A Public/Private Extension of Conway's Accessor Model

Karen M. McCann and Maurice Yarrow

Computer Sciences Corporation, Mail Stop T27A-1
NASA Ames Research Center, Moffett Field, CA 94035
{mccann,yarrow}@nas.nasa.gov

Abstract

We present a new object-oriented model for a Perl package, based on Damien Conway's "accessor" model. Our model includes both public and private data; it uses strategies to reduce a package namespace, but still maintains a robust and error-trapped approach. With this extended model we can make any package data or functions "private", as well as "public". (Note : "namespace" in this context means all the names, variables and subs, associated with a package.)

Motivation and Background

For the past six months we have been working on a project called "ILab: The Information Power Grid Virtual Laboratory". This is a Perl program which provides a Perl/Tk user interface and generates scripts for parameter studies being run on the Grid. (Note that the expression "Grid" is now being used to describe the world-wide network of supercomputers, both parallel and serial, that scientists are using to run scientific problems that involve data sets too large to fit on desktop systems.) This program has an extensive user interface; we already have about 15,000 lines of Perl, and we anticipate about 50,000 lines when all options are implemented. With such a large project, it became relatively important for us to organize our data and operations into packages, in order to speed up development and debugging. We have searched the Perl literature for reliable models to use for our packages, and by adding some new ideas we have developed a new object-oriented approach.

Many Perl authors have lamented the fact that package member variables in Perl cannot be completely protected from outside access. In spite of various methods of "protecting" variables, there is always some way that a programmer can access the protected variables. In comparison, variables declared to be private in C++ or Java can only be accessed by a class's own functions, or by the functions of a derived class or "friend" class (C++). Any attempt to access private variables from outside the class will cause a compile-time error. In C++ and Java, this data protection is built in to the language itself; in Perl, data protection has to be constructed, and a adequate construction method has not yet been presented in the current Perl literature. The methods we have seen so far make it inconvenient, not impossible, to access data members from outside a package.

Conway's Accessor Model

The most interesting of these methods was presented by Damien Conway [1]. He describes a truly clever and useful way of creating a Perl "closure" inside a constructor function. This package, called `Soldier` (see Listing 1), illustrates a method for hiding the data members of a Perl package, thereby making these members private in a manner similar to C++ and Java. The strategy is that users (i.e., programmer-users) of package `Soldier` cannot access the data members directly, only through the "get" or "set" functions associated with those members.

In Conway's example, the strings `name`, `rank` and `serial_num` are put in an array and then used as keys in hash `%data`; the anonymous sub whose reference is `$accessor` is the only way to access this hash. The anonymous sub is a closure, i.e., any local data accessed by this contained sub is made non-transient as long as the reference to that sub does not go out of scope. Even though `%data` is a local (`my`) variable, it does not disappear after the call

to `new` returns. The `new` sub (which is package `Soldier`'s constructor) then blesses and returns the anonymous sub reference, instead of blessing and returning the `%data` hash as `$self` (the most commonly used Perl strategy for creating packages). As a result, the reference returned from the call to `new` can be used not only to call the subs attached to this package, but can also be used to get/set the data members.

Here, all data members except `rank` have been protected from assignment by putting restrictions in the accessor sub, and by not providing set functions for them. This brings Conway's `Soldier` package very close to being completly object-oriented: a) the package's data members are not visible to outside users except as specified by the get/set functions, as they would be in C++ or Java; b) the package's interface is made up of functions, not data

