`[incr Tcl]` – Object-Oriented Programming in Tcl

Michael J. McLennan

AT&T Bell Laboratories
1247 S. Cedar Crest Blvd.
Allentown, PA 18103
michael.mclennan@att.com

ABSTRACT

Tcl was designed to be an interactive command language, not a structured programming environment. Small projects flourish in Tcl, thereby enticing programmers to build larger systems. But large systems quickly become unwieldy as the growing body of Tcl procedures becomes more and more difficult to maintain or extend. What is needed is a way of better organizing procedures and data into packages with well-defined interfaces. **`[incr Tcl]`** is a set of Tcl extensions designed to provide such support. In this paper, I describe these extensions and show how they can be used to manage complexity.

INTRODUCTION

`[incr Tcl]` provides object-oriented extensions to Tcl, much as C++ provides object-oriented extensions to C. The emphasis of this work, however, is not to create a whiz-bang object-oriented programming environment. Rather, it is to support more structured programming practices in Tcl without changing the flavor of the language. More than anything else, **`[incr Tcl]`** provides a means of encapsulating related procedures together with their shared data in a local namespace that is hidden from the outside world. It encourages better programming by promoting the object-oriented "library" mindset. It also allows for code re-use through inheritance.

The fundamental construct in **`[incr Tcl]`** is the class definition. Each class acts as a template for actual objects that can be created. Each object contains a unique bundle of data and provides a number of procedures or "methods" that operate on the data.

Class definitions are expressed with the following syntax:

```
itcl_class className {
    inherit baseClass ?baseClass...?
    constructor args body
    destructor body

    method name args body
    proc name args body

    public varName ?init? ?config?
    protected varName ?init?
    common varName ?init?
}
```

As an example, suppose we create a class to represent a toaster. Before we begin writing the class definition, we must decide what features of the toaster we will need to use in our programming. Many toasters, for example, have a heat setting that allows the user to adjust the darkness of the toast. Toasters also have a tray that catches crumbs from the toast. This tray must be emptied from time to time, or it could become so full that the crumbs would touch the heating elements and catch fire. Notice that both of these toaster features are related: If the heat control is set toward the dark side, the toast will tend to leave more crumbs in the tray, and the tray will have to be emptied more often.

Our hypothetical toaster could be encapsulated in the class definition shown in Figure 1. Each toaster has it own heat setting "**`heat`**" and crumb count "**`crumbs`**." The heat setting is implemented as a public variable so that it can be configured by the class consumer (read "programmer using a toaster"). The crumb count, on the other hand, is kept as a protected variable, and is modified indirectly by methods which operate on each toaster individually. In addition to these variables which were explicitly defined, all objects have a built-in protected "**`this`**" variable which is initialized with the object name.

Most of the useful work associated with a toaster is accomplished through its methods. Methods provide a clean interface to an object, providing ways to manipulate its data without giving direct access to the data itself. In our **Toaster** class definition, the "**toast**" method performs the action of toasting bread, adding crumbs to the crumb tray. We use a simple formula that multiplies the heat setting by the number of bread slices to compute the number of crumbs generated by each toasting. The "**clean**" method clears the crumb tray by resetting the crumb count to zero. The "**check**" method reports the current level of crumbs in the tray. Finally, the "**config**" method is used to change public variables such as the heat setting. The "**constructor**" and "**destructor**" are special methods that are invoked automatically whenever an object is created or destroyed. They are used to initialize and clean up after

```
itcl_class Toaster {
    constructor {config} {}
    destructor {
        if {$crumbs > 0} {
            puts stdout "$crumbs crumbs ... what a mess!"
        }
    }
    method config {config} {}

    method toast {nslices} {
        if {$nslices < 1 || $nslices > 2} {
            error "bad number of slices: should be 1 or 2"
        }
        set crumbs [expr $crumbs+$heat*$nslices]
        if {$crumbs >= $maxcrumbs} {
            puts stdout "== FIRE! FIRE! =="
            set crumbs $maxcrumbs
        }
        return [check]
    }

    method clean {} {
        set crumbs 0
        return [check]
    }

    method check {} {
        set level [expr $crumbs*100.0/$maxcrumbs]
        return [format "crumb tray: %.0f%% full" $level]
    }

    proc resize {newsize} {
        set maxcrumbs $newsize
    }

    public heat 3 {
        if {$heat < 1 || $heat > 5} {
            error "invalid setting $heat: should be 1-5"
        }
    }
    protected crumbs 0
    common maxcrumbs 40
}
```

