A class oriented interface for Unix systems

Unix is an operating system that is based on text files for configuration.
Configuration is the act by which a generic piece of software is adapted to the rest of the system and the need of the user.
Unix is the only operating system that is worth targeting, because it is the only operating system that is installed on real computers. Windows sucks.
Other operating systems are theoretical as far as i see.
Most important software out there uses the POSIX API. And even if the above statements are not true and 235 people show me 235 other operating systems, I am happy with the massive amount of software that exists for Unix.
Despite that, Unix gained popularity even on end user systems and although many improvements are made in the field of operating systems, POSIX compatibility is nearly a must for such advancements. GNU Hurd is a POSIX compatible system.

Configuring software on Unix systems means editing some configuration file that is read by the parser of a program. Another way is to give command line arguments to programs. That is a good solution so far, because there is no need for a special configuration program to configure software, an editor is all that is needed.
The problem, or should i say nightmare that arises comes from the fact that every software that gets written for Unix must choose what format that configuration file should be written in, because there is no standard that suggests a proper way. The good thing about Unix is its standardization.
Every Unix compatible system out there supports a set of standard commands and if they are not standard enough, it is practically always possible to install some GNU utilities. Configuration files is a field where there are no standards.

Another thing to mention are graphical user interface programs that are targeted at system configuration. One of these (a famous one) is webmin.
Another example that comes to my mind is gnome-system-tools.
But webmin is not the right solution and gnome-system-tools are a step towards the right solution, but two or three more steps are missing.
What these GUI tools have in common is that they offer a front end that is specifically designed for some configuration, for example boot loader configuration. There is a more general way that i will talk about below.
gnome-system-tools are a more advanced architecture than webmin, because the problem with the latter is, that it parses configuration files and builds some HTML, whereas gnome-system-tools have an XML layer between the
GUI and the configuration file. What those two examples do right is that they are backward compatible with the text file configuration style.
That means they do not use a new database backend to manage the configuration.
That would also be hard to do, because every software that gets managed by
those configuration helper programs would need to be patched to access its
configuration in this new style data store.

The Linux Registry is a project, that does this and thats a reason that will make it hard for the Linux Registry to establish itself.

So webmin is a bunch of perl libraries and perl CGIs. There is no middle layer other than the interface of the perl libraries. gnome-system-tools is more language independent, as it has XML between its backends (which happen to be written in perl) and its frontend which are coded in C.

When programming a larger piece of software, it is nearly always the case to use object oriented paradigms nowadays. Some people just use some class library in their procedural code, but when the program to build is a little bigger, nearly everyone uses object orientation. Practically any important language has object oriented features. Thats for a good reason. I will now skip the advantages that object orientation offers to the user and just mention an example: on unix, everything is a file. You can cat the file, count its lines. You can do open(2), read(2), write(2) on that file no matter what the real filesystem implementation is (ext2, reiserfs, nfs). And to some degree, even network connections are files that support the methods read(2) and write(2). To be more precise files and network connections are subclasses of the more general concept of a stream.
So what would the world look like if you had to use different programs to print a file on a ext2 filesystem to the screen or one on an NFS mount.
That would be horrible. Thats just one example of the benefits of object orientation. If you study the C++ STL or the Java standard class library you get more examples.

So when object orientation is helpful when one wants to make a wide range of similar things look like they are identical and there is so much software out there that does many similar things but there are also so many different ways to make those similar things happen, then it beats the eye like a sunray:
why not employ object oriented concepts to the domain of software configuration on unix systems. Or to put it another way: why not raise the level of standardization to configuration files? Or in another way: Why not use a common layer, shared by a variety of programs that configures that programs and that makes it possible to: encapsulate complexity behind a set of abstractions, present that layer to the user in different ways (via a GUI or accessible to shell scripts). So what we move to is a 3 tier architecture.
The backend of that architecture are the configuration files that exist already. So the whole Infrastructure that is build on that backend is backward compatible with existing software and existing administrators that insist to use an editor.
The middle layer would be a class library, as we use object oriented paradigms to achive encapsulation and abstraction.
The frontend layer would be compatible to shell scripting. Another frontend would be a GUI.

To be compatible to the command line and shell scripts, one important design principle is to be completely text based. A Method has an arguments array (which C programmers know as argc/argv). It has standard input, output and error streams (which C programmers know as

stdin, stdout and stderr).
So when one calls a method on the command line that means just executing a process.

For a proper object oriented system one needs classes or datatypes. So we take a directory and put some method files into it. That means we put some executables into it. When an object is used those executables are put into scope. When the user changes its current working directory to that of one of our object, his PATH environment variable is modified to include all methods that the class has defined. So the current set of runnable programs depends on the current directory that you are in just as each object in real programming languages has a namespace of its own for methods that does not interfere with eaually named methods of other classes.

With the method interface being as it is, it is clear that methods can be written in any programming language, just like now commands on unix systems can be written in any language. It turns out that in order to design a good class library, a single method does much less than a normal unix command.
So it is inacceptable to spawn a process for each method invocation. Currently, methods written in C++ can be dynamically linked into the current process. When a method is called, the stdin/out/err cannot be pipes then.
The solution is to use string streams. The method that is called must only read and write from or to streams. When it is executed in a process of its own, these streams will be pipes. When it is loaded into the current process these streams will be string streams.
There are (not 100% complete) language bindings for perl and javascript at the moment. So when a method is invoked, that is implemented in perl, a perl interpreter is dynamically loaded into the current process which then executes the script. The standard streams of the perl method are wrappers to string streams. Javascript makes no difference and every scripting language can be made dynamically linkable by just programming a binding to C++ iostreams and some code that invokes methods. So the bindings that are necessary for a scripting language are finite, even when new classes are created, because the classes are accessed through a text format.