Listing 1: Conway's `Soldier` *package*

```
#-----------------------------------------------------------------------------
package Soldier;
$VERSION = 1.00;
use strict;

use Carp;

my @attrs = qw(name rank serial_num);

sub new
{
    my ($class, %args) = @_;
    my %data;
    @data{@attrs} = @args{@attrs};
    my $accessor =
        sub
        {
            my ($cmd, $attr, $newval) = @_;
            croak "Invalid direct access: use the ${cmd}_$attr method instead"
                unless caller()->isa("Soldier");
            return $data{$attr}
                if $cmd eq "get";
            return $data{"rank"} = $newval
                if ($cmd eq "set" && $attr eq "rank");
            croak "Cannot $cmd attribute $attr";
        };
    bless $accessor, ref($class)||$class;
}
# These methods provide the only means of accessing object attributes
# (note that only rank can be changed)

sub get_name    { $_[0]->('get','name') }
sub get_rank    { $_[0]->('get','rank') }
sub get_serial_num    { $_[0]->('get','serial_num') }

sub set_rank
{
    my ($self, $newrank) = @_;
    $self->('set','rank',$newrank);
}


no strict;
package main;

my $s = Soldier->new(name=>"ryan",rank=>"private",serial_num=>"1000000");

print $s->get_name(), "\n";
```

```
print $s->get_rank(), "\n";
print $s->get_serial_num(), "\n";
$s->set_rank("general"), "\n";

$s->("set","rank","Colonel");
print $s->("get","serial_num");

print $s->get_name(), "\n";
print $s->get_rank(), "\n";
print $s->get_serial_num(), "\n";

##############################################################################
#                                                                            #
#  Copyright (C) 1999 by Manning Publications Co. All Rights Reserved.       #
#                                                                            #
#  This code is free software. It may be used, redistributed                 #
#  and/or modified under the terms of the Perl Artistic License              #
#  (see http://www.perl.com/perl/misc/Artistic.html).                        #
#                                                                            #
##############################################################################
#-----------------------------------------------------------------------------
```

Accessor Closure

This is an amazing trick! The Perl interpreter is very flexible and the reference to `$accessor` still works for calling the package's subs, even though `$accessor` is a reference to an anonymous sub, instead of a reference to a `$self` hash. And, it does indeed make the data members private, i.e., not accessible from outside the package. However, it has some unexpected effects: (1) some variables (`rank` and `serial_num`) can only be set through the constructor call to `new`, and the assignment in the constructor is not error-trapped; (2) there are no public data members - all data members are private; (3) the interface to the package is only functions: no data (note that this effect may be a desirable one); (4) the package's own subs can't set any data members except `rank`: the error-trap in the anonymous sub, which is meant to prevent outside setting of data members, also prevents inside setting of the same data members; (5) a package derived from Soldier can't have its own data, only its own functions (the Perl interpreter won't let you treat that `$accessor` value as a `$self` hash).

Now, this is carrying protection too far; we don't want to protect the package variables from the package's own functions. Furthermore, setting variables only through the constructor may not be a good design.

Some Design Requirements

We would like a design that satisfies the following considerations: (1) the package's private variables should be only accessible from outside through get/set functions; (2) the package's own functions should be able to freely access all variables; the constructor should not be the only function that can set any private variable; (3) reasonable error-trapping and initialization should be possible; (4) large amounts of code, or too many repetitions of the same pieces of code, should be avoided; (5) the namespace of the package should not be unduly affected by these constraints (in other words, we should not have to add too many functions or variables to our package in order to accomplish this data hiding); (6) the design should not require large amounts of code and error-traps in the constructor itself; (7) we would like to be able to have public data members, as well as private data members.

It is easy to see that Conway's design only satisfies considerations (1), (3), and (arguably), (4) and (5).
Nigel Chapman has presented a similar method [4]. Chapman's method is also effective, but does not satisfy (2) and (6) above.

Why Private Variables?

The use of private variables and functions is really the cornerstone of object-oriented programming. According to object-oriented theory, an "object" should have its own data and its own functions; it should read itself, write itself, draw itself, error-check itself, etc. The object's private data and functions should be those data and functions that are necessary to these private operations; the public data and functions should be those that users need to set and call in order to use the object; these public data and functions are said to make up the object's interface. Many C++ programmers take the approach that all class data should be private except for the data that needs to be public; that, ideally, all object data should be private, and accessible only through get/set functions. However, in practice this ideal is rarely accomplished, since: (1) very few objects are completely free from "lateral" dependence on other objects; (as opposed to "vertical", or inherited, dependence); (2) mostly, C++ programmers do not bother to write get and set functions for every single data variable; (3) truly private data, in C++, introduces many lexical, logical, and procedural complications; this is due to the large set of C++ keywords that are used for different types of data and function protection. In Java, the same kind of complications are introduced, just fewer of them. This brings us to the contrary position:

Why Public Variables?

