Proceedings of the 7th USENIX Tcl/Tk Conference

Austin, Texas, USA, February 14–18, 2000

FEATHER: TEACHING TCL OBJECTS TO FLY

Paul Duffin

Feather: Teaching Tcl objects to fly

Paul Duffin
IBM Hursley Laboratories, UK
pduffin@hursley.ibm.com, http://purl.oclc.org/net/pduffin/home

December 16, 1999

Abstract

Feather is a set of extensions which enhances the capabilities and flexibility of Tcl objects, builds a framework around them to enable new data types to be easily added and also provides a rich set of new data types. This paper starts off by giving quite a detailed description of how Tcl objects work and then moves on to describe the features of **Feather**, how they work and what they can be used for.

1 Introduction

One of the major complaints raised against Tcl by advocates of other scripting languages such as Python, Lisp and Perl is that it has very few data structures.

Feather is the remedy.

The introduction of Tcl objects in Tcl 8.0 not only greatly improved the scalability of Tcl in terms of speed and memory usage, it also provided the foundation for **Feather** to build on.

Feather is a set of extensions which not only provides lots of new data types, it also provides frameworks into which new data types can be easily inserted. It was decided early on that **Feather** must not require any changes to the Tcl core in order to make it as easy as possible for people to use. At times it seemed that the decision might have to be changed, but in the end a core change was not needed.

As there is not enough space here to describe all the features of **Feather** completely, this paper aims to provide enough information to prove that it is technically feasible and to show you some of the exciting things which **Feather** makes possible.

The paper starts off by giving quite a detailed description of how Tcl objects work and their limitations and restrictions. This should provide a sound basis for the next few sections which describe the various enhancements that **Feather** makes. The final sections describe what **Feather** uses these enhancements for, gives a brief summary of what could not be included and a look towards the future.

2 Tcl objects

This section aims to give a description of how Tcl objects work, it describes why they were introduced, gives an overview of them and then goes into more details, especially of the subtleties which cause the most problems for users.

2.1 History

Versions of Tcl prior to 8.0 used C NUL terminated character strings as their primary data type. This approach has the following scalability problems and data representation limitations.

- Everything from lists to procedure bodies to loop bodies had to be parsed every time they were used.
- Strings had to be copied whenever a long lasting reference to it was required.
- Binary data was not easily supportable because in general no length was associated with the strings.

Hence the Tcl objects were created.

2.2 Basic characteristics

The starting point for Tcl objects is that as far as the Tcl programmer is concerned they must behave exactly like strings do. In other words the Tcl programmer must be able to treat *everything as a string*.

In order to provide the necessary functionality each Tcl object has the following information associated with it.

- Reference count.
- Type.
- Dual representations
  - A string representation, including length.
  - A type specific, or internal representation.

2.2.1 Reference counted

Much of the string copying done in older versions of Tcl was done because a private reference to the data was required and there was no way of protecting the data from being changed, or of detecting when it needed to be freed.

Tcl objects solve both of these problems by using a simple reference count to keep track of how many references there are. The object can only be freed when the reference count reaches 0, and it can only be changed if the reference count is less than or equal to 1, i.e. only the code which is changing the object has a reference to it.

```
set a "hello"
set b $a
```

In the above code `a` is a reference to the literal object `"hello"` which has a reference count of 1. The assignment of `$a` to `b` simply makes `b` a reference to the object and increments its reference count to 2. The equivalent code in a string based version of Tcl would have to copy the string value of `a`.

2.2.2 String representation

This is the most important information stored in a Tcl object as it allows it to behave like a string. The length of the string is also stored in the object which means that it can handle binary data with embedded NULs.

2.2.3 Typed

A Tcl object's type basically determines the format of the internal representationn; the actual type which an object has is determined by the context it is used in. For example, an object which is used as a number will have a number type and an object which is used as a list will have a list type. If an object does not have the correct type for the current context it is automatically converted to the correct type if possible, otherwise an error is reported.

Literal strings which are found in source code have no internal type at all so they are untyped.

```
set a "1 2 3 4"
set b [lindex $a]
```

In the above code `a` starts as a reference to the literal object `"1 2 3 4"`, `[lindex]` then changes the type of the object to **list** generating an internal list representation in the process. The string representation is not changed.

