Class Building Design Notes

Author: John M. Długosz
dhcgnd702@sneakemail.com http://www.dlugosz.com/Perl6

Version 1.1

Table of Contents

1 Perl 6 Class Building

The way a class is created in Perl 6 is very interesting. While most languages you are familiar with treat the declarations as pure compile-time directives, Perl builds a class object by executing instructions written in Perl. This allows much more flexibility and more give-and-take than the approach which separates the compiler from the run time.
The purpose of this paper is to explain what is known about this thus far, and to serve as a sounding board to flesh out details and complete the design.

It is particularly important that the way the "meta class" is driven to create the class is standardized, so that portable code can be written to do interesting things with it such as interface with other languages or create an entirely new class system with its own dispatch mechanics.

2 How a Class is Made

Consider the following simple class:

```
class Dog {
  is Mammal;
  does Pet;
  has $.name = "fido";
  method foo () { ... }
  }
```

This might look similar to class declarations in C++, Java, etc. where everything is simply declarations for the compiler to understand innately.

In Perl 6, the situation is very different.

The block of the class is executed at compile-time, in the same manner as a BEGIN block.

And those things that look like declarations are actually function calls on the "class builder". The grammar makes them look and feel like declarations, but they are active statements to execute.

For example, the statement is Mammal will make a call to the meta object to add a base class to the definition it's building. The statement does Pet will likewise make a call to the builder to add a role. The next line tells the builder to register an attribute, with the name $.name and a closure that produces a default value for it if needed later. Finally, the last statement tells the builder to add a method, passing in the name and the sub block.

If you change the syntax to something more familiar to what it's really doing, it would look something like this:

```
my Metaobject $dogclass = P6opaque_builder.new("Dog");
with $dogclass {
   .addBase(::Mammal);
   .addRole(::Pet);
```

```
 .addAttribute( :name<$.name>, default => {"fido"} );
 .addMethod( :name<foo>, block => { ... } );
 .buildit;
}
```

2.1 Need for Standard

What functions really are there to be called on the builder is what we need to design, rather than wait for every implementation to come up with something different.

The point is that this is not some deep dark magic, but a well-defined part of Perl 6. It is possible to write entirely new class systems as well as class representations. This is how we will interoperate with classes in other languages and virtual machines, as well as allow CPAN modules such as Self/Javascript style objects which are quite different from classes at all.

The exact same interface should be used for creating roles too. So who knows what composition feature might be invented in the future, and implemented ready-to-run in portable Perl 6?

2.2 Executing Code at Compile Time

Here is another example that you should look at in light of how the class block is really like a BEGIN block.

```
class Fustian {
  my $var1;
  has $.a;
  our $var2;
  sub helper () {  say $var1; }
  method foo ()
  {
   $var1 = self.a;
   helper;
  }
}
```

The my and our lines are perfectly ordinary code, with meaning you know from normal blocks. It means the same thing here!

The sub definition is perfectly ordinary, and again means the same thing here!

So what happens with the definition of foo? The block forming its body is a closure, and that closure closes over the $var1. It has no problem calling helper, as an ordinary sub whose name is in scope when the closure is compiled.

The processing of the block of the method needs something to be aware of the stuff in the class, but the things that have nothing to do with the class builder or the meta object just work as they always did in any code.

In fact, it makes you wonder if you could write code in the block of a class like this:

```
class Fustian {
```

```
my $var1;
has $.a;
if $?OS ~~ /Windows/ {
    has $.b;
    say "adding Windows support";
  }
...
```

The answer is Yes! The code can compute things, call functions, and do whatever it likes as well as issues the has/is/does/method lines. The block is executed only once, at compile time, just like a BEGIN block. Executing that block will create and configure the class builder that describes the class, including the meta object used to *bless* instances of that class.

Now let's try something a little more flamboyant.

```
sub add_attributes (Int $defval)
{
 has Int $.a = $defval;
 has ::?CLASS $.next;
 has Str $.c;
}

class George {
  has Int $.d;
  add_attributes(5);
  }
```

Now this is a course a toy example, and I'm not really suggesting doing something bizarre instead of using a role for attribute reuse. But, imagine that add_attributes was reading configuration information from an XML file, or a .NET manifest, and automatically built classes to order. Now are you seeing the possibilities?

So, is this code legal? Specifically, the add_attributes function is outside of the scope of the class body itself.

Well, the use of ::?CLASS is a problem, as it is intended to be lexically scoped. But that is no big deal, we just need to add a contextual variable for the same purpose, or pass it in as a parameter. So, change ::?CLASS to ::+Class on that line.

