OO for Tcl

or "How I Learned to Stop Worrying and Write the Code"

Donal Fellows
<donal.k.fellows@manchester.ac.uk>

For the past two years, I have been working on developing a new OO system for Tcl that is intended to serve as a basis for a wide range of OO styles. In this paper, I will describe and explain the current status of the work, discuss the issues involved in producing a high-performance flexible OO system, and describe a number of issues that have been encountered during work (with Arnulf Wiedemann) to build a version of [incr Tcl] on top of the core OO system.

As many of you know, I have been writing an object system for Tcl for a couple of years now. The intention was that this object system should focus on just the core task of making a fast method dispatch system as well as seeding the very heart of the inheritance hierarchy. This is in contrast to the other major object systems (e.g., [incr Tcl], XOTcl, Snit) that provide a much larger set of features, but at the cost of being far more complex. By sticking to the fundamentals, my code will be well placed to focus on how to be fast and readable, allowing the other OO systems to focus on "added value" such as collection management systems, rich application support, etc. Like that, it would allow us to have the power of the object-programming paradigm without enormous amounts of effort or tearing up the large number of existing scripts that depend on the features of the previously existing object systems.

This paper does not describe the detailed programming interface for the object system: I covered that previously[1]. Instead, it goes into more detail about the details of what makes for a fast and flexible object system. There has been a major change since I presented the initial proposal for this work two years ago though: after much discussion, I decided that any object system that goes into the core must have a substantial amount of practical "in-deployment" experience first. In order to gain this experience, I redesigned my object system to work as an extension package using the TEA build system. In addition, by doing this, I made it far easier for other people to work with the system during development. More eyeballs really do mean fewer bugs!

This has resulted in the TclOO extension, which you can use with any sufficiently recent version of an 8.5 core (i.e., after the sixth alpha release). It has documentation and a test suite, and I know that it builds and works correctly on both Windows and Linux. It even exports its own API via a stubs table, making it even easier to build your own extensions on top. There are down-sides to this though: how they have been dealt with is one of the topics of this paper.

[1] See my paper in the Tcl 2005 conference, or TIP #257, which was derived from it.

Flexibility and the Art of Code Writing

One of the major driving requirements of the TclOO package has been that it should be possible for third party code to extend it, and in as many different ways as possible. Thus, you can define not just new methods, but new *kinds* of methods and (currently experimental) new ways of invoking objects. However, doing this, especially for the long term, requires both the definition of structures (so that typing information can be provided in a sane fashion) and the rigorous hiding of the internal details of those structures that are private to the TclOO package itself.

The concealment of the internal details of private structures is relatively straightforward in practice: when those tokens even potentially pass outside the control of the package, I conceal their real types and they are just an abstract pointer[2]. I then provide a set of accessor functions to allow third-party code to extract the information within the real structures without exposing details that can change between versions.

Ensuring that public structures are future-proof is more complex. The structures that require this treatment are there to express the type of something, and instances of those structures will typically be compiled as constants in extension code. This binds the binary versions of those extensions inherently to the version of the API they use. Luckily, this problem has already been resolved in Tcl for structures such as the `Tcl_Filesystem` and `Tcl_ChannelType`, which would otherwise have the identical problem. These handle versioning by putting a version number directly into the structure: by knowing what version of the structure declaration the code was compiled against, the set of valid fields can be understood. This allows structures for purposes such as the definition of types of methods or metadata to be migrated into the future at minimal cost.

Another thing that has come from the TclOO work has been the way that some parts of the Tcl core are much easier to extend than before. For example, the Tcl `info` command is now an ensemble, as this allows the addition of new `info class` and `info object` subcommands in a simple fashion. The alternative would have been a special mechanism just for the `info` command itself, which would have required extensive testing instead of being just an application of a more general facility.

Inheriting Diamonds

One of the things that I wished to support was multiple inheritance, since that is a feature that is often very useful; e.g., a school bus is both a road vehicle and a passenger transportation device, and yet those superclasses are fundamentally distinct (compare with dump trucks and cruise ships!) And yet this opens up the way to a classic problem where you have a class that is a subclass of two other classes that define conflicting methods: the key to the problem being which method is “more important”? Since this can involve almost arbitrary amounts of additional confusing complexity, this problem is genuinely hard. (Arguably, this should not happen as method names should never clash like this, but method names model human language, and language is messy and imprecise.)

---

[2] In C, a pointer is abstract if it points to a type that the compiler does not know the size of. The classic example of an abstract pointer is `void*`, but pointers to a structure of unknown size are better in practice for many things, since they require an explicit cast to be converted to another type.

