A class library for Unix

...and software running on it

or

how to turn a bunch of buzzwords into a useful program

Meta

- this talk is best consumed with knowledge of: object oriented programming, Unix
- it is about work in progress, nothing you can expect to be important for your job tomorrow
- Unix hackers might also find some techniques interesting
- you are invited to participate under the term of the GPL
- now lets move on to the facts

Overview

- It is for Unix systems what C++ is for C
- it introduces explicit types to Unix
- it reduces the complexity for the user / administrator
- it is a design step further than webmin, more generic
- it is compatible to Unix, a method can be used anywhere a command execution is now used
- you get all the benefits of components, abstract interfaces, polymorphism, encapsulation and a more functional style

Why Object Orientation?

- The goal is to **classify** the diversity of software
- Abstract classes can help to integrate a wide variety of programs
- Classification can be used as a basis for documenting software, so even documentation can be shared between programs
- Abstractions help reduce the development time and they also reduce the time necessary to learn which functionality a program has and how to use it

It is not

- a programming language (you can use any language)
- kernel code (except for a virtual file system module)
- very efficient (but shell scripts aren't either)

Problems addressed

- Unix has a great variety of software that is independently developed and so is very diverse
- Standardization stops before we come to configuration files
- Scripts often address a particular software package and not a class of similar packages
- There is no general GUI concept for Unix configuration and the GUIs (webmin, gnome-system-tools) are not easily scriptable

State of the art I

- There are already lots of implicit classes
  - packages/ports (install, remove)
  - services (start, stop, restart, reload)
  - kernel modules (load, unload)

State of the art II

- Many ways to name the same thing
  - Apache: BindAddress/Port
  - wu-ftpd: -p
  - squid: http_port
  - inetd: first column of an entry
  - xinetd: after the service keyword
  - tcpserver: second non-option argument
- Nice to play memory games
- Especially ugly to automate

State of the art III

- Debian has update-inetd which encapsulates inetd.conf or xinetd.conf behind the same interface and update-rc.d which does the same for different init configuration styles
- Gnome system tools have backends that abstract system objects (for example grub and lilo)
- debian menus, which are used by a variety of desktop environments

Benefits for advanced users

- makes administrative tasks more portable
- no need to write parsers, just get the values from the objects
- gives Unix a more functional character and a more programming character generally

Benefits for end users

- Less documentation to read to gain access to more programs
- easier to compare different programs
- easier to switch from one program or component to another
- objects and classes are closer to natural thinking
- less information to keep in mind
- end users normally play different memory games

Benefits for the curious

- Open object model which is easier to inspect than that of a real programming language
- Learn about Unix software in a well structured format
- Does not focus on depth, but on covering a wide range
- Maybe it can be transformed into some sort of wikipedia for Unix software

Benefits for developers

- writing methods can be as simple as writing a shell script
- write code for a consistent framework
- use your favourite programming language

Basic Architecture

- Programming language independence
- Text oriented inter-method communication
- argv – style method invocation
- 3 tiers (Storage, Object Model, Frontends)
- Backward compatible with existing configuration files
- Scriptable
- Graphical user interface(s)
- Comfort before efficiency

Implementation status

- Alpha status
- Not a prototype anymore
- Brings some application independent classes (Scheduler, Virtual File System, ...)
- Language bindings: perl, c++, javascript, bourne shell
- Backends: file system, original configuration files via virtual RAM file system
- Frontends: command line, HTML, XML
- 30k lines of C++, 20k lines shell and perl

Anatomy of a class

- A class is a list of key/value mappings
  - inherit: a list of strings naming superclasses
  - methods: a list of files of method implementations
  - methdsc, varidsc: documentation for methods and variables
  - clsdesc: the class documentation
  - methsrc: method source files
  - include: code shared by a number of methods
  - members, bmember, fmember: variables