Should all package data members be made private? In object-oriented theory, are there reasons for not error-trapping data members of a package by providing get/set subs? In other words, creating public data members, which can be assigned to without any error-trapping? (Note that the `$self` hash strategy gives us public data members in a package.) Well, certainly! Here are five reasons, and we are sure that readers could provide more: (1) error-trapping is not needed; UNDEF fields are valid (example: consider a data field called "Comment" that a user can fill in during the course of creating some document, where the edit box for entering "Comment" only allows 128 chars.); (2) error-trapping was done in the user interface (examples: a float or integer field which user can only enter with a slider that already has min-max values; an edit box that only allows character entry; a dialog that will not accept a filename unless the field is not NULL and the file exists; etc.); (3) the setting of the data, and the error-trapping of the data, are done in different packages; (4) the package's data member was a copy of another package's data member, and the other package had error-traps (a special case of ((3)); (5) you're developing the code and you don't know yet what error conditions should be put on some data members, etc.

Philosophical Basis for Privatization

In object-oriented theory, there are three reasons for privatizing data and operations:

(1) Organization, enforced isolation, "encapsulation": Essentially, each object is a "library" of functions and associated data (OR: data and associated functions; depending on the design requirements of the object. Although objects are usually modelled on sets of data, occasionally objects need to be modeled on a particular operation.) If objects are well designed, code will be "modular": relatively free from interdependencies, and easy to debug since object operations are limited in scope to that object. This encapsulation allows internal representation to be changed without affecting any users of an object. There are many types of objects that do not fall into this library category; however, the organizational principle still applies. It can be argued that organizational considerations are independent of privatization; that is, we get the organizational improvements of organizing code into objects, whether we make the object data and functions private, or not.

(2) Error-trapping: Since the initialization and assignment of private variables is error-trapped, it should be either difficult or impossible for a programmer to assign incorrect data to the private members of an object. Typically, the set functions associated with each data member should prevent assignment to values that are not allowed. This prevents a large amount of possible programming mistakes.

(3) Paranoia: "Programming Correctness": This appears to be one of the primary "philosophical" motivations behind private data. The theory is that programmers, who have to use objects designed by other programmers, cannot introduce incorrect data, or incompatibilities, into the used/derived objects as long as the base objects have private data and functions. At its worst, this kind of thinking leads to the delusion that correctly designed objects will not only provide for every kind of error that might ever possibly happen, but also, that these correctly designed objects will never need to be re-designed.

The organizational and error-trapping benefits of programming with objects and hidden data are generally accepted as “Good Things” by programmers. The third consideration - which, as pointed out above, is based on Programming Correctness - is a sticky point that generates much debate, both formal and informal.

Interestingly enough, it can be argued that private data and functions do nothing to increase the extensibility of an object, and in fact may interfere with it. Typically programmers extend objects by deriving new objects from base objects; the derived object has everything the base object has, plus more if needed, and the base object’s functions can be overwritten by the derived object, also as needed. In C++ this can be especially problematic; if the base object’s data is private (in C++, only visible to the base object) when it should be protected (in C++, also visible to any objects derived from the base object), the programmer of the derived object can find that it is impossible to extend the base object as necessary. Since the programmer of the base object typically does not know what future design considerations will be brought to bear upon objects derived from the base object, this kind of “cut-off” happens much more often than it should; and, when it does, the programmer of the derived objects cannot do anything except modify the base class. (However, sometimes this may not be possible; we provide for this case in our derived class, Listing 3.)

However, we would still like to get the benefits of hidden data out of a Perl package, and especially without being too strict about it. Here is an extension of Damien Conway’s `Soldier` class, which satisfies more of the criteria mentioned above, and gives us some added benefits, as well:

Listing 2: Extended Soldier package

```
#-------------------------------------------------------------------------
#    Soldier.pl
#    Sample package that illustrates Object Oriented data hiding by
#    using a Perl "closure".
#    Based on "soldier.pl" written by Damien Conway.

package Soldier;
$VERSION = 2.00;
use strict;

sub new    {                    # CONSTRUCTOR for package SOLDIER
    my ($class, %args) = @_;    # hash "args" is values passed in

    # Create a $self hash, init/add the one public variable to it :
    my $self = {};

    # Array of PUBLIC data member names; assign if passed in :
    my @PublicNames = qw(SoldierName);
    foreach my $attr (@PublicNames)
        { if ( defined( $args{$attr} ))
            { $self->{$attr} = $args{$attr}; } }

    # Create anonymous subs and put refs to them into the $self hash :
    my %PrivateHash;            # this hash made permanent by closure
    $self->{SoldierGet} =
        sub { my ($attr) = @_; return $PrivateHash{$attr}; };

    $self->{SoldierSet} =
        sub {  my ($attr, $newval) = @_;
            my ($package, $filename, $line) = caller();
            if ( $package->isa('Soldier'))
                { return ($PrivateHash{$attr} = $newval); }
            else
                { print "ERROR : illegal attempt to set ",
                        "a closed variable\n"; return; } };
            # Alternate error trap:
            # if ( $package->isa('Soldier') && $filename eq 'Soldier.pl' )

    bless $self, ref($class)||$class;
```

```
    # An array of PRIVATE data member names; assign if passed in by
    #    calling the SAME NAME subs :
    my @PrivateNames = qw(Rank SerialNumber);
    foreach my $attr (@PrivateNames)
        { if ( defined( $args{$attr} ))
            { $self->$attr( $args{$attr} ); } }

    return $self;
}

# These methods are the only way to access "private" object attributes
#-------------------------------------------------------------------------
sub Rank {
    my ($self, $value) = @_;

   # Only allow rank to be set to one of these values :
    my @RankNames =
        qw( Private Sergeant Lieutenant Corporal Major Captain General );
    if ( defined( $value ))
        { if ( grep {$value eq $_} @RankNames )
            { return $self->{SoldierSet}->('Rank', $value); }
        else
            { print "attempt to set rank to an non-allowed value\n"; } }
    else
        { return $self->{SoldierGet}->('Rank'); }
}

sub SerialNumber {
    my ($self, $value) = @_;
    if ( defined( $value ))
        { # Don't allow serial number to be set to a negative value:
        if ( $value > 0 )
            { return $self->{SoldierSet}->('SerialNumber', $value); }
        else
            { print "attempt to set SerialNumber to less than 0\n"; } }
    else
        { return $self->{SoldierGet}->('SerialNumber'); }
}

sub DumpSoldier {
    my $self = shift;
    print "Name = ", $self->{Name}, ", Rank = ", $self->{SoldierGet}->('Rank'),
        ", Serial Number = ", $self->{SoldierGet}->('SerialNumber'), "\n";
}

#-------------------------------------------------------------------------
no strict;
package main;

    my $s = Soldier->new(Name=>"Ryan",Rank=>"Private",SerialNumber=>"1000000");
    print "\nSoldier after call to new = \n";
    $s->DumpSoldier();

    print "\nSetting the name, public variable\n";
    $s->{Name} = "GI Joe"; $s->DumpSoldier();
    print "\nSetting the rank, private variable\n";
    $s->Rank("General"); $s->DumpSoldier();
    print "\nSetting the serial number, private variable\n";
```

```
    $s->SerialNumber("1010101"); $s->DumpSoldier();

    print "\nGetting the name\n"; print $s->{SoldierName}, "\n";
    print "\nGetting the Rank\n"; print $s->Rank(), "\n";
    print "\nGetting the SerialNumber\n"; print $s->SerialNumber(), "\n";

    print "\nTest: trying to set private variable Rank via accessor\n";
    $s->{Set}->( 'Rank', 'General' );
#-------------------------------------------------------------------------
```

Get/Set Functions and the Same Names

In Perl it is easy to "collapse" the get and set functions into one function. If an additional argument is passed into the function, it is a set operation; else, it is a get operation. This reduces our get/set namespace by 50%: only one get/set function per data member.

Here is another simplification: in the above example, the data members have the same names as their associated get/set functions. These data names are really just strings that key into the `%PrivateHash` hash, and they are hidden by the Perl closure. The effect of this strategy is to further reduce the namespace of variables and associated get/set functions; again, reduced by 50%. By collapsing the get/set functions, and using "Same Names", the *effective* namespace for data members and their get/set functions is reduced by a total of 75%. Over the course of a large program, this effect is non-trivial: there are fewer names for a programmer to remember or reference; the code is simpler, making it easier to write, debug, and modify.