Each type has to define the following operations:

- Free the internal representation.
- Duplicate the internal representation.
- Convert to type.
- Update the string representation.

2.2.4 Type specific representation

As its name suggests the nature of this information depends on the type of the object.

2.3 Behaviour and rules

Although Tcl objects are quite simple structures the rules governing their behaviour can be a little confusing. When Tcl objects were first introduced it took quite a long time to remove all of the problems caused by their unforeseen behaviour.

2.3.1 Dual representation

As mentioned above Tcl objects have two representations, a string one and an internal or type specific one. Each type provides mapping functions to convert from one to another. The built in Tcl types fall into two broad categories depending on the mapping function they use.

A *transparent* type is one whose string representation contains all the information that the internal representation does. They have the following characteristics.

- Mapping from the string representation to the internal representation requires no other information.
- Changes to the internal representation requires changes to the string representation.
- Objects of this type have no seperate identity, an identical but seperate object can be created simply by copying the string and converting to the correct type.
- The internal representation is unique to a Tcl object.
- Objects of this type are automatically freed when they are no longer needed.

A *handle* type is one whose string representation is simply an identifier for the internal representation.

- Mapping from the string representation to the internal representation requires additional information such as a hash table.
- Changes to the internal representation does not require changes to the string representation.
- Objects of this type have their own identity, copying the string and converting to the correct type, simply results in another object which refers to the same internal representation as the original.
- The internal representation may be shared between many Tcl objects.
- Objects of this type have to be explicitly freed.

Integer, **double**, **boolean**, **string**, **regular expression** and **list** are all *transparent* types. **Command**, **index**, **window** and **font** are all *handle* types.

2.3.2 Conversions

Converting from one type to another is done as follows

1. Update the string representation.
2. Invalidate the existing internal representation.
3. Parse the string representation and create the new internal representation.
4. Change the type.

Step 2. calls the free internal representation operation of the current type which means that it is difficult to distinguish between converting from one type to another and deleting the object.

2.3.3 Shared objects

An object is shared when its reference count is greater than 1 and you have to be very careful when working with objects which are or could be shared. In fact you should assume that any object you are working with is shared, unless you know otherwise.

The one rule that you must follow when working with shared objects is that you must **never** modify the string representation: you can do just about anything else to it but that. If no string representation exists then it is valid to generate it but once it exists you must not change it. i.e.

```
set a "hello"
set b $a
append b ", world"
puts $a
```

In the above code the assignment of `a` to `b` causes the literal string object `"hello"` to have a reference count of 2 and hence is shared. If `[append]` modified the string representation of that shared object then the code would output `"hello, world"` which is not correct according to Tcl's semantics. Instead, if `[append]` has to append to a variable whose value is shared it first duplicates the value object and works with that duplicate ensuring the correct behaviour. This is called *copy-on-write* semantics.

2.3.4 Lossy conversions

A lossy conversion is one which loses information, e.g. converting from a floating point number to an integer can lose information. If an object is subject to a lossy conversion it is most important that the string representation is generated before the conversion happens.

```
set a 1
incr a
set b [expr {$a * 0.1}]
puts $a
```

In the above code `a` is initialised with an object which has a string representation of `"1"`, a type of integer and a value of 1. It is then incremented, resulting in it being assigned a new object[1] with no string representation, a type of integer and a value of 2. `[expr]` causes the object to be converted to a double type, with a value of 2.0. If that conversion did not generate the string representation from the integer value then the final line would output 2.0, instead of 2 which is of course incorrect.

2.3.5 Literal handling

When parsing, Tcl replaces all literal strings with a Tcl object and identical literal strings are replaced with the same Tcl object. In the following code all occurrences of `.b` are replaced with the same Tcl object.

```
button .b -command {
  puts "Button pressed"
}
pack .b
```

This behaviour has lots of advantages in that it reduces the memory requirements and if the object is used in the same context as shown above it can also improve the performance by allowing subsequent uses to benefit from any cached information. However, because this process increases the number of shared objects it also can have disadvantages because of the increased chance that shimmering (see below) will occur.

