TclOO: Past, Present and Future

Donal Fellows
University of Manchester / Tcl Core Team
donal.k.fellows@manchester.ac.uk

Abstract

This paper looks at the state of Tcl's new object system, TclOO, looking at the forces that lead to its development and the development process itself. The paper then focuses on the current status of TclOO, especially its interesting features that make it well-suited to being a foundational Tcl object system, followed by a look at its actual performance and some of the uses to which it has already been put. Finally, the paper looks ahead to some of the areas where work may well be focused in the future.

1. TclOO: Past

Genesis (with Phil Collins)

One of the longest-standing complaints about Tcl has been its lack of an object system; this has not always been an entirely fair criticism, as Tcl has in fact had rather too many of them! The most well known ones are Tk (in a sense), [incr Tcl], XOTcl and Snit. But each of them has had a number of restrictions that has made it not quite suitable for use as a general Tcl object system, and to people not immersed in the details of Tcl this has meant that Tcl has *appeared* to completely fail to support those.

The problems with and good properties of the existing object systems listed above are these:

Tk: Tk is focussed entirely on supporting graphical widgets, and so has never actually been usable as a general OO system. It has however established a key baseline for how we expect object instances to work at the script level, particularly in the use of subcommands as a way of interacting with the instance.

[incr Tcl]: Itcl (as it is more commonly known) is the object system that has come closest to being universally deployed in Tcl installations to date, and the way in which you declare classes within it is exceptionally easy to use. On the other hand, it shows its heritage in the C++ model of the early 1990s, being relatively inflexible and unable to support many advanced features of object systems.

XOTcl: In many ways, XOTcl represents the polar opposite of itcl. It has a great set of basic semantic features and is enormously flexible, both features that fit the spirit of the Tcl language itself very well, but it is awkward syntactically and has a strong insistence on adding a lot of methods that are not needed everywhere, making the interface exposed by objects cluttered.

Snit: Snit is the most interesting of the purely scripted object systems (the others all have substantial C components). Its focus is on the use of delegation to achieve its goals rather than inheritance, and this makes it very suitable for many of the tasks involved in building megawidgets. However, it pays a substantial performance penalty for being written in Tcl.

It is these essential problems, each of which is totally inherent to the way that each object system works, that led to the effort to develop a new object system that would be deeply integrated and yet of a more limited scope. The aim was (and is!) to enable people to build whatever they want on top of it.

The Plotters Revealed

Tcl 8.6 is the first version of Tcl to ever include a standardized core object system, which also goes by the name of "TclOO" and is also available as an extension for Tcl 8.5. It was written to try to take

the very best of these other object systems and to provide a flexible and dynamic basis for the creativity of the Tcl community.

TclOO really started life at a Tcl Conference. Specifically, it arose originally out of a conversation between Will Duquette and myself at the 2003 conference in Ann Arbor, where we talked about Snit (which Will had been presenting) and what it would take to make it faster, and where I had been presenting how to assemble of some of the features of Tcl 8.4 to make a simple object system. That conversation led to the development of ensembles for Tcl 8.5, a feature that is proving to be quite useful in itself through enabling things like simple compilation of parts of commands like info and string without the previous complexity of the command compilers.

Later in the development of 8.5, we (together with Steve Landers) revisited the ideas from that conversation and asked what was working about the new features of 8.5, and what was not. We agreed that while ensembles were great, they didn't do nearly as much as we desired. In particular, we instead wanted a "real" object system, with things like inheritance and separated public and private interfaces, as Will was hitting inherent limits in the design of Snit when applied to large systems. So we started by asking what was wrong with the existing systems, and we concluded that Tk was utterly inflexible, itcl deeply dated (though nice to use), XOTcl polluted though semantically interesting, and Snit far too slow and limited.

Given that, we took the one with the strongest theoretical basis (XOTcl) and looked at what were the smallest changes that we could make to it to make it suitable for producing clean interfaces (we'd identified that cleanliness in that area would be a required feature of any real core OO system). Alas, once you start picking at XOTcl's syntax, many of its features don't work well; it depends very strongly on how it processes its init methods as a way of setting up objects. In particular, we had a strong desire to allow authors of classes to be able to handle arbitrary arguments, just like normal Tcl procedures, rather than forcing arguments that consisted of hyphen-prefixed method names to be interpreted as calls to that method. After all, that is a feature that can be added by a particular class if desired. When we combined that principle with the general cleanup (and moving the configuration of classes and objects into another command) we ended up with essentially the interface we have today.

To clear up a misconception that sometimes raises its head, it has never ever been our intention to force anyone to use TclOO, just as we do not force anyone to use any other Tcl feature (even doing without procedures is possible, if inclined towards making life difficult for yourself). In particular, we do not force all values to be objects, unlike a number of other languages. Tcl retains its standard "everything is a string" semantics, and objects are really just another way to create commands.

In the Developing World

In terms of development, TclOO had a difficult gestation. In particular, it took an exceptionally long time to get to first base, as for a long time I struggled with trying to build a prototype in Tcl so that I could test the script-level API. The real breakthrough came when I stopped trying to script a prototype and instead built a system that would do method dispatch (initially, as just a single-parent inheritance tree) and seeded an initial population of classes (i.e., the class of objects and the class of classes). With that basic core done, even if totally unoptimized at that point, it became possible to see the way ahead and to start making effective use of collaboration tools like version control systems.

The development of TclOO was done initially as a branch off a particular point in the development of Tcl 8.5, and I took a general decision that I would not track any changes originating outside the branch while doing the development. This meant that it was stuck with whatever bugs were present in the exact version of Tcl I happened to start from, a matter of some frustration to other Tclers who wished to review the code, but also meant that the only changes on the entire branch were ones that related to the things being developed. This "pure feature development" branch technique makes best use of the way that CVS manages branches, especially as I took care to ensure that I had set a tag on the point where the branch originated from. Only once I had versions that were substantively feature-complete did I work on integration with the rest of Tcl at all, a task made far easier by the fact that the large majority of the code changes were in new source files.

In retrospect, the key visible milestones were:

1. Implementation of the core class structure.
2. Working method invocation.
3. Creation of the oo::define command.
4. Beginning a test suite.
5. Prototype build system.
6. Restructuring methods so they are done with a generic management structure instead of just special cases.
7. Addition of introspection, enabling a full test suite.

8. Release to other people as a package.
9. Integration into Tcl.

Most of these were obviously milestones at the time. For example, once you can do a rapid cycle of build and test, you can iron out bugs at a greatly increased rate. Note also that introspection came relatively late, not because it was unimportant, but because it was easy; it was only forced in the end by the fact that it was needed for increasing the test coverage from checking for errors to detailed inspection of the consequences of reconfiguration.

There were a few invisible milestones as well, primarily in the development of the method call chain cache. Key points there were when it was first introduced (very early, once it was clear that generating method call chains would be expensive), the point where the cache started being maintained at least partially in the method name Tcl_Obj internal representations, and the point where the cache was made not specific to a particular object when that object was stereotypical of its class (the normal case by far).

