CASTE: A class system for Tcl *

Michael S. Braverman

571 Evans Hall
University of California
Berkeley, CA 94720
braver@cs.berkeley.edu

Abstract

This paper introduces CASTE (Classes, A Sensible Tcl Extension), a class system for Tcl that, in its simplest form, enables the creation and manipulation of structured objects, but more generally provides an entire object-oriented class mechanism with the inheritance of slots and methods. Methods may be defined either in Tcl or C. CASTE is largely modeled after the behavior of "standard-class" classes and "standard-object" objects in CLOS, the Common Lisp Object System [1]. However, unlike CLOS, in which methods are invoked via the application of generic functions, methods in CASTE are invoked using a message passing paradigm that is more consistent with the syntax and semantics already used by Tcl/Tk for interactions with its built-in classes. CASTE supports the development of class libraries, with class definitions "auto-loaded" on an as-needed basis. The system is both efficient and flexible, and experience with it has demonstrated its ability to promote software re-use in Tcl.

1 Introduction

Tcl currently provides limited facilities for managing structured data objects. While arrays and lists can be used to group heterogeneous data, using them to simulate structured objects typically requires the manipulation of global variables throughout a program, making such programs unnecessarily complex and error-prone. This paper introduces CASTE (Classes, A Sensible Tcl Extension), a class system for Tcl that, in its simplest form, enables the creation and manipulation of structured objects, but more generally provides an entire object-oriented class mechanism with the inheritance of slots and methods, including *before*, *after*, and *around* daemon methods. The inheritance mechanism greatly enhances the possibilities for software re-use in Tcl. In particular, it facilitates the development of families of user interfaces that share common properties; a programmer need only write methods specific to the application at hand without having to construct the entire interface from scratch.

CASTE is largely modeled after the behavior of "standard-class" classes and "standard-object" objects in CLOS, the Common Lisp Object System [1]. CASTE follows CLOS's "standard" policies for resolving multiple inheritance in the class hierarchy, for determining the default initialization of slot values, and for ordering method invocations when more than one method may apply to a given object. However, unlike CLOS, in which methods are invoked via the application of generic functions, methods in CASTE are invoked using a message passing paradigm that is more consistent with the syntax and semantics already used by Tcl/Tk for interactions with its built-in classes (canvases, menus, etc. . .).

CASTE is implemented in C and has been used to build several projects. One of these projects, a simple proof-tree browser and editor, demonstrates the utility of inheritance. For example, proof-tree nodes can be drawn as rectangles, circles, or ovals. When a node is selected, control points are displayed on the corners of its bounding box. Rather than writing separate code for all three shapes of objects, an abstract class corresponding to objects with rectangular bounding boxes was created and methods for the display and manipulation of control points for that class were defined. A specific class, inheriting from the abstract class, was defined for each node shape. Each of these shape-specific classes has methods tailored

*This material is based in part upon work supported by the National Science Foundation under Infrastructure Grant No. CDA-8722788.

```
> defclass node {
        x
        y
        sheet
  }
node
```
(a)

```
> node
node1
> node1 set y 4
4
> node1 set y
4
> node1 y
4
```
(b)

```
> node? node1
1
> node1 destroy
> node1
invalid command ...
> node? node1
0
```
(c)

```
> defmethod {node mult_y} {factor} {
        puts "Hi, I'm $self."
        return [expr $factor*[$self y]]
  }
> node1 mult_y 10
Hi, I'm node1.   ←printed
40               ←value returned
```
(d)

Figure 1: Class and Method Definition

to its type of outline, but the common operations on control points are handled automatically via inheritance. Three other significant applications have also been developed using CASTE: a 2D structured drawing program, an interactive 3D modeling program, and a hypertext system[2]. All three of these applications share a common interface that was defined at an abstract level using the class system. Application-specific characteristics of each interface were specified by simply writing application-specific methods.

2 System Overview

There is insufficient space in a paper of this length to describe, comprehensively, all of CASTE's features. Instead we will illustrate, in terms of three examples, the system's principal characteristics. In the figures that follow, expressions that a user might send to a Tcl interpreter are prefixed with a **>** and appear in a **bold** font; values that are returned or are printed appear in a sans-serif font. It is assumed that the reader has a basic understanding of object-oriented programming and, of course, programming in Tcl.

2.1 Class and method definition

CASTE may be used merely to define and manipulate structured objects. Figure 1a illustrates the use of **defclass** to define a class named **node** whose objects are to contain three slots named **x**, **y**, and **sheet**. In addition to defining the slots of the class, **defclass** also creates a Tcl command named **node** that may be invoked to create an instance of the **node** class and a command named **node?** that will test if an object is an instance, or sub-instance, of that class. The value returned from the **node** command will be a string of the form **node<N>** (**<N>** some integer), which will serve as a handle for the object.