A study of the literature for dynamic object systems (static systems like C++ have other constraints that did not concern me) indicated that the best solution was to think in terms of first converting the inheritance graph (as viewed from a particular point) into a tree back to the object root, where any node may appear multiple times. Then you walk the tree "pre-order" to produce a traversal list, traversing the parents of each node in "natural" order (i.e., the order specified in the definition of the class). Finally, you remove every reference to any method on the list *except the last one*. The resulting list of methods (see Figure 1) turns out to be exactly what you want.

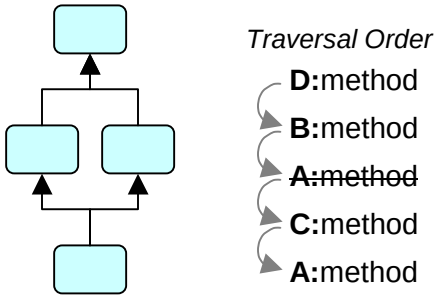

Figure 1: Diamond Inheritance Pattern

Well, almost. TclOO also supports mixins and filters, which add to the complexity. Mixins are classes that are added to objects; they are great for modelling roles and orthogonal behaviour, and come in the inheritance order before conventional classes (which model types better.) Filters are a way to decide whether to skip the evaluation of a method or perform some other kind of wrapping evaluation on a per-method basis, and are implemented as a list of method names to call before calling the real method. With both mixins and filters added, you have the model used by TclOO and XOTcl. (Other Tcl object systems typically have either simplifications of this model – [incr Tcl] is like this – or are done in a totally different way – the Self-modelled ones are in this category.)

Caching for Fun and Profit

The algorithm for calculating the method call chain described above is distinctly expensive, as you can imagine. The only way to get reasonable speed out of it is to be strict about using caching. And yes, TclOO uses caches a lot.

In particular, it caches method chains carefully so that the second time you call an object's method, the chain can be retrieved rapidly and the method dispatched in double-quick time. But care must be taken when doing such caches that the values retrieved from them are valid; if a class in the inheritance tree is modified, it can mean that all your assumptions about what the method chain looks like are wrong! Luckily, it turns out that it is easy to build a system for detecting potential problems that is also cheap. Just as with Tcl's bytecode engine, I use epoch counters. When an incompatible change happens, the appropriate epoch is updated – every object has its own epoch counter, but classes use a global one because they may be used outside themselves – and the code that retrieves values from the cache can just check two epochs (the object epoch and the global epoch) against the values saved when the chain was created. When both epochs match, the chain of implementations for that particular method name is correct and can be dispatched immediately.

Of course, if you have experience with the `Tcl_Obj` value system you might expect that the caches would be kept in the method name value itself. After all, that is exactly where Tcl's ensembles and functions like `Tcl_GetIndexFromObj` keep their caches. But this is not actually a safe thing to do, since we have per-object methods (and mixins and …) and without a strong classical object typing system I must keep the caches in the object itself and not the method name value. This is a significant difference between a subcommand dispatch scheme designed to support an ensemble and one for objects and their methods.

Getting [Incr]ementally Better

As mentioned earlier, one of the major aims of this work was to support the building of other object systems on top. This is a good thing to aim for since they have historically reached very deep into Tcl's innards in order to get speed, and that has left them inclined to be tightly bound to particular versions of Tcl. Not exactly the Stubs promise!

Instead, I have been working (with much prompting from Arnulf Wiedemann) on providing an API that allows these other extensions to build their style of methods on top of my core ones without having to pry deep inside my code. For the moment, this API is not public – I do not know yet whether the functions and structures involved are at all stable – but it is my intention to make it available. In particular, it allows for code to do things like adjusting the command resolution scheme specifically for the body of the method instead of by doing strange things with the overall command resolution system. This limits the effects and makes it easier to increase the performance. Other areas that have an internal extension point are the mechanisms for deciding how to implement a particular method call, for deciding the exact level of privacy enjoyed by a class method, and to allow classes to control the name of the namespaces of the objects they create.

The net result of this (and much work by Arnulf) is that it has proved possible to implement a new version of [incr Tcl] on top of the TclOO core and get it to sufficient quality where it passed the itcl test suite. One of the biggest gains is that this new version, currently known as itcl-ng, can do this without need to deal with direct allocation of structures that are in the Tcl core. This in turn makes it likely that future versions of itcl will be compliant with the broader Tcl Stubs promise: that a new minor version of Tcl will not force the rebuilding of extensions built against the old version.

