## An Extensible Viewer for your Personal Finances

# THE BOTTOM LINE

www.sxc.hu

A helpful Perl script gives you an immediate overview of your financial status, adding the balances of multiple accounts and share depots. It even allows users to add their own plugins. **BY MICHAEL SCHILLI**

Contrary to popular belief, rich people are not unhappier than people without a penny to their names. You can hear the sighs of relief all over the country – gone are the fears of wealth-induced miserliness. And for the first time in years, people again dare to check their financial status. Read on to find out how.

With the exception of a few eccentrics who prefer to horde their wealth under the floorboards of their villas, people are increasingly turning to programs such as Gnucash to manage their accounts and depots. Account management programs help you tidy up your accounts, giving you neat formatting or even graphical output. This said, the open source tools are still very much in the vein of Quicken and Microsoft Money and entail a lot of highly disciplined work.

Amateur accountants typically lack the time to complete all those painstaking entries, not to mention the complex installation that Gnucash requires. In other words, this is not a simple tool and not easily extensible. In contrast to this, our Perl script is designed for the rest of us, who prefer not to spend more than ten minutes a month updating their bank balances but still like to check out the bottom line of their shareholdings on a daily basis.

The system supports modular plugin-based extensions, allowing users to take currency conversion or tax scenarios into consideration and providing customization without drifting toward bloatware.

It is so easy to overdo things. For example, it doesn't make sense to develop a module for share splitting, which occurs once every few years; a few manual steps are all it takes to handle the split (in other words, don't try to be everyone's darling.) Our *beancounter*

**Listing 1: beancounter**

```
01 #!/usr/bin/perl -w
02 ##################################
03 # beancounter - Money
04 #        Counting Interpreter
05 # Mike Schilli, 2004
06 # (m@perlmeister.com)
07 ##################################
08 use strict;
09
10 use lib
11 '/some/where/in/module/land';
12
13 use Log::Log4perl qw(:easy);
14 Log::Log4perl->easy_init(
15   $ERROR);
16
17 use Plugger;
18 my $string = join '', <>;
19
20 my $plugger = Plugger->new();
21 $plugger->init();
22 $plugger->parse($string);
```

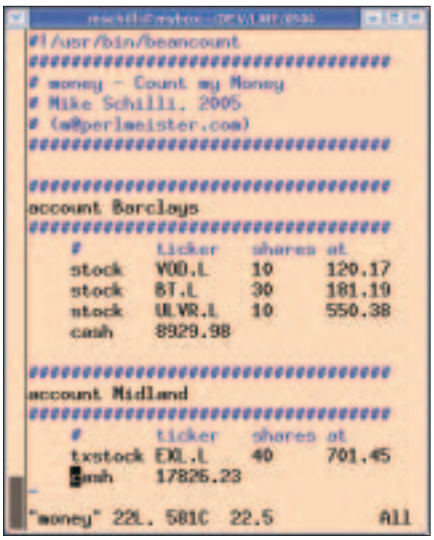


**Figure 1: A configuration file defines the account holder's details; at the same time, the file is an executable script.**

script tries to find the golden mean. It has basic functionality that gives users a sum total for multiple accounts and shareholdings, but it leaves enough scope for users to handle their own specialized requirements.

## Interpreter Script

Users can define their account data in a file called *money*, as shown in Figure 1. The *account* keyword defines a new account; a shareholding starts with *stock*; and *cash* is self-explanatory. The *beancounter* script in Listing 1 is the interpreter for this financial data. It parses the account definitions, ascertains share prices, and adds profits and losses to give you a financial statement. Typing *beancounter money* in the command line launches the script, but there is an even easier way. Make the *money* configuration file executable and add the *beancounter* interpreter to the she-bang line. In other words, *money* does not use *perl* as its interpreter, but *beancounter*.