By default, the **node** class will have a number of methods defined for slot access. The simplest of these is **set**. Like the Tcl **set** command, if no value is supplied, then the current value of the indicated slot is returned. As a convenience, slot reading methods, with the same names as the corresponding slots, are automatically defined. The creation of a **node** object and the use of these methods are illustrated in Figure 1b.

Normally an object will exist for the lifetime of the Tcl interpreter in which it was created. However, if an object is no longer needed, its storage may be reclaimed by "sending" it the **destroy** method, as in Figure 1c.

Custom methods for a class are defined with the **defmethod** command. Figure 1d demonstrates the definition and use of a method named **mult_y** on the **node** class; this method takes a single argument named **factor**. When a method is invoked, the local variable **self** is automatically bound to the handle of the object to which the method was passed.

2.2 Inheritance, initialization, and daemons

Figure 2a shows how a more specific class named **s_node**, with two additional slots named **shape** and **contents**, could be defined to inherit from the **node** class. This class definition differs from that in Figure 1a in three important ways. First, rather than having a class name appear after the **defclass** command, a list of class names appears. The first of these is the name of the class that is being defined and the subsequent names, called the direct super-classes (there is only one in this example), are those classes from which the class being defined should *directly* inherit. The defined class will *indirectly* inherit from all those classes from which the direct super-classes inherit. The order of the class names is used to compute the global inheritance order of the class hierarchy. Gen-

```
> defclass {s_node node} {
      {shape init "abstract"}
      contents
   } –slot_args
s_node
```

(a)

```
> s_node –shape round
s_node1
> s_node1 shape
round
> s_node
s_node2
> s_node2 shape
abstract
> s_node2 set y 3
3
> s_node2 y
3
```

(b)

```
> defmethod {s_node mult_y before} {factor} {
      $self export y
      puts "Running Before"
      incr y 5
  }
> defmethod {node mult_y after} {factor} {
      puts "Running After"
  }
> s_node2 mult_y 10
Running Before        ←printed
Hi, I'm s_node2.      ←printed
Running After         ←printed
80                    ←value returned
```

(c)

Figure 2: Simple Inheritance, Initialization, and Method Daemons

erally speaking, classes are listed in left to right order from most specific to least specific. Thus, methods and slots of **s_node** will shadow those of **node** and any classes from which **node** might inherit.

The second and third significant differences in the **defclass** statement are related: the presence of the **–slot_args** option at the end of the statement and the manner in which the **shape** slot is defined. Rather than just having its name appear, there is a list specifying the optional **init** attribute of the **shape** slot. The **init** option specifies a default expression that may be evaluated (in the global environment) at object creation time to initialize the slot. The **–slot_args** option arranges for the **node** object creation command to allow initialization arguments for the slots specified in the **defclass** statement. These arguments will have the same name as the corresponding slot, except with a "–" prepended to the slot name. Thus, the arguments will look like the switches that are used to initialize Tcl/Tk widgets. If an initialization argument is used when creating an object, its value will take precedence over that specified by any **init** option for the same slot; in this case, the **init** option's expression will not be evaluated since it may potentially have side-effects.

Figure 1b should clarify this discussion. First **s_node1** is created using the **–shape** argument with the value **round**; hence, its **shape** slot has the value **round**. Next, **s_node2** is created without the **–shape** argument, and so the **init** option is used to initialize the slot, giving it the value **abstract**. Note that the **x**, **y**, **sheet**, and **contents** slots are unbound for both **s_node**s because no initializations were specified for them. These slots can be set and read in the usual manner, as shown in the figure, after the object is created. Mechanisms exist in CASTE for the inheritance of slot initializations, for creating initialization arguments for only a subset of a class's slots, and for generating initialization arguments that are named differently than the slots themselves, but space does not permit their description here. CASTE also allows the definition of initialization arguments that may be used to do more than just set an object's slots. This feature will be discussed in section 2.3.

Figure 2c illustrates the definition of *before* and *after* daemon methods. A method defined in the usual manner, as in Figure 1d, is called a *primary* method. When an object is passed the name of a method, the most specific primary method with that name is invoked.[1] However, when the primary method is invoked *all* the associated *before* and *after* methods are run before and after the primary method, respectively, in an order determined by the inheritance hierarchy. Whether or not *before* or *after* methods exist, the value returned for a particular method invocation is that which is returned from the primary method. Thus, in Figure 2c, a *before* method is defined for the **mult_y** method (earlier defined in Figure 1d) on the **s_node** class. An *after* method is also defined for **mult_y**, but it is specified on the superclass **node**. Thus, **node** objects will "see" only the *after* method, whereas both the *before* and *after* methods will be run for **s_node** objects. A third type of daemon method, the *around* method (not demonstrated here), can be defined to modify the value returned by the primary method or to selectively keep the primary method and

[1] The primary method, if it wants, may use a command named **call_next_method** to invoke the method that it is immediately shadowing in the class/method hierarchy. The predicate command **exists_next_method?** may be used to determine if there is a shadowed method to call.