Citizens' Advice Cupboard

In terms of advice for others taking on projects of a similar scale, here are a few tips:

1. Make a plan first, so that you know when to declare success. You can change the plan, but without one it is impossible to progress.
2. Study the literature and the competition. It is much easier to reuse other people's good ideas than it is to have your own!
3. Until you have the heart of things done, you cannot make any good use of the tools and principles of open source development; people only help when they can do so incrementally.
4. Don't mix bug fixes and feature development up. Even with the help of systems like GIT, it's still better to have a clean description of what your feature is so that you can present it to others as a neat package. (Be prepared to do special demonstration merges from time to time, but don't waste effort on maintaining them.)
5. You won't get everything right until you have full in-service testing. It is just impossible to anticipate everything that the users of the code will want to do (well, assuming you're successful that is) so do not put effort in early on getting it all right. It is far better to plan to have a beta period to give the adventurous a chance to work out where your bloopers are.
6. Don't do releases until you are passing all your tests on at least one system. Be prepared for problems when other people are building with any architectural configuration that you have not personally checked, and prepare to spend time that you don't want to spend on helping people.

2. TclOO: Present

TclOO is designed to be fast, small, stable, scriptable, and deeply integrated with Tcl. The hope is that by having one piece of code dig in this deep, others will be able to reap the performance benefits without going to all the trouble.

Faster, Pussycat! Kill! Kill!

It is well known that it is a bad idea to optimize too early, so much so that people are frequently enjoined from optimizing at all until they are experts. So why did we put so much effort into optimizing TclOO? Quite simply, it is that we know for certain that different languages will be compared for the performance of their object systems, no matter how little this applies to whether the language or the object system is a good choice for the problem. I therefore made boosting the performance a key priority.

The speed of TclOO comes from the fact that it has been carefully optimized for performance. In particular, the following critical paths have had much attention: the (repeated) method call, especially when the object is a prototypical member of its class, the creation of a new non-class object, and the destruction of a non-class object. Because it is anticipated that classes will have lifespans that are substantially longer than that of instances, much less effort has been put into optimizing their creation and destruction.

For method calls, the heart of the performance comes from two things. Firstly, the process of converting the object and method name into a sequence of method implementations to call (a "call chain") is slow, so it is vital that it is avoided wherever it is safe to do so. This is done through extensive caching, within the internal representation of the method name, within the object, and within the class (so that when objects are plain instances of a class — by far the most common case — they can share caches for a substantial reduction in overall performance cost). The only disadvantage of doing all this caching is that the determination of when a cache has to be invalidated is important to get right, possibly even requiring erring on the side of caution, and the cost when the cache *is* invalidated is

rather high. In addition, the cost of checking whether the cache is invalidated must be low. These are achieved by the use of epoch counters that are updated whenever the definitions of objects or classes are modified; when the current epoch does not match the one where a cached call chain was created, the old call chain is discarded and a new one is created. (In fact, there are multiple epoch counters. This means that updates of the definition of a single object or a class with no instances or subclasses do not cause the rebuild of the whole world, these being important common cases.)

The other source of method call performance comes from careful tracing of just where the costs are. In particular, the two most costly operations that are likely to happen in most high-performance code are memory allocation and hash-table lookups. It is therefore clear that to boost performance, it is important that these expensive operations are only used where they are strictly necessary. Once I came to the point where actual high-performance implementations were required, I also avoided using techniques like Tcl_Preserve to manage lifetime of objects that have non-trivial cleanup, because a custom reference count system requires less space and fewer lookups internally.

For an example of the sort of optimizations that have been applied, it is useful to consider the creation of an object. This requires the creation of a namespace and two commands, one inside this new namespace and the other with either the name of the namespace or a value supplied by the caller. By being careful here, it was possible to avoid a moderately complex series of lookups in the creation of the internal object command (i.e., my) since it is always created in a new namespace; there is never any need to check if there is an existing command in the way. This means that it is safe to directly manipulate the namespace structure itself when creating the command, an option that is strongly not recommended for code outside the Tcl core! Similarly, the name of the current object is also only generated lazily and is shared as much as possible, only being discarded when the object is renamed.

Tiny Tim Strikes Back!

Almost all of the development of TclOO was a one-man band, and yet it has been possible to produce a very high quality object system in a short time this way. The key to this achievement is that the focus was always on doing as little as possible, keeping features out if they didn't *need* to be in.

Most object systems come with a substantial class library (or equivalent). After all, this is a very useful thing for most people. But TclOO is not intended to be the total solution to everyone's needs, and in any case, there are many wonderful existing libraries. Redoing them all would have taken a large amount of work and would have detracted from the primary mission: getting the basic object system done. So such effort as I had, needed to be spent wisely.

Where did it go?

The method dispatch engine, as previously mentioned, is the single most important part of the code, as it is highly exposed to users and the key to how everything works. It received much attention as you might expect.

Another area that took a lot of work to get right was the initialization and finalization of both objects and the entire system. Of particular difficulty was ensuring that everything took itself apart neatly when the world was ripped out from under its feet in strange ways, particularly by the whole interpreter being deleted or through deletion of the global namespace.

The third area that consumed a lot of effort was the implementation of methods themselves. This is because they are hooked very deeply into the code that manages procedures, sufficiently so that they in fact use a new private API created specially for them called TclObjInterpProcCore. (This API is why the TclOO package for Tcl 8.5 cannot work in older versions of Tcl.) In Tcl 8.6, things are different again, because methods fully integrate into the non-recursive calling engine so that new commands like tailcall and yield will work smoothly in methods.

The final area that consumed masses of effort was the build system for the stand-alone TclOO package. While things were compounded by the fact that TclOO was exporting a stub table of its own so that code built against it could continue to work when used with Tcl 8.6, it was plain issues of building that caused difficulties time and time again, even with the help of TEA for the basics. If the code had not required the use of Tcl's internal headers, it would have been far simpler. (By comparison, no problems of a similar nature were ever encountered when working on the code as integrated into Tcl.)

How to Bolt Stable Doors Earlier

A critical goal of TclOO has been stability of implementation, so that it is as solid a platform as possible for others to build on. This is founded

upon two aspects: freedom from crashes and freedom from memory leaks.

In respect of the first aspect, crash-free code is something that any Tcl programmer expects. Even if they do something unwise, a good Tcl command will detect it and respond with a sensible error message rather than falling over in a nasty mess. Checking that this is actually true depends on the use of a test suite that exercises as many failure modes as possible (too few arguments, too many arguments, wrongly formatted arguments, clashes between commands, etc.) and knowing what to check for requires a tricky mind. My thanks to Don Porter for his key insights into the myriad ways in which finalization could occur.

The only aspect of Tcl where nasty failure can happen is with memory allocation, because it is exceptionally difficult to check for in a portable fashion except at the point when you can no longer do anything nice about it. It is therefore essential that memory is not leaked; when an object is deleted, all memory associated with it must be cleaned up, and even when things are dynamically altered, nothing must ever be left in an undetermined state. Though the TclOO test suite does check for leaks from the low-level machinery (the method call engine, the caches) it does not, and cannot, ensure that programs are leak-free. You can still get into a lot of trouble with programs that create lots of objects and do not delete them again.