If you do not use the hyper-modern Zsh shell, but use good old Bash instead, you will not be able to add a script to the she-bang line. Instead, you need a C wrapper, which is what the following C program, *beancount.c*, gives us:

```
main(int argc, char **argv) {
    execv("/usr/bin/➥
    beancounter", argv); }
```

Now compile *beancount.c* by entering the following

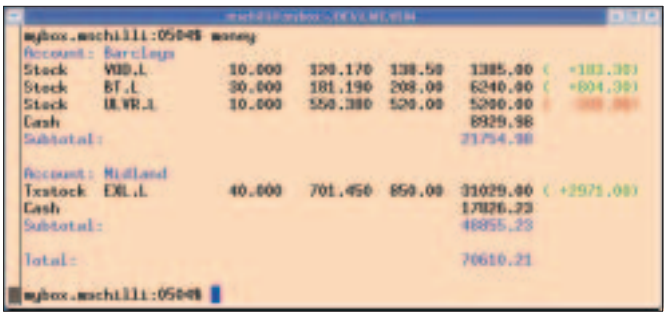


**Figure 2: Money counter *beancounter* in action. Calling the script in the command line gives you a colorful overview of your accounts and total financial status.**

```
cc -o beancount beancount.c
```

This allows you to use the *beancount* executable as the interpreter for the financial data in the she-bang:

```
#!/usr/bin/beancount

account Barclays
#####################################
  #       ticker   shares at
  stock   VOD.L    10     120.17
  # ...
```

If the file with this code, *money*, is executable, you can simply enter *money* to launch the money counter. Although it looks like a configuration file, we actually have an executable script. Figure 2 shows you the output. Practical!

## Extending the Interpreter with Plugins

The interpreter in Listing 1 is quite sparse: it creates an instance of a *Plugger* type object, calls *init()* to initialize the underlying plugin architecture, and passes the configuration file, which it read from standard input using < > previously, to the plugin system's *parse()* method. The *Plugger* framework in Listing 2 interprets the first word in each line as a command. But without plugins, it can't interpret a single thing. In fact, all it does is ignore the *money* file's comment lines, which all start with #.

*Plugger.pm* automatically parses any modules added to the *Plugger/* directory during compilation. Line 7 pulls in the the Module::Pluggable CPAN module, which handles all this. Lines 8 and 9 set the *require* flag and the search path to the plugins relative to the current directory or *@INC* path.

Our Plugins do not have a *new()* constructor, in contrast to the typical object-oriented approach, but an *init()* function, which is called by *Plugger.pm*, the master of all plugins, one by one for each plugin module it finds. Module::Pluggable automatically adds the *plugins()* method to its host, the *Plugger* class. *plugins()* returns a list with the names of all discovered plugins. Lines 31 and 32 use this mechanism to iterate through the *init()* functions of all the plugins.

To allow a plugin to know its caller, and run the caller's methods if required, *Plugger.pm* passes a *$ctx* (context) refer-

