, a lot of the things that languages provide (like, for example garbage collection and class inheritance) are really driven by limited memory resouces. For offsiders, this is just not an issue. A typical computer system will have many orders of magnitude more file-system space than memory space. It is very difficult to fill up your hard-drive with offsiders. (It's hard enough to fill up a usb stick with them!).

expectations

I am labouring the point here, but I will repeat. Offsider development is not like using any existing programming language. One of the reasons to be so insistent on this point is that if you don't understand this fundamental difference, then your expectations for the framework will be unrealistic. Offsiders offer power and flexibility in systems development that no programming language currently offers. On the other hand, offsiders can't possibly compete with any programming language in terms of:

speed Offsiders were not designed for speed, they were designed for persistence, flexibility and ease of use. The were designed for extreme late binding of everything. They were designed for simultaneous concurrent access by multiple processes.

familiar features I have already mentioned things like class inheritance, garbage collection and control structures. These are natural, and even neccessary for most languages, but don't really fit well into the offsider paradigm. You shouldn't miss them, because they are simply not neccessary. You use computer languages to write the individual methods. The offsider framework ties it all together so it works as a single object.

So are offsiders objects?

Yes, yes and yes. You interact with an offsider by sending it messages. It's just not like any object-oriented programming language you have used.

Offsiders as systems

You can think of a computer system as a collection of inter-related data and executables. For example, a web server will have the server itself, the startup script in `/etc/init.d`, and any `perl` scripts you have written to query the log files, do maintenance and so on. It will also have data, in the form of configuration files, the error log, the access log, `.htaccess` files and whatever.

This is exactly what an offsider is. It is a collection of inter-related data and executables.

The difference is that all interaction with an offsider is by means of messages, and all of the data and executables are stored in the same place, as the offsider infrastructure, within the offsider's base directory.

You can work around the limitation that everything needs to be stored within the base directory (just write methods that point elsewhere). You can ignore the message interface and access the data and methods using the raw `unix` toolset.

This analogy also puts into context the speed and resource usage issues that the framework has. After all, do you really care if the startup method takes 0.1 seconds more than an equivalent script in `/etc/init.d`? It's only going to get run once every 10 years, after all! :)

There are other, more subtle differences, however. The object-oriented nature of the framework means that when you build systems using offsider techonology, it will tend to be much more modular. It will tend to have lots of smaller executables, which reference each other. A complex system will also tend to grow lots of subsystems which are themselves offsiders (perhaps implemented as child offsiders of the overall system). Your configuration data will also tend to be stored as individual keys, rather than in a single and rather complex configuration file, which needs special tools for parsing, and special care for editing. You will be more likely to provide individual methods for performing complex or tedious configuration changes.

Something else you will discover is that interactive development becomes much easier and more natural. It is trivial for you to write and immediately test individual methods. This can be done without the need to shut down and restart anything. Offsiders themselves are very interactive. You don't need to connect to a server, login to a session or anything like that. You can make changes to methods or data interactively, and those changes will take immediate effect, even for processes that are still running.

Again, the `Weave` is instructive. This is a system of moderate complexity. The way it has been implemented is quite dissimilar to how such a system would normally be designed and implemented.

One Weave I implemented handles about 5,000 new items per day. It is in constant use, and is a key component in a production system with 24/7 uptime. After its initial installation I was able to continue development on the system without disrupting the system or putting it out of action for any length of time. It is difficult to describe how different this development paradigm is, but I can assure you that it is significant.

The difference is in the paradigm, and the framework provided.

Offsiders as databases

Finally, we consider an offsider as a database.

It is obvious that an offsider can contain data. It implements keys and attachments by default, and both of them are by nature data. You can also create heirarchical structures by implementing child offsiders.

You can create an offsider that acts as a template for an offsider that can resemble a row in a table (just a set of keys, really). You can then create an offsider that uses the template to build a collection of child offsiders, each of which represents one row. This offsider models a table. You can then use a collection of these tables to model a relational database. Issues of data integrity can be implemented by appropriate methods. Have a standard template for a row, another for a table, another for a database. Now create databases and tables as required.

You can create an offsider which acts as an interface to a *real* database. For example an offsider that is simply a front end to an SQL server. Or an offsider that is part front-end, partly implements the data itself, partly retrieves the data from the net using `http`. The options are limitless, and you can hide the complexity from the user by providing a single, simple generic interface.

You can also implement completely non-standard data structures. For example, the `Weave` manages a collection of items implemented as anonymous offsiders. It provides the capability of creating navigation pathways between those items, which enables you to implement arbitrary data structures as required. By implementing appropriate methods, you also ensure the integrity of those structures.

You can also provide a database interface by providing methods for updating and retrieving data. These methods in turn can implement any consistency checks that are appropriate.

You can provide methods for report generation, methods for web-based interaction. You can have individual items within the database provide their own methods that override the standard methods.

An implication of this way of thinking is that cgi scripts are designed somewhat differently. The typical way to write such a script is that the script connects to a database, issues a data request, retrieves the data, reformats the data as `html`, adds an `http` header and sends the document back to the client.

The way to do it with offsiders is that the sgi script sends a message to an offsider somewhere, and the offsider sends back the complete `http` document, with headers. The cgi script simply sends that document back to the client unchanged. The cgi doesn't need to know anything about the database or the significance or meaning of the data (information hiding). In its turn, the database itself might create the document outline, and then leave it up to each indivudual data structure or item to fill in the detail, adding formatting or other markup as deemed appropriate by that item.

Conclusion

The purpose of this document is to give you broad perspective. Different ways of thinking about this strange, new technology known as **offsiders**. Hopefully it will have alerted you to the fact that this is indeed different, and to beware of making too many assumptions based on the rather misleading terms I have been forced to use for want of better ones.