Thus, it is vitally important that introspection starting at the oo::object class be able to find all currently extant objects, i.e., objects cannot get lost. Given that, deleting a class results in the deletion of all instances of that class and all subclasses too, making tidying up much simpler. One easy way to do this when an object is simply tied to the lifetime of a particular scope (procedure, lambda expression) is to use the try command's finally clause to ensure that the objects created are also deleted. An alternative is to use a variable unset trace on a local variable of the scope. Another technique that works well for coroutines is to set a deletion trace on the coroutine command, as that is based on information easily looked up with info coroutine.

In addition, TclOO also supports tying the lifetime of one object to another that "owns" it, a concept that is called "composition aggregation" in UML (see Figure 1 for an illustration of this as applied to TDBC). You do this by creating the owned object in the internal namespace of the owner (you may also rename the object into the namespace to achieve this). Once that is done, the contained object will be automatically destroyed when the container goes away; no explicit destructor is required. The only thing to be careful of is that when the contained object is deleted, it is impossible to call anything on the container. TDBC uses this feature (see Figure 1) to manage the lifetimes of statements and result sets within database connections.

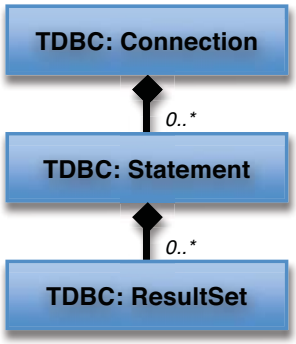

Figure 1. UML class diagram showing object ownership in TDBC

A Scripted Entrance

One of the key features of TclOO which makes it a much more suitable feature for a general Tcl object system is that it is scriptable. In particular, it is possible to introduce new features of the base system in pure Tcl code; use of the C API is not necessary at all.

Not only can you reconfigure core classes like oo::object and oo::class (though perhaps this is not all that wise; library code might not expect it) through the virtues of dynamically modifiable classes, but you can also create your own subclasses of them and use those to build up your infrastructure. In particular, making a subclass of oo::class (a feature that came from XOTcl) allows you much greater control over how construction of objects happens, which makes writing classes that work in complex ways much easier. For example, when creating megawidgets you can use an unknown method handler to intercept calls that follow the classic Tk pattern ("*className* .foobar …") and redirect that to creating an instance of the class.

```
oo::class create MegaBase {
  superclass oo::class
  method unknown {m args} {
    # Looks like a widget name?
    if {[string match ".*" $m]} {
      # Hand off to creation method
      tailcall my create $m {*}$args
    }

    # Default handling
    next $m {*}$args
  }
}
MegaBase create MegaExample {
  method create {w args} {
```

```
    # Code omitted for clarity ...
  }
}
MegaExample .foobar -text "example"
```

But that is not the only way that you can extend the behaviour of TclOO. For example, you can add a command that lets a method defined by a class refer to variables declared on the class and not the instances, so that the variable is shared between all instances of the class. Adding of this command to the visibility of all methods is done by just creating it in the special namespace ::oo::Helpers, as that is added to the path of all object namespaces (by default, it contains the next and self commands). Thus, you can declare the command like this:

```
proc ::oo::Helpers::classvar \
    {name args} {
  # Get reference to class's namespace
  set ns [info object namespace \
      [uplevel 1 {self class}]]

  # Double up the list of varnames
  set vs [list $name $name]
  foreach v $args {lappend vs $v $v}

  # Link the caller's locals to the
  # class's variables
  tailcall namespace upvar $ns {*}$vs
}
```

One thing to note about that code (in the line that is marked with bold text) is that it makes use of a combination of two of TclOO's introspection features to discover where the class keeps its variables. The self class command reports what class declared the current method (normally obvious, but not in this case) and the info object namespace command reports what the private namespace of any object is, allowing code to peek behind the curtain of the official interface any object (including classes). Since all variables of objects are kept in a private per-object namespace, it is a simple step to go from obtaining the namespace name to manipulating the variables.

Even more sophisticated elaborations of the TclOO core functionality are possible. The oo::define command works by evaluating its definition argument in another namespace[1], so adding another command to that namespace will enhance the oo::define command directly. Here, I demonstrate how to make a command to create class methods:

```
proc ::oo::define::classmethod \
    {name {args ""} {body ""}} {
  # Create the method on the class if
  # the caller gave arguments and body
  set argc [llength [info level 0]]
  if {$argc == 4} {
    uplevel 1 [list \
        self method $name $args $body]
  } elseif {$argc == 3} {
    return -code error \
        "wrong # args: should be\
        \"[lindex [info level 0] 0]\
        name ?args body?\""
  }

  # Get the name of the current class
  set cls [lindex [info level -1] 1]

  # Get its private "my" command
  set my \
      [info object namespace $cls]::my

  # Make the connection by forwarding
  tailcall forward $name  $my $name
}
```

As can be seen, all a class method is can be captured through defining a method on the class and arranging for the instances to forward calls of the method on to the class. By forwarding through the my command (which is always created in every object's namespace), the link between object and class is preserved even if the class is renamed (though that is almost certainly a bad idea…)

I have demonstrated three ways in which the behaviour of TclOO may be extended in new ways by adding implicit method calls, class variables and class methods. This demonstrates that the core functionality of TclOO is extensible through pure scripting, establishing that the users of it can do almost anything rather than requiring evolution to be driven from the Core Team. (A few things probably do remain to be done, as will be seen later.)

It should be noted that I do not guarantee that the precise names of the namespaces used in the preceding two examples will be maintained across all versions. This is because they strictly form part of the implementation, and not the specification.

What Happened Next

One of the most important uses for TclOO has turned out to be TDBC, which makes extensive use of a number of key features. In particular, the ability to handle object lifetime automatically with minimal code (given the ownership of some objects clearly by another, as discussed earlier) and the ability to define custom methods at the C level in a simple fashion have both proved to be important. This is not to say that TDBC requires TclOO to exist; it clearly does not and it has been carefully kept that way. As long as a database package meets

[1] After all, why would I write a complex code parser when I could just reuse the Tcl interpreter?

the TDBC interface specification, it is part of the general universe. But the ability to provide the core of the system using TclOO classes allows new database interfaces to be written with much less effort, which is tremendously valuable.

TclOO has also already been used in production applications. As was reported back at EuroTcl 2009, Ansaldo STS use it to form the OO backbone for a track viewer used in railway maintenance. They report[2] that TclOO is easy to use for this application and a strong improvement to Tcl.

Another key claimed feature of TclOO is that it enables the foundation of other object systems on top of it. A particular case in point is [incr Tcl] 4.0, which is included as contributed package with Tcl 8.6. This version which was done by Arnulf Wiedemann, and which is founded on top of the basic facilities provided by TclOO, passes the whole itcl test suite (including the really obscure parts) and means that in the future, the substantial existing body of community code that makes use of itcl will be able to work with new versions of Tcl without lots of adaptation on the part of script authors (unlike with Tcl 8.5, where the port of itcl took quite a long time).

Supporting this has meant a number of changes to TclOO and just goes to show that the last 10% of compatibility often takes by far the most work. Examples of the changes that were required were hooks for additional models of method visibility (because itcl has a C++-like notion of friend classes), a method to allow classes to control what namespaces their instances create (because unlike commands, namespaces cannot be renamed safely), and a way to rewrite method names before invocation (so an object can be invoked as if it was an instance of a superclass).