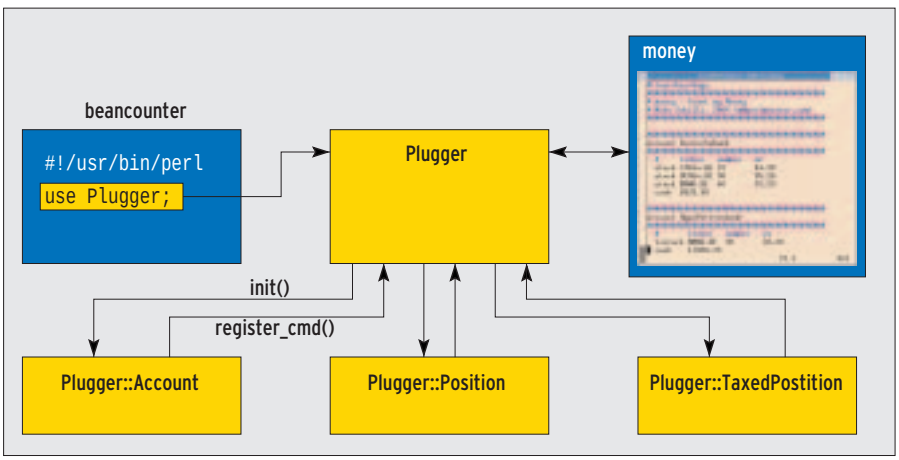


**Figure 3: The *Plugger.pm* plugin manager uses Module::Pluggable to parse the plugins below *Plugger::* and calls their *init()* functions. The plugins then call *register_cmd()* in turn to register commands with the caller.**

ence to the *init()* method of each plugin. This is simply a reference to the only existing Plugger object, the plugin manager. This reference allows a plugin to issue instructions to the *Plugger* manager. As *Plugger* interprets the commands in a configuration file, the plugin calls the *register_cmd()* management method to register new commands.

## Argument Assist for Account Plugin

Listing 3 gives you an example of a plugin in the *Plugger/* directory: *Account.pm* uses the *register_cmd()* mechanism described earlier in this article to teach the plugin manager the *account* command:

```
$ctx->register_cmd("account",
    \&start, \&process, ➥
    \&finish);
```

Within the confines of the framework, this two-liner specifies that on interpreting the *account* keyword in the configuration file, Plugger should call the *process()* function in *Plugger/Account.pm* and pass it the split elements of the configuration line as arguments. *Plugger.pm* also calls the *start()* function shown in Listing 3, line 21, before starting to interpret the configuration file, and finishes off by calling the *finish()* function (line 84).

The account plugin makes use of this mechanism to set the total value for all defined accounts, which is stored in the global variable *account_total*, to zero before starting to parse. We still need to decide where to define a counter of this kind, which *Account.pm* and other plugins will need to access. The *Plugger.pm* module creates a hash called *%MEM* for this purposes. The module passes a reference to the *%MEM* hash to anything that uses the *mem()* accessor in line 36 to ask for a reference. For example, a plugin such as *Account.pm* can do the following:

```
$ctx->mem()->➥
{account_total} = 0;
```

to set a variable which other plugins with a reference to the *Plugger* plugin manager can access, thanks to *$ctx*. In fact, this technique demonstrates how the *Account.pm* and *Position.pm* plugins pass information: *Account.pm* first sets *account_total* to zero. *Position.pm*, which is used

### Listing 2: Plugger.pm

```
01 ##################################
02 package Plugger;
03 ##################################
04 use strict;
05 use warnings;
06
07 use Module::Pluggable
08   require     => 1,
09   search_path =>
10       [qw(Plugger)];
11
12 our %DISPATCH = ();
13 our %MEM      = ();
14
15 ##################################
16 sub new {
17 ##################################
18   my ($class) = @_;
19
20   bless
21     my $self = {}, $class;
22
23   return $self;
24 }
25
26 ##################################
27 sub init {
28 ##################################
29   my ($self) = @_;
30
31   $_->init($self)
32     for $self->plugins();
33 }
34
35 ##################################
36 sub mem { return \%MEM; }
37 ##################################
38
39 ##################################
40 sub parse {
41 ##################################
42   my ($self, $string) = @_;
43
44   for (sort keys %DISPATCH) {
45     $DISPATCH{$_}->{start}
46       ->($self)
47       if $DISPATCH{$_}
48         ->{start};
49   }
50
51   for (split /\n/, $string) {
52
53     s/#.*//;
54     next if /^\s*$/;
55     last if /^__END__/;
56     chomp;
57
58     my ($cmd, @args) =
59       split ' ', $_;
60
61     die
62       "Unknown command: $cmd"
63       unless
64       exists $DISPATCH{$cmd};
65
66     $DISPATCH{$cmd}
67       ->{process}
68       ->($self, $cmd, @args);
69   }
70
71   for (sort keys %DISPATCH) {
72     $DISPATCH{$_}->{finish}
73       ->($self)
74       if $DISPATCH{$_}
75           ->{finish};
76   }
77 }
78
79 ##################################
80 sub register_cmd {
81 ##################################
82   my ($self, $cmd, $start,
83     $process, $finish)
84     = @_;
85
86   $DISPATCH{$cmd} = {
87     start   => $start,
88     process => $process,
89     finish  => $finish,
90   };
91 }
92
93 1;
```

by every *stock* or *cash* definition, evaluates this and adds it to *account_total*.

