Extending Tcl for Dynamic Object-Oriented Programming

David Wetherall and Christopher J. Lindblad
Telemedia Networks and Systems Group
Laboratory for Computer Science
Massachusetts Institute of Technology

category	Tcl/Tk	Object Tcl	[incr Tcl]
object			
creation	button .b -height 10	Button b -height 10	Button B -height 10
			Button #auto -height 10
		Class Button ...	itcl_class Button ...
destruction	destroy b	b destroy	B delete
		Button destroy	
procedure			
creation	proc flash	Button instproc flash	Button ... method flash
		Button proc flash	Button ... proc flash
		b proc flash	
invocation	flash	b flash	B flash
			virtual flash
			flash
		Button flash	Button :: flash
			Button::flash
variable			
creation	set msg	b set msg	Button ... public msg
			Button ... protected msg
		Button set msg	Button ... common msg
reference	upvar msg	b instvar msg	
introspection			
	info <option>	b info <option>	B info <option>
		Button info <option>	itcl_info <option>

Table 2: Usage Analogies

specialized, a capability seen in languages such as Self [13]. This is an outgrowth of a strategy that we refer to as the *object command* approach. We believe it is well suited to Tcl/Tk since it directly supports the grouping of each widget object with its behaviors and related data. Our approach also shares goals with languages such as Dylan [5], which attempt to be practical on small machines while providing many of the language features found in CLOS and other advanced object systems.

This paper presents Object Tcl, emphasizing the language design and implementation issues that we encountered. We begin with the *object command* approach for grouping data and operations in Tcl, and the basic objects that result from taking this approach to its conclusion. Then we consider the additional abstraction capabilities provided through inheritance mechanisms, and through a class system. This progression is intended to reflect increasing levels of both functionality and design, much as we followed in the development of Object Tcl. We assume a familiarity with object-oriented programming techniques and Tcl/Tk.

2 Comparison to Tcl/Tk & [incr Tcl]

Before presenting Object Tcl, we provide a rough comparison with Tcl/Tk and [incr Tcl]. An analogy with Tcl/Tk helps to show how Object Tcl is designed to extend Tcl. An analogy with [incr Tcl] (the most popular of the object extensions) helps those familiar with it to gauge Object Tcl.

Features

Table 1 summarizes a comparison of features, and should be taken as a promise: by the end of the paper, each item in the table will be addressed. Many of the differences stem from the different biases of C++ and Flavors. [incr Tcl] classes are defined as a single block, while Object Tcl classes are defined incrementally across as many blocks as needed, typically one block per method and with few ordering constraints. This difference goes hand-in-hand with extensibility. Methods and classes are encouraged to be individually changed at any time in Object Tcl, while they tend to be grouped and declared once in [incr Tcl][1].

The support for run-time extensibility in conjunction with introspection is the main reason we describe Object Tcl as "dynamic". Tcl is designed so that much information on variables and procedures is accessible via Tcl commands and new variables and procedures can be created freely. The same is true of Object Tcl, further extended to allow attributes such as the class of an object to be discovered and changed as needed.

The combination of introspection and extensibility permits a dynamic style of programming, where programs interrogate and modify programs. We believe this is a natural style for an object extension to Tcl, since it matches the capabilities of Tcl itself. Conversely, the static organization style of C/C++ programs is mainly a consequence of requiring that many decisions be made at compile-time, a consideration that is not appropriate for Tcl.

Usage

Table 2 summarizes a comparison of language syntaxes. Syntax often receives scant attention in evaluating a language, but we believe it is an important consideration here because we are attempting to extend an existing language, rather than to create a new one. Accordingly, Object Tcl mirrors Tcl command names where appropriate. A `set` method, for example, is used to create instance variables, rather than introduce a different notation. Its syntax is also designed to be compatible with the usage of Tk widgets.

A second difference when compared to [incr Tcl] is that Object Tcl provides fewer ways of forming expressions, particularly for method invocation. This is because we attempted to include only a core set of facilities. For example, a special syntax for the automatic naming of objects (the #auto in [incr Tcl]) is not included, since it is easy to provide via a separate command[2].