Additionally, we want to make sure that the accessor reference returned from the "new" function cannot be used outside of the package's own functions. Conway shows us how to do this: use the `caller` function, which returns the package name, file name, and line number of the calling function, from within a called sub. By using function `caller` inside the accessor closure, we can tell where the accessor was called from; one line of code suffices to eliminate calls that did not come from inside package `Soldier` or a descendant package. Now our data members are truly private, i.e., not visible from outside a package, but easily accessible from inside the package. As far as code using (or calling) package Soldier is concerned, the effect is even more amazing: the data members of the package are as easy to access as public variables, but they are actually private!

From within a package's functions, it is now more inefficient to get a private variable than a public variable:

`$self->{SoldierGet}->('Rank');` (a private variable)

- is a function call plus a hash lookup;

`$self->{Name}` (a public variable)

- is only a hash lookup. The overhead from the extra function call is not very much, but if a package function needs to use a private data member in a loop, the function can create a local copy to use in the loop:

```
my $localRank = $self->{SoldierGet}->('Rank');
```

- otherwise, just put `$self->{SoldierGet}->('Rank')` in-line, as you would put `$self->{Name}` in-line.

Now, functions inside package Soldier can get/set any data member; we don't have to put error-trapping inside the accessor function, as in the first example. Error-traps can be put where they belong, in the get/set functions. In the above example, data member `Rank` is limited to a small set of allowable values, and data member `SerialNumber` is restricted to values greater than 0; data member `SoldierName` has no error traps. We do initialization in the `new` sub, as well: 1) for public data members, simply assign by keying into the `%args` passed in; 2) for private data members, also key into `%args`, but call the Same Name subs instead of assigning. If it were advisable to initialize any data members that were not set in the constructor arguments, then we would call a separate initialization function near the bottom of the constructor. This function would assign default values to any data members that are still undefined after the constructor arguments are processed.

In the first version of Soldier, the setting of data members in the constructor is not error trapped. The line of code:

```
@data{@attrs} = @args{@attrs};  # set member values to args values
```

- simply assigns to data members the values passed in, without doing any error-checking. Note that the call to `new` makes use of the Perl "hash" assignment syntax in the argument list:

```
my $s = Soldier->new(Name=>"ryan",Rank=>"private",SerialNumber=>"1000000");
```

Our method of calling the Same Name functions catches errors in the constructor call; for example, if the constructor was called with argument `Rank => 'General'`, the line of code in the second example :

`$self->$attr( $args{$attr} );`
- will call the sub `Rank` with argument `'General'`, which will test the argument for an allowed value. Note that if we disallowed or ignored any arguments passed in to the constructor, users of package Soldier would be forced to set data members one by one with the get/set functions. We wanted to avoid this, since the hash assignment syntax in an argument list is standard Perl, and using it for several variables at once is very convenient.
In the first example, Conway returned `$accessor` from the constructor, only to cut off access to it from outside (in "Variation for the Paranoid"); what, then is the point of returning it from the constructor? In the second version, instead of returning `$accessor`, we return `$self`; we have *two* accessors (one for getting, one for setting), and we just plug references to these functions into the `$self` hash. The accessors are restricted as before so no user of package Soldier can call them from outside the package. Package `Soldier` now has *two* hashes of data members: a hash called `%PrivateHash` which contains all private data members, and the `$self` hash that we are accustomed to using, which will contain public data members.

Improvements

Note that since data member `Name` is now public, we didn't provide a get/set function for it; users of package `Soldier` can get or set data member "Name" directly through the `$self` hash. 1) the get accessor doesn't need a "caller/isa" trap: any outside code can "see" the private variables, but still has to go through an error-trap to set them; 2) we have both public and private variables in package `Soldier`; 3) we can still allow hash assignment syntax in the call to the constructor, and this hash syntax can be used for both public and private variables, and, any private variables in the hash syntax are still error-trapped; (Note that the use of an Init function is recommended by OOP experts.) 4) private data is readily accessible by all package subs, and all derived package subs. Note the ease of using our new package Soldier: private and public data members are named in the same way, and the difference of accessing them is just trivial syntax:

```
$s->{Name} = "GI Joe";
```

- is an assignment to a public data member, while:

```
$s->Rank("General");
```

is a function call that assigns a value to a private data member. This, after all, is the point of our package model: we are willing to put up with some extra lines of code in a package, if it makes the calling code simpler, more robust, and easier to use; since, over the course of the large ILab project, we are going to have a lot more calling code than package code. The error-trapping is all in the package; we don't have to bother with error-trapping in the calling code.