If the literal string looks like an integer then the parser will actually create an integer object, rather than simply a literal string object.

2.3.6 Shimmering

This is what happens when an object is repeatedly converted from one type to another and happens most often with shared objects but can happen with any object, because the problem is caused not by the number of references but by the number of different uses made of the object. These conversions can take up a lot of time causing performance problems. It is very difficult to actually quantify the effect shimmering has on a program as there is no way at the moment to monitor the conversions which take place. This also means that it is difficult to find these problems, unless you have a deep understanding of the internals of Tcl.

```
set b 1.0
for {set a 0} {$a < 200} {incr a 2} {
  set b [expr {$b / 2}]
}
```

In the above admittedly slightly contrived example the literal integer object 2 is used both as an integer, by `[incr]` and as a double, by `[expr]`. This means that it will be repeatedly converted from one type to another with the consequent loss of performance.

[1] If the object is not shared then it would be the same object.

2.3.7 Threads

Accesses to Tcl objects are not serialised because the overhead would be too great and for the most part unnecessary. Because of this you must not use an object from two different threads at once. It is possible to pass objects from one thread to another but they must not be used concurrently. Also, shared objects must never be passed from one thread to another because it is almost certain that this rule would be broken as you do not have control over how and when the object is to be used.

3 Interfaces

One of the biggest limitations of Tcl objects is that they can only have one internal representation which is the primary cause of shimmering. Adding additional internal representations is not really a suitable solution for the following reasons.

- Increased memory usage in both the Tcl_Obj structure itself and for all the internal representations which now exist concurrently.
- Increased complexity managing the internal representations, choosing which one to free, finding an internal representation of the correct type, etc.
- It will most likely introduce major incompatabilities with existing extensions.

The approach that **Feather** takes is to keep one internal representation but to allow each object type to behave in different ways depending on the context, which means that in many cases conversions become unnecessary. This does not solve all shimmering problems but it does eliminate a large set of them which for the most part tend to be impossible for the Tcl programmer to do anything about.

These additional behaviours are provided by *interfaces*. If an object type can behave like a command then it provides an implementation of the command *interface*, if it can behave like a foobar then it provides an implementation of the foobar *interface*.

Conversely, if a particular context needs an object which behaves like a foobar then it checks to see if it provides a foobar *interface*. If it does then the *interface* is used, otherwise the context could try and convert the object to a known type which does provide a foobar *interface*. If the conversion failed, or no such type existed then an error would be reported.

3.1 Implementation

An *interface* definition is simply a structure of function pointers and possibly other related data. The Tcl_ObjType structure is the *interface* which Tcl object types must provide. An *interface* implementation consists of a set of functions and an initialised instance of the structure.

Rather than go into details of exactly how the *interface* mechanism works I will save space and time by just describing its important features.

- It is very dynamic.
- The Tcl_ObjType is extended to allow a set of *interfaces* to be associated with it. This is done dynamically to avoid any incompatabilities with existing extensions.
- Each *interface* definition has a unique fixed name. e.g. the **Feather** command *interface* is called "feather::command".
- Each *interface* definition used is dynamically allocated a key. This key is used to retrieve an *interface* from a Tcl_ObjType in O(1) time.
- *Interfaces* have to be added to, and removed from, Tcl_ObjTypes dynamically.

While developing the *interface* mechanism I did take a brief look at the way Python worked but rejected it immediately because it only seems to support a fixed set of *interfaces* which are hard coded into its structures. The definition of PyTypeObject in the Python header file object.h has space reserved in it for each *interface* that is supported with unused space for future expansion. This seemed far too restrictive for Tcl which prides itself on its dynamism and lack of limits.

3.2 Polymorphism

A polymorphic function, or command, is one which can work with different types of objects. *Interfaces* support polymorphism in just the same way as abstract virtual classes in C++ [1] do,

4 Opaque objects

As mentioned before Tcl only supports a few data types; strings, numbers, lists and arrays. While the latter two are very powerful they can be difficult to use and may also be relatively inefficient. **Feather** attempts to solve these problems by providing a rich set of new types.

