bsda:obj for FreeBSD

Unleashing the FreeBSD Shell[1]

Abstract

This is a micropaper written by Dominic Fandrey as background material for his talk “Binary Package Management and Object Oriented Shell Scripting under FreeBSD”, held at the EuroBSDCon 2010 in Karlsruhe/Germany.

FreeBSD's default shell is an Almquist shell derivative. The Almquist shell was originally developed to imitate the Bourne shell[Man1SH]. In conjunction with the tools in the FreeBSD base system and a sufficiently twisted mind it yields near limitless power to those who know its use.

This paper provides entry points into the world of object oriented shell programming and provides a quick hackers' guide to the author's shell object framework `bsda:obj`.

1. Introduction

In 2005 the author migrated his IBM Thinkpad notebook from MS Windows XP to FreeBSD 5.3, his first serious encounter with a UNIX style operating system. By 2010 he claims to be one of the most outspoken, polarizing and well-known (though of doubted popularity) members of the German BSD user community.

2. Discovering the Shell

The two things that got him hooked were the FreeBSD ports system, and the power of the FreeBSD shell. Commuting between home, friends and his university (the TU Darmstadt at the time) he wrote a shell script that was invoked by FreeBSD's `devd(8)`[Man8DEVD] system, triggered by an ACPI event occurring when the WLAN power switch was activated. This shell script searches for available networks, sorts them by connection strength and connects to them in sequence until it succeeds or went through them all. Depending on the network encountered the script executes small configuration scripts that automatically open VPN connections and add additional routes.

A version of this script is still running on his notebook with more than 60 network configurations at the time of this writing.

On modern systems and working environments shell scripting poses two challenges. Synchronizing work distributed over several processes and handling huge data structures.

In March 2008 a new version of the script named `pkg_libchk` was released, which solved the synchronization issue through file system locking using FreeBSD's `lockf(1)`[Man1LOCKF] command. The second issue proved far more challenging and was first tackled with the beginning of `bsda:obj` development, which was boldly versioned 1.0 in its first incarnation on the GPN8 conference[GPN8Code], because it already contained all the features the author expected to need. At the time of this writing the framework has the version 1.16 and is 2218 lines long, instead of the 452 lines of version 1.0.

1 Last edited Friday 8 October 2010

3. Objects in the Shell

On the 10th November, 2009 the author wrote a short note about his effort to bring object orientation to the shell on the `comp.unix.shell` newsgroup[CUSHELL2009A], where he discovered that he was far from being the first person to venture this way. A post by Brian Hiles[CUSHELL2009B] featured a list of nine projects and papers in the field, three of which should not go unmentioned, each of them for entirely different reasons.

3.1. SHOOP

When discussing the subject of object oriented shell scripting this is in the author's perception, the one project, if any, people usually know about.

Three developers, Joey Hess, Adam Heath and Gergely Nagy[ShoopAuthors] have worked on SHOOP until the last commit to the project source repository in 2001. All three can be found on the Debian developers list[DebianPeople].

The `README` of the project[ShoopReadme] states:

```
Every language under the sun these days is
Object Oriented.  In an effort to make POSIX
shell more buzzword compliant, and to show
that it's really not a big deal for a language
to lack built-in OO support, we have added
object orientation to plain old shell script.
```

It goes on to list the features of SHOOP:

```
Specifically, we have implemented classless OO
with introspection, finalization,
serialization, and multiple inheritance.
```

The most notable differences to `bsda:obj` are the lack of classes and scope checks.

The syntax does not appeal to this paper's author, but this is hardly a fair statement, because unlike `bsda:obj`, SHOOP conforms to POSIX.

3.2. UNIX Shell Objects

The book by Christopher A. Jones was at the beginning of the list posted by Brian Hiles. UNIX Shell Objects was published by IDG in 1998[Jones1998]. The book is most notable for its sheer magnitude, outlined on the back of the book:

> UNIX Shell Objects shows you how to break the mold of traditional shell programming and use a shell-based Object Request Broker to create multitiered, distributed-object applications that bridge networks and platforms.

Jones' Korn shell based framework has a “one file, one class” philosophy. The syntax for creating a class requires a lot of `eval` calls, thus the framework offers a preprocessor named `shcc`, which converts abstract code to the executable syntax.

The framework has no concept of scope and access rights, however all attribute access is through access functions which allows the implementation of workarounds, this is announced on page 5 in the book:

> When you develop classes in the UNIX shell, you will not be using private and public specifiers, but you will implement access functions. Examples in this book, however, demonstrate how access rights can be specified and internal data “locked” down if needed. In theses examples, internal code is not included in the “header” area at the top of the class file, ...