(a)	(b)

<pre>**> defclass nodeset {**
    {nodes init {}}
  } –slot_args
    {init_args –xpos –ypos}
    {arg_defaults –xpos 50 –ypos 70}
nodeset
 	
> defmethod {nodeset initialize after} {inits} {
    parse_inits_into {–xpos xp –ypos yp} $inits
    foreach node [$self nodes] {
      $node set x $xp y $yp
    }
  }
> defmethod {nodeset destroy before} {} {
    foreach node [$self nodes] {$node destroy}
  }</pre>
	<pre>**> nodeset –nodes {s_node1 s_node2} –xpos 100**
	nodeset1
	> s_node1 x
	100
	> s_node1 y
	70
	> nodeset1 destroy
	> nodeset1
	invalid command name "nodeset1"
	> s_node1
	invalid command name "s_node1"
	> s_node2
	invalid command name "s_node2"</pre>

Figure 3: Customizing Initialization and Destruction

its associated *before* and *after* methods from being invoked.

The definition of the **mult_y** *before* method also demonstrates the use of the **export** method. This method allows a procedure to map between an object's slots and local variables in the current scope. This mapping ability allows a program to have extended direct access to a slot without the overhead associated with the repeated invocation of slot accessor methods. Hence, given that the **y** slot of **s_node2** was set to **3** (in Figure 2b), when the **mult_y** method is invoked with the argument **10**, the *before* method is first run, incrementing the **y** slot of **s_node2** to **8**, and, thus, the value returned from the primary method, following the execution of the *after* method, is **80**.

2.3 Custom initialization and destruction

When a CASTE object is created, its slots are first allocated and then the object is initialized using a pre-defined method named **initialize**. Normally, as described earlier, this method initializes slots according to the initialization arguments passed to the object creation command and, in their absence, uses the **init** option in the individual slot definitions to determine the initial values of the slots. Later, if a **destroy** message is sent to an object, the object and its slots are simply deallocated. Sometimes, however, this default behavior is insufficient. For instance, it is often useful to pass initialization arguments that do not simply set the value of a slot of the object being created. Similarly, certain cleanup operations may have to be run before an object is destroyed. The default behavior of object initialization and destruction may be modified through the definition of daemon methods.

Consider the **nodeset** class defined in Figure 3a. Objects of this class will contain a single slot, **nodes**, whose initial value defaults to the empty list. The class definition uses the **–slot_args** option to define an initialization argument for the slot, but it also declares two additional initialization arguments, **–xpos** and **–ypos**, using the **defclass** option named **init_args**. Further, the class definition uses another **defclass** option, **arg_defaults**, to specify default expressions to associate with **–xpos** and **–ypos** should these arguments not be passed at object creation time.

Following the class definition, an *after* method for **initialize** is defined on the **nodeset** class. When the **nodeset** object creation command is invoked, the single **inits** argument will be bound to the list of initialization arguments (if any) that are passed. The **parse_inits_into** command parses out the requested initialization arguments and binds local variables to their associated values. If a requested argument is not passed, then **parse_inits_into** will consult the default values defined by **arg_defaults** in the class definition.

Figure 3b illustrates the modified initialization behavior for **nodeset** objects. First, the object creation command is called with the two initialization arguments **–nodes** and **–xpos**, creating the object **nodeset1**. Since the initialization method in Figure 3a is an *after* method, the default initialization method will be run before it is executed, and so the **nodes** slot of **nodeset1** will already be initialized in the usual manner before it starts. The *after* method then loops through the nodes in **nodes** setting their **x** and **y** slots according to the values associated with **–xpos** and **–ypos**. Since **–ypos** was not passed to the object creation command, its value is taken from that specified in the **arg_defaults** option. Hence, we find, in Figure 3b, that the **x** and **y** slots of **s_node1** (and **s_node2**, for that matter) have the values **100** and **70**, respectively, after **nodeset1** is initialized.

Figure 3b also demonstrates the effect of the **nodeset destroy** *before* daemon method. After **nodeset1** is destroyed, we find that **s_node1** and **s_node2** were also destroyed per the instructions in the *before* daemon method.