Unfortunately, neither a *transparent* or a *handle* type was suitable for these new types. *Transparent* objects must in general be copied before being changed. This would be very inefficient for the types that I wanted to provide and make them much harder to use. On the other hand the fact that *handle* objects need to be explicitly freed makes them unsuitable because of the increased risk of memory leaks. Therefore, I created a new category of object types called *opaque*. *Opaque* objects combine parts of *transparent* and *handle* types and have the following characteristics (labelled to indicate whether they were taken from *transparent* or *handle* type)

- Mapping from the string representation to the internal representation requires additional information such as a hash table. (*Handle*)
- Changes to the internal representation does not require changes to the string representation. (*Handle*)
- Objects of this type have their own identity, copying the string and converting to the correct type, simply results in another object which refers to the same internal representation as the original. (*Handle*)
- The internal representation may be shared between many Tcl objects. (*Handle*)
- Objects of this type are automatically freed when they are no longer needed. (*Transparent*)

4.1 Mutable objects

Mutable objects consist of those *opaque* objects whose internal representation can be changed. Support for *mutable* objects was the primary reason for creating *opaque* objects although as can be seen later it is not the only one. The majority of the new **Feather** types are *mutable*.

The introduction of *mutable* objects into Tcl brings with it the ability for Tcl scripts to create cycles of Tcl objects. These cycles will not be cleared up by simple reference counting and **Feather** does not currently support garbage collection.

4.2 Implementation

The implementation of *opaque* objects is quite simple but does reveal some subtleties about their use which mean that they are not suitable for all purposes. While the following implementation is not the only way to implement *opaque*-like types I have found it to be the best both in terms of simplicity of implementation and versatility from the Tcl programmer's perspective. Also, remember that **Feather** provides a framework for creating *opaque* objects and therefore has to cope with problems in a very general manner.

The internal representations of *opaque* objects need to be reference counted because each one can be referenced by multiple Tcl objects. In fact it is the internal representation which uniquely identifies the *opaque* object, not a Tcl object.

In addition to the normal Tcl_ObjType operations, *opaque* objects also need to support the *opaque interface*. This *interface* consists of the following operations:

- Set internal representation.
- Get internal representation.
- Increment reference count.
- Decrement reference count.

The main reason why this *interface* is needed is to allow an object whose string representation refers to an *opaque* object to be converted back to that object whatever type it may be.

The following describes the Tcl_ObjType interface that *opaque* objects implement and also the *opaque interface* that they use.

4.2.1 Update the string representation

This operation has to create a string representation which uniquely identifies the internal representation amongst all *opaque* objects. At the moment **Feather** objects uses a combination of the type name, for readability, and the address of the internal representation, for simplicity.

As soon as, but not before, the string representation is generated it becomes necessary to provide a way to convert an object from its string representation to the correct type.

```
set opaque [createFooObject]
eval useFooObject [list $opaque]
```

In the above code `[eval]` concatenates the string representations of the literal string `useAsFooObject` and the string representation of the list containing the *opaque* object referred to by `opaque` and then evaluates it. At this point `[useAsFooObject]` is passed an untyped Tcl object which it needs to convert to an *opaque* **Foo** type.

While this conversion back from the string representation to the *opaque* object type is not strictly necessary, without it the objects are almost unusable because they will not work properly with `[eval]` or `[uplevel]` which are very commonly used.

Therefore, once this operation has generated the string representation it then stores the internal representation and the type of the *opaque* object into the table of *opaque* objects using the string representation as the key.

4.2.2 Convert to type

This operation uses the string representation as a key to find the internal representation and object type in the table of *opaque* objects. If it found them it then checks that the stored type and the type to which it is being converted match and only then does it do the conversion. If either the key was not valid, or the check failed then an error is generated and the conversion does not go ahead.

The conversion is actually done by the *set object from internal representation* operation from the *opaque* interface.

4.2.3 Duplicate the internal representation

If the object is *mutable* then this operation simply has to create a duplicate of the internal representation and associate it with the duplicated Tcl object, just like *transparent* objects.

If on the other hand, the object is not *mutable* then this operation can either create a duplicate of the internal representation as above, or it can just increment the reference count of the existing internal representation.