A class can have variables too. Variables are implemented through a method. The method is for example called 'string'. One can use the string method to store or retrieve a variable. The string method internally opens a file to either store its current argument as the new value of that variable or it reads the current value from the file and prints it to stdout if no arguments are given.
Variables can be turned into HTML input elements and are accessible through a GUI generally. A method that takes no parameters can be activated by clicking on a button. A method that takes a single element can be activated by a single line text box.

These implementation details (which are actually implemented at the time of writing) are not really essential to understand the broad concept and are in fact a specialization of the general idea. So the last few paragraphs were just a technical interplay. Lets move on to some actual classes.

Classes already exist in current unix systems. These are almost exclusivly
implicit classes. Debian users know /etc/init.d/service start|stop|restart.
Every service that is listed in /etc/init.d/ presents a common interface to the user. This interface is made up of the methods start, stop and restart at least. There are some others, like reload, which must not necessarily be implemented. So here's one problem of implicit classes. There is no way to find out which methods are supported.

Another example is pidfile creation. Most daemons allow to create a pid file from which their current process id can be read. The statement in the configuration to specify the location of the pidfile varies from software to software. And with the pidfile mechanism one can do a lot of things. One can send signals to the daemon, find out how much ram the daemon uses and much more. So here's an example class hierarchy:

process

pidfile extends process

methods of process are: send-signal, show-ram-usage
a single abstract method of process is: list-pids

methods of pidfile is simply: list-pids

So every daemon that supports pidfiles can be made compatible with the pidfile class and so be made compatible with the process class.
So when one wants to send a signal to a specific daemon, one only needs to know which object is associated with that daemon and can type:

$ cd /path/to/daemon; send-signal TERM

instead of opening a file and using kill -TERM <pid from the file>

Say you want to configure the port that a network server should listen on:

cd /path/to/server; port 23

No need to learn that apache uses a completely different configuration name for the same thing than wu-ftpd.

So port configuration would be handled by the class Inet/server, process status would be handled by the class process. The apache server would support both of these interfaces. You could find out what configuration a server supports by just listing its implemented interfaces. On the command line. And then change some variables and finally call a method start to bring up the server.
All at a single object. Or you could do all that not from the command line but via a graphical user interface.

Or you have the class PacketManger that defines method that must be implemented by class that allow to install software packages. The

namespace of software packages can then be incorporated into the already existing class namespace.
Instead of doing
$ cd /usr/ports/bla/fasel; make install
or
$ apt-get install bla-fasel

one would type:

cd /some_prefix/Package/bla-fasel; install

This incorporation of foreign namespaces (as the names of software packages) into the namespace of objects has the advantage that you can list all available packages by some means that is implemented in terms of the object system and not via a specialised tool like apt-cache. So to list the available packages one uses the method list in the apropriate object and one would also use the same method name to list all network interfaces, as they too would be in the object namespace. A particular network interface could be named:
/some_prefix/Interface/eth0

and to list all interfaces one does:

cd /some_prefix/Interface; list

comparable to:

cd /some_prefix/Package; list

To check whether a package is installed one could use the method is-installed.
But there is a more general solution: the class 'toggle'.
The class 'toggle' has one of two states: on or off. Like a light switch.

So the package class inherits from toggle and to check whether a package is installed one writes:

cd /some_prefix/Package/bla-false; is-on

And to see wheter a network interface is up or down can also be handled by the toggle class:

cd /some_prefix/Interface/eth0; is-on

What emerges from the examples is a way to name things on your computer: You name entities with a object name and you name things that modify these entities with methods.
All you have to do is to learn a complete new set of command names. But when you are done with that, new software that gets written and has been adapted by someone to the class library can be used with no further overhead. Ideally you wouldn't even have to read the man page to set up the software, cause you just get a listing of implemented interfaces which you already know how to handle.

Ten years ago i would have expected such a redesign of the interface to unix systems. There are many half-baked solutions out there that address a particular encapsulation or abstraction problem but are incompatible to each other.
So nothing is more obvious than to unify those attempts and to simplify or unix computer and still be scriptable and have a GUI.

Then you can write administrative scripts that magically work with all the network servers out there or configure the network in a specific way on every unix flavor out there. No need to write parsers, because parsers are part of the framework. They are a mapping from configuration files to the object model. You want to include algorithms into your configuration files to make some setting depend on some value you get at runtime. No need to write a configuration file generator, because algorithms are already supported in the object model. You just override a variable with a method that does whatever you want.

The text approach has its advantages. But the 3 tier object oriented approach is just a step further. And you don't have to say goodbye to your old configuration files, you can still edit them, because they are the database backend. And nobody says that setting a variable means clicking some GUI element or invoking a command on the command line. Objects can be transformed into a text file and after editing the text file can be transformed back into an object with the advantage that there are not dozens of names to be learned for setting some configuration item. When it is the same thing then it has the same name.

You want to use an xpath expression to find out, which network servers are up and running and which are not? Making an XML tree of the object tree is just a matter of writing some code once to work with every object afterwards.

You want to know which package supports configuring bindings to more than one port. As this property of the program is reflected in the class that the package implements, you can search all classes whether they implement this class of this special property you are interested in.

You can query the database for every web server and get a listing of them. You can use the configuration of your old web server and change to a new one that works instantly, because the configuration syntax is compatible.

You can have a basic configuration for heavy load network servers or low load network servers that you use when setting up some server and you inherit from that base configuration and just adapt it by overwriting some methods and variables. And when you later want to change some low load server into a high load server configuration you just change the class.

The downside is at the time of this writing only 50% of the mentioned things are actually implemented, segfaulting 5% of the time. Thank god the unimplemented features do not segfault.