What a Performance!

So let's have a look at some performance figures! After all, it's better to know what careful attention on performance (at the expense of deep richness of class library) has produced.

Firstly, these tests were all performed on a quiescent MacBook Pro with a 2.6 GHz Intel Core 2 Duo processor and plenty of memory (i.e., these tests are not in any way constrained by lack of memory). Except where noted, they are run in ActiveTcl 8.5.2.0.284846, mostly because that's the version I have conveniently installed and hooked up to a local teacup-managed repository. The version of Tcl 8.6b1.1 is the current HEAD version from the Tcl CVS repository as of August 30, 2009, and was a threaded and optimized build on the same hardware platform.

Secondly, these tests try to perform equal work. Where an object system makes available the distinction between direct method calls and virtual method calls (to use C++ terminology) both are tested; note that most OO systems for Tcl use virtual calls as they fit the Tcl ethos better. Similarly, because the usual reason for building an object is so that it can be used, the object creation test also includes a trivial method call internally so that circumstances are like those that would be experienced in production code. (Arguably this means that TclOO is favoured slightly, but that's into the whole "lies, damned lies, and benchmarks" debate.)

Thirdly, these figures are all quoted to only 3 significant figures. Even that may be more accurate than they really are; multiple runs of each test are performed, and only the best is picked each time.

However, as we look across all of these performance measurements, we see that TclOO is *consistently* the fastest. In some cases, it is even competitive with not using an object system at all.

(The source transcript – though with some floundering omitted – from the run used to produce these performance figures is supplied with this paper in the file ppf_transcrtipt.txt. It is also in the appendix to this paper.)

Basic method call

For method calls, we first compare calling the same method on the same object over and over, where the method takes a numeric argument returns that value plus one. The state of the object is not accessed at all.

```
# Plain old procedure
proc plus {x} {
  incr x
}
```

This is the fastest way of implementing the procedure in Tcl 8.5.2, and the same body was used for all the examples described in Table 1 and Figure 2.

Table 1. Execution speed for basic method calls

Tcl	Object System	Calls (s^{-1})
8.5.2	Plain old procedure	1,630,000
8.5.2	TclOO 0.6.1	1,240,000

[2] This was not the focus of their presentation, which was mainly about what they were doing with GUIs and how Tcl was doing very will overall in their commercial application.

8.5.2	XOTcl 1.6.3	803,000
8.5.2	[incr Tcl] 3.4	514,000
8.5.2	Snit 2.2.3	1,080,000
8.5.2	Stooop 4.4.3(D)	996,000
8.5.2	Stooop 4.4.3(V)	75,300
8.6b1.1	Plain old procedure	936,000
8.6b1.1	TclOO 0.6.1	645,000
8.6b1.1	[incr Tcl] 4.0b3	363,000

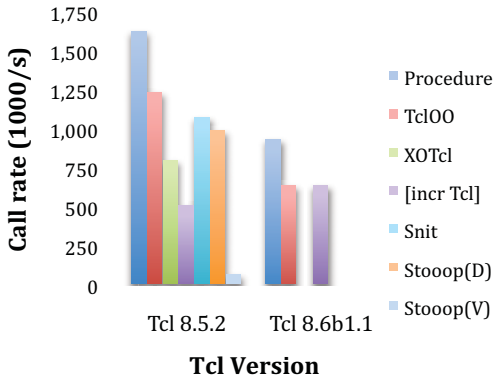

Figure 2. Execution speed for basic method calls

As you can see, Tcl 8.6b1.1 is considerably slower than 8.5.2 was, at least at the moment of testing. This is because of the introduction of the non-recursive execution engine, which significantly impacts the speed of normal command invocation at the moment. Also of note are the two variants of calls for Stooop; this is because virtual calls (which are effectively what all the other object systems do anyway) take a massive hit in Stooop.

Stateful method call

This differs from the previous one in that the object has to maintain state between the calls. In specific, the object has to maintain an accumulator. For plain old procedures, this is done by using a global variable, but all other object systems have to use an instance variable (or nearest equivalent).

```
# The TclOO class declaration
oo::class create Example {
  variable acc
  method accumulate {x} {
    incr acc $x
  }
}
```

The above code is a fast implementation because it uses variable resolvers to hook from the method's internal variable table to the object-wide variables. This is very fast, especially for per-object methods where it is possible to compile in direct references to the variables rather than having to look up. (As can be seen in Table 2 and Figure 3, the difference isn't gigantic but it is enough to make it faster than a plain procedure call in 8.5.2.)

Table 2. Execution speed for stateful method calls

Tcl	Object System	Calls (s^{-1})
8.5.2	Plain old procedure	1,130,000
8.5.2	TclOO 0.6.1(cls)	1,120,000
8.5.2	TclOO 0.6.1(obj)	1,140,000
8.5.2	XOTcl 1.6.3(1)	911,000
8.5.2	XOTcl 1.6.3(2)	432,000
8.5.2	[incr Tcl] 3.4	490,000
8.5.2	Snit 2.2.3	819,000
8.5.2	Stooop 4.4.3(D)	776,000
8.5.2	Stooop 4.4.3(V)	69,700
8.6b1.1	Plain old procedure	766,000
8.6b1.1	TclOO 0.6.1(cls)	597,000
8.6b1.1	TclOO 0.6.1(obj)	626,000
8.6b1.1	[incr Tcl] 4.0b3	308,000

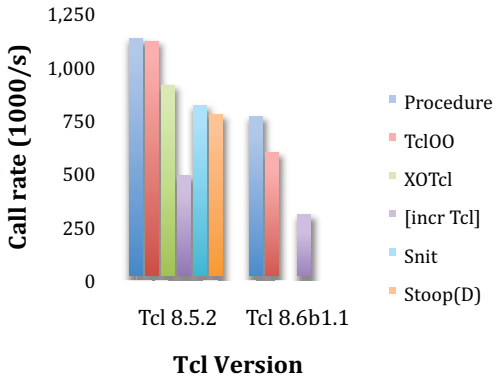

Figure 3. Execution speed for stateful method calls

Of interest here is the fact that TclOO has a very small performance hit by comparison with the equivalent plain Tcl code, and that with XOTcl it depends heavily on how you access the variable: (1) is directly calling the built-in incr method which is implemented in C, and (2) is using the instvar method to pull the variable reference into a normal method's scope.

Object create and delete

While it is really object creation that is interesting in performance terms, for the number of objects created to be large enough for sensible numbers they must be deleted as well so that we are not measuring the effects of allocating large amounts of memory.

```
# The itcl way to create and delete
itcl::class Example { }
itcl::delete object [Example obj]
```

As you can see in Table 3 and Figure 4, this is just about the simplest test of construction and destruction possible.

Note also that from here on, there is no entry for simple Tcl operations; object creation is roughly where things go beyond the level where such equivalents exist.

