Objects in TCL

Introduction

C++ and Java are two well-known languages that offer primitives and in-language support for object oriented programming. That does not mean that object oriented programming is impossible in other languages. Object orientation is really just a way of thinking; it has more to do with design than with implementation. Look at the source code of the TCL interpreter for a great example of object oriented programming in C (not C++!). If you are disciplined enough to always pass a pointer to a struct as the first parameter to a function, you can see such a function as a 'method' of the struct. You do not need actual language support to create object-oriented code.

TCL does not offer object oriented primitives, but it is flexible enough to accomodate new primitives. This paper decribes a well known technique, *object commands*, to add some object primitives to TCL. Once you understand how object commands work, you will be able to figure out the code of most object packages and extensions.

This paper assumes that you are familiar with TCL, and that you have written at least a few simple scripts in TCL.

Existing extensions

Many extensions of TCL exist, that add flavours of objects and classes to the basic language. Some of these extensions are written in C, and must be compiled and linked to the TCL library to make them available in the language. A good example is >> [Incr Tcl], an extension that introduces primitives such as `class`, `method` and `constructor`. Other extensions, such as Jean-Luc Fontaine's >>Stooop, are written in TCL itself. They don't require any recompilation. You may wonder how it is possible to extend TCL with new primitives written in TCL itself. This paper answers that question by zooming in on the techniques of *object commands* and *class commands*. We will not handle the category of extensions that requires compilation.

Another important difference is that between extensions with static and dynamic classes. With *static classes*, the members of a class cannot be changed at runtime. You *can* introduce new classes into a running system, but once a class is created, you cannot add new methods or data members to it. Similarly, you can create new instances of a class, but you cannot (easily) change the class of an existing instance.

[Incr Tcl] is an example of a TCL extension with static classes. You cannot add methods or variables to an existing class or object. You can, however, change the implementation of any method of a class (just like rewriting a procedure body in pure TCL). And of course, you can inherit from an existing class and add new methods and variables in the derived class.

But since TCL is a dynamic language, in which you can introduce new procedures and new variables at run-time, it seems more appropriate to also allow the creation of new methods and

member variables at run-time. That requires a *dynamic class mechanism* such as offered by >> OTcl.

Both static and dynamic object-oriented extensions of TCL can make use of the techniques described in this paper.

Many thanks for bob Techentin for sharing his [Incr Tcl] knowledge.

A simple example

Suppose we want to manipulate objects that store a single attribute, for example the name of a color. Each object has its own color. We also need to give each object a unique identifier or number to distinguish it from other objects.

We store the object colors in a TCL array, indexed by the object name. For example:

```
set a_color(a1) green
set a_color(a2) yellow
set a_color(a3) red
```

We now have three objects a1, a2, a3, each with its own color. Even an extremely simple approach like this one is already useful in many cases where you need to map object attributes to their values. Simplicity is not a bad property of designs, but a very good one.

We can make this more attractive and hide the array, by writing two access procedures:

example1/apples.tcl

```
proc get_color {obj_name} {
   global a_color
   if { [info exists a_color($obj_name)] } {
      return $a_color($obj_name)
   } else {
      puts "Warning: $obj_name has no color!"
      return "transparent"   ; # return a default color
   }
}

proc set_color {obj_name color} {
   global a_color
   set a_color($obj_name) $color
}
```

We now access the colors of objects as follows:

```
set_color a1 green
puts "a1 has color [get_color a1]"
```

The next step is to introduce some syntactic sugar: just a small improvement that makes the syntax look better, but does not really change anything fundamental. We create the following procedure:

example2/apples.tcl

```
proc a1 {command args} {
   if { $command == "get_color" } {
      return [get_color a1]
   } elseif { $command == "set_color" } {
      set_color a1 [lindex $args 0]
   } else {
```

```
      puts "Error: Unknown command $command"
   }
}
```

Using this procedure, we can now access the color of the a1 object as follows:

```
a1 set_color yellow
puts "a1 has color [a1 get_color]"
```

As you can see, all this really does is swap the positions of the object name and the name of the `get_color` or `set_color` procedure. Not very useful in itself, but it makes the syntax look more like an object invocation. It looks as if we invoke the 'method' `set_color` on the 'object' a1.

Procedure `a1` is called an *object command*. Its first argument is the name of a method that you want to invoke on the object. The object has the same name as its object command. Its data is stored in a global array, in this case `a_color`, but that is hidden from the programmer by the object command.