By-passing the Error-traps

As Conway has pointed out, a *determined* programmer can still bypass the error-trapping in the set accessor, by re-opening package `Soldier` outside of file Soldier.pl, and then issuing a direct set call as in the test above. Here is his example [1,p. 301-302]:

```
my $soldier = Soldier->new(name=>'Alexander', rank => 'General' );
package Soldier;
$soldier->('get', 'name'); # call the accessor directly
```

The main purpose of all this error-trapping and privatizing is not to "keep programmers out of" a piece of code: instead, it is to make it easier for *any* programmer to see where a mistakes are made during code development.

On the other hand, a *more determined* programmer might extend the error-trap in the accessor function, by using the alternate error trap. Instead of:

```
if ( $package->isa('Soldier'))
    { return $PrivateHash{$attr} = $newval; }
```

use:

```
if ( $package->isa('Soldier') && $filename eq 'Soldier.pl')
    { return $PrivateHash{$attr} = $newval; }
```

Well, now the merely determined programmer is in trouble: re-opening the package won't work anymore. The only way to bypass the error-trapping is to add code to the file Soldier.pl! But, that is the point: if a programmer has to modify a base class, then the programmer should modify the base class, but should not do the modification in an outside file where it is hard to find.

As pointed out above, it is a mistake to assume that a base class will never need to be modified. In our second version of Soldier, for example, the sub `Rank` doesn't include all possible strings describing `Rank`, and a user of package Soldier might well need to extend it. Of course this user could derive a new package from Soldier, and give this derived package its own private variable named `Rank`, and overwrite the `Rank` sub; this might be preferable in some programming situations.

Derived Packages and Namespaces

Note that this additional error-trap in the accessor function means that derived packages can't use the get/set accessors directly, either: like any other outside package, a derived package must use the base package's Same Name functions for the base package's private data. In C++ and Java, derived and parent classes can have data variables with the same names: the class namespaces are distinct. In Perl, the namespace situation is more complicated.

In C++, Java, and Perl, if a derived object has a function with the same name as a function in the parent object, the derived object's function is said to overwrite the parent's function; i.e., the derived object's function is called instead of the parent's function. In this situation it is often necessary for the derived object's function to call the parent's function, as well as to execute its own statements. In all of these languages, the parent's function does not disappear, but is still available through the syntax `ParentName::FunctionName`.

In Perl, if parent and derived packages both have public data members with the same names, the derived package's data member overwrites the parent's data member: this is because the parent and derived packages are sharing the same namespace, which in this case is the `$self` hash. Derived Perl packages share their parent's data namespace, but the function namespace remains distinct. As a result, Perl derived packages that wish to have their own data - in addition to the parent's data - must in their "new" functions call the parent's "new" function. For example:

```
package SoldierBoy;
use base qw( Soldier );
sub new {
    my ($class, %args) = @_;
    my $self = SUPER::new( %args ); # SoldierBoy gets Soldier's data
    $self->{Member1} = undef;       # SoldierBoy's public variables
    $self->{Member2} = 10;
    bless $self,$class;             # re-blessing of $self
    return $self;
    }
```

Here, package `SoldierBoy` is derived from package `Soldier`; instead of instantiating `$self` as a empty hash, `SoldierBoy` calls `Soldier` new, and uses the return value as its `$self` hash. `SoldierBoy` can add as many values as it wants to this `$self` hash; in this way, the `SoldierBoy` package has its own public data. Note the danger, though : if `Soldier` also had data members called `Member1` and `Member2`, `SoldierBoy`'s initialization of these two members would wipe out the values that `Soldier` had put in the same members. If `Soldier`'s values were needed for some reason, `SoldierBoy` would have to either make copies of those members under different names, or re-name its own members so that there are no name collisions. (The first of these alternatives is not a good design approach.)