3 Introducing Objects to Tcl

The Tool Command Language (Tcl) is apt for combining primitive commands into complete applications because of its simplicity and extensibility. In Tcl, all commands and values are represented as strings. It is easy to pass data between the Tcl interpreter and commands written in C code in an application: the C code need only be able to convert the internal representations to and from strings.

When data too complex to be readily converted to and from strings must be managed, the implementation may be pushed further into the command written in C. Using this approach, each object of complicated data is registered with the interpreter as a single command. Operations on an object are performed with subcommands by using the first argument to the command to specify the operation.

We refer to this strategy as the *object command* approach, since each command may be interpreted as an object and each subcommand as a message to that object. It is implicit in Tk and the majority of object extensions, and has proved effective for structuring programs as well as for managing complicated data. Objects complement and extend the inbuilt Tcl organizations of lists, arrays, procedures, and commands. They provide a natural way to group related operations and data, and lead to object-oriented programming techniques.

Objects

Object Tcl formalizes the object command approach, allowing object models to be realized without replicating the infrastructure that would otherwise be necessary. Objects support the same flexibility in combining Tcl and C as does the Tcl language. Methods may be implemented in C or Tcl as appropriate, leading to objects with a mixture of

```
Object astack

astack set things {}

astack proc put {thing} {
  $self instvar things
  set things [concat [list $thing] $things]
  return $thing
}

astack proc get {} {
  $self instvar things
  set top [lindex $things 0]
  set things [lrange $things 1 end]
  return $top
}

astack put toast    ==> toast
astack get          ==> toast
astack destroy      ==> {}
```

Figure 1: Making a stack object by specializing a generic object.

[1] But this will be addressed as part of a major new release.

[2] Perhaps called `gensym`.

implementation; however, only the Tcl interface is presented in this paper. CASTE also includes this capability, as will a forthcoming release of [incr Tcl].

Each object in Object Tcl is manipulated via a single command registered with the interpreter. Simple objects are created with the `Object` command, as shown in Figure 1. Here, the object `astack` is first created, then built up to implement a stack by adding one instance variable and two methods, and finally tested and destroyed.

To understand the example, first notice that the `astack` object is individually crafted, without requiring a class definition for all stack objects. Not only may an object's data differ from every other object, its code may differ as well. This is in contrast to Tk widgets and other object extensions. Yet it is a natural extension of the object command approach (since objects are synonymous with commands and different commands typically hold different behaviours) and should lead to a natural style of Tcl programming.

name	description
`class`	set class
`destroy`	destroy object
`proc`	define Tcl method
`set`	define Tcl variable
`instvar`	link to Tcl variable
`info procs`	list Tcl methods
`info args`	list formals of Tcl methods
`info body`	list body of Tcl methods
`info commands`	list Tcl/C methods
`info vars`	list Tcl variables
`info class`	get class

Table 3: Object Methods

Methods & Instance Variables

The `get` and `put` methods of the `astack` object are introduced with the `proc` method, which mirrors the behavior of the Tcl `proc` command. When later invoked, methods appear as procedures with the addition of three special variables (Table 4) that describe the method context. For example, the reference to `self` in the method bodies holds the name of the object for which the method is executing. It allows other calls to the the same object; `self` is the equivalent of `this` in C++ and [incr Tcl].

Similarly, the `things` instance variable is introduced with the `set` method, which mirrors the behavior of the Tcl `set` command. To access it during method execution, the `instvar` method is first used in the method bodies. It is analogous to the `export` method in CASTE, while no method is used in [incr Tcl]. We chose a declaration scheme because we felt it was important to distinguish instance variables from local variables, much as the `global` and `upvar` declarations are used prior to the variables they make accessible.

name	description	type
`self`	name of the object	*variable*
`proc`	name of the method	*variable*
`class`	class method defined	*variable*
`next`	next shadowed method	*method*

Table 4: Method Environment

All instance variables and methods are public in the sense of C++ and [incr Tcl] since Object Tcl does not support sealing. We felt that protection was not necessary for well-written programs and found it difficult to reconcile with our goals and Lisp heritage. Protection sometimes interferes with introspection and automatic method combination (discussed later) since the goal of hiding information affects the goals of self-description and flexible code reuse.