Duplicating the internal representation requires that the string representation of the duplicated object be invalidated because otherwise there would be two different *opaque* objects with the same string representation. This breaks the rule defined in the section above that the string representation has to uniquely identify the internal representation.

Invalidating the string representation of the duplicated object may seem like a dangerous thing to do but it is actually quite safe. The duplication process was designed with *copy-on-write* semantics in mind so it was never intended to create a clone of the original object but rather a copy of the object which could be modified. The string representations of duplicated objects are usually invalidated after they have been created. Because *opaque* objects do not support *copy-on-write* semantics they never need to be implicitly duplicated, rather they are explicitly duplicated when necessary.

As far as the Tcl programmer is concerned the difference between duplication and incrementing the reference count is that the former changes the string representation of the object whereas the latter does not.

4.2.4 Free the internal representation

This operation simply decrements the reference count of the internal representation. Only when the reference count reaches zero is the internal representation actually freed. Freeing the internal representation also removes the information associated with the string representation in the table of *opaque* objects.

4.2.5 Set internal representation

This operation takes a Tcl_Obj pointer and a pointer to the internal representation and converts the object to the correct type and attaches the internal representation to it.

```
set opaque [createFooObject]
eval useOpaqueObject [list $opaque]
```

In the above code `[useOpaqueObject]` is passed an untyped string object which it needs to convert to an *opaque* object, however, it does not know what the type of the object should be. The conversion process uses the string representation as a key to find the internal representation and the object type in the table of *opaque* objects. If it found them it then calls this operation to do the actual conversion.

4.2.6 Get internal representation

This operation takes a Tcl_Obj pointer and returns a pointer to the internal representation.

4.2.7 Increment reference count

This operation increments the reference count of the internal representation.

4.2.8 Decrement reference count

This operation decrements the reference count of the internal representation.

4.3 Interaction with interfaces

The process of retrieving an *interface* implementation from an object is complicated by the introduction of *opaque* objects. Without them, a single query to the object is sufficient to determine whether it supports the *interface*. With them, the object has to be converted to an *opaque* object first if necessary and then queried.

4.4 Limitations

Opaque objects have some limitations which mean that they are not suitable for all uses. n

4.4.1 Conversions

You have to be very careful when using *opaque* objects that you do not convert them to other types as this will cause the reference from the Tcl object to the internal representation to be released which may cause the internal representation to be freed.

transparent objects do not have this problem because they can be recreated from the string representation if necessary and *handle* objects have to be explicitly freed.

4.4.2 Callbacks

Opaque objects do not work well with string based callbacks.

```
set opaque [createFooObject]
scrollbar .sb -command [list command $opaque]
unset opaque
```

In the above code `[scrollbar]` takes a copy of the string representation of the list and stores it away. By the time the scrollbar command is evaluated `[unset]` has caused the *opaque* object to be freed.

The solution to this is to make sure that the *opaque* object exists as long as the `-command` option refers to it.

The following code fails in the same way, even though button commands are stored as Tcl objects, because the Tcl parser concatenates `bCommand` and the string representation of the *opaque* object before calling `[button]`.

```
set opaque [createFooObject]
button .b -command "command $opaque"
unset opaque
```

5 Container objects

A container object is simply an object which provides a container *interface*.

5.1 Interface

The container *interface* defines operations to read and write elements, to remove elements, to check that elements exist, to get the size, contents and keys of the container and also to get an iterator for the container. It can handle both single dimensional, e.g. lists, and multi-dimensional containers, e.g. matrices.

Container iterators have their own *interface* which defines operations to increment and decrement it, to read and write elements, to free it and to check that it has reached the end.

The **list** type is the default container type. This means that if an object being used as a container does not have a container *interface* and is not *opaque* then it is converted to a **list** object, which does have a container *interface* courtesy of **Feather**.

5.2 Implementation

The implementation of container objects is very simple. There is one thing however which is worth mentioning because it affects how the Tcl programmer uses them. The string representation of the *opaque* container objects contains a space to enable polymorphic functions like `[loop]` and `[container]` to differentiate between the string representation of an *opaque* container object and the string representation of a list containing a single *opaque* container object.