## Adding Color

Imagine you want *Account.pm* to output the top line of an account and the balance in blue and in bold type. The CPAN Term::ANSIColor module handles this nicely. Adding a *:constants* tag to the *use* statement exports text attribute constants, such as *BLUE*, *BOLD*, and *RESET* (back to standard type) to the namespace of the calling script. This allows you to issue *print* statements such as

```
print BLUE, BOLD,
      "In blue and bold!",
       RESET;
```

to output ANSI sequences that output the blue and bold text in the current terminal before calling *RESET* to fall

**Listing 3: Account.pm**

```
001 ##############################
002 package Plugger::Account;
003 ##############################
004 use strict;
005 use warnings;
006 use Term::ANSIColor
007   qw(:constants);
008
009 ##############################
010 sub init {
011 ##############################
012   my ($class, $ctx) = @_;
013
014   $ctx->register_cmd(
015     "account", \&start,
016     \&process, \&finish
017   );
018 }
019
020 ##############################
021 sub start {
022 ##############################
023   my ($ctx) = @_;
024
025   $ctx->mem()
026     ->{account_total} = 0;
027 }
028
029 ##############################
030 sub account_start {
031 ##############################
032   my ($ctx, $name) = @_;
033
034   print BOLD, BLUE,
035     "Account: $name\n",
036     RESET;
037
038   $ctx->mem()
039     ->{account_subtotal} = 0;
040   $ctx->mem()
041     ->{account_current} =
042     $name;
043 }
044
045 ##############################
046 sub account_end {
047 ##############################
048   my ($ctx, $name) = @_;
049
050   print BOLD, BLUE;
051   printf "%-47s %9.2f\n\n",
052     "Subtotal:", $ctx->mem()
053     ->{account_subtotal};
054   print RESET;
055 }
056
057 ##############################
058 sub account_end_all {
059 ##############################
060   my ($ctx) = @_;
061
062   print BOLD, BLUE;
063   printf "%-47s %9.2f\n\n",
064     "Total:", $ctx->mem()
065     ->{account_total};
066   print RESET;
067 }
068
069 ##############################
070 sub process {
071 ##############################
072   my ($ctx, @args) = @_;
073
074   my $c =
075     $ctx->mem()
076     ->{account_current};
077   account_end($ctx, $c)
078     if $c;
079   account_start($ctx,
080     $args[1]);
081 }
082
083 ##############################
084 sub finish {
085 ##############################
086   my ($ctx) = @_;
087
088   my $c =
089     $ctx->mem()
090     ->{account_current};
091   account_end($ctx, $c)
092     if $c;
093   account_end_all($ctx);
094 }
095
096 ##############################
097 sub position {
098 ##############################
099   my (
100     $type,  $ticker,
101     $n,     $at,
102     $price, $value,
103     $gain
104     )
105     = @_;
106
107   unless (defined $ticker) {
108     printf "%-47s %9.2f\n",
109       $type, $value;
110     return;
111   }
112
113   my $clr =
114     $gain > 0 ? GREEN: RED;
115
116   printf
117     "%-8s %-10s %9.3f %9.3f"
118     . " %7.2f %9.2f"
119     . " %s(%+9.2f)%s\n",
120     $type, $ticker, $n, $at,
121     $price, $value, $clr,
122     $gain, RESET;
123 }
124
125 1;
```

back to normal type for any following print statements.