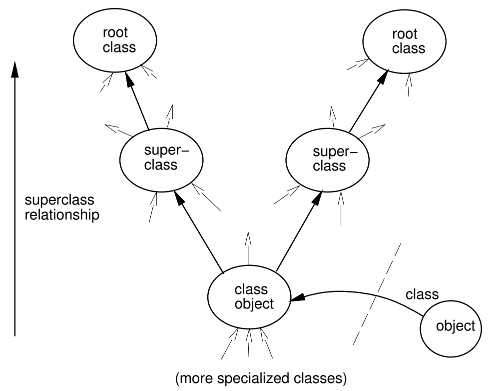

Figure 2: Superclasses and Method Dispatch

4 Inheritance

By themselves, objects allow related data and procedures to be grouped. Inheritance mechanisms strengthen this model by allowing the sharing of functionality between different objects, thus encouraging code reuse.

Inheritance in Object Tcl is based on *classes* and their *superclasses*. Each object has a class, and aAfter its own specializations, it inherits functionality from that class and that class's superclasses. Object Tcl supports multiple inheritance — each class may have many superclasses, and these may be organized in any directed acyclic graph. The class of an object is discovered with the `info class` method and changed with the `class` method. Similarly, the superclasses of a class are discovered with the `info superclass` method and changed with the `superclass` method.

A method is dispatched by searching first the object table of methods, then methods associated with the class, and then up through the superclasses (Figure 2). The first method located is invoked, so methods closer to the object shadow more distant ones. Method dispatch is thus conceptualized as a series of command lookups.

```
Class Safety

Safety proc create {acollection} {
  $self next $acollection
  $acollection set count 0
}

Safety instproc put {thing} {
  $self instvar count
  incr count
  $self next $thing
}

Safety instproc get {} {
  $self instvar count
  if {$count == 0} then { return {} }
  incr count -1
  $self next
}

Class Stack

Stack proc create {astack} {
  $self next $astack
  $astack set things {}
}

Stack instproc put {thing} {
  $self instvar things
  set things [concat [list $thing] $things]
  return $thing
}

Stack instproc get {} {
  $self instvar things
  set top [lindex $things 0]
  set things [lrange $things 1 end]
  return $top
}

Class SafeStack -superclass {Safety Stack}

SafeStack s
s put toast    ==> toast
s get          ==> toast
s get          ==> {}
s destroy      ==> {}
```

Figure 3: A class that guarantees safety in a collection and a class that implements a stack collection are mixed to form a safe stack.

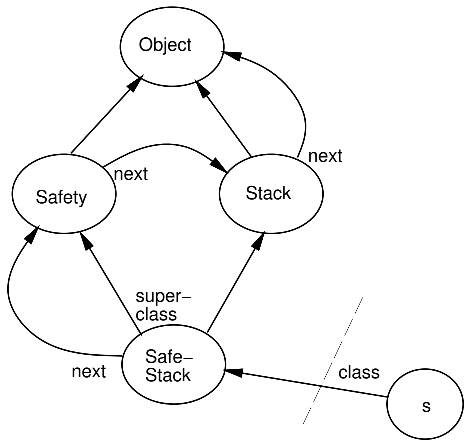

Figure 4: Class Hierarchy and Ordering for Method Dispatch of a SafeStack

The programmer does not explicitly determine the global order in which the superclasses are searched, however. As in CLOS, Object Tcl automatically determines the global order from the local superclass relations. It uses a CLOS-like topological sort algorithm in which direct superclasses of a class take precedence from indirect superclasses. For efficiency, the determined order is then cached since the algorithm's running time is proportional to the size of the superclass graph.