Collecting Examples

But you would rather see code, right?

There are many features in the TclOO system that are of interest at the scripted level. Although it only defines two classes (being `oo::object`, the class of objects, and `oo::class`, the class of classes) these classes have many abilities, some of which I shall show off here. Firstly, let us define some simple collection classes.

Since we want to allow objects to be automatically deleted when they are no longer referenced, it greatly helps to start with a reference-counting infrastructure. The following class creates objects that maintain a reference count just like those for `Tcl_Obj` values, deleting themselves when the count drops below one.

```
oo::class create Refcountable {
    constructor {} {
        variable refcount 0
        next
    }

    method incrRefCount {} {
        variable count
        incr count
    }
    method decrRefCount {} {
        variable count
        if {[incr count -1] <= 0} {
            my delete
        }
    }
}
```

On top of this class, we then build some simple collection classes. This list class can have reference counted objects added to it, searched for in it, removed from it, and can also iterate over the list of objects. When the list is destroyed, it will automatically remove its references to its contents (possibly deleting them in turn, of course).

```
oo::class create List {
    superclass Refcountable
    constructor {} {
        variable list {}
        next
    }
    destructor {
        my foreach object {
            $object decrRefCount
        }
        next
    }

    method add args {
        variable list
        foreach object $args {
            lappend list $object
            $object incrRefCount
```

```
        }
    }
    method has object {
        variable list
        expr {$object in $list}
    }
    method remove object {
        variable list
        set idx [lsearch -exact $list $object]
        if {$idx >= 0} {
            set list [lreplace $list $idx $idx]
        }
        return
    }
    method foreach {var body} {
        variable list
        upvar 1 $var v
        foreach v $list {
            uplevel 1 $body
        }
    }
}
```

But as we all know, lists are not the only sort of collection. The other major kind is the map. This map class maintains a mapping (in a dictionary) from strings to objects. The objects are naturally reference counted. It supports methods to put (add or update) a mapping, get the object from a mapping, delete a mapping or list the keys in the mapping. Aside from the other features, one interesting thing to note here is the `Decr` method, which is hidden from use by things outside the class. This happens automatically when the method name does not start with a lower-case letter.

```
oo::class create Map {
    superclass Refcountable
    constructor {} {
        variable map {}
        next
    }
    destructor {
        variable map
        dict for $map {key object} {
            $object decrRefCount
```

```
        }
        next
    }

    method Decr key {
        variable map
        if {[dict exists $map $key]} {
            [dict get $map $key] decrRefCount
            return 1
        }
        return 0
    }
    method put {key object} {
        variable map
        $object incrRefCount
        my Decr $key
        dict set map $key $object
        return
    }
    method get {key} {
        variable map
        return [dict get $map $key]
    }
    method unset {key} {
        if {[my Decr $key]} {
            variable map
            dict unset map $key
        }
        return
    }
    method keys {} {
        variable map
        return [dict keys $map]
    }
}
```

As you can see, it is quite easy to build all the trappings of a conventional object system. Or at least it is if you do not permit renaming of objects with `rename`. When objects may be renamed, things get quite a bit more complex since you can no longer safely store the

object's name; instead, you need to use some kind of unique identifier that is never modified: the name of the object's private namespace serves this purpose well. This class also demonstrates how the object system can use other features of Tcl (in this case, namespaces and traces) to achieve its aims

```
oo::class create Renamable {
    superclass Refcountable
    constructor {
        variable ::objforname
        set objforname([namespace current]) [self]
        trace add command [self] rename \
                [namespace code {my Renamed}]
        next
    }
    destructor {
        variable ::objforname
        unset objforname([namespace current])
        next
    }
    method Renamed {from to op} {
        variable ::objforname
        set objforname([namespace current]) $to
    }
    method uid {} {
        return [namespace current]
    }
    method getFromUid {uid} {
        variable ::objforname
        return $objforname($uid)
    }
}
```

Updating the list and map classes to use this new class's features by storing the unique identifier values instead of the object names is left as an exercise for the reader.

Wrapping Widgets

One key rite of passage for an object system is integrating with Tk. Everyone wants to do it so they can make megawidgets and create other sorts of enhanced functionality. Here I demonstrate how to do this in a simple example using an entry widget:

```
oo::class create Entry {
```

```
    self.unexpose create
    constructor {widgetName args} {
        entry $widgetName {*}$args
        variable realName __$widgetName
        rename $widgetName $realName
        rename [self] $widgetName
        trace add command $realName delete \
                [namespace code {my delete ;#}]
    }
    method unknown {method args} {
        variable realName
        return [$realName $method {*}$args]
    }
    unexpose unknown
}
```

Note that I use the special unknown `method` here to direct any method invocations not otherwise known to the subcommands of the real widget. This, very much like Snit's delegation, makes it simple to override a method without having to maintain the whole list of subcommands (a traditional problem with [incr Tk]).