Storing classes in the filesystem

- a directory can be made into a class by creating some files that describe the class
- all object attributes begin with a dot and are either files (.inherit, .members) or directories (.methods, .methsrc, .fmember)
- directories may contain other directories, there is one subdirectory for each programming language in the .methsrc and .include directories for example

Using a class

- A class is either used when a method is called on it or it is in the list of superclasses of an object that is used
- When a class is accessed, the .inherit file is read to obtain a list of superclasses
- When a method is called, all superclasses methods directories are scanned for a matching implementation
- That is a late binding that uses the file system at runtime (or generally the currently used backend)

Using classes from the shell

- in the bash PROMPT_COMMAND is set to point to a program that tries to interpret the current directory as a class and sets the shell environment accordingly
- PATH is modified to include all .methods directories of all superclasses
- after that the method at-enter is called in which other modifications can be made (for example to install completions for methods that are now in scope)
- a method can be invoked like a command as it is in the path now

Using classes from other languages

- There is a C++ class that is (for historic reasons) called Environment that has all information about an object
- There is a C++ class called MethodFinder which is instanciated with an Environment object and has a method method() that is used to call a method
- Bindings exist for scripting languages to get access to the above classes

The execution environment of a method

- Very similar to a unix command
- stdin, stdout, stderr streams
- argc/argv arguments
- an integer return value
- a method can either be in a process of its own or dynamically loaded into the running process
- so one writes little programs where stdin/stdout may be string streams and which do not call exit(2) to terminate and which must free all memory they acquire (comparable to mod_perl vs. CGIs)

Examples

- cd /tmp; create -i foo bar; destroy bar
  - creates the object bar in /tmp that inherits from foo, then deletes it
- @ universal list-methods [-r]
  - lists all methods (one per line) that are members of the universal class [and superclasses]
- cdo servers/apache; restart
  - change object to the apache server and call method restart
- @ servers/apache restart
  - like above

More examples

- @ /etc/fstab/0 device
  - print the device of the zeroth line of the fstab
- @ ~/.mozilla/cache clean
  - delete the browser cache
- for s in $(@ services list); do @ $s reload; done
  - tell every background daemon to reload the configuration
- @ Package/zsh install
  - install a software package

Reusing interfaces

- class: collection, method: count
- @ /etc/fstab count
- @ /etc/inetd count
- @ User count
- @ Network/Interface count
- @ Network/Route/Ipv4 count
- @ User/root/Session count

Reusing interfaces II

- class: cleanable, method: clean
- @ Apt/unstable/cache clean
- @ ~/.firefox/cache clean
- @ Server/squid/cache clean
- @ Server/apache/error-log clean
- @ Cache/universal clean

Other uses

- @ packetfilter allow Server/ftpd
- cdo Server/apache
  - create -i Logger/syslog error-log
  - create -i syslog error-log
  - create -i Logger/readproctitle error-log
  - create -i Logger/file error-log
  - create -i Logger/my-custom-logger error-log
  - destroy error-log
  - disable error-log

One step beyond

- @ Meta/class implements [-r] cleanable
  - list all classes that implement the cleanable interface [including subclasses]
- @ Meta/class search server network time
  - search for all classes that have above strings somewhere in their documentation (e.g. an ntp server)
- @ Server save-state; @ Server restore-state
  - save and restore the volatile state of all objects under Server/* (e.g. which of them is running)

Things you may find useful

- C++ virtual file system framework
- C++ text template processing framework
- C++ User space scheduler/event framework
- C++ compile time class configuration framework
- perl bindings to iostreams and stl containers

Work Todo

- Implementing parsers and configuration file generators
- Writing more language bindings
- Maybe a standalone GUI
- Writing documentation
- Classifying all the software out there
- Porting away from (debian) linux

Research/Advanced Work Todo

- Generic interface to access scripting language interpreters from C++ (the opposite direction of what SWIG does)
- Style of the presentation of the object model to the user (the GUI)
- Identifying existing abstractions and designing classes
- Implement advanced programming concept (multiple dispatch, ...)