Figure 3 shows an example of multiple inheritance. Two classes that adhere to a protocol for managing collections are defined. The `Stack` class captures the behavior of the `astack` object that was described previously, allowing many stack objects to be instantiated. The class differs from the object in that the `get` and `put` methods are introduced with the `instproc` method, indicating that they are available for all stack instances, and there is a `create` method that initializes the internal list for every stack object that is created.

The `Safety` class adds safety checking to collections, keeping track of a count of items and catching attempts to withdraw from empty collections. These classes are combined using multiple inheritance to form a `SafeStack` class, which is then demonstrated by creating and testing the object `s`. Figure 4 shows the corresponding class hierarchy and method dispatch. Note that the `Safety` class can be dynamically combined with any type of collection that adheres to the protocol (for example, stack, queue, and set) but it is not required by any collection.

Method Combination

The example also demonstrates the use of method combination, where the total behavior of a method is split across classes, rather than simply being shadowed or not. This increases the abstraction capabilities of inheritance, and is found in most advanced object languages. Method combination is the mechanism used to order the constructors and destructors of C++ classes, though it is not recognized by that name.

In Object Tcl, method combination is achieved with the `next` method. When called within a method, it causes the next most shadowed implementation of the method to be invoked. Unlike C++ and [incr Tcl], the next method is not named explicitly, but is determined automatically according to the precedence ordering. This mechanism is similar to `around` methods in CASTE.

Method combination is used in the `get` and `put` methods of the `Safety` *mixin* class. First, the methods contribute their safety checks, and then the next method is called with `next`. For the `SafeStack` class, this ensures that the `get` and `put` methods of the `Stack` class are called, even though the `Stack` class has no direct relationship to the `Safety` class. The overall ordering followed in the combination is shown in Figure 4.

This style of programming permits the modular combination of methods in an intuitive manner: simply use `next`, and the system will ensure that the superclass relations hold. In contrast, both C++ and [incr Tcl] require that the next most shadowed method be named explicitly. We believe that this is a major weakness of their multiple inheritance model. It complicates code reuse because it requires programmers to include non-local information about the class hierarchy in method implementations, forcing them to deal with global class orderings. Further, our example would fail in C++ and [incr Tcl] because their inheritance model does not allow calls across the hierarchy other than to superclasses.

5 The Class System

In Object Tcl, classes are objects themselves, with the added ability to create and manage other objects. They support the superclass relation and serve as a repository for methods on behalf of their objects. This approach lets us apply the power of the object system to influence their behavior.

name	description
`create`	make new instance
`superclass`	set superclasses
`instproc`	define Tcl method for instances
`info instances`	list instances
`info superclass`	list superclasses
`info instprocs`	list Tcl methods for instances
`info instargs`	get formals of a Tcl method for instances
`info instbody`	get body of a Tcl method for instances
`info instcommands`	list Tcl/C methods for instances

Table 5: Class Methods

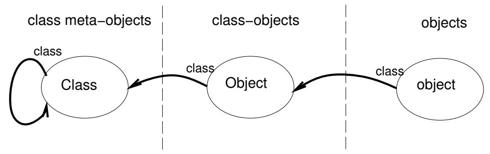

Figure 5: Class Relationships

Table 5 lists the methods for class objects. The basic class object that defines the behavior of objects, and from which other classes inherit by default, is `Object`. Because classes are simply special objects, new classes may be created and destroyed in the same manner as regular objects. Each uses a `create` method to manufacture new instances, the equivalent of a constructor in C++ and [incr Tcl]. The `create` method is also the default method, allowing the name of an instance in the method position to cause its creation. Thus, in the examples `SafeStack s` was interpreted as `SafeStack create s`, resulting in the creation of a new object.

Since classes are themselves objects, they are in turn managed by other classes. These managing objects form a third category of objects called *class meta-objects*. They are closed under the class relationship, and so are responsible for managing themselves. Figure 5 shows these relationships. The standard class meta-object is called `Class`. New class meta-objects may be manufactured too, and in conjunction with customizing `Object`, this allows the behavior of classes to be widely changed.