What about the syntax for has statements? Does the grammar only recognise that while inside a class definition, or is it always good? It actually seems like more overhead to change the grammar in scope than to just leave them there, so unless there is some conflict, I think these should always work and direct their "calls" to the currently being built class.

3 Class Dispatch

The dispatching of methods (SMD) on an object is completely up to the instance itself. It is possible to implement hooks to foreign objects this way, as well as implement entirely new object systems that may not even be class based.

When a method call is seen, such as $x.foo, Perl 6 does not innately know what to do with that. It must communicate with the meta object to look up the proper function to call.

Conceivably, that function can do anything it wants, in terms of how it organizes method code for instances.

But, it should not be limited to blind run-time lookup by name, but provides a rich enough compile-time interface to allow Perl 6 to build fast jump tables and do type checking, *without* needing to innately know how methods are organized.

So at compile time, the compiler could notice the declared type of $x and build a method table, notice that foo is number 5 in that table, and emit code to just go there on this line, arrange for tables of derived types to be supersets of the base tables, and reuse everything for all usage of that type in the program. The compiler wants to do a lot to generate strongly-typed fast and checked code, but the meta object is still in charge.

4 Terminology

4.1 AST (Abstract Syntax Tree)

The parser produces an AST representation, and allows Perl code to examine and modify the tree before compilation continues. This is specified for "macro" features, for writing more interesting grammatical constructs, and will be needed as part of the Metaobject interface.

4.2 bless

In Perl 5, anything that will act as an object (that is, do virtual dispatch) has a pointer to a package that defines the methods. The keyword bless installs this reference and turns the data structure into an object.

In Perl 6, it is not clear yet how close the situation is to being exactly the same. Every object has a reference to a "meta object" that controls dispatch, so blessing would refer to installing this meta object reference. But is the meta object the same as a package? That is yet to be determined, but I think it must be. But, keep in mind that there might be another layer of abstraction here to be explored, and don't assume that the package is the meta object is a class.

4.3 class builder object

The "class builder" is something that is created when a class is being defined, and has methods called on it to build up the meaning of the class based on the contents of the class block in the source code.

It does not need to continue to exist after the class is finished, but another can be created to re-open an existing class (classes in Perl 6 are "open"). The class builder is not the same object as the meta object.

4.4 meta object

The “meta object” is the object that is in charge of dispatching normal (single) dispatch method calls.

This is not called the meta class because it does not have to be any particular type. The standard should explain how to create a working meta object in order to implement an object having virtual function dispatch, using only other Perl 6 features.

The meta object is also expected to respond to dispatches to so-called “class methods” on that type.

The HOW method of any object returns the meta object.

4.5 Meta Object Protocol (MOP)

A systematic way of using meta objects is called the MOP. The normal built-in P6 class system is one MOP.

The P6 Standard MOP (probably) makes a meta object for each defined class, and treats the meta object in the manner of a meta class, in a traditional class-based object system.

It’s possible to customize the dispatching of the P6 Standard MOP without replacing it completely, in order to interface with foreign objects, implement proxies, etc.

The concept of having different, configurable ways of storing private data (e.g. in a Hash that is the blessed object) is a feature of the P6 Standard MOP, not a concept that needs to be promoted to the Metaobject fundamental definition.

4.6 SMD

The virtual function single dispatch system, as distinct from the MMD (multi method dispatch) system. The meta object is in charge of SMD. MMD is a different feature and is understood in a deeper way by the Perl language itself.

5 Meta Object Design Details

5.1 Bootstrapping

The first concern with writing a dispatcher as another object is the bootstrapping problem. Suppose that $x.foo works by calling

```
my $m = $x.HOW;
my $method = $m.can(“foo”);
$method();
```

So how does $m.can get dispatched? It expands into an infinite recursion.

The meta object needs to be called using a more primitive system than method dispatch. The more primitive way of organizing multiple functions into a unit is with a package containing subs and data.

So, we require that the meta object be implemented as a blessed package. We know that certain primitive capabilities may be invoked by calling the sub in that package, without having to look for it (so if it is inherited, you have to make an alias in this package too).

But it needs to be blessed as an object as well, because if you *do* write code like above, it will not realize that you need the special way of calling and will dispatch the method. So the meta object of the meta object is a form that makes the required organization work with normal dispatching.

So the call to $x.foo, using only the simplest run-time binding, generates code like this:

```
my ::M := $x.HOW;  # I know it is a package
my $method = M::can("foo");
$method;
```

That is, the call to can is a simple namespace-qualified call, not another dispatch.