FIGURE 1 – Definition for a class of **Toaster** objects.

each object. In our **Toaster** class, the constructor does nothing aside from allowing us to configure public variables (more on this later). But if anyone is foolish enough to destroy a toaster without cleaning it first, the destructor prints a message reporting the number of crumbs that were spilled.

We could create, use, and destroy an instance of class **Toaster** with the following code:

```
wish: Toaster Serial001 -heat 2
Serial001
wish: Serial001 toast 2
crumb tray: 10% full
wish: Serial001 toast 1
crumb tray: 15% full
wish: Serial001 clean
crumb tray: 0% full
wish: Serial001 config -heat 5
wish: Serial001 toast 2
crumb tray: 25% full
wish: Serial001 delete
10 crumbs ... what a mess!
```

The first line in this example follows the syntax:

className objName ?args...?

It creates a new **Toaster** object with the name "**Serial001**" and passes the remaining arguments ("**-heat 2**") to the constructor. The new object is then manipulated by invoking its methods, using the syntax:

objName methodName ?args...?

In this example, we toast a few slices of bread, clean the toaster, change the heat setting, toast a few more slices, and delete the toaster. Since we deleted the toaster without cleaning it, the destructor reports the number of crumbs spilled from the crumb tray. Note that the "**delete**" method was not included in our class definition, but is a built-in method that is automatically defined for all classes. Other built-in methods are summarized in the **[incr Tcl]** manual page.

All class methods—constructor included—follow the same conventions for matching actual argument values to the names in the formal argument list. These conventions are identical to the ones used for ordinary Tcl **proc**'s, with one addition: If the last formal argument is named "**config**," all remaining arguments are treated as "**-***name value*" assignments, where *name* refers to a public variable. If optional *config* code was included in the public variable definition, this code is automatically executed after each assignment is made. If this code returns an error, the variable is reset to its previous value and the method is aborted, returning the error. In general, this *config* code can be used to recognize changes to public variables and update the object accordingly. In our example, the *config* code for the heat setting merely checks to see if the new setting is valid, and returns an error if it is not. Note that any method can use "**config**" as a formal argument, although the constructor is a natural place for it to appear.

From time to time, it is useful to have data that is shared among all objects in a class. For example, the size of the crumb tray is a design variable that is common among all of our toasters. This parameter is represented by the common variable "**maxcrumbs**" in the class definition. Now suppose that people begin using the toasters, and they complain that the crumb tray is too small. We could change the size of the crumb trays for all toasters instantly, just by modifying the "**maxcrumbs**" variable. But common variables are like protected variables in that they are only accessible within the scope of the class. We need to define a procedure that is local to the class—and therefore has access to common variables—that can be invoked without reference to a specific object. In C++, this type of procedure is referred to as a "static" member function. In **[incr Tcl]**, it is simply called a class "**proc**." Note that the class definition in Figure 1 contains a "**resize**" proc that modifies the size of the crumb tray. This proc could be invoked as follows:

```
wish: Toaster :: resize 100
```

Notice that this command requires spaces around the "**::**" delimiter. Thus, the same command "**Toaster**" used earlier to create a new object is also used to invoke procs in class **Toaster**. The extra spaces around the "**::**" delimiter are necessary from a Tcl language standpoint, but are unfortunate from the programming standpoint. As we will see shortly, the "**::**" delimiter appears in other contexts within **[incr Tcl]**, and in these contexts, spaces around the delimiter are forbidden!

This toaster example, foolish as it may seem, illustrates a simple solution for many of the problems facing Tcl programmers. Many programs have clumps of related procedures that share the same data. Typically, this sharing is accomplished using global variables, and related procedures are kept together in source files. The object-oriented paradigm helps to organize this kind of programming better by explicitly grouping procedures and associated data into a class definition. It keeps private data hidden from the global namespace, and provides well-defined access mechanisms for public variables.