Since our version of `Soldier` has both private and public data, we would like a design where any package derived from `Soldier` can have its own public and private data, as well. It is actually very easy to accomplish this. `SoldierBoy` can have its own get/set accessor references and `%PrivateHash`, in the same way that `Soldier` does; the only caveat is that `SoldierBoy` must not overwrite the parent's get/set accessor values in the `$self` hash that they share. This is why we have given long names - `SoldierGet` and `SoldierSet` - to the parent's accessor references: `SoldierBoy` will have to have similar unique names (`SoldierBoyGet` and `SoldierBoySet`) so that there are no namespace collisions. Listing 3 illustrates a version of `SoldierBoy` that overwrites one of `Soldier`'s private variables.

Listing 3: Derived Package SoldierBoy.pm

```
#-----------------------------------------------------------------------
use Soldier;
package SoldierBoy;
use base qw( Soldier );
use strict;

sub new {
    my( $class, %args ) = @_;
    my $self = new Soldier( %args );    # call parent's constructor

    # Array of PUBLIC data member names, assign if passed in :
    my @PublicNames = qw(PublicField);
    foreach my $Name (@PublicNames)
        { if ( defined( $args{$Name} ))
            { $self->{$Name} = $args{$Name}; } }

    # CLOSURE : accessor functions which make PrivateHash a permanent
    #    feature of this instance of this package
    my %PrivateHash;
    $self->{SoldierBoyGet} =
        sub { my ($attr) = @_; return $PrivateHash{$attr}; };

    $self->{SoldierBoySet} =
        sub {  my ($attr, $newval) = @_;
            # Find out who called this anonymous sub; if not from
            #    THIS package or descendant, dump 'em out :
            my ($package, $filename, $line) = caller();
            if ( $package->isa('SoldierBoy') && $filename eq 'SoldierBoy.pm')
                { return ($PrivateHash{$attr} = $newval); }
            else
                { print "SoldierBoy ERROR : illegal attempt to set ",
                        "a closed variable\n"; return; } };

    bless $self, ref($class)||$class;

    # An array of PRIVATE DATA names; call Same Name functions to init
    #    private members to any values passed in :
    my @PrivateNames = qw( Rank DateOfBirth HealthCode );
    foreach my $Name (@PrivateNames)
        { if ( defined( $args{$Name} ))
            { $self->$Name( $args{$Name} ); } }

    return $self;
}

sub DateOfBirth {
    my ($self, $value) = @_;
    if ( defined( $value ))
        { # Don't allow DateOfBirth to be set to a negative value :
        if ( $value > 0 )
            { return $self->{SoldierBoySet}->('DateOfBirth', $value); }
        else
            { print "attempt to set DateOfBirth to less than 0\n"; }
        }
    else
        { return $self->{SoldierBoyGet}->('DateOfBirth'); }
}

sub HealthCode {
```

```
    my ($self, $value) = @_;
    if ( defined( $value ))
        { if ( $value eq 'OK' || $value eq 'FlatFooted' )
            { return $self->{SoldierBoySet}->('HealthCode', $value); }
        else
            { print "attempt to set HealthCode to non-allowable string\n"; }}
    else
        { return $self->{SoldierBoyGet}->('HealthCode'); }
}

sub Rank {
#   Derived class's overwritten sub for setting OWN version of a PRIVATE
#    VARIABLE
    my ($self, $value) = @_;
    # Only allow rank to be set to one of these values :
    my @RankNames =
qw( Private Sergeant Sergeant-Major Lieutenant Corporal Major Captain General );
    if ( defined( $value ))
        { if ( grep {$value eq $_} @RankNames ) # if value is in RankNames
             { return $self->{SoldierBoySet}->('Rank', $value); }
          else
             { print "Soldier: attempt to set rank to non-allowed value\n"; }}
    else
        { return $self->{SoldierBoyGet}->('Rank'); }
}

sub DumpSoldier {
    my $self = shift;
    print "SOLDIER BOY dump soldier\n";
    print "Name = ", $self->{SoldierName},     # parent's public member
        ",\nSerial Number = ", $self->SerialNumber(), # parent's private member
        ",\nRank = ", $self->Rank(),           # own overwritten private member
        ",\nPublic field = ", $self->{PublicField},   # own public member
        ",\nDateOfBirth = ", $self->DateOfBirth(),    # own private members
        ",\nHealthCode = ", $self->HealthCode(), "\n";
}
1;
#-------------------------------------------------------------------------
```