We can now create as many objects as we want: just write a procedure like `a1` for each object, replacing each occurrence of `a1` by the name of another object. Sounds like a lot of work? It is. We will soon see how you can automate this. Writing a separate procedure for every object is not only tiresome; it also imposes heavy resource burdens on the application, because procedures take up space in the TCL interpreter.

The first improvement is that we can write a single dispatcher procedure like this one:

```
example3/apples.tcl
proc dispatch {obj_name command args} {
   if { $command == "get_color" } {
      return [get_color $obj_name]
   } elseif { $command == "set_color" } {
      set_color $obj_name [lindex $args 0]
   } else {
      puts "Error: Unknown command $command"
   }
}
```

The object commands can now be written with only a single line of code:

```
proc a1 {command args} {
   return [eval dispatch a1 $command $args]
}
```

Creating a procedure of this form for each object consumes less memory, simply because the procedure is shorter. But it is still quite cumbersome to write a procedure every time you want to instantiate an object. To simplify this task, we write yet another procedure, one that creates object commands! It looks like this:

```
example4/apples.tcl
proc apple {args} {
   foreach name $args {
      proc $name {command args} \
         "return \[eval dispatch $name \$command \$args\]"
   }
}
```

We call this procedure the *class command*, because it is like a class type that you can

instantiate. Instantiating and manipulating objects is now as simple as this:

```
apple a1 a2 a3
a1 set_color green
a2 set_color yellow
a3 set_color red
puts "a1 has color [a1 get_color]"
puts "a2 has color [a2 get_color]"
puts "a3 has color [a3 get_color]"
```

The class command creates objects of class 'apple'. Each apple has its own color, which can be accessed through the methods `get_color` and `set_color` of the class.

There are still some pieces missing in the puzzle. First of all, we now have a way of creating new objects, but we cannot delete objects yet. This leads to memory leaks, so we need to provide a procedure for deleting apples:

```
example5/apples.tcl
proc delete_apple {args} {
   global a_color
   foreach name $args {
      unset a_color($name)    ; # Deletes the object's data
      rename $name {}    ; # Deletes the object command
   }
}
```

We can also set up the array `a_color` in such a way that `$a_color(obj)` is always filled in for every object. We do this in the class command:

```
proc apple {args} {
   foreach name $args {
      proc $name {command args} \
         "return \[eval dispatch $name \$command \$args\]"
      set a_color($name) green
   }
}
```

This makes the class command act like a constructor that sets up the default values for object attributes. In this case we picked green as the default color. We now use the complete set of procedures like this:

```
apple a1 a2 a3
a2 set_color yellow
a3 set_color red
puts "a1 has color [a1 get_color]"   ; # Uses default color green
puts "a2 has color [a2 get_color]"
puts "a3 has color [a3 get_color]"
delete a1 a2 a3
```

Summary

To summarize, we have followed these steps:

- Store attributes in a global array
- Write a procedure for each 'method' of the object; this method takes the name of the object as its first argument.
- Write a dispatch procedure to call one of those methods.
- For each object, write a procedure (object command) with the same name as the object.

Its first argument is the method name. It calls 'dispatch'.
- For each class, write a procedure (class command) that creates the object commands automatically. The class command can also fill in default attribute values.
- For each class, write a delete procedure to reclaim resources of an object and destroy its object command.

That's it. You now know enough to start using object commands and class commands in TCL. The rest of this paper offers a few more tips and tricks, plus (pointers to) real-life examples where object commands are used.

More attributes

We will give our `apple` class some more attributes, to show you how multiple attributes can be handled. We give each apple a size and a price (both are integers). These are again stored in global arrays, for example `a_size` and `a_price`. Both are indexed by the name of the object, just as for the `a_color` array we've been using so far. And again we can write get/set procedures to access these new attributes. The code is very similar to that for the color attribute, so I will not show it here.