## Online Share Prices with Temporary Storage

The *Position* plugin in Listing 4 retrieves current (that is 20 minutes delayed) share prices from the Yahoo financial page using another CPAN module, Finance::YahooQuote. An exported function, *getonequote()*, gives you ticker symbols such as *VOD.L* for Vodafone shares on the London stock exchange, or *EBAY* for the Ebay shareprice on the Nasdaq. There is a useful list of UK ticker symbols at [3].

As *beancounter* may need the same share price multiple times, *Position* stores the price in a cache for 10 minutes. The CPAN Cache::Cache module has an extremely simple interface with *set()*, which sets a cache entry, and *get()*, which gets the cache entry back later. The implementations include an in-memory cache called *Cache::MemoryCache* and *Cache::FileCache* a file-based persistent cache. *Position.pm* uses the following

```
my $cache = Cache::➥
FileCache->new(
  { namespace          =>➥
   'Beancount',
    default_expires_in ➥
    => 600,
  });
```

to create a cache object and handles all the details, such as efficient storage in

### Listing 4: Position.pm

```
001 ##################################
002 package Plugger::Position;
003 ##################################
004 use strict;
005 use warnings;
006 use Log::Log4perl qw(:easy);
007 use Finance::YahooQuote;
008 use Term::ANSIColor;
009
010 ##################################
011 sub init {
012 ##################################
013   my ($class, $ctx) = @_;
014
015   DEBUG "Registering @_";
016
017   $ctx->register_cmd(
018     "stock",   undef,
019     \&process, undef
020   );
021   $ctx->register_cmd("cash",
022     undef, \&process_cash,
023     undef);
024 }
025
026 ##################################
027 sub process {
028 ##################################
029   my ($ctx, $cmd, @args) =
030     @_;
031
032   my $value =
033     price($args[0]) *
034     $args[1];
035   my $gain =
036     $value - $args[2] *
037     $args[1];
038
039   Plugger::Account::position(
040     ucfirst($cmd),
041     @args[ 0 .. 2 ],
042     price($args[0]),
043     $value,
044     $gain
045   );
046
047   my $mem = $ctx->mem();
048   $mem->{account_subtotal} +=
049     $value;
050   $mem->{account_total} +=
051     $value;
052 }
053
054 ##################################
055 sub process_cash {
056 ##################################
057   my ($ctx, $cmd, @args) =
058     @_;
059
060   my $mem = $ctx->mem();
061   $mem->{account_subtotal} +=
062     $args[0];
063   $mem->{account_total} +=
064     $args[0];
065
066   Plugger::Account::position(
067     ucfirst($cmd),
068     (undef) x 4,
069     $args[0], undef);
070 }
071
072 use Cache::FileCache;
073
074 my $cache =
075   Cache::FileCache->new(
076   {
077     namespace => 'Beancount',
078     default_expires_in =>
079       600,
080   }
081   );
082
083 ##################################
084 sub price {
085 ##################################
086   my ($stock) = @_;
087
088   DEBUG
089     "Fetching $stock quote";
090
091   my $cached =
092     $cache->get($stock);
093
094   if (defined $cached) {
095     DEBUG "Cached: $cached";
096     return $cached;
097   }
098
099   my @quote =
100     getonequote $stock;
101
102   die "$stock failed"
103     unless @quote;
104   $cache->set($stock,
105     $quote[2]);
106
107   return $quote[2];
108 }
109
110 1;
```

temporary files without colliding with other applications. Users simply call *$cache->set()* and *$cache->get()*.

## Rich versus Poor

Of course, *beancounter* is totally over-engineered for a simple statement of account tool – obviously the work of an architecture astronaut; thanks to Joel Spolsky for hitting the nail on the head at [4]. The plugger framework really comes to its own when it is asked to add user-specific functionality without modifying the original code.