5.3 New container objects

Unfortunately, there is not enough space to describe all the details of each of the container types that **Feather** provides. The following list just provides a brief description of each one.

chain A *mutable* linked list which is O(1) for insertion and removal and O(N) for indexing.

hash A *mutable* hash table similar to, but faster and more efficient than, a Tcl variable array. The main reason for this is that it uses Tcl objects and not strings for the keys.

map A *mutable* container based on a red-black or 2-3-4 tree. The objects are stored in order in the tree, and searching and inserting are all very efficient. This is the perfect container to use to implement a set construct.

sequence A *transparent* container which can be used to efficiently create repetitions of objects or arithmetic series.

structure A *mutable* container very similar to C structures.

vector A *mutable* container very similar to Tcl lists.

Creation of a container object is done by calling the Tcl command of the same name with the first argument as `create`. This command also provides type specific operations. Operations which can be done through the container *interface* are all done using `[container]`.

5.4 Polymorphic container commands

5.4.1 container

This command provides access container objects through the container *interface*. The following code creates a vector and a hash object and then gets the first element in the vector (indexed by 0) and the colour of a frog.

```
set vector [vector create alpha beta gamma]
set colourOf [hash create frog green ]

container get $vector 0
container get $colourOf frog
```

5.4.2 loop

This command provides a way to iterate over containers. The following code iterates over the contents of the vector and prints them on stdout.

```
set vector [vector create alpha beta gamma]
loop letter $vector {
    puts $letter
}
```

6 Command objects

Command objects are Tcl objects which can be used in place of a normal Tcl command. The distinguishing feature of *command* objects is that they implement the *command* interface.

Although all existing **Feather** *command* objects are *opaque* they do not need to be. In fact Tk windows are a prime example of a *handle* object type just waiting to be converted into a *command* object.

6.1 Interface

The command *interface* consists of one function whose prototype is the same as a Tcl_ObjCmdProc.

cmdName is the default command type.

6.2 Implementation

Command objects are implemented by modifying the behaviour of the **cmdName** type in a similar way to Tcl Blend [2]. Unfortunately, I did not discover this until well after I had implemented by own version.

What differentiates **Feather** from Tcl Blend is that **Feather** provides a framework for others to add in their own *command* objects.

One important distinction between *opaque command* objects and other *opaque* objects is that the string representation of an *opaque command* object contains no characters which are treated specially by `[eval]`, i.e. spaces, newlines and semicolons. The reason for this is to allow them to easily pass through `[eval]` without having to worry about using `[list]` to protect them.

6.3 Lambda objects

This is basically an unnamed procedure which is wrapped up in an *opaque* object. It is *opaque* in order to hide the arguments and the body.

A lambda *command* object is created by using `[lambda]` which takes the same arguments as `[proc]` apart from the name. It should behave identically to a named procedure with the same arguments and body.

```
set command [lambda $parameters $body]
eval $command $arguments
```

The above code should behave identically to the following code ignoring any differences due to the different names.

```
proc command $parameters $body
eval command $arguments
```

Just like `[proc]`, `[lambda]` does not support static scoping, however by combining lambda objects and curried objects which are described later it is possible to emulate it.

6.3.1 Examples

The following code creates a lambda *command* object which returns the result of multiplying its argument by itself. When applied to 12 this returns 144, or 12^2.

```
set square [lambda {x} {
  expr {$x * $x}
}]
$square 12
```

The following code creates a button which, when pressed, outputs `"Pressed"`.

```
button .b -command [lambda {} {
    puts Pressed
}]
pack .b
```

6.3.2 Performance

Lambda objects are almost as fast as normal procedures and the difference between them is mainly due to the overhead arising from the overriding of the **cmdName** type. With a suitable patch to the Tcl core command objects should be just as fast as normal procedures.

6.4 Curried objects

A curried object is an *opaque command* object which encapsulates a *command*, or *command* object and a list of other Tcl objects. When the curried object is invoked it in turn invokes the encapsulated *command* with its arguments appended to the end of the encapsulated objects.