INHERITANCE

Object-oriented programming also provides a mechanism for code re-use through inheritance. Suppose we want to design a slightly better toaster that automatically cleans itself whenever the crumb tray gets full. This new toaster would be largely identical to the original model, with the addition of a monitoring device that automatically invokes the "**clean**" method as needed. Using inheritance, we might implement this new toaster as shown in Figure 2.

The first statement ("**inherit Toaster**") effectively copies the definition of class **Toaster**, causing class **SmartToaster** to inherit all public, protected, and common variables, as well as all methods and procs. The next statement redefines the "**toast**" method to clean the crumb tray as needed and then perform the usual toasting operation. Since the "**toast**" method has been redefined, the previous "**toast**" method must be referenced using its full name "**Toaster::toast**."

Any inherited member can always be referenced using the syntax "*class*::*member*." However, the short name (without the "*class*::" prefix) refers to the first member found by searching the class hierarchy from the current class scope upward through all base classes; base classes are searched in the order that they are defined in the "**inherit**" statement for each class. Within the scope of class **Toaster**, therefore, the name "**toast**" refers to "**Toaster::toast**," but within the scope of **SmartToaster**, "**toast**" refers to "**SmartToaster::toast**." Within the scope of class **SmartToaster**, "**clean**" refers to "**Toaster::clean**," since this method was not redefined within class **SmartToaster**.

If a command is not recognized within the scope of a class or its base classes, it is passed to the main interpreter for evaluation. Commands can also be sent explicitly to the main interpreter using the "::*command*" syntax. Note that since command resolution includes the main interpreter, widget commands can be invoked transparently within a class, but have their effects at the global scope.

Object-oriented programming buffs may recognize that to have a command act as a "virtual" method within some class scope, the method name could be prefaced by the object name: *e.g.*, "**$this toast**." This forces the command to be executed in the main

```
itcl_class SmartToaster {
    inherit Toaster

    method toast {nslices} {
        if {$crumbs >= [expr $maxcrumbs-10]} {
            clean
        }
        return [Toaster::toast $nslices]
    }
}
```

FIGURE 2 – Definition for a class of **SmartToaster** objects.

interpreter—where the object name is recognized as a command—and resets the scope to the most specific class. Virtual methods are an extremely important part of object-oriented programming. They allow the programmer to create a default behavior at the highest level of abstraction, and to specialize this behavior in derived classes. To support this functionality, a special "**virtual**" command is available within the scope of each class. This command changes the scope of execution to be the most specific class, and invokes the remainder of its arguments as a command in that scope.

We use a **SmartToaster** in much the same way that we used a **Toaster**:

```
wish: SmartToaster Serial002 -heat 5
Serial002
wish: Serial002 toast 2
crumb tray: 25% full
wish: Serial002 toast 2
crumb tray: 50% full
wish: Serial002 toast 2
crumb tray: 75% full
wish: Serial002 toast 2
crumb tray: 25% full
```

However, the **SmartToaster** cleans itself automatically—notice the sudden drop in the crumb tray during the last toasting operation.

Without inheritance, much of the code needed to implement the basic toaster would have to be duplicated for each new toaster class. This makes a program larger and more difficult to maintain. If a bug were found in the original code, the fix would have to be propagated to all of the clones. Another alternative is to develop one grand "supertoaster" that can account for all possible toaster behaviors; but this leads to complex code that is also difficult to maintain. Inheritance provides a cleaner solution, allowing for code to be shared among classes.

This "smart toaster" example shows how inheritance can be used to specialize the behavior of a class of objects. Inheritance can also be used to add the same common behavior to many different classes of objects.

For example, suppose our toaster business is so successful that we begin making other appliances: **Blender**'s, **Mixer**'s, **Juicer**'s, *etc.* Suppose we realize that all of these appliances require electricity to operate. We could create a class **Outlet** to represent an electrical outlet supplying power, but then each of our appliance classes would need an extra bit of code to deal with the outlet. Much of this code—plugging into an outlet and consuming power—would be the same across all appliances. Indeed, we can arrange for classes to share this code by having them inherit from a common base class **Appliance**.