The *Plugger/TaxedPosition.pm* plugin in Listing 5 gives you an example. *Plugger/TaxedPosition.pm* subtracts 50 percent tax from (possible) share profits defined by *txstock*. This "Desert island dreams" mode gives you the bottom line if you were to cash your shares and pay taxes at 50 percent. *TaxedPosition* does not subtract anything if your shares have made a loss, but simply gives you the face value of the shares after liquidating the losers.

Depending on the scenario, users can write plugins for new keywords, add them to the framework, and modify the system. As *TaxedPosition.pm* references the *price()* function defined in *Position.pm*, it makes sense to use an inheritance or interface mechanism to link *TaxedPosition.pm* and *Position.pm*. As the plugger framework does not have classes, *TaxedPosition.pm* in line 10 of Listing 5 simply defines an *AUTOLOAD* handler which channels calls from unknown functions to *Position.pm*.

To facilitate screen output and ensure manageability, the *Position.pm* plugin handles all screen output. The *position()* function expects the data for an output item: type, ticker symbol, number, buying price, current price, current total value, profit and loss, and provides neatly formatted output of the results. Cash entries only need the left and right columns.

Your own plugins should utilize Plugger::Account's position method "printing" just like *TaxedPosition.pm*. The *price()* function in *Position.pm* should also prove useful for your own plugins.

## Installation

Both the *beancounter* script (Listing 1) and the compiled C wrapper *beancount* need to be stored in */usr/bin* and must be executable. *Plugger.pm* (Listing 2) and all the plugins below *Plugger/* should be in one of your Perl environment's *@INC* paths. If not, you could use a line such as

```
use lib '/home/mschilli/perl-modules';
```

in the *beancounter* Perl script to publish the path, assuming that Plugger & co. are stored in the directory specified here. The Module::Pluggable, Finance::YahooQuote and Term::ANSIColor modules are available from CPAN. The easiest way to install them is to use a CPAN shell. After doing so, there is nothing to stop Perl from ruling your finances! ■

### INFO


[1] Listings: *http://www.linux-magazine.com/Magazine/Downloads/53/Perl*

[2] Module::Pluggable tutorial: *http://www.perladvent.org/2004/6th*

[3] Symbols for popular UK shares: *http://uk.biz.yahoo.com/p/uk/cpi/cpia0.html*

[4] Joel Spolsky, "Don't let Architecture Astronauts scare you" in *Joel on Software*: Apress 2004.


### Listing 5: TaxedPosition.pm

```
01 ##################################
02 package
03       Plugger::TaxedPosition;
04 ##################################
05 use strict;
06 use warnings;
07 use Log::Log4perl qw(:easy);
08
09 ##################################
10 sub AUTOLOAD {
11 ##################################
12   no strict qw(vars refs);
13
14   (my $func = $AUTOLOAD) =~
15  s/.*::/Plugger::Position::/;
16   $func->(@_);
17 }
18
19 ##################################
20 sub init {
21 ##################################
22   my ($class, $ctx) = @_;
23
24   $ctx->register_cmd(
25     "txstock", undef,
26     \&process, undef
27   );
28 }
29
30 ##################################
31 sub process {
32 ##################################
33   my ($ctx, $cmd, @args) =
34     @_;
35
36   my $value =
37     price($args[0]) *
38     $args[1];
39   my $gain =
40     $value - $args[2] *
41     $args[1];
42
43   my $tax = $gain / 2;
44
45   $value -= $tax
46     if $gain > 0;
47   $gain -= $tax if $gain > 0;
48
49   Plugger::Account::position(
50     ucfirst($cmd),
51     @args[ 0 .. 2 ],
52     price($args[0]),
53     $value,
54     $gain
55   );
56
57   my $mem = $ctx->mem();
58   $mem->{account_subtotal} +=
59     $value;
60   $mem->{account_total} +=
61     $value;
62 }
63
64 1;
```