Table 3. Execution speed for object create/delete

Tcl	Object System	Calls (s^{-1})
8.5.2	TclOO 0.6.1	174,000
8.5.2	XOTcl 1.6.3	127,000
8.5.2	[incr Tcl] 3.4	149,000
8.5.2	Snit 2.2.3	8,710
8.5.2	Stooop 4.4.3	45,800
8.6b1.1	TclOO 0.6.1	151,000
8.6b1.1	[incr Tcl] 4.0b3	8,170

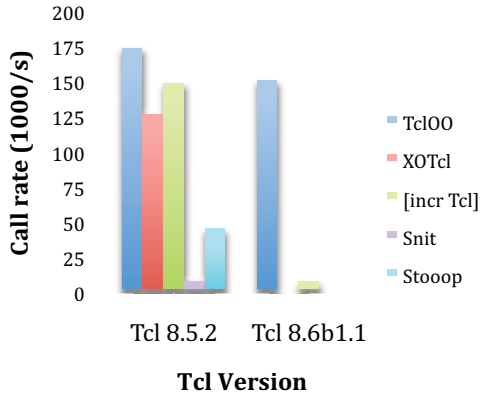

Figure 4. Execution speed for object create/delete

As you can see, at the moment there seems to be a definite issue with creation/destruction in itcl 4.0. Given that the performance problems with Snit have been well understood, it is probably the case that itcl 4.0 is currently performing too many actions at object creation time rather than class creation time.

Object create, method call, and delete

This category of performance measurement illustrates whether the costs of working with calls to one method on many objects of the same class are excessively expensive. This particularly tests whether the mapping of method names to implementations at the class level is efficient. Because method calls are approximately 10 times faster than object creation, ten calls to the same stateful method will be performed for each create/delete cycle.

```
# How to make a suitable Stooop class
stooop::class OCMCDst {
  proc OCMCDst {this} {}
  proc ~OCMCDst {this} {}
  proc accumulate {this x} {
    incr ($this,acc) $x
  }
  stooop::virtual proc \
        vaccumulate {this x} {
    incr ($this,acc) $x
  }
}
```

Also note that the timings for the Tcl-only object systems are slightly less accurate as they have fewer analysis runs.

Table 4. Execution speed for create/call/delete

Tcl	Object System	Calls (s^{-1})
8.5.2	TclOO 0.6.1	62,600
8.5.2	XOTcl 1.6.3	53,700
8.5.2	[incr Tcl] 3.4	36,400
8.5.2	Snit 2.2.3	5,240
8.5.2	Stooop 4.4.3$^{(D)}$	27,800
8.5.2	Stooop 4.4.3$^{(V)}$	5,950
8.6b1.1	TclOO 0.6.1	36,900
8.6b1.1	[incr Tcl] 4.0b3	6,200

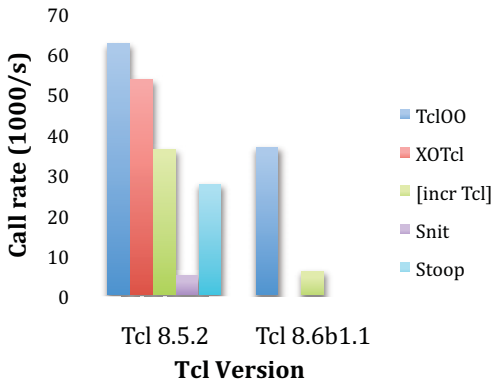

Figure 5. Execution speed for create/call/delete

One of the main stories here (as shown in Table 4 and Figure 5) is that Stooop remains mostly competitive with the C-coded object systems despite being in pure Tcl, though this is only if you are not using virtual methods. Rather like the difference between normal and virtual methods in C++, this is an important distinction in terms of flexibility but, if you're prepared to live with the restrictions, it is still a reasonable choice. The other main story remains the problems with itcl 4.0b3.

Superclass call

This final performance measurement looks at how the use of a class hierarchy impacts on performance. To make things simpler, I have every object system doing the same thing. The superclass method implementation is maintaining a counter of the number of times it has been called and returning the current value, and the subclass method implementation is accumulating the argument value plus the current result of the superclass's implementation.

```
# TclOO implementation of used classes
oo::class create SCo_base {
  variable count
  method foo {} {
    incr count
  }
}
oo::class create SCo_derived {
  superclass SCo_base
  variable acc
  method foo x {
    incr acc [next]
    incr acc $x
  }
}
```

Note that this test excludes the use of Snit, as that does not support inheritance, and forces the use of Stooop virtual methods.

Table 5. Execution speed for superclass call

Tcl	Object System	Calls (s^{-1})
8.5.2	TclOO 0.6.1	607,000
8.5.2	XOTcl 1.6.3	176,000
8.5.2	[incr Tcl] 3.4	287,000
8.5.2	Stooop 4.4.3	386,000
8.6b1.1	TclOO 0.6.1	358,000
8.6b1.1	[incr Tcl] 4.0b3	62,800

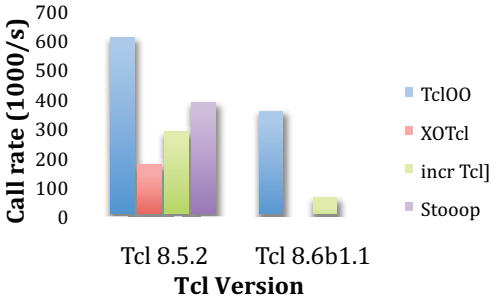

Figure 6. Execution speed for superclass call

Looking at these figures (see Table 5 and Figure 6), it's clear that TclOO is substantially faster than the others at handling inheritance and calling superclass implementations of methods. (It's not clear that the value for Stooop is correct in this case; I may have made an error when constructing the test code.)

3. TclOO: Future

Where do we go from here? Well, there are a few features that it would be exceptionally nice to have but which did not make the cut due to a shortage of development time.

Note that many of these areas may touch scripts written that use TclOO. These will be specifically noted below where they are known.

Talking Trash

TclOO follows the model of using explicit destruction of objects. This is very much the model that has been used in the past by Tcl-based object systems (all the ones in §1 do this) but it does not address all the problems that occur. For example, it is too easy to create an object and then have an error thrown which causes you to lose sight of the object. While it is possible to discover the object again through TclOO's introspection facilities, it is not trivial to go from knowing the name of an object to whether that object is garbage. In addition, if an

object has an explicit name from the user, the object probably should not be subject to garbage collection anyway. Classes are the classic example of this; you hardly ever want to get rid of a class just because you don't have any instances of it at the moment. Moreover, explicitly named objects (including classes) may well be referred to from files that have not been sourced yet; it would be wrong to destroy them.

But the object names that come from the new construction method are synthetic; they have to be kept in variables or otherwise in a script in order to be used correctly. This means that it is potentially possible to garbage collect them when they are no longer referred to. The simplest way of doing this would be to make the trigger for this be when the name Tcl_Obj of the object ceases to have a reference to it and is deleted. That will allow the greatest proportion of existing scripts to function correctly with no additional work, even in complex cases where the name is passed via a master interpreter. It also meets the guarantees of the destructor semantics, as they are not guaranteed to run in all situations (e.g., if the program receives an unhandled signal or destroys the interpreter). In addition, we could easily have it that renaming the object causes it to leave the "GC system" and require manual lifetime management, which is effectively the current situation anyway.