Example code for an electrical **Outlet** is presented in Figure 3. Each outlet has a public variable "**owner**" which contains the billing address for the owner, and a protected variable "**usage**" which keeps track of power usage. Appliances request power from an outlet by invoking the "**use**" method. The "**sendBill**" method computes the charges for current power usage and sends a bill to the owner of the outlet via electronic mail. Note that the "**rate**" used for computing charges is a common variable, so any rate change will affect all outlets simultaneously; the class proc "**rate**" is used to configure this variable. At the end of each month, the power company responsible for the outlets can invoke the class proc:

```
wish: Outlet :: bill
```

to automatically send bills for all **Outlet**'s in existence.

During the billing process, the "**bill**" proc needs a list of all **Outlet**'s in existence. The **Outlet** class could itself keep track of this by having each **Outlet** add/remove itself from a common list in the constructor/destructor. But since **[incr Tcl]** keeps track of such information automatically, it is easier to simply query the information using the "**itcl_info**" command:

```
wish: Outlet x -owner mmc@gewurtz
x
wish: Outlet y -owner mmc@gewurtz
y
wish: itcl_info objects -class Outlet
x y
```

Such query commands are normally invoked at the global scope, as shown above. Because of the way that **[incr Tcl]** handles scope, however, they can also be invoked transparently

```
itcl_class Outlet {
    constructor {config} {}
    method config {config} {}

    destructor {
        if {$usage > 0} bill
    }

    method use {power} {
        set usage [expr $usage+$power]
    }

    method sendBill {} {
        if {[catch “open /tmp/bill w” fout] != 0} {
            error “cannot create bill in /tmp”
        } else {
            set amount [format “$%.2f” [expr $usage*$rate]]
            puts $fout “----------------------------------------”
            puts $fout “/////////// MEGA-POWER, INC. ///////////”
            puts $fout “----------------------------------------”
            puts $fout “   Customer: $owner”
            puts $fout “     Outlet: $this”
            puts $fout “      Usage: $usage kilowatt-hours”
            puts $fout “      Usage: $usage kilowatt-hours”
            puts $fout “                                        “
            puts $fout “ Amount Due: $amount”
            puts $fout “----------------------------------------”
            close $fout
            exec mail $owner < /tmp/bill
            set usage 0
        }
    }

    proc bill {{customer *}} {
        foreach outlet [itcl_info objects -class Outlet] {
            set owner [$outlet info public owner -value]
            if {[string match $customer $owner]} {
                $outlet sendBill
            }
        }
    }

    proc rate {{newval “”}} {
        if {$newval == “”} {
            return $rate
        }
        set rate $newval
    }

    public owner {}
    protected usage 0

    common rate 0.05
}
```

FIGURE 3 – Definition for a class of electrical **`Outlet`** objects.

within a specific class scope. When this command is invoked in the "**bill**" proc, for instance, it is not recognized as a built-in Tcl command nor as a method or proc of class **Outlet**; therefore, it is automatically sent out to the global scope for evaluation, and it returns a list of all **Outlet**'s in existence.

In order to use these electrical outlets, our appliances must know which outlet they are connected to, and they must use power from that outlet during their operation. We can abstract this behavior into the base class **Appliance**, shown in Figure 4(a). This class contains a public variable "**outlet**" that can be configured with the name of an **Outlet** object supplying power. The "**power**" method handles the details of checking to see if the outlet is valid and using the outlet to request power.