But we want to do something fancier with this new capability. We do this by Here's a new kind of entry widget that we can flash like a button:

```
oo::class create FlashEntry {
    superclass Entry
    method flash {{times 5}} {
        set bg [my cget -bg]
        set fg [my cget -fg]
        for {set i 0} {$i < $times} {} {
            my configure -bg $fg -fg $bg
            update idletasks
            after 200
            my configure -bg $bg -fg $fg
            update idletasks
            if {[incr i] < $times} {
                after 200
            }
        }
    }
}
```

Now we can use this like this, which (apart from the slightly different creation sequence) is now just like using a normal widget, except it has this extra capability:

```
FlashEntry new .e
pack .e
bind .e <Return> {%W flash}
```

Tackling Threads

As you might expect, the TclOO package is completely thread-safe. This means that we can use it with the Thread package with very little fuss. For example, here is a small thread pool manager that also looks after getting the results from the pool back and cleaning up after itself:

```
package require Thread
oo::class create Parallel {
    constructor {lambdaTerm $args} {
        variable term $lambdaTerm
        variable pool [tpool::create {*}$args]
        variable posted {}
    }
    destructor {
        variable pool
        variable posted
        if {[dict size $posted]} {
            my cancel
        }
        tpool::release $pool
    }
    method start {values} {
        variable term
        variable pool
        variable posted
        if {[dict size $posted]} {
            error "pool still busy"
        }
        variable results {}
        foreach v $values {
            dict set posted [tpool::post -nowait $pool \
                    [list apply $term $key]] $v
        }
    }
    method wait {} {
```

```
        variable pool
        variable posted
        variable results
        set done [tpool::wait $pool [dict keys $posted]]
        foreach j $done {
            dict set results [dict get $posted $j] \
                    [tpool::get $pool $j]
            dict unset posted $j
        }
        return [dict size $posted]
    }
    method cancel {} {
        variable pool
        variable posted
        variable results
        set left [tpool::cancel $pool [dict keys $posted]]
        foreach j $left {
            tpool::wait $pool $j
            dict set results [dict get $posted $j] \
                    [tpool::get $pool $j]
        }
        set posted {}
    }
    method results {} {
        variable results
        return $results
    }
}
```

The thread pool manager can be used to execute lambda terms on many values in parallel; for example, this simple example demonstrates how to compute Fibonacci numbers in a somewhat foolish fashion:

```
Parallel create Fib {x {fib $x}} –maxthreads 6 –initcmd {
    proc fib x {
        if {$x <= 2} {return 1}
        expr {[fib [incr x -1]] + [fib [incr x -1]]}
    }
}
```

```
Fib start {10 20 30 40 50 60 70 80 90 100 110 120 130 140}
while {[Fib wait]} {}
array set fibonacci [Fib results]
puts "got part way..."
Fib start {150 160 170 180 190 200 210 220 230 240 250 260}
while {[Fib wait]} {}
array set fibonacci [Fib results]
Fib delete

parray fibonacci
```

Working with WebServices

Of course, we can also do things with objects and WebServices. Indeed, this is how they are typically created in most of the rest of the WS community.

```
package require WS::Server
package require WS::Utils
oo::class create Service {
    self.unexport new
    constructor {args} {
        global Config
        variable ServName [self]
        ::WS::Server::Service -service [self] \
                -host $Config(host):$Config(port)
                {*}$args \
                -premonitor  [namespace code {my}] \
                -postmonitor [namespace code {my}] \
                -checkheader [namespace code {my CHECK}]
    }
    method PRE {service operation argList} {
    }
    method POST {service operation status results} {
    }
    method CHECK {
        service operation caller httpHeaders soapHeaders
```

```
    } {}


    # A Simple Utility Method
    method DateTime {instant} {
        clock format $instant –format {%Y-%m-%dT%H:%M:%SZ} \
                -gmt yes
    }
    # A Utility Method
    method type {name definition} {
        variable ServName
        ::WS::Utils::ServiceTypeDef Server $ServName \
                $name $definition
    }


    # Utility method
    method operation {nameInfo argList doc body} {
        variable ServName
        set args {}
        set name [lindex $nameInfo 0]
        oo::define [self] method $name $args $body
        set body2 [namespace code [list my $name]]
        foreach arg $argList {
            append body2 " $" [lindex $arg 0]
            lappend args [lindex $arg 0]
        }
        ::WS::Server::ServiceProc $ServName $nameInfo \
                $argList $doc $body2
    }


    # Utility method for producing operation results
    method Result args {
        set op [uplevel 1 {self method}]Result
        upvar 1 _RESULT_ result
        if {![info exists result]} {set result {}}
        if {![dict exists $result $op]} {
            dict set result $op {}
        }
        dict set result $op {*}$args
```

```
    }
}
```