So what is the cost? Only the core "everything is a string" semantics of Tcl itself. The issue is that names of objects manifestly cease to be just strings; they are imbued with additional magic that has an impact upon the lifetime of the object they talk about. Of course, some extensions have already taken this step (TclJava uses it for objects from the Java world, Tcom requires it for correctly interfacing with and manipulating COM entities on Windows, etc.) but it is a large step from doing these as additional behaviours that people opt into and making them a core part of Tcl itself.

The alternatives are those suggested earlier, such as tying to the lifetime of a particular scope (though it is possible to have a method to change the object's governing scope to the parent's, which handles the single largest problem with this) or command. Also, it is possible and not difficult to add manual reference counting, but requiring Tcl scripts to use this would be onerous.

Because of these fundamental issues, I believe it is unlikely that the method of GC that is probably best in practice is unlikely to be implemented in any version of Tcl before 9.0. Only then can we risk venturing into this deep pit of semantic vipers.

☠ **Storing of new Method Output in Non-Preserving Containers** (e.g., substrings)

Note again that most scripts are unlikely to see any problems, precisely because the new method currently delivers names that are not under the caller's control and most places that people are likely to store method are actually containers (variables, lists or dictionaries).

In a Snit Over Methods

One thing that has not yet been as successful as I wished was adopting features from Snit. Snit is notable for two key features: delegation and submethods (which are also a feature of Tk, though Tk is not a general, flexible OO system).

TclOO does support delegation through method forwarding, and this can be made flexible by delegating to a command in the current object's namespace rather than a global command. Like that, controlling the delegation target for multiple methods simply becomes a matter of adjusting that single command:

```
oo::class create Delegator {
  constructor {target} {
    interp alias \
       {} [namespace current]::target\
       {} $target
  }
  forward len  target length
  forward idx  target index
  forward bit  target range
  method target {} {
    return [interp alias {} target]
  }
}

# Demonstrate by going to string
set str [Delegator new ::string]
$str len "abc"        ;# → 3
$str idx "abc" 1      ;# → b
$str bit "abcde" 1 2  ;# → bc
$str target           ;# → ::string
```

However, TclOO does not support the other important feature of Snit, submethods, at all smoothly. Firstly, a definition: a submethod is where you can directly define methods that are ensembles of lesser methods, and is a key feature of how a large API (like Tk's canvas and text widgets) is made manageable. For example, the text widget's bindings are managed through a group of submethods so you can do ".txt tag add …", ".txt tag bind …", etc.

In Snit, you declare a submethod by using a list (of length greater than one) for a method name; the first element becomes the master method name and the second (and later) elements become the sub-

method name. Thus it would work like this (using TclOO syntax):

```
oo::class create SubmethodExample {
  method {foo bar} {a b c} {
    # ...
  }
  method {foo boo} {} {
    # ...
  }
  forward {foo foo} somewhere
}
```

This would create a single method, foo, which is composed of three submethods (bar, boo, and foo) into an ensemble. While it is possible to construct a work-alike using a real ensemble and forwarding (as with the delegation example) it is exceptionally messy. It is also badly supported by the introspection facilities, which expose the implementation details rather than the model that should be exposed to users.

Thus a plan for the future is to create proper support in TclOO for submethods, including working out exactly how introspection of this is to be done. This leads to another point where compatibility with the current implementation might be not maintained in the future:

☠ **Method Names which are Well-Formed Multi-Element Lists**

If you create methods with spaces in the name, they are likely to have a different interpretation applied to them once we do support submethods.

Slot You In Sometime Next Year?

A feature of XOTcl that it would be good to adopt is slots. A slot is a special class for managing a collection of values associated with classes or objects (for example, the lists of superclasses and declared variables). The idea is that there should be a single relatively rich scheme for managing these things, which are currently only exposed as lists that you can set as a whole.

This would immediately enable simpler schemes for adding to and rearranging these lists, which would be a strong benefit. Right now, they have to be given in one place. This is not particularly onerous for the superclasses to be honest, but rather painful and complex when dealing with the declared variables.

The main down-side to doing this is that it requires a syntactic space to be opened up so that it will be clear what are the slot methods and what are the things being manipulated. The other requirement is that the current behaviour must remain the default if no slot method is given; without this, existing code will break in the common case. However, the simplest way of achieving this is to make the slot method names begin with a hyphen so they look like the options used in places like namespace import. For example:

```
oo::class create oo::slot {
  method -append {args} { ... }
  method -clear {args} { ... }
  # etc.
}

oo::class create SomeClass {
  variable a b c
  variable -append d e f
  # As if: variable a b c d e f
}
```

How exactly the slots would be plumbed into the definition system, I do not currently know. (It's even possible that it might be possible with just scripting.) But the area of incompatibility is relatively clear.

☠ **Names for Things Beginning with Hyphens**

Not all names will be affected, of course, and method names will (with the exception of filters) be not subject to this scheme. Most others will be.

TclOO Extispicy[3] for Fun and Profit

Of course, if we are going to go down the road of having slots on the object, then we have the question of what slots should there be. Of course, there are the existing configurable lists (objects have three at the instance level: mixins, filters and variables; classes have those at the class level plus the list of superclasses) but that does not make those the only ones that are desirable. If we were to add a slot, what would its semantics be? If we allow the addition of new behaviours, should we really hard-code the current arrangement? Should there be mechanisms to allow greater control of the basic features of TclOO?

Right now, there is a single large system in TclOO that is not highly flexible, and that is the code to generate the call chain in response to a method call. Yes, on one level you can configure it by defining new methods, changing what mixins are configured, etc. But the algorithm itself is inviolate. Filters always come before method chain traversal. The mixins of a class always precede its superclasses.

Wouldn't it be interesting to be able to change this?

[3] As all good students of Etruscan civilization know, this means divination through inspection of a creature's guts. As you can see, Tcl is highly educational!

Of course, what exactly this would look like, I have no idea at all. Right now, there are a few internal APIs that Itcl 4.0 uses to be able to maintain precise compatibility with previous versions (e.g., other ways of following a method chain than using the next command). However, these are very much hacks that are put in precisely to support high compatibility, and they do not sit particularly well with TclOO as they inhibit the use of filters at later stages of the method chain.

It would be better if there were a way to override the mapping of the object's state and method name to a call chain so that non-standard behaviours could be *systematically* supported. Mind you, because any implementation of such an override would need to understand the internal data structures of the TclOO implementation, this would always be an API that is officially unsupported. It may even be a route that we never actually go down in a general fashion; unlike the other things mentioned which are in principle driven from what has already been found to be useful, this instead originates in the desire to expose everything. Such internal reasons are rarely a good driver for action.

It's Got a Widget!™

One of the main features that I believe will come in Tk soon now is a megawidget support class (or classes) so that people can write their own widgets more simply than before. Yes, they have always been possible to script in Tcl, but it has been difficult to do so well. The best alternative to date in terms of sophistication has been Snit, but it does not scale well, and BWidget looks badly dated on some platforms.