The changes required to convert our existing **Toaster** into an **Appliance** are minor, as

```
itcl_class Appliance {

    method power {power} {
        if {[itcl_info object $outlet] == ""} {
            set outlet {}
        }
        if {$outlet == ""} {
            error "cannot use $this: not plugged in"
        }
        $outlet use $power
    }

    public outlet {}
}
```

```
itcl_class Toaster {
    inherit Appliance
    ...

    method toast {nslices} {
        power [expr 0.03*$heat]
        if {$nslices < 1 || $nslices > 2} {
            error "bad number of slices: should be 1 or 2"
        }
        set crumbs [expr $crumbs+$heat*$nslices]
        if {$crumbs >= $maxcrumbs} {
            puts stdout "== FIRE! FIRE! =="
            set crumbs $maxcrumbs
        }
        return [check]
    }

    method clean {} {
        power 0.5
        set crumbs 0
        return [check]
    }

    ...
}
```

FIGURE 4 – (a) Base class for **Appliance** objects that deal with electrical **Outlet**'s.
(b) Changes required for **Toaster** class to act as an **Appliance**.

shown in Figure 4(b). The "**inherit**" statement makes the **Appliance** behavior available within class **Toaster**, and the "**toast**" and "**clean**" methods are modified to take advantage of it. Both methods request power for each operation. For toasting, the amount of power depends on the current heat setting, while for cleaning the power has a large, constant value (image the crumbs being vaporized!). Note that since class **Toaster** has been updated with these changes, class **SmartToaster** will receive them automatically through inheritance.

We use a **Toaster** or a **SmartToaster** as before, except that now we must be careful to specify an **Outlet** for each appliance:

```
wish: Toaster t1 -heat 2
t1
wish: t1 toast 2
cannot use t1: not plugged in
wish: Outlet 110v -owner mmc@gewurtz
110v
wish: t1 config -outlet 110v
wish: t1 toast 2
crumb tray: 10% full
```

MULTIPLE INHERITANCE

Suppose our appliance business becomes quite successful, despite the fact that our toasters catch fire from time to time. Suppose our legal department is concerned about our liability with respect to these fires, and they would like us to track hazardous appliances so that at any point we would know how many are in operation, and how many accidents have occurred. We could build this tracking behavior into the **Appliance** base class, but this solution might be overkill—not all appliances are hazardous. Instead, we create a separate base class **Hazard** for hazardous products, and add this base class to the "**inherit**" specification for appliances that need to be tracked.

We will be gathering statistics for many different appliance classes. For each class, we need to keep track of the total number produced, the number that are currently active, and the number of accidents that have been reported. To simplify this process, we define the **HazardRec** class shown in Figure 5. Each **HazardRec** object represents information for one class of appliances, and contains the class name and the relevant statistics as protected variables. The "**change**" method is invoked to increment or decrement a particular statistic, and the "**report**" method is used to generate a current report. Note that the constructor for this class takes a single argument—the class name associated with the statistics. Since this class has no public variables, the usual "**config**" argument is of no use.

The **Hazard** base class which collects these records is shown in Figure 6(a). The constructor for this class starts off by querying the most-specific class name for the object being constructed. This is accomplished using the built-in "**info**" method to query the class name. For this method to return the most-specific class name, however, it must be invoked in the most-specific class scope; as discussed earlier, this is accomplished via the "**virtual**" command: "**virtual info class**."

If no previous record of this class exists, a new record is created. Note that the command used to create a new record:

```
HazardRec #auto $class
```

specifies "**#auto**" as the object name. This directs **[incr Tcl]** to use an automatically generated name for the object, which is returned as the result of the command. Automatically generated names have the form "*class*<*number*>" where *class* is the class name and <*number*> is some unique number. The name of each new **HazardRec** is associated with a particular class name in the common array "**recs**." The constructor finishes by using the **HazardRec** to record a new instance of some hazardous class.

The destructor for the **Hazard** base class uses the same technique to query the most-specific class name, and then uses the appropriate **HazardRec** to record one fewer active instance. An "**accident**" method is also provided to make it easier for derived classes to report accidents. Each call bumps the accident count for the appropriate class, and an message describing the accident is printed to standard error. At any time the class proc "**report**" can be invoked to report the statistics for a particular class.

The changes required to add this tracking to our **Toaster** class are again minor, as shown in Figure 6(b). The **Hazard** base class is added to the "**inherit**" statement, and the "**toast**" method is updated to use the "**accident**" method to report fires. Again, since class **Toaster** has been updated, class **SmartToaster** will inherit the same changes automatically.

