

# Structured Classes in Perl

*Robby Walker* \*

## Abstract

Perl's class system is like no other – its utter lack of structure makes it very simple and elegant. Unfortunately, this design also robs Perl of some power that more structured class systems can offer. While I do not suggest that Perl's class system should be changed, as its down and dirty approach can be really useful, sometimes a more orderly design is useful. For this purpose, I present the `Class::Structured` module, and discuss its implementation.

## 1 Conflicting class structures

Consider a super class `Text`, which stores some string variable `contents` containing the `Text` object's string data. `Book` inherits from `Text`, but `Book` also defines a variable `contents`, which in this case refers to the book's table of contents.

One of the nicest features of a Perl object is that its reference may point to anything, be it an array, hash, or simple scalar. This allows for a nice flexible set of options for a class developer. If both `Text` and `Book` use a hash for their internal storage, then both `Text` and `Book` write to the same location `$self->{'contents'}`, thus overwriting each others data, which neither module expects to happen. Even worse, if `Text` expects its subclasses to use a hash, but `Book` uses an array for internal storage, the program will complain when it tries to access an array like it was a hash, and vice versa.

Assuming this variable name clash can be avoided, `Book` still may need to call `Text`'s constructor. Perl does not call the constructor of a super class automatically and pushes this responsibility off to the subclass. Constructors look just like regular functions to the Perl.

Finally, what if I want to define an abstract function (a function whose implementation is deferred to subclasses, although it is declared in the base class) `search` in `Text`, and ensure that all subclasses of `Text` implement this function? Perl provides no way to enforce this check automatically.

To address these problems I wrote `Class::Structured`, which provides solutions to these problems by creating private variables, constructors, and abstract functions. It is still in the early phases of development, and does not implement inheritance types (like `private`, `protected`, and `public`), protected variables, or destructors yet. However, it provides a more structured approach to classes than standard Perl.

Before I get too wrapped up in the benefits of `Class::Structured`, I need to first consider its one severe drawback: speed. Using `Class::Structured` introduces some amount of overhead into your programs, and should not be used if speed is critical. However, where design is favored over speed, perhaps to aid in code maintenance, `Class::Structured` can be a very useful tool.

---

\*webmaster@cd-lab.com

## 2 Using Class::Structured

Using Class::Structured is easy. Code listing 1 defines our Book class.

---

Code Listing 1

---

```

1 package Book;
2
3 # inherit from Text
4 use Text;
5 our @ISA = qw(Text);
6
7 # use the structured class module
8 use Class::Structured qw(:all);
9
10 # define the private type variable
11 define_variables 'contents' => 'private';
12
13 # define our constructor
14 constructor 'new',
15 { 'Text' => 'new' },
16 implementation {
17     my $self = shift;
18     $self->contents = shift;
19 };
20
21 # declare the abstract draw method
22 declare_abstract 'search';

```

---

I start by inheriting from class Text. Line 8 imports the Class::Structured module and all of its symbols.

At line 11 I define my class variables using the `define_variables` function. In general, the syntax for this command is a list of variable name and access type pairs.

```
define_variables 'varname' => 'accesstype', 'varname' => 'accesstype', ...;
```

where `varname` refers to the variable being defined and `accesstype` must be `private`. I include the `accesstype` field in order to allow for `protected` variables in the future.

Once I have called this function, I can access the private `contents` variable to retrieve its value or to set it by using it as an lvalue subroutine. Class::Structured makes use of a fairly recent Perl feature to define subroutines so that they can be assigned to. If you try this without Class::Structured, or something else that defines lvalue subroutines for you then Perl will rightly complain that it cannot compile the program.

```
my $x = $self->contents;
$self->contents = "Three blind mice...";
```

Class::Structured supports public variables through its `public` method. Public variables are available to all of the classes in the hierarchy.

```
my $x = $self->public("name");
$self->public("name") = 'A Shape';
```

The public variable space is shared by all classes in a hierarchy, and public and private variables of the same name can co-exist. A normal method call to `contents` accesses the private variable, while the `public` method can still access public variables, even if they are the same name.

```
$self->contents = "A long time ago...";
$self->public("contents") = "Chapter 1...";
```

Going back to code listing 1, I defined my constructor (lines 14 - 19) next. The syntax for constructor definition includes the constructor's name, a declaration of its super class, and its implementation.