One of the main difficulties with megawidget systems is that there are several different approaches to building a megawidget in the first place. For example, you could take an existing widget and add more methods and behaviours, or you could put several widgets together in a frame and treat them as an overall widget. You could even build a whole dialog in a toplevel and offer that without dressing it up officially like a widget, much like the way that tk_getOpenFile works on Unix. Each of these has its own advantages and disadvantages, and perhaps different base classes are needed for each.

In the case of simple extension of a single Tk widget, the basic widget facilities (e.g., naming, configuration, focus, selection) are already in place and all that is needed is to slot the object system in smoothly and to add any options and methods as required. I already have experimental code to do this (see http://wiki.tcl.tk/21103 on the Tcler's Wiki) though I have not done any studies of performance of creation and usage. Of course, there are a few issues left in Tk itself, notably the fact that you can't control the class of the widget when used for the initial lookup of options, but these are less of a problem than they used to be, since extensive use of the X11 option database is now much less common. A scripted workaround is also possible, but would really hurt performance.

With composed megawidgets, it is harder since there are far more of the basic facilities that have to be built and more things may need to change in Tk for the integration to be natural to both creators and, especially, users of megawidgets. One thing I know of is that using pack or grid to put another normal widget inside the megawidget will result in the widget being arranged with respect to the outermost widget, and not the position that the megawidget appears to offer (this would have mattered if the labelframe widget had been done as a megawidget).

For dialogs, the main issues relate to what the proper support for them should be at all. Maybe they are even fine without explicit official support, or support with only the most basic features.

Thus, while it is clear that TclOO is probably suitable for megawidgets, the amount of effort to provide support is high because it is probably the case that Tk is not yet suited to being used that way. What will take the time is resolving what corrections to Tk are required and implementing those.

Appendix: Log of Performance Measurements

Note that certain steps are omitted from this log (e.g., deleting of objects after measurements are complete) and it has been split up into sections for clarity. The trace for Tcl8.6b1.1 is very similar (i.e., the input script is identical where appropriate) other than having different versions, fewer packages and different timings.

Basic Method Call

```
% package require Tcl
8.5.2
% proc plus x {incr x}
% time {plus 1} 1000000
0.613694793 microseconds per iteration
% expr {1000000/0.613694793}
1629474.473967062

# ------------------------------------------------------------------
% package require TclOO
0.6.1
% oo::class create TclOOExample {
    method plus x {incr x}
}
::TclOOExample
% TclOOExample create eg
::eg
% time {eg plus 1} 10000000
0.8083755836000001 microseconds per iteration
% expr {1000000/0.8083755836000001}
1237048.7435390174
% eg destroy

# ------------------------------------------------------------------
% package require XOTcl
1.6.3
% xotcl::Class XOEg
::XOEg
% XOEg instproc plus x {incr x}
% XOEg create eg
::eg
% time {eg plus 1} 1000000
1.2447734590000001 microseconds per iteration
% expr {1000000/1.2447734590000001}
803359.0311311417
% eg destroy

# ------------------------------------------------------------------
% package require Itcl
3.4
% itcl::class ITEg {
    method plus x {incr x}
}
% ITEg eg
eg
% time {eg plus 1} 1000000
1.946722511 microseconds per iteration
% expr {1000000/1.946722511}
513683.8940061962
% itcl::delete object eg

# ------------------------------------------------------------------
% package require snit
2.2.3
% snit::type SnitEg {
```

```
    method plus x {incr x}
}
::SnitEg
% SnitEg eg
::eg
% time {eg plus 1} 1000000
0.9236280330000001 microseconds per iteration
% expr {1000000/0.9236280330000001}
1082686.9305297493
% eg destroy

# -------------------------------------------------------------------
% package require stooop
4.4.1
% stooop::class StooopEg {
    proc StooopEg {this} {}
    proc ~StooopEg {this} {}
    proc plus {this x} {incr x}
    ::stooop::virtual proc vplus {this x} {incr x}
}
% set eg [stooop::new StooopEg]
2
% time {StooopEg::plus $eg 1} 1000000
1.00375688 microseconds per iteration
% expr {1000000/1.00375688}
996257.18132064
% time {StooopEg::vplus $eg 1} 1000000
13.272679443 microseconds per iteration
% expr {1000000/13.272679443}
75342.73725923512
```

Stateful Method Call

```
% package require Tcl
8.5.2
% proc acc x {global acc; incr acc $x}
% time {acc 2} 1000000
0.883275877 microseconds per iteration
% expr 1000000/0.883275877
1132149.1122303116

# -------------------------------------------------------------------
% package require TclOO
0.6.1
% oo::class create TclOOExample {
    variable acc
    method acc x {incr acc $x}
}
::TclOOExample
% TclOOExample create eg
::eg
% time {eg acc 1} 1000000
0.890838787 microseconds per iteration
% expr 1000000/0.890838787
1122537.5618944622
% oo::object create egobj
::egobj
% oo::objdefine egobj {
    variable acc
    method accumulate x {incr acc $x}
}
% time {egobj accumulate 1} 1000000
0.8782098199999999 microseconds per iteration
% expr 1000000/0.8782098199999999
1138680.0480094838
```

```
# -------------------------------------------------------------------
% package require XOTcl
1.6.3
% xotcl::Class XOEg
% XOEg create eg2
::eg2
% time {eg2 incr acc 1} 1000000
1.097875676 microseconds per iteration
% expr 1000000/1.097875676
910849.9458184554
% XOEg instproc acc x {
    ::xotcl::my instvar acc
    incr acc $x
}
% time {eg2 acc 1} 1000000
2.31492083 microseconds per iteration
% expr 1000000/2.31492083
431980.215928162

# -------------------------------------------------------------------
% package require Itcl
3.4
% itcl::class ITEg {
    variable acc 0
    method accumulate x {incr acc $x}
}
% ITEg eg3
eg3
% time {eg3 accumulate 1} 1000000
2.040263924 microseconds per iteration
% expr 1000000/2.040263924
490132.6677577425

# -------------------------------------------------------------------
% package require snit
2.2.3
% snit::type SnitEg {
    variable acc
    method accumulate x {incr acc $x}
}
::SnitEg
% SnitEg eg4
::eg4
% time {eg4 accumulate 1} 1000000
1.221689523 microseconds per iteration
% expr 1000000/1.221689523
818538.5739777683

# -------------------------------------------------------------------
% package require stooop
4.4.1
% stooop::class StooopEg {
    proc StooopEg {this} {}
    proc ~StoopEg {this} {}
    proc accumulate {this x} {incr ($this,acc) $x}
    stooop::virtual proc vaccumulate {this x} {incr ($this,acc) $x}
}
% set eg [stooop::new StooopEg]
4
% time {StooopEg::accumulate $eg 1} 1000000
1.304988281 microseconds per iteration
% time {StooopEg::vaccumulate $eg 1} 1000000
1.288521466 microseconds per iteration
% expr 1000000/1.288521466
776083.3066323166
```