Toaster's can be used as before, and statistics can be reported as follows:

```
wish: Outlet 110v -owner mmc@gewurtz
110v
wish: Toaster t1 -heat 5 -outlet 110v
t1
wish: t1 toast 2
crumb tray: 25% full
wish: t1 toast 2
crumb tray: 50% full
wish: t1 toast 2
crumb tray: 75% full
wish: t1 toast 2
== FIRE! FIRE! ==
crumb tray: 100% full
wish: Toaster t2
t2
wish: t2 delete

wish: Hazard :: report Toaster
Toaster: 2 produced, 1 active, 1
accidents
```

From this example it is clear that the **Hazard** base class is working correctly. Two objects were constructed ("2 produced"), one object was destroyed (only "1 active"), and one accident was reported. The careful reader might wonder: "How did **Hazard::constructor** and **Hazard::destructor** get invoked?" After all, class **Toaster** has both a constructor and a destructor of its own, and neither of these mentions the **Hazard** base class. Any constructors in a class hierarchy that are not explicitly invoked within the normal constructor for an object are implicitly invoked by **[incr Tcl]**. The same rule applies to destructors as well. For example, the "normal constructor" for a **SmartToaster** is the inherited **Toaster::constructor**. When this method has been invoked, **[incr Tcl]** searches through the derivation hierarchy for any other constructors that have been overlooked and invokes them implicitly. For our

```
itcl_class HazardRec {

    constructor {cname} {
        set class $cname
    }

    method change {var inc} {
        if {![info exists $var]} {
            error "bad field '$var'"
        }
        incr $var $inc
    }

    method report {} {
        return "$class: $total produced, $actives active, $accidents
accidents"
    }

    protected class {}
    protected total 0
    protected actives 0
    protected accidents 0
}
```

FIGURE 5 – Class **HazardRec** holds statistics for objects tracked by class **Hazard**..

```
itcl_class Hazard {

    constructor {} {
        set class [virtual info class]
        if {![info exists recs($class)]} {
            set recs($class) [HazardRec #auto $class]
        }
        $recs($class) change total +1
        $recs($class) change actives +1
    }
    destructor {
        set class [virtual info class]
        $recs($class) change actives -1
    }

    method accident {mesg} {
        set class [virtual info class]
        $recs($class) change accidents +1
        puts stderr $mesg
    }

    proc report {class} {
        if {[info exists recs($class)]} {
            return [$recs($class) report]
        } else {
            error "no information for class '$class'"
        }
    }
    common recs
}
```

```
itcl_class Toaster {
    inherit Appliance Hazard
    ...

    method toast {nslices} {
        power [expr 0.03*$heat]
        if {$nslices < 1 || $nslices > 2} {
            error "bad number of slices: should be 1 or 2"
        }
        set crumbs [expr $crumbs+$heat*$nslices]
        if {$crumbs >= $maxcrumbs} {
            accident "== FIRE! FIRE! =="
            set crumbs $maxcrumbs
        }
        return [check]
    }

    ...
}
```

FIGURE 6 – (a) Base class for **Hazard** objects that are monitored for accidents.
(b) Changes required for **Toaster** class to be tracked as a **Hazard**.

example, this takes care of the elusive **Hazard::constructor**.

When multiple inheritance is involved, the derivation hierarchy for a particular class can become quite complicated. **[incr Tcl]** supplies a number of commands that can be used to query hierarchy information. In particular, the built-in "**info**" command returns inheritance information for a class or for a particular object:

```
wish: Toaster :: info inherit
Appliance Hazard

wish: Toaster t3
t3
wish: t3 info inherit
Appliance Hazard
```

Sometimes it is convenient to get a list of all base classes in an inheritance hierarchy, as opposed to the most immediate base classes. The "**info**" command can be used to query this as well:

```
wish: SmartToaster :: info heritage
SmartToaster Toaster Hazard
Appliance
```

The "**heritage**" for a particular class includes the class itself, as well as all base classes reported in the order that they are searched for resolving name conflicts, implicit invocation of constructors, *etc*.