---

Code Listing 2

---

```
1 constructor 'name',
2 { 'superclass' => 'constructor', ... },
3 implementation {
4     my $self = shift;
5     ... your code here ...
6 };
```

---

In code listing 2, the `name` string refers to the constructor's name, which in code listing 1 was the traditional `new`. The `superclass` part specifies which parent class's constructors to use, since Perl cannot recognize a constructor on its own and a class may have as many as it likes. The `superclass` declaration can be omitted, in which case the parent's default constructor will be used. A default constructor is either the first constructor to be defined by a class, or the constructor defined with the `default_constructor` keyword instead of the normal `constructor`.

To finish my constructor declaration, I use the keyword `implementation` followed by a code block and terminated with a semicolon. Within the `implementation` section, I do not bless my own class variable. The `Class::Structured` module handles this and passes the result as the first argument instead of the class type. If I do not use a semicolon my program will not compile because `constructor` is a regular function unlike Perl's keyword `sub`.

### 3 A Review of Symbol Tables

In order to explain how `Class::Structured` works, I first must recall some facts about Perl's symbol tables. The majority of `Class::Structured`'s implementation deals with adding functions to a package's symbol table.

Perl's symbol table works essentially just like a hash so it allows variables with names that are not valid Perl variable names. For instance, while `!structured!.abstracts` is not a legal variable name, I can use a symbolic reference to create it anyway.

```
${ '!structured!.abstracts' } = 'foo'; # aha!
```

In the implementation of `Class::Structured` I often use variables in this way so that I can create variables that `Class::Structured` can use while also minimizing the chance of collisions with actual variables used by the package whose symbol table I manipulate. The general format of a name shows its package and variable name.

```
$package . '::!structured!.' . $varname
```

So, in `Class::Structured`, if I am dealing with the `abstracts` variable for the `Foo::Bar` package, the variable name is

```
{ 'Foo::Bar::!structured!.abstracts' }
```

I can also add a function to the `$package` package. This is especially useful because it lets me use a closure as a package function. A Perl closure is simply a subroutine which refers to a lexical variable that has gone out of scope. It does not have a name and does not usually have a symbol table entry, but through `Class::Structured` it can.

```
*{ $package . '::' . $function_name } = sub { my $x = 'foo'; $x };
```

## 4 How `Class::Structured` Works

With the review of symbol tables out of the way, I can explain the guts and innards of `Class::Structured`.

Recall that I am adding three concepts to Perl's class system: constructors, private variables, and abstract functions. The easiest of these to implement, and the one I will discuss here, is the abstract function. Remember that an abstract function is a function whose implementation is deferred by a class to its subclasses. A class with abstract functions left to be defined should not be instantiable.

Code listing 3 shows the implementation of the function used to define a new abstract function.

---

Code Listing 3

---

```

1 use Carp;
2 use Set::Scalar;
3
4 sub declare_abstract {
5     my $function_name = pop; # get last param as function name
6     my $package = caller;
7
8     # update the abstract list
9     # (use a weird name so we don't collide with a real variable)
10    my $list_name = $package.'::'. '!structured!.abstracts';
11
12    ${ $list_name } = Set::Scalar->new() unless defined ${ $list_name };
13    ${ $list_name }->insert( $function_name );
14
15    # declare the function
16    *{ $package.'::'.$function_name } =
17        sub {
18            croak "$function_name in class $package is declared abstract, " .
19                " and cannot be called";
20        };
21 }
```

---

This code determines what package to modify by using the `caller` function, which, when called with no arguments, will return the name of the package that called the subroutine. If code in package `A` calls the subroutine, `caller` will return `A`.

Lines 12 and 13 update a `Set::Scalar` object that contains the list of defined abstract function names. This variable is used later to check that all abstracts have been implemented.

Finally, on lines 15 - 20 I define the abstract function with the same name that will cause the program to stop with an appropriate error message.

`Class::Structured` defines two functions, `list_abstracts` and `check_abstracts` that can be called with a package name to list all outstanding abstract functions and to check that all abstract functions have been implemented, respectively. The latter is used by constructor functions to make sure a class has defined all abstract functions before instantiating an object.

The implementation of such constructors as well as private variables can be found in `Class::Structured`'s source code, which is available on the Comprehensive Perl Archive Network. Please feel free to ask me any questions regarding how they work, or to offer suggestions on this module since it is still in the early phases of development.

## 5 References

Comprehensive Perl Archive Network – <http://www.cpan.org>

`lvalue` subroutines are discussed in the `perlsub` man page

`Set::Scalar`