In chapter 6 on page 139 the author starts to explain why and how to integrate shell objects with Java. The second paragraph on this page states some of the book's author's reasons:

> ... Also, the shell is single-threaded, only allowing one line of execution throughout an application. While you can have different processes communicating through IPC, threading is more powerful when it comes to certain operations, especially at the device and network level. The shell also has no support for sockets. ...

In the opinion of this paper's author it boils down to leaving the shell for I/O with high performance requirements.

Summarizing Jones' entire book is beyond the scale of this paper. Go ahead and read it.

3.3. A New Object-Oriented Programming Language: sh

The last project by another party to introduce here is a paper written by Jeffrey S. Haemer and released in the proceedings of the USENIX Summer 1994 Technical Conference[Haemer1994].

As the author states in chapter 5, the practical value of his approach is in doubt:

> “Cute idea,” you say, “but is this good for implementing real applications?” Probably not.

The appealing components of the paper are the clear and simple concept as well as the author's humour:

> While the system is unconventional, only a toy, and downright slow, its implementation is straightforward and its use instructive.

The concept of Haemer's approach is summarized in a single paragraph:

> In what follows, object classes are shell scripts and objects are running processes. Methods are invoked by messages passed to objects through FIFOs (named pipes). The methods themselves are implemented as shell functions; function polymorphism is guaranteed because separate programs have separate name spaces. A class hierarchy is provided by the file system itself.

The performance implications of making every single object its own process have already been established, still the concept appeals in our n-core world. This paper is being written on a dual core notebook with 8GB of RAM. Not a meter away stands a new quad core in a tower that actually needs less energy than the notebook.

Though the old clock frequency race appears to have started again, after all there is only so much you can do in parallel, this paper's author would not be surprised to have hundreds or even thousands of cores at his disposal in a consumer grade CPU, ten years from now.

4. Using bsda:obj

The latest version of the `bsda:obj` framework can always be downloaded from the BSD Administration Scripts[BSDA] source repository at SourceForge.net:

> http://bsdadminscripts.svn.sourceforge.net/viewvc/bsdadminscripts/bsdadminscripts/?view=tar

4.1. Libraries

The framework is part of the BSD Administration Scripts package, releases can be found in the FreeBSD Ports system under `sysutils/bsdadminscripts`[FPBSDA]. The `bsda:obj` code can be found in the file `src/bsda_obj.sh`. The package also contains libraries using the framework, easily identified by their names matching the glob pattern `bsda_*.sh`.

For the beginning this subsection is just to provide a context for what is possible, those interested in how something was done, should look into the code.

Three libraries are of sufficiently generic purpose to be of interest to most shell programmers:

- `bsda_messaging.sh`
- `bsda_scheduler.sh`
- `bsda_tty.sh`

The `bsda:messaging` library provides communication classes that allow processes to talk to each other through file system based message queues. Semaphores ensure safe many-read and single-write access to a queue.

The `bsda:scheduler` library offers interfaces and two simple schedulers to manage the control flow of a process. Because every scheduler can also run as a process schedulers can be nested to create a primitive priority mechanism. To hand process control over to a primitive scheduler has the advantage of making it possible to serialize an entire process, which offers many possibilities, such as resume after an interruption or even moving a process to another machine through a tcp connection established with `nc(1)`.

The `bsda:tty` library offers the `bsda:tty:Terminal` class to control the terminal the application is running in. It allows features like multiple status lines, output duplication and offers helper methods to convert numeric values to common byte size representations. It also offers a wrapper method around `printf`[Man1PRINTF] that has the ability to scale strings down to make them fit the terminal. All this works for arbitrary sized terminals.

4.2. bsda:obj Syntax and Internals

A very thorough and complete introduction into the `bsda:obj` syntax can be found in the first ~1000 lines of the `bsda_obj.sh` file. It is a good idea to have it at hand to check details and try one's own bits of code. This paper concentrates on interesting features and concepts instead of completeness.

Classes, Objects and Returning Data

The framework consists of a collection of functions. The most significant function is named `bsda:obj:createClass()`. It takes a bunch of parameters and creates a class and a constructor function, which creates objects. Parameters to the `createClass()` function, which cannot be interpreted are ignored, this way comments can be embedded into class definitions.