The following code creates a curried object containing a command and the literal objects `1` and `2`. Invoking the curried object with the literal objects `3` and `4` results in `[command]` being invoked with the literal objects `1`, `2`, `3` and `4`.

```
proc command {args} {
    puts $args
}
set object [curry command 1 2]
$object 3 4
```

As mentioned above curried objects can be used with lambda objects to allow most if not all functional programming constructs to be used. Curried objects are also very useful for callbacks and for implementing Tk window-like object systems.

6.4.1 Examples

The following code creates a procedure, `compose`, which takes two commands as its arguments, constructs another command object by composing those two commands together and then returns it. It then uses `[compose]` to create a command object which returns the result of multiplying its argument by itself and then multiplying the result of that by itself. When applied to 3 this returns 81, or $(3^2)^2$ or 3^4.

The currying is necessary because Tcl does not support static scoping and apart from substitution, which will not work properly with command objects, there is no way for the composed command to get access to the creating commands variables.

```
proc compose {f g} {
  curry [lambda {f g x} {
    $f [$g $x]
  }] $f $g
}
set quad [compose $square $square]
$quad 3
```

It is also very easy to create simple object oriented interfaces. For each instance that you want to create you first create a *mutable* object which contains the state of that instance, and then you encapsulate that along with the command which manages the state of the object in a curried object and return that as your objects command.

```
proc command {clientData method args} {
  switch -- $method {
    :
    : Modify the $clientData object here.
    :
  }
}
```

```
proc factory {args} {
  set clientData [createStateObject]
  set command [curry command $clientData]
  return $command
}
```

6.4.2 Performance

When passing large lists through `[eval]`, or `[uplevel]` it is much faster to wrap them in curried objects than using `[list]` to protect them. This is because it eliminates the need to generate and then reparse the string representation of the list.

7 Additional features of Feather

Unfortunately, there has not been enough room to describe in detail all of the features of **Feather** so here is a brief description of some of the things which were not included.

Tables This is the unimaginative name that I have given to the very useful data type upon which the *interface* mechanism is built.

Per interpreter data This is an AssocData-like mechanism (using *tables*) which is faster O(1) and requires less memory per key than the existing mechanism based on hash tables.

Overrides This is a mechanism to change the behaviour of existing commands in much the same way as `[rename]` and `[proc]` can but it does not pollute namespaces with renamed procedures and the overrides can be removed in any order.

Handles These are general purpose handles which can be used to safely pass Tcl objects through hazardous environments such as string based callbacks.

Generic interface This allows a Tcl object type to choose what interfaces each object of that type provides.

Generic object The generic object uses this feature to expose interfaces to the Tcl programmer.

8 The future of Feather

Feather is nowhere near finished yet, here is a selection of the things which need doing to it and with it.

- Work with Scriptics to integrate some of the **Feather** stuff into the Tcl core.
- Finish off the existing container types, and create new ones such as matrices.
- Define new *interfaces* such as a numeric one for use by `[expr]`.
- Serialisation of objects in both binary and ascii format.
- Create yet more object oriented systems using the **Feather** concepts.
- Try and define a mechanism to allow one object to be wrapped around another.
- Garbage collector to clean up reference loops created by *mutable* objects.
- Create some new types to provide efficient communication between threaded interpreters.

9 Acknowledgments

Thanks to everyone who helped with the design and implementation of **Feather** and also to those who helped with this paper.

A special thank you to Jean-Claude Wippler for yet another good idea and to Alexandre Ferrieux for spending the time to convince me that conversion from string to opaque object was needed.

The biggest thank you must however be reserved for my wife Julie who read about 4 or 5 different versions of this paper even though she did not under it.

10 Availability

At the time of writing I have not yet managed to get permission to make the source code freely available. As soon as I get it I will announce it on comp.lang.tcl and comp.lang.tcl.announce.

References

[1] Bjorn Stroustrup, *The C++ Programming Language, 3rd Edition*, Addison-Wesley Publishers (1997).

[2] Tcl Blend,
http://www.scriptics.com/products/java/,
http://ptolemy.eecs.berkeley.edu/ cxh/java/

[3] comp.lang.tcl

The reference section is very short because I have not had access to conference papers before and most of my knowledge about what is happening in the rest of the Tcl world is obtained from [3].