6 Implementation

Object Tcl is available as a Tcl/Tk extension implemented in C. Originally, it was embedded in the VuSystem toolkit [8], but we have recently developed a standalone version for evaluation. It is available via anonymous ftp from `ftp.tns.lcs.mit.edu` in `/pub/otcl`.

Our implementation goal was to form a compact extension by leveraging the Tcl implementation without changing its core. Methods are implemented in the same manner as Tcl procedures. They are stored in Tcl closure format in an object hash table by their name, and are evaluated by Tcl using its implementation of the `proc` command, its interpreter, and its calling conventions. Similarly, instance variables are brought into scope using the same mechanism as the `upvar` and `global` commands.

The result of our efforts is 2000 lines of C code that have little overlap with the Tcl implementation. This is a pleasingly small extension compared to other object extensions and the Tcl core itself. Further, directly using the Tcl implementation of procedures guarantees that Tcl commands that access information further up the call chain (like `uplevel` and `info`) will function correctly.

The performance cost of our approach also seems reasonable. A basic object method in Object Tcl takes approximately the same time to dispatch as a Tcl procedure with three arguments. A measurement of the time for these operations is 80 and 70 microseconds, respectively[3]. The corresponding measurement for [incr Tcl] is 60 microseconds. For Object Tcl, methods further up the inheritance chain and combined methods may take longer to dispatch, with a time dependent upon the superclass graph. We may revisit them later with the goal of making inheritance and combination as cheap as direct method invocation. However, we feel that absolute performance is less important than our emphasis on compactness and reuse of the Tcl implementation.

One interesting aspect of the implementation is the autoloading of Tcl code. Demand loading is important to reduce startup time. It is complicated for object-oriented programming because objects and methods may be referred to in more ways that previously. A class, for example, may refer to other classes as its superclasses, or methods may combine with each other along the inheritance chain. Object Tcl addresses this problem by allowing the user to fill a method with a Tcl code stub that forces its complete loading, and by attempting to load undefined classes as they are referenced. Demand loading at the method level as well as the class level permits a finer-grained distribution of startup time.

[3]DEC Alpha 3000/400, code compiled -O, and null bodies.

The implementation also features a C application programmer interface that parallels the Tcl level interface described so far. This allows objects and classes to be created and deleted from C, as well as methods whose bodies are implemented in C to be added to objects.

7 A Visual Programming Example

The VuSystem [8] multimedia application of Figure 6 demonstrates the use of Object Tcl. It shows a visual programming interface of a video motion segmentation program. The interface is used to manipulate the program as it runs [14].

The video program is written as an Object Tcl script (using an older embedded version of the language). Code that manipulates the video frames is written in C, while code that controls the user interface is written in Tcl. The program displays the video windows as part of its normal operation, but the graph structure that is drawn on the screen as a program representation is dynamically produced at the user's request.

The visual programming interface is constructed by using introspective features to query the program objects for their organization. Each video processing object is first asked what it is connected to in order to enumerate all objects, and then asked to draw a representation of itself on the screen.

Most objects draw themselves by inheriting a generic method implementation. They appear as the labelled rectangles linked to each other. Some objects, such as those representing a group of local objects or remotely executing objects, require a special representation. This is accomplished by shadowing the generic method with a specialized implementation. Thus, some of the labelled rectangles in Figure 6 represent a group of processing modules that appear to function as a single larger module.