A demonstration of how to use this code is naturally in order. This is adapting from the

```
Service create wsExamples \
        -description {Tcl Example Web Services}
wsExamples type echoReply {
    echoBack {type string}
    echoTS   {type dateTime}
}

wsExamples operation {
    SimpleEcho {type string comment {Requested Echo}}
} {
    {TestString {type string comment {Text to echo back}}}
} {Echos a string back} {
    my Result $TextString
}
wsExamples operation {
    ComplexEcho {type echoReply comment {Requested echo+ts}}
} {
    {TestString {type string comment {Text to echo back}}}
} {Echos a string back with a timestamp attached} {
    my Result echoBack $TestString
    my Result echoTS   [my DateTime [clock seconds]]
}
```

This example, based on the code on the Web Services for Tcl website, is already considerably simpler for the application of simple object technology. But deeper support should be possible in the future. After all, ideally a web service should not be significantly harder to write syntactically than a conventional Tcl namespace; there is more than enough other complexity to deal with!

Accelerating with Aspects

Another thing you can do with TclOO is create aspects. An aspect is a way of “cross-cutting” a program so that code does not need to deal with everything in one place. Instead, you can have each part be a specialist in what it does, perhaps by adding logging or persistence to some existing code that would otherwise need significant reengineering. For example, below we define a special class that is used for applying transparent caches to an object. This is great when you are dealing with methods that can take a long time to

execute because of computation, though care must be taken with it because it does not understand object internal state.

```
oo::class create cacheAspect {
    filter Memoize
    method Memoize args {
        # Do not filter the core method implementations
        if {[lindex [self target] 0] eq "::oo::object"} {
            return [next {*}$args]
        }

        # Check if the value is already in the cache
        my variable ValueCache
        set key [self target],$args
        if {[info exist ValueCache($key)]} {
            return $ValueCache($key)
        }

        # Compute value, insert into cache, and return it
        return [set ValueCache($key) [next {*}$args]]
    }
    method flushCache {} {
        my variable ValueCache
        unset ValueCache
        # Skip the cacheing
        return -level 2 ""
    }
}
```

You can then apply this to any object to add memoization to that object's methods by mixing the class in. For example:

```
oo::object create demo
oo::define demo {
    method compute {a b c} {
        after 3000 ;# Simulate deep thought
        return [expr {$a + $b * $c}]
    }
```

```
}
```

This object just does some simple calculations, but takes a long time over it.

```
puts [demo compute 1 2 3]      prints "7" after delay
puts [demo compute 1 2 3]      prints "7" after delay
puts [demo compute 1 2 3]      prints "7" after delay
```

Time to add that memoization!

```
oo::define demo mixin cacheAspect

puts [demo compute 1 2 3]      prints "7" after delay
puts [demo compute 1 2 3]      prints "7" instantly!
puts [demo compute 1 2 3]      prints "7" instantly!
puts [demo compute 4 5 6]      prints "34" after delay
puts [demo compute 4 5 6]      prints "34" instantly!
puts [demo compute 1 2 3]      prints "7" instantly!
```

If we change things, we need to flush the cache…

```
oo::define demo method compute {a b c} {
    after 3000
    return [expr {$a * $b + $c}]
}
puts [demo compute 1 2 3]      prints "7" instantly, wrongly!
demo flushCache
puts [demo compute 1 2 3]      prints "6" after delay, right!
puts [demo compute 1 2 3]      prints "6" instantly
```

And all this from just the application of a mixin and a filter. The `demo` object itself knows nothing at all about how to do caching, but we waved our magic wand and added the functionality after the fact. Can aspects make *your* programming tasks easier?

Future Directions

Thanks to the help I have received from many people (especially Arnulf Wiedemann), the TclOO package is almost ready for public release. The main thing left to do is to discover what features have I left out that are critical, and that is something which is best done by letting other people try to use and break it. As always, the code probably needs more work so that it goes faster. I also want to really encourage everyone to take my code and find cool ways to use it to do things that are relevant to you.