3 Efficiency and Flexibility

CASTE was designed not only to simplify programming in Tcl, but also to be efficient. At the time that a class or a new method for a class is defined, the inheritance hierarchy for the class and the concomitant ordering of its methods, along with that of their *before*, *after*, and *around* daemons, are cached. Hence, the time needed for object creation and method invocation is independent of the size and complexity of the class hierarchy. Measurements on the current CASTE implementation indicate that slot access via methods takes roughly one and a half to twice the time of a direct **set** command on a regular Tcl variable. However, this overhead can be eliminated using methods such as **export** (described in section 2.2). Greater efficiency may also be achieved by defining methods directly in C. CASTE provides a C function, **defCmethod**, that allows the programmer to register a standard Tcl/C call-back function as a method for use in a particular interpreter.

CASTE is very flexible, allowing methods and classes to be defined in any order. The only requirement is that all the super-classes of a class be defined before attempting to create an object of that class. All classes implicitly inherit from a "top" class named **T**. Default methods for all classes are specified in CASTE by defining methods on the **T** class. As a result, the user may replace, or modify the behavior (via method daemons) of any or all the default methods supplied, as with the example in section 2.3. Error events (e.g., writing to undefined slots or reading from unbound slots) are also handled within the class system via methods; hence, class specific handling of errors is possible by merely writing custom error handler methods to shadow the system defaults. A complete list of the default methods, grouped roughly according to function, is provided in Table 1.

4 Tcl/Tk Compatibility

As with procedure definitions in Tcl, classes and methods may be defined and later redefined in the same interpreter. Classes dependent on those redefined are automatically updated by the system.

A Tcl program may use the **slot_ref** method to gain direct access to an object's slot via a global variable. This feature allows Tcl to trace operations on slots. By extension, it also allows Tk widgets, such as **checkbutton**s, to effectively use a slot as a referent in a **–variable** option.

For complete compatibility, CASTE will allow the programmer to provide a desired handle name when creating an object. As a result, object creation can be made to look exactly like that for Tk widgets. Moreover, the **cinfo** method can provide the default initialization for any given slot of an object. Together, the **cinfo** and **set** methods can be used to define a **config** method, providing access to slots with a syntax completely consistent with that used with Tk widgets.

CASTE is implemented so as to require no modifications to the standard Tcl distribution code. A Tcl interpreter may be extended to use CASTE by simply calling, from C, the single function **Init_Caste** and passing a pointer the interpreter's **Tcl_Interp** structure. A slight change to Tcl's **auto_mkindex** procedure allows CASTE to provide class libraries that are "auto-loaded" on an as-needed basis.

5 Conclusion and Future Directions

This paper has presented an overview of CASTE, a comprehensive class system for Tcl. The system is both powerful and efficient and its syntax and semantics integrate well with those already existing in Tcl/Tk.

Future versions of CASTE will allow "class slots", wherein all instances of a given class will be able to share one or more of their slots. This will allow CASTE to subsume much of the functionality needed by those who desire to have a "module" mechanism, with shared private variables, in Tcl.

CASTE currently allows classes to be redefined, but it does not update any existing objects that are in-

Method and Arguments	Brief Description
initialize inits	Initialize object according to inits list.
destroy	Destroy object.
set [slot value]* slot [value]	Set the slots to the corresponding values, returning the last value or value of the last slot given.
append slot value [value]*	Perform Tcl append operation on slot with values.
lappend slot value [value]*	Perform Tcl lappend operation on slot with values.
unset slot [slot]*	Unset the indicated slots.
slot_bound? slot	Return 1 or 0 depending if the slot is currently set.
export slot [slot]*	map slots of object to local variables of the same name
export_to slot var [slot var]*	map slots of object to the corresponding indicated local variables
slot_ref slot	Return global variable that refers to slot.
print	Print the values of all the object's slots.
cinfo request [arg]*	Return information concerning object's class.
no_applicable_method umethod arglist	Produce Tcl error: undefined method umethod was applied with the indicated argument list.
no_next_method method arglist	Produce Tcl error: call_next_method was called and no method exists higher in the class hierarchy.
slot_unbound slot	Produce Tcl error: attempted to read unbound slot.
slot_missing slot op [arg]	Produce Tcl error: attempted to perform operation [with the indicated arg] on an undefined slot.

Table 1: Default Methods

stances of the redefined class; only a warning is printed if an outdated object is used. In the future, objects will be automatically updated to the new class definition.

Finally, a default **trace** method, that is aware of CASTE's inner workings, needs to be defined so that methods, rather than procedures, may be called when accessing the slots of an object.

Acknowledgements

I would like to thank Narciso Jaramillo for using CASTE to build a number of interesting applications. His use of CASTE aided in the debugging process and helped to point out deficiencies in the original design. I would further like to thank him and Brian Smith for useful suggestions concerning features that CASTE should include.

References

[1] Steele, Guy L., Jr., *Common Lisp: The Language.* 2nd ed. Digital Press (1990), 770–864.

[2] Jaramillo, Narciso, personal communication.