In this manner, the inheritance is used to capture the common patterns while delegation permits special cases to be handled as they arise. Further,

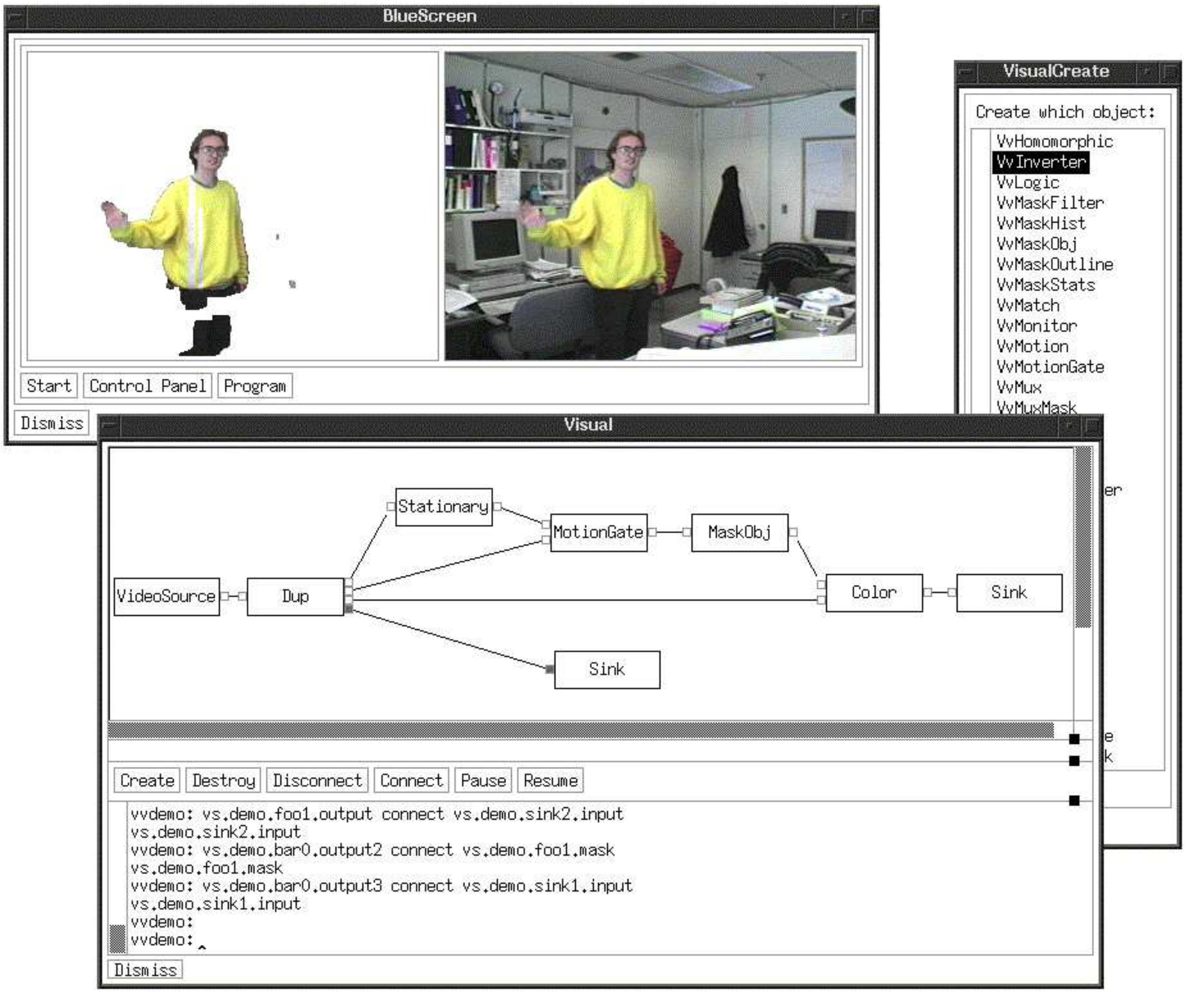

Figure 6: A Motion Segmentation program built with the VuSystem

the operation polymorphism of object-style naming means that the same code may be used to construct visual programming interfaces for a range of video programs, even if the interfaces appear to be quite different.

The dynamic extensibility of Object Tcl is also important to the user interface. Manipulations of the interface are translated into object commands that are evaluated directly, and in this way the running program is reconfigured. Video processing modules may be deleted and new ones created, or their pattern of processing changed. Further, new classes may be manufactured by grouping existing objects. This is analogous to the mega-widgets of [incr Tk] [10].