An interesting alternative is to use an array for every *object*, rather than an array for every *attribute*. TCL allows us to create a procedure and an array variable with the same name, so we can call our object command 'a1' and use an array 'a1' to store the attributes of that object. The code of all our procedures now changes slightly:

example6/apples.tcl
```
proc get_color {obj_name} {
   upvar #0 $obj_name arr
   return $arr(color)
}

proc set_color {obj_name color} {
   upvar #0 $obj_name arr
   set arr(color) $color
}

proc dispatch {obj_name command args} {
   if { $command == "get_color" } {
      return [get_color $obj_name]
   } elseif { $command == "set_color" } {
      set_color $obj_name [lindex $args 0]
   } else {
      puts "Error: Unknown command $command"
   }
}

proc apple {args} {
   foreach name $args {
      proc $name {command args} \
         "return \[eval dispatch $name \$command \$args\]"
      upvar #0 $name arr
      set arr(color) green
   }
}

proc delete apple {args} {
```

```
    foreach name $args {
       upvar #0 $name arr
       unset arr        ; # Deletes the object's data
       rename $name {}  ; # Deletes the object command
    }
}

# Note the advantage of using an array per object:
# 'delete_apple' can just 'unset arr' instead of having to
# remove one entry in three different arrays.
```

A third alternative is to use only a single, global array, indexed by the object name *and* the attribute name. To find the color of apple a1, you would have to access `$attributes(a1,color)`. The advantage of having only a single array to maintain, has to be weighed off against the disadvantage of having to delete several array entries in the `delete_apple` procedure.

Configuring the attributes

Another improvement that we can make, is to get rid of all those annoying get/set methods. We do this by introducing two new methods for each class, called `configure` and `cget`. The first gives new values to some attributes, the second reads the value of an attribute. We can implement these procedures for the `apple` class as follows:

```
proc dispatch {obj_name command args} {
   upvar #0 $obj_name arr
   if { $command == "configure" || $command == "config" } {
      foreach {opt val} $args {
         if { ![regexp {^-(.+)} $opt dummy small_opt] } {
            puts "Wrong option name $opt (ignored)"
         } else {
            set arr($small_opt) $val
         }
      }

   } elseif { $command == "cget" } {
      set opt [lindex $args 0]
      if { ![regexp {^-(.+)} $opt dummy small_opt] } {
         puts "Wrong or missing option name $opt"
         return ""
      }
      return $arr($small_opt)

   } elseif { $command == "byte" } {
      puts "Taking a byte from apple $obj_name ($arr(size))"
      incr arr(size) -1
      if { $arr(size) <= 0 } {
         puts "Apple $obj_name now completely eaten!  Deleting it..."
         delete_apple $obj_name
      }

   } else {
      puts "Error: Unknown command $command"
   }
}

# We also change the implementation of the "constructor",
# so that it accepts initializing values for the attributes.
proc apple {name args} {
   proc $name {command args} \
```

```
        "return \[eval dispatch $name \$command \$args\]"

    # First set some defaults
    upvar #0 $name arr
    set arr(color) green
    set arr(size) 5
    set arr(price) 10

    # Then possibly override those defaults with user-supplied values
    if { [llength $args] > 0 } {
        eval $name configure $args
    }
}
```

Attribute access now looks exactly as it does for Tk widgets. Compare these two fragments of code:

```
button .b -text "Hello" -command "puts world"
.b configure -command "exit"
set textvar [.b cget -text]

apple a -color red -size 5
a configure -size 6
set clr [a cget -color]
```

Some widget libraries that are written in pure TCL, use object commands and configure/cget methods to make the widget syntax the same as in Tk. But obviously, this technique also works for other kinds of objects.

Object persistence

We will now cover a more exotic topic: object persistence. This means that you can save an object on disk, and recover it later, in the same or in another application, even in another process. The recovered object has exactly the same attributes as the one you saved.

In languages such as C++, object persistence is quite a challenge (especially if you want to save an object on one platform, and recover it on another platform with different endianness or with a different compiler). But the flexibility of TCL makes object persistence a piece of cake! We will save our objects in a text file, then treat that file as an Active File to read the objects back (Read more about the Active File pattern in my paper on TCL file formats, or on >> Nat Pryce's web site).

We only need a single Tcl procedure (!) to give objects of all classes the ability to make themselves persistent:

```
example8/apples.tcl
proc write_objects {classname args} {
    foreach name $args {
        upvar #0 $name arr
        puts "$classname $name \\"
        foreach attr [array names arr] {
            puts "   -$attr $arr($attr) \\"
        }
        puts ""
    }
}
```

The idea is that this procedure is invoked as follows:

```
write_objects apple a1 a2 a3
```

The implementation above shows that the procedure makes the objects a1, a2, a3 of class 'apple' persistent, by simply outputting a call to the class command 'apple' followed by the object name and all its attributes. The resulting output is stored in a file and looks like this:

```
apple a1 \
   -price 10 \
   -size 5 \
   -color green \

apple a2 \
   -price 10 \
   -size 3 \
   -color yellow \

apple a3 \
   -price 12 \
   -size 5 \
   -color red \
```

It is now extremely easy to read these persistent objects back from disk: just `source` the file! The `source` command executes all class commands in the file, creating instances with exactly the same attributes as the ones we saved earlier. Object persistence in Tcl is indeed a piece of cake.