5.2 Required Functions

The main point of the meta object is to provide for dispatch. It is not always asked to call the located method, or to call it now, so its principle feature is a way in which to look up methods by name and return Code objects that may be called as a separate step.

The synopses mentions several things that can be performed once the meta object is obtained:

- Stringification
- attributes - get list of attributes
- bless - bless an instance into this class
- can – look up method
- does – test against role or base
- isa – test against base
- methods – get list of methods

and I've identified others already:

- buts – get list of properties aggregated into the single meta object
- re-open the class (return a new "class builder" object)

Other derived requirements will be listed here.

5.3 Method Lookup

Here begins the design discussion notes for "how does this object allow the compiler to interoperate with its method management system"?

The simplest thing is a can function that works like in Perl 5. But according to S12: Unlike in Perl 5 where `.can` returns a single `Code` object, Perl 6's version of `.HOW.can` returns a "WALK" iterator for a set of routines that match the name, including all autoloaded and wildcarded possibilities. In particular, `.can` interrogates any class package's `CANDO` method for names that are to be considered autoloadable methods in the class, even if they haven't been declared yet. Role composition sometimes relies on this ability to determine whether a superclass supplies a method of a particular name if it's required and hasn't been supplied by the class or one of its roles.

This makes it, at best, awkward to use in this capacity. Yes, I want full introspection to explore all the inherited forms and sort them any which way. But most of the time I want *one* answer, and that is the method to call.

Because of the differences in return type, I think these should be different functions: full blown query, and simple “tell me the method to call”.

Even in the simple case, where I’m relying on the meta object to know the rules of inheritance and delegation and all that and just want a single result, I can imagine calling it in different circumstances:

- Look up a method to call right away, as with run-time binding and no type information.
- Look up a method at compile time knowing only the static type, to be called many times at run time.
- Look up methods to populate a fast jump table based on the type.
- Look up methods that might not exist, but are added at run-time after the class was analyzed at compile time.

And even without doing anything rude to the class system, Perl 6 easily supports delegation and wildcards. So the compile-time lookup might get back information that says “maybe, check at run-time” instead of a definite “no”.

For populating tables, the code returned can be closures which repeat the lookup when called.

So, I’m thinking that the can function will take various adverbs that indicate what the caller is really doing, e.g. if the method will be used immediately, or used to populate a table; whether it has an instance to look at or only the type itself, etc.

I also might need to know for sure I’m pointing to the real function Routine, not a thunk that repeats the lookup first. For example, when addressing the Routine itself in order to check its properties or wrap it. And does a method have to be a Routine, or will the more primitive Code do?

5.4 More than Jello

To nail things down:

The name Metaobject exists as a role. Classes implementing that role extend Package and use the proper compatibility implementation for the class storage, all specified

by items in the role.

The name MetaMetaobject exists as a Package. It contains implementation of a Metaobject that provides standard dispatching on the required structure of the Metaobject layout. This is not the same as P5hash (it uses Package, not Hash).

6 Class Builder Design Details

6.1 Semantic Level

In order to work, the functions that accept items to add to the class definition need to interact with the compilation of those items.

The code blocks need to be processed to handle private data access. This indicates that the blocks should be passed as ASTs, not as fully-compiled Code objects. ASTs need to be well documented in order to allow people to write macros, anyway.

The parsing of code found within a class block needs to know about methods and other contents of the class.

6.2 Class-Building Environment

When a class, role, or other feature based on Class Builder is pending, the current Class Builder will be available as a contextual variable. It may also be available as a lexical variable within the scope of the class block.

Proposal: $?CLASS_BUILDER and $+CLASS_BUILDER.

6.3 Scenario

A declaration such as a has item will be parsed by special grammar rules that collect the parts using its special syntax. The parts are fed as parameters to a method call on $+CLASS_BUILDER.

Code blocks, including the blocks of methods, are parsed into AST but then needs to involve the The Class Builder before continuing to process it into a Code object. The Class Builder has a method for this purpose, that will go over an AST and replace generic “private attribute access” nodes with inline code that performs that access.

The Class Builder might also act as a server for local names, to provide parse-time knowledge of class contents including which names are types or functions, even though this is not a normal lexical scope in the block being compiled.

6.4 Required Functions

One function for each kind of thing to add to the class definition, more or less. Note that “is” is overloaded: will it determine earlier whether I am adding a base class or setting an option like “is rw”, and making different calls? The the kind of thing to add needs to be determined early not late.

We need the function to complete the class (called with the block is closed).

We need the AST fixer.

We need the name server for the parser.