8 Future Work

We plan to extend Object Tcl in three directions to make it more configurable, efficient, and usable.

Meta-object protocols (MOPs) [6] let the language user incrementally modify and extend the language definition. The language then occupies a region of design space, rather than a point, and consequently is more widely applicable. For Object Tcl, we would like to exploit the meta-objects to make object traits such as creation, destruction, method dispatch, and precedence orderings accessible to the user.

Standard program browsing tools would make effective use of the introspective abilities of the system. Each object could be described in detail, and the entire class inheritance graph displayed. In conjunction with a MOP, these tools could monitor events such as object creation and destruction and aid in debugging.

Optimizations, such as caching method dispatches, may help to boost overall performance. The present implementation achieves relative efficiency through its tight integration with the Tcl implementation, but it could be tuned.

9 Conclusions

We feel that the design of Object Tcl has several advantages that differ from other object extensions. Object Tcl has grown out of the object command approach of manipulating compliated data through a single Tcl command. It retains this style of programming by allowing individual objects to be specialized, while supporting the construction of object models without replicating the infrastructure that would otherwise be necessary.

More than anything else, we attempted to extend Tcl with a core of object-oriented programming facilities rather than to import a separate object programming language to Tcl. Object Tcl largely maintains the runtime extensibility and introspective features of Tcl, as well as Tcl and Tk syntax plus compatibility with C. We feel this is a good match for an object extension.

We also sought an economy of design and implementation. Object Tcl provides few ways of forming expressions, yet it incorporates many of the features found in languages such as Flavors and CLOS. It supports multiple inheritance, method combination, and a metaobject-based class system.

By leveraging the Tcl implementation without changing the core, Object Tcl is both compact and tightly integrated with Tcl. At 2000 lines of C code, it is a small extension that is easily managed. By reusing the Tcl implementation with little overlap, a high degree of compatibility with Tcl is achieved.

Object Tcl is available for evaluation via anonymous ftp from `ftp.tns.lcs.mit.edu` in `/pub/otcl`.

References

[1] Micheal S. Braverman. CASTE: A class system for Tcl. In *Proceedings of Tcl/Tk Workshop 1993*, Berkeley, CA, May 1993.

[2] Wayne A. Christopher. Writing Object-oriented Tcl-based Systems using Objectify. In *Proceedings of Tcl/Tk Workshop 1993*, Berkeley, CA, May 1993.

[3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[4] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, 1983.

[5] Apple Computer Inc. *Dylan: an object-oriented dynamic language.* Apple Computer Inc., April 1992.

[6] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1992.

[7] C. J. Lindblad. Using Tcl to Control a Computer-Participative Multimedia Programming Environment. In *Proceedings of 1994 USENIX Symposium on Very High Level Languages*, Santa Fe, NM, October 1994.

[8] C. J. Lindblad, D. Wetherall, and D. Tennenhouse. The VuSystem: A Programming System for Visual Processing of Digital Video. In *Proceedings of ACM Multimedia 94*. ACM, October 1994.

[9] Michael J. McLennan. [incr Tcl] - Object-Oriented Programming in TCL. In *Proceedings of Tcl/Tk Workshop 1993*, Berkeley, CA, May 1993.

[10] Michael J. McLennan. [incr Tk] : Building Extensible Widgets with [incr Tcl]. In *Proceedings of Tcl/Tk Workshop 1994*, New Orleans, LA, June 1994.

[11] David A. Moon. Object-Oriented Programming with Flavors. In *Proceedings of ACM Conference on Object-Oriented Systems, Languages, and Applications (OOPSLA) 1986.* ACM, September 1986.

[12] John Ousterhout. *An Introduction to Tcl and Tk.* Addison Wesley, 1994.

[13] D. Ungar and R. B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, 1987.

[14] David Wetherall. An Interactive Programming system for Media Computation. Technical Report MIT/LCS/TR-640, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1994.