This derived package has its own public data and its own private data. In addition, one of its private data members, `Rank`, *overwrites* a private data member in parent package `Soldier`. Note that the `SoldierBoy` constructor calls `Soldier`'s constructor, passing in `%args`. Inside `Soldier`'s constructor, when the lines of code :

```
   my @PrivateNames = qw(Rank SerialNumber);
    foreach my $attr (@PrivateNames)
        { if ( defined( $args{$attr} ))
            { $self->$attr( $args{$attr} ); } }!
```

are executed, instead of calling `Soldier's Rank` sub, `SoldierBoy's Rank` sub gets called. (Likewise, for any other calls inside package `Soldier`'s subs to sub `Rank`.) In this way, `SoldierBoy` has replaced `Soldier`'s private variable Rank with its own version of `Rank` - and also, of course, its own version of the (Same Name) `Rank` sub. The advantage of this design feature is that a user of package `Soldier` is no longer prevented from replacing a parent's private variable if necessary. If this programmer-user cannot modify the code in package `Soldier` for some reason, (e.g., if the `Soldier` package was in some library format, or the programmer-user couldn't modify the distributable `Soldier` package) this feature can be a life saver, since the sins of the parent are no longer visited upon the children (so to speak).

Drawbacks

Of course, our design model has a few ”gotchas”. Here they are :
1) the parent’s subs must not call the accessors directly *except* in the Same Name subs; otherwise, these calls will be accessing the *parent’s* version of the private variable, not the child’s version. If this happens it will probably introduce subtle bugs that will be hard to find.
2) if the child wants to overwrite a parent’s variable, the child must not only have its own variable in its own private hash, it must also have its own (overwritten) version of the parent’s Same Name sub.
3) when a parent’s private variable is overwritten in this way, it could easily introduce bugs into the parent’s subs, since the variable may now take on values not allowed for in the parent’s subs.
4) parent and child packages have to have unique names for the accessor sub references.
This design model is somewhat ”stiff”, i.e., non-flexible, in the sense that once the parent package is written this way, the child packages should follow the same design model (although, they don’t have to). Overall, we feel that the advantages outweight the disadvantages.

Varieties of Packages

Now we have many options available for encapsulation. Our packages can have:
1)all private data (just don’t put in a `$self` hash, and return the `$accessor` from the constructor);
2) all public data (with the `$self` hash, this is the way we have been doing it anyway);
3) a mixture of public and private data, as in the second `Soldier` example;
4) private data that is invisible from outside the package (put the `caller` error-trap in the get accessor, as well as in the set accessor)
5) private functions that are not callable from outside (just put the `caller` error-trap inside the function that you want to be private.)

Summary

By using Damien Conway’s accessor design, along with collapsed get/set functions, Same Name naming convention, an error-trapped constructor, and two data member hashes, we have a package design that gives us private variables that are still easy to access, and public variables as well. This approach is not only robust, but also gives us a simple package design without sacrificing flexibility or requiring large amounts of code. In C++ and Java, making a variable or function private means restricting its lexical scope, without any error-trapping considerations; in our example, ”private” means error-trapped *and* lexically restricted. In addition, a derived package can overwrite its parent’s private *data* as well as its parent’s functions (-provided that the base package’s subs only access the private data members through the Same Name functions, and do not call the accessor directly.)
We propose using the term “closed variables” to refer to data members that are made private in this way, since these data members are not really private as in C++ or Java.
Overall, we expect this approach to cut down development and debugging time in addition to the speed and ease of use already offered by Perl. Our opinion is that Perl takes object-oriented programming to a more evolved level than C++ or Java, since Perl is just as powerful, but does not have the “decoration” of multiple keywords.

References

[1] Damien Conway, “Object-Oriented Perl”, Manning Publications (2000), 299–302

[2] Jon Orwant, “Perl5 Interactive Course Certified Edition,” Waite Group Press (1998), Chapter 7.

[3] L.Wall, T.Christiansen, and R.Schwartz, “Programming Perl”, O’Reilly and Associates, Inc. (1996) 289–325.

[4] N.Chapman, “Hiding Data Objects with Closures”, *The Perl Journal*, Vol.4, No. 3, Fall 99, 50-55.