Adding new classes

So far, we have worked with only a single class `apple`. If we want to add a new class to our example, we need to write a new class command and a new dispatcher procedure.

Suppose we also want to have objects of class `fridge` (in which we will want to store apples of course). We need to duplicate the effort we did on the `apple` class:

```
example10/classes.tcl
proc dispatch_fridge {obj_name command args} {
   upvar #0 $obj_name arr
   if { $command == "configure" || $command == "config" } {
      array set arr $args

   } elseif { $command == "cget" } {
      return $arr([lindex $args 0])

   } elseif { $command == "open" } {
      if { $arr(-state) == "open" } {
         puts "Fridge $obj_name already open."
      } else {
         set arr(-state) "open"
         puts "Opening fridge $obj_name..."
      }

   } elseif { $command == "close" } {
      if { $arr(-state) == "closed" } {
         puts "Fridge $obj_name already closed."
      } else {
         set arr(-state) "closed"
         puts "Closing fridge $obj_name..."
      }
```

```
    } else {
       puts "Error: Unknown command $command"
    }
}

proc fridge {name args} {
    proc $name {command args} \
       "return \[eval dispatch_fridge $name \$command \$args\]"

    # First set some defaults
    upvar #0 $name arr
    array set arr {-state closed -label A}

    # Then possibly override those defaults with user-supplied values
    if { [llength $args] > 0 } {
       eval $name configure $args
    }
}
```

This laborious task can also be partly automated by a procedure called `class` which accepts the name of a new class, a list of its member variables, and a list of its method names. It then automatically sets up the necessary procedures such as the class command and the dispatcher proc. The only thing we still need to implement by hand, are the methods of the class. The whole thing could be set up as follows:

example11/classes.tcl

```
proc class {classname vars methods} {

    # Create the class command, which will allow new instances to be created.
    proc $classname {obj_name args} "
       # The class command in turn creates an object command.  Careful
       # with those escape characters!
       proc \$obj_name {command args} \
    \"return \\\[eval dispatch_$classname \$obj_name \\\$command \\\$args\\\]\"

       # Set variable defaults
       upvar #0 \$obj_name arr
       array set arr {$vars}

       # Then possibly override those defaults with user-supplied values
       if { \[llength \$args\] > 0 } {
          eval \$obj_name configure \$args
       }
    "

    # Create the dispatcher, which dispatches to one of the class methods
    proc dispatch_$classname {obj_name command args} "
       upvar #0 \$obj_name arr
       if { \$command == \"configure\" || \$command == \"config\" } {
          array set arr \$args

       } elseif { \$command == \"cget\" } {
          return \$arr(\[lindex \$args 0\])

       } else {
          if { \[lsearch {$methods} \$command\] != -1 } {
             uplevel 1 ${classname}_\${command} \$obj_name \$args
          } else {
             puts \"Error: Unknown command \$command\"
```

```
            }
        }
    "
}
```

The `class` procedure basically just creates two new commands for us (a class command and a dispatcher).

The code looks pretty messy, because it contains two levels of indirection: a proc that creates a proc that creates yet another proc. This involves a bit of backslash-escape sourcery, which can be confusing. Richard Suchenwirth has a very nice solution to make this kind of proc-creating-proc more readable: he creates a template with names containing a special character such as the '@' sign; then he replaces those names by the actual class and instance names, using `regsub`. See his page on >> gadgets for an example. Using this technique, our implementation becomes a lot simpler:

example12/classes.tcl

```
proc class {classname vars methods} {

    # Create the class command, which will allow new instances to be created.
    set template {
        proc @classname@ {obj_name args} {
            # The class command in turn creates an object command.
            # Fewer escape characters thanks to the '@' sign.
            proc $obj_name {command args} \
                "return \[eval dispatch_@classname@ $obj_name \$command \$args\]"

            # Set variable defaults
            upvar #0 $obj_name arr
            array set arr {@vars@}

            # Then possibly override those defaults with user-supplied values
            if { [llength $args] > 0 } {
                eval $obj_name configure $args
            }
        }
    }

    regsub -all @classname@ $template $classname template
    regsub -all @vars@ $template $vars template

    eval $template

    # Create the dispatcher, which dispatches to one of the class methods
    set template {
        proc dispatch_@classname@ {obj_name command args} {
            upvar #0 $obj_name arr
            if { $command == "configure" || $command == "config" } {
                array set arr $args

            } elseif { $command == "cget" } {
                return $arr([lindex $args 0])

            } else {
                if { [lsearch {@methods@} $command] != -1 } {
                    uplevel 1 @classname@_${command} $obj_name $args
                } else {
                    puts "Error: Unknown command $command"
                }
            }
```

```
        }
    }

    regsub -all @classname@ $template $classname template
    regsub -all @methods@ $template $methods template

    eval $template
}
```

You see that this simplifies the code. We use the '@' sign because it is not frequently used in normal TCL code. We postpone the evaluation of `$classname` and other variables until we are out of the inner procedure body, so that the number of escape characters is reduced to almost zero.

With or without this "template" technique, we can now create our original classes `apple` and `fridge` in a more compact way:

```
example12/classes.tcl
class apple {-color green -size 5 -price 10} {byte}
proc apple_byte {self} {
    upvar #0 $self arr
    puts "Taking a byte from apple $self"
    incr arr(-size) -1
    if { $arr(-size) <= 0 } {
        puts "Apple $self now completely eaten!  Deleting it..."
        delete $self
    }
}

class fridge {-state closed -label A} {open close}
proc fridge_open {self} {
    upvar #0 $self arr
    if { $arr(-state) == "open" } {
        puts "Fridge $self already open."
    } else {
        set arr(-state) "open"
        puts "Opening fridge $self..."
    }
}

proc fridge_close {self} {
    upvar #0 $self arr
    if { $arr(-state) == "closed" } {
        puts "Fridge $self already closed."
    } else {
        set arr(-state) "closed"
        puts "Closing fridge $self..."
    }
}
```

There are several things to note in this implementation:

- Creating new classes is indeed a lot simpler than before. We only need one line with the class "declaration", plus one proc for each of the class methods.
- Each method is implemented as a global proc which has the instance name as its first argument. Any other arguments are optional.
- In the implementation of each method, we access the object's array directly. We could make the methods less dependent on the actual implementation of the object by using `configure` and `cget` instead, for example

example13/classes.tcl

```
proc fridge_close {self} {
   if { [$self cget -state] == "closed" } {
      puts "Fridge $self already closed."
   } else {
      $self configure -state "closed"
      puts "Closing fridge $self..."
   }
}
```

This is less implementation-dependent, and perhaps slightly more readable. It is less efficient though, because the `configure` and `cget` implementations add an extra level of procedure calls with a couple of `if`s. You should probably decide for yourself which of the two ways you are going to use, depending on the importance of efficiency in your application.

- Also note that we can implement the `class` procedure in a slightly different way, without actually knowing in advance the list of all the variables and methods of the class. The new implementation could look like this:

example14/classes.tcl

```
# No more 'methods' argument here; 'vars' is optional
proc class {classname {vars ""}} {

   # Create the class command, which will allow new instances to be created.
   set template {
      proc @classname@ {obj_name args} {
         # The class command in turn creates an object command.
         # Fewer escape characters thanks to the '@' sign.
         proc $obj_name {command args} \
            "return \[eval dispatch_@classname@ $obj_name \$command \$args\]"

         # Set variable defaults, if any
         upvar #0 $obj_name arr
         @set_vars@

         # Then possibly override those defaults with user-supplied values
         if { [llength $args] > 0 } {
            eval $obj_name configure $args
         }
      }
   }

   set set_vars "array set arr {$vars}"
   regsub -all @classname@ $template $classname template
   if { $vars != "" } {
      regsub -all @set_vars@  $template $set_vars template
   } else {
      regsub -all @set_vars@  $template "" template
   }

   eval $template

   # Create the dispatcher, which does not check what it
   # dispatches to.
   set template {
      proc dispatch_@classname@ {obj_name command args} {
         upvar #0 $obj_name arr
         if { $command == "configure" || $command == "config" } {
            array set arr $args
```

```
            } elseif { $command == "cget" } {
               return $arr([lindex $args 0])

            } else {
               uplevel 1 @classname@_${command} $obj_name $args
            }
         }
      }

      regsub -all @classname@ $template $classname template

      eval $template
   }

   # ...

   fridge f1 -state open
   f1 close

   # Even after 'f1' is created, we can add a new method to the 'fridge'
   # class.  'f1' automatically gets the new method.
   proc fridge_paint {self color} {
      puts "Painting fridge $self $color ..."
   }

   f1 paint green
```