Object Create and Delete

```
% package require TclOO
0.6.1
% time {[oo::object create ::obj] destroy} 100000
5.7581981099999995 microseconds per iteration
% expr 1000000/5.7581981099999995
173665.43854462836

# ----------------------------------------------------------------------
% package require XOTcl
1.6.3
% xotcl::Class XOEg
::XOEg
% time {[XOEg2 create eg] destroy} 100000
7.893295159999999 microseconds per iteration
% expr 1000000/7.893295159999999
126689.80187990336

# ----------------------------------------------------------------------
% package require Itcl
3.4
% itcl::class Example { }
% time {itcl::delete object [Example obj]} 100000
6.7288695700000005 microseconds per iteration
% expr 1000000/6.7288695700000005
148613.37251332693

# ----------------------------------------------------------------------
% package require snit
2.2.3
% snit::type Sniteg {}
::Sniteg
% time {[Sniteg create e] destroy} 100000
114.74785177 microseconds per iteration
% expr 1000000/114.74785177
8714.76009855413

# ----------------------------------------------------------------------
% package require stooop
4.4.1
% stooop::class StooopEg {
    proc StooopEg {this} {}
    proc ~StooopEg {this} {}
}
% time {stooop::delete [stooop::new StooopEg]} 100000
21.82625304 microseconds per iteration
% expr 1000000/21.82625304
45816.38443242386
```

Object Create, Method Call, and Delete

```
% package require TclOO
0.6.1
% oo::class create OCMCDo {
    variable acc
    method accumulate x {incr acc $x}
}
::OCMCDo
% time {
    OCMCDo create obj
    obj accumulate 1; obj accumulate 2; obj accumulate 3; obj accumulate 4
    obj accumulate 5; obj accumulate 6; obj accumulate 7; obj accumulate 8
    obj accumulate 9; obj accumulate 10
    obj destroy
```

```
} 100000
15.9744709 microseconds per iteration
% expr 1000000/15.9744709
62599.88241613686

# -------------------------------------------------------------------
% package require XOTcl
1.6.3
% time {
    xotcl::Object create obj
    obj incr acc 1; obj incr acc 2; obj incr acc 3; obj incr acc 4
    obj incr acc 5; obj incr acc 6; obj incr acc 7; obj incr acc 8
    obj incr acc 9; obj incr acc 10
    obj destroy
} 100000
18.61800578 microseconds per iteration
% expr 1000000/18.61800578
53711.44535115726

# -------------------------------------------------------------------
% package require Itcl
3.4
% itcl::class OCMCDi {
    variable acc 0
    method accumulate x {incr acc $x}
}
% time {
    OCMCDi obj
    obj accumulate 1; obj accumulate 2; obj accumulate 3; obj accumulate 4
    obj accumulate 5; obj accumulate 6; obj accumulate 7; obj accumulate 8
    obj accumulate 9; obj accumulate 10
    itcl::delete object obj
} 100000
27.50890204 microseconds per iteration
% expr 1000000/27.50890204
36351.86888033282

# -------------------------------------------------------------------
% package require snit
2.2.3
% snit::type OCMCDsn {
    variable acc
    method accumulate x {incr acc $x}
}
::OCMCDsn
% time {
    OCMCDsn obj
    obj accumulate 1; obj accumulate 2; obj accumulate 3; obj accumulate 4
    obj accumulate 5; obj accumulate 6; obj accumulate 7; obj accumulate 8
    obj accumulate 9; obj accumulate 10
    obj destroy
} 30000
190.97659513333335 microseconds per iteration
% expr 1000000/190.97659513333335
5236.24373605485

# -------------------------------------------------------------------
% package require stooop
4.4.1
% stooop::class OCMCDst {
    proc OCMCDst {this} {}
    proc ~OCMCDst {this} {}
    proc accumulate {this x} {incr ($this,acc) $x}
    stooop::virtual proc vaccumulate {this x} {incr ($this,acc) $x}
}
% time {
```

```
    set obj [stooop::new OCMCDst]
    OCMCDst::accumulate $obj 1; OCMCDst::accumulate $obj 2
    OCMCDst::accumulate $obj 3; OCMCDst::accumulate $obj 4
    OCMCDst::accumulate $obj 5; OCMCDst::accumulate $obj 6
    OCMCDst::accumulate $obj 7; OCMCDst::accumulate $obj 8
    OCMCDst::accumulate $obj 9; OCMCDst::accumulate $obj 10
    stooop::delete $obj
} 30000
35.96880766666666 microseconds per iteration
% expr 1000000/35.96880766666666
27801.86680824366
% time {
    set obj [stooop::new OCMCDst]
    OCMCDst::vaccumulate $obj 1; OCMCDst::vaccumulate $obj 2
    OCMCDst::vaccumulate $obj 3; OCMCDst::vaccumulate $obj 4
    OCMCDst::vaccumulate $obj 5; OCMCDst::vaccumulate $obj 6
    OCMCDst::vaccumulate $obj 7; OCMCDst::vaccumulate $obj 8
    OCMCDst::vaccumulate $obj 9; OCMCDst::vaccumulate $obj 10
    stooop::delete $obj
} 30000
168.15855409999998 microseconds per iteration
% expr 1000000/168.15855409999998
5946.768544437669
```

Superclass Call

```
% package require TclOO
0.6.1
% oo::class create SCo_base {
    variable count
    method foo {} {
        incr count
    }
}
::SCo_base
% oo::class create SCo_derived {
    superclass SCo_base
    variable acc
    method foo x {
        incr acc [next]
        incr acc $x
    }
}
::SCo_derived
% SCo_derived create obj
::obj
% time {obj foo 1} 1000000
1.647879855 microseconds per iteration
% expr 1000000/1.647879855
606840.3573026262

# -------------------------------------------------------------------
% package require XOTcl
1.6.3
% xotcl::Class SCx_base
::SCx_base
% SCx_base instproc foo {x} {
    xotcl::my incr count
}
% xotcl::Class SCx_derived -superclass SCx_base
::SCx_derived
% SCx_derived instproc foo {x} {
    xotcl::my incr acc [xotcl::next]
    xotcl::my incr acc $x
}
```

```
% SCx_derived create obj
::obj
% time {obj foo 1} 1000000
5.67232063 microseconds per iteration
% expr 1000000/5.67232063
176294.68875774747

# ------------------------------------------------------------------
% package require Itcl
3.4
% itcl::class SCi_base {
    variable count
    method foo {} {
        incr count
    }
}
% itcl::class SCi_derived {
    inherit SCi_base
    variable acc
    method foo x {
        incr acc [SCi_base::foo]
        incr acc $x
    }
}
% SCi_derived obj
obj
% time {obj foo 1} 1000000
3.483079099 microseconds per iteration
% expr 1000000/3.483079099
287102.29414172715

# ------------------------------------------------------------------
% package require stooop
4.4.1
% stooop::class SCs_base {
    proc SCs_base {this} {}
    proc ~SCs_base {this} {}
    stooop::virtual proc foo {this} {
        incr ($this,count)
    }
}
% stooop::class SCs_derived {
    proc SCs_derived {this} SCs_base {} {}
    proc ~SCs_derived {this} {}
    stooop::virtual proc foo {this x} {
        incr ($this,acc) [SCs_base::_foo $this]
        incr ($this,acc) $x
    }
}
% set obj [stooop::new SCs_derived]
170011
% time {SCs_derived::foo $obj 1} 100000
15.915702479999998 microseconds per iteration
% expr 1000000/15.915702479999998
62831.031257126146
```