The following are the contents of the `bsda_obj_demo.sh` file:

```
1  #!/bin/sh
2  #
3  # A small demo of "bsda_obj.sh", which demonstrates
4  # return by reference. Note that this even works
5  # safely when the variables within a method have the
6  # same names as the variables in the caller context
7  # (such as is the case for recursive methods).
8  #
9  # These features are really just useful byproducts
10 # of my desire to write object oriented shell
11 # scripts.
12 #
13
14 # Import framework.
15 bsda_dir="${0%${0##*/}}"
16 . ${bsda_dir:-.}/bsda_obj.sh
17
18 # Declare the class.
```

```
19 bsda:obj:createClass Demo \
20     w:value \
21         This is a comment \
22     x:fibonacciRecursive \
23         "This is a comment, too. <== my preferred style" \
24
25 #
26 # Implementation of the fibonacciRecursive method for the
27 # Demo class.
28 #
29 # Yes I know that this is the least efficient way of
30 # doing this, but it demonstrates what I want it to.
31 #
32 # @param &1
33 #   The variable to store the fibonacci value in.
34 # @param 2
35 #   The index of the fibonacci value to return.
36 #
37 Demo.fibonacciRecursive() {
38     # Terminate recursion.
39     if [ $2 -le 2 ]; then
40         $caller.setvar "$1" 1
41         return 0
42     fi
43
44     local f1 f2
45
46     $this.fibonacciRecursive f1 $(($2 - 1))
47     $this.fibonacciRecursive f2 $(($2 - 2))
48
49     $caller.setvar "$1" $((f1 + f2))
50 }
51
52 # Create instance.
53 Demo demo
54
55 # Call the fibonacci method from instance and ...
56 # ... store the result in the value variable.
57 $demo.fibonacciRecursive value 8
58 # ... print the result.
59 $demo.fibonacciRecursive '' 8
60
61 # Set an attribute.
62 $demo.setValue $((value - $($demo.fibonacciRecursive '' 6)))
63
64 # Get an attribute and ...
65 # ... store the result in the value variable.
66 $demo.getValue value
67 # ... print the attribute.
68 $demo.getValue
```

What can be seen in line 19 is that the first parameter to the `createClass()` function is always treated as the class name. The following parameters either define attributes or methods. The `w:value` parameter defines the attribute `value`. The `w:` prefix triggers automatic generation of the methods `getValue()` and `setValue()`. The `x:` prefix on line 20 announces a method.

Methods are created as regular shell functions following the syntax `<class>.<method>()`. The method implementation in line 37 reveals a very convenient feature of `bsda:obj`.

Obfuscating details like parameter checks have purposefully been left out. What this example illustrates is that complex values are not returned through `stdout` or global variables. Instead, an arbitrary number of method parameters can be dedicated to passing on the names of variables results are expected in.

The method then has the possibility to use the `$caller.setvar` function to write into variables in the calling context.

It is a solid convention to only do that to variables whose names have been passed on. Otherwise the caller would require undesired amounts of insights into the method and the danger that an internal change breaks something grows.

How Methods Work

The constructor function, named after the class, creates an object context for each object it creates. This approach uses a technique that is called closures by functional programmers. It creates specialized versions of the class methods, providing them with the current object context and the special variables `$this`, `$class` and `$caller`.

Because the FreeBSD shell does not support real closures, they are imitated by using unique prefixes and a call stack for returned values. The object context is generated by object functions that populate the `$this`, `$class` and `$caller` variables, perform access scope checks and call the class method implemented by the programmer. After that function has terminated its local scope has been left and the data provided with `$caller.setvar()` can be taken from the stack and written into the context of the caller.

The `$this` variable simply contains the prefix of the current object. The following is an interactive `sh` session, that includes the previous example:

```
1  $ . bsda_obj_demo.sh
2  21
3  13
4  $ echo $demo
5  BSDA_OBJ_bsda_obj_Demo_0f47995cc363303f_1286381988_35944_0_
6  $
```

The long string in line 5 is the object ID of the `Demo` instance created in line 53 of the `bsda_obj_demo.sh` file. `BSDA_OBJ_bsda_obj_` are two prefixes provided by the framework, because it was created with a vague idea of creating several compatible frameworks, that can generate interacting classes and objects. The `Demo_` prefix is generated from the class name. `0f47995cc363303f_` is a hexadecimal presentation of a random 64bit session ID, this makes sure that object IDs are unique under any given circumstances. The next number `1286381988_` is the UNIX time in seconds the session was started. The following number is the current process ID `35944_` and the final `0_` a class instance counter (starting with 0).

If a script forks it is necessary to use the `bsda:obj:fork()` function to make the new PID known to the forked process and make sure they create objects with different IDs.

Additional Features

This example hopefully suffices to get started and get an idea how the framework works.