This implementation shows that you can add new methods to an existing class, simply by implementing a new global procedure named `classname_methodname` with `self` as its first argument. The dispatcher procedure will find this new method even though it did not yet exist at the time the class was created. The same is true for member variables (this has silently been the case in all previous examples, in fact): just call `configure` with a new variable name, and it will end up in the object's array correctly. Only variables specified in the `class` procedure get a default value, though; Other variables do not exist before they are first set by `configure`!

Advanced techniques

After posting a news message about this paper, I received so many interesting responses that I should really make time to write a sequel. Many more advanced topics need to be covered here! For now, I don't have enough time to go over them in detail, so I just made a short list of topics that deserve more attention. Obviously, your input is more than appreciated.

Memory leaks in TCL

Instead of storing object attributes in global arrays, you could create *local* arrays and pass them around with `upvar`. That way, these arrays disappear when they go out of scope. Stops a lot of memory leaks. Then set a trace on the array so that when it goes out of scope, you also delete the object command. Thanks to Richard Suchenwirth for this tip.

Introspection

In the spirit of TCL, classes and instances should offer introspection. For example:

- Ask an instance what class it has (store "class" as a special attribute).
- Ask the list of all classes currently available.
- Ask a class for a list of all its instances.

Persistence revisited

- Saving objects can now be done without giving the class name (thanks to introspection).
- In fact, you can now write a proc that saves all objects in the system, in one fell swoop.
- Combine the two in one ftion: If you pass some instance names as parameters, only those instances are saved; otherwise, all instances are saved.
- Save the class command with the instances, so that the script that invokes "source" does not need to know in advance what an 'apple' is.

From object-based to object-oriented

Virtual functions, inheritance, ... Not yet written. Also operator overloading a la C++, cfr Wiki's Gadgets etc.

Smarter objects

A la Python, give the objects special procs for printing themselves, saving themselves, converting to/from string (easy in TCL :-), ... This already goes some distance towards roles.

Generic programming

Is generic programming required in a scripting language? Is it useful? How does it look? Generalized data structures?

Roles

Explain the general concept of *roles*, then show some simple examples. This is about roles in TCL, not about roles in C++. Languages like C++ make it very hard to do this kind of stuff, but in TCL it should be easier. Refer to roles in the g2 preprocessor.

Namespaces

Will's paper covers this in a lot of detail.

Dynamic classes/instances

Adding new variables/methods to existing classes/instances. Prototype-based language; copying an object to another one (copy its variable array, copy the object command too).

Reference links

Find out more about objects and Tcl at any of the following pages:

- >>SWIG is an excellent tool for making C and C++ objects available in TCL. You can combine SWIG with some of the techniques in this paper to provide an object-oriented TCL interface for your C++ classes.
- >> Otcl is an object-oriented extension of the Tcl language.
- >>Stooop is another object-oriented extension, but entirely written in TCL.
- > Todl is a tool that generates TCL code from a high-level class description, so that you can have TCL objects up and running in very short time.
- Richard Suchenwirth pointed me to the Lightweigth Object System for TCL ("LOST") on the Tcl'ers wiki, and the very elegant TCL gadgets on which it is based.
- >> Will's Guide to Creating Object Commands is another paper on using object commands in Tcl. It covers more advanced topics such as dealing with object commands in namespaces.
- >> Nat Pryce describes a pattern called *widget commands*, which is very close to the class commands in this paper.
- >>Cetus Links maintains a page *full* of links about Tcl; some of them are about objects too.

Many thanks to everybody on comp.lang.tcl who read this paper and helped me correct some mistakes and make many improvements. Special thanks to Richard Suchenwirth, Jean-Luc Fontaine, and Bob Techentin for their technical insights.

Koen Van Damme (koen.vandamme1 at pandora.be)
kvd's home