The `bsda:obj` framework however has many more notable features than those introduced here, the following list is an attempt to provide the complete feature set:

- Classes
- Interfaces
- Multiple inheritance
- Private, protected and public access scopes
- Namespaces
- Serialization and recursive (deep) serialization
- Reflection and introspection
- Automatic getter and setter generation
- Access scope widening (e.g. make automatic getters public and leave the setter private)
- Caller resolution through the `$caller.getObject()` and `$caller.getClass()` functions
- Convenient return stack through the `$caller.setvar()` function
- Constructors and destructor methods can call optional init and cleanup methods that can prevent object creation/deletion (besides all the other useful stuff like initializing attributes or recursive data structure removal)

The contents section of the `bsda_obj.sh` file provides an overview over the framework documentation:

```
# TABLE OF CONTENTS
#
# 1) DEFINING CLASSES
# 1.1) Basic Class Creation
# 1.2) Inheritance
# 1.3) Access Scope
# 1.4) Interfaces
# 2) IMPLEMENTING METHODS
# 2.1) Regular Methods
# 2.2) Special Methods
# 3) CONSTRUCTOR
# 4) RESET
# 5) DESTRUCTOR
# 6) COPY
# 7) GET
# 8) SET
# 9) TYPE CHECKS
# 9.1) Object Type Checks
# 9.2) Primitive Type Checks
# 10) SERIALIZE
# 10.1) Serializing
# 10.2) Deserializing
# 11) FORKING PROCESSES
# 12) REFLECTION & REFACTORING
# 12.1) Attributes
# 12.2) Methods
# 12.3) Parent Classes and Interfaces
# 13) COMPATIBILITY
# 13.1) POSIX
# 13.2) bash - local
# 13.3) bash - setvar
# 13.4) bash - Command Substitution Variable Scope
# 13.5) bash - alias
```

Note that despite the framework's size of more than 2200 lines, only about 650 of these actually contain code. The remaining lines contain white space, comments and documentation.

5. Conclusion

As has hopefully been shown getting started is not difficult. The code of the framework and the libraries is in many places difficult to read and obscure. But it is also extensively documented.

I greatly welcome any mention of where the code is difficult to understand and insufficiently documented or simply broken. I can be contacted via e-mail at <kamikaze@bsdforen.de>. If you speak German visit the German BSD community at http://bsdforen.de, my usual hangout on the net.

6. Bibliography

[BSDA] Dominic Fandrey, BSD Administration Scripts, Geeknet, Inc., 2010, http://sourceforge.net/projects/bsdadminscripts/

[CUSHELL2009A] Dominic Fandrey, Object Oriented Shell Scripts, comp.unix.shell, 2009, http://groups.google.com/group/comp.unix.shell/msg/abcdff404971eac0?dmode=source

[CUSHELL2009B] Brian Hiles, Re: Object Oriented Shell Scripts, comp.unix.shell, 2009, http://groups.google.com/group/comp.unix.shell/msg/20c66ac2fb145ad6?dmode=source

[DebianPeople] Various, Debian - Project Participants, SPI, 2010, http://www.debian.org/devel/people

[FPBSDA] Dominic Fandrey, FreshPorts - sysutils/bsdadminscripts:, DVL Software Limited, 2010, http://www.freshports.org/sysutils/bsdadminscripts/

[GPN8Code] Dominic Fandrey, Objektorientierte Shell-Skripte , Entropia e.V. CCC Karlsruhe, 2009, http://entropia.de/wiki/GPN8:Code#Objektorientierte_Shell-Skripte

[Haemer1994] Jeffrey S. Haemer, A New Object-oriented Programming Language: sh, USENIX Summer 1994 Technical Conference, 1994

[Jones1998] Christopher A. Jones, UNIX Shell Objects, IDG, 1998, ISBN 0-7645-7004-8

[Man1LOCKF] John Polstra, lockf(1) - Execute a Command While Holding a File Lock, The FreeBSD Project, 1998

[Man1PRINTF] Various, printf(1) - Formatted Output, The FreeBSD Project, 2005

[Man1SH] Kenneth Almquist, sh(1) - Command Interpreter (shell), The FreeBSD Project, 2010

[Man8DEVD] M. Warner Losh, devd(8) - Device State Change Daemon, The FreeBSD Project, 2005

[ShoopAuthors] Joey Hess, Adam Heath and Gergely Nagy, AUTHORS, Shoop: SHell Object Oriented Programming, 2001, http://shoop.cvs.sourceforge.net/viewvc/shoop/shoop/docs/AUTHORS?revision=1.2

[ShoopReadme] Joey Hess and Adam Heath, README, Shoop: SHell Object Oriented Programming, 2000, http://shoop.cvs.sourceforge.net/viewvc/shoop/shoop/docs/README?revision=1.26