Finally, it is sometimes useful to query a particular object, to see if it belongs to a certain class. This is accomplished using the built-in "**isa**" method:

```
wish: SmartToaster st
st
wish: st isa Toaster
1
wish: st isa Blender
0
```

This method returns a non-zero value if the given class can be found anywhere in an object's heritage. Thus, a **SmartToaster** is a **Toaster**, but not a **Blender**.

Multiple inheritance is an extremely useful facility. It allows different behaviors to be packaged neatly into base classes and "mixed in" to a class as needed. In many programming languages, multiple inheritance imposes severe performance penalties on applications that use it. For **[incr Tcl]**, however, multiple inheritance is not much more expensive than single inheritance. Still, it can add considerable complexity to an application, and should not be used excessively.

The only limitation in the multiple inheritance model used for **[incr Tcl]** is that the same base class may not appear twice in a class heritage. That is, a class cannot inherit from the same base class more than once. Suppose, for example, we create a class **Oven** which inherits from **Appliance**. Suppose we wanted to create a **ToasterOven** by inheriting from both **Toaster** and **Oven**. **[incr Tcl]** would flag this as an error, since the base class **Appliance** would appear twice in the heritage of the new class—once from **Toaster** and once from **Oven**—and references to **Appliance::** members would either be ambiguous or would clash with one another.

Many times a compositional ("has-a") relationship is more effective than an inheritance ("is-a") relationship. For example, we modeled a **Toaster** as both an **Appliance** and a **Hazard**; both base classes were used to supply different behaviors. Instead of creating a base class **Hazard**, however, we could have created a **HazardMonitor** class and had each **Toaster** create/destroy a **HazardMonitor** object automatically within its constructor/destructor. From this standpoint, a **Toaster** "is-a" **Appliance**, but "has-a" **HazardMonitor**. Both solutions would work equally well, and both will have their proponents. Suffice it to say that multiple inheritance is useful, but is not always the best solution or the only solution for a particular problem.

APPLICATIONS

One application area that the object-oriented paradigm addresses particularly well is the construction of "mega-widgets." It is often useful to package a collection of primitive widgets together as a new widget—using buttons and entries and a listbox to create a file browser, for example. Until now, it has been difficult to do this and make the result look like a widget, and not a collection of procedures. Using **[incr Tcl]**, however, new widget classes can be defined entirely in Tcl code, and the resulting widget objects look like normal Tk widgets.

I have created a class **ColorEditor**, for example, that packages a canvas, a scale and an entry widget together to form the color editor shown in Figure 7, with a Hue-Saturation-Value (HSV) color model. Any color name typed into the entry widget is automatically displayed in terms of its hue, saturation and value components. Hue and saturation are represented by a position within the circle drawn on the canvas; value is shown directly by the scale widget. As the user adjusts these components by clicking on the canvas or adjusting the scale, the entry is updated to display the current color value in "**#rrggbb**" format.

A new **ColorEditor** widget is created like any other Tk widget:

```
ColorEditor .editor -size 2i \
  -borderwidth 3
```

The args "**-size 2i -borderwidth 3**" are passed to the constructor and treated as assignments to the public variables "**size**" and "**borderwidth**". This widget can be packed and treated like any other widget:

```
wish: pack append . .editor top
wish: winfo class .editor
ColorEditor
```

The class also includes a "**config**" method which can be used to change widget attributes (*i.e.*, public variables), an "**install**" method which loads a color value, and a "**get**" method which returns the current color value. Note that although any number of **ColorEditor** objects can be created, each will have its own protected variables containing the H, S and V color components.

Inheritance is an added advantage, allowing mega-widgets to be specialized with new features. I have created a class **ListBox**, for example, that displays a list of items in a text widget, and automatically manages its own scrollbar. When the list is longer than the display area, the scrollbar is packed into the widget frame; when the list is shorter, it is removed. I then derived the class **SelectBox**, which inherits the display capability from **ListBox**, and adds facilities for selecting items in the list.

Compositional ("has-a") relationships can also be used to build larger widgets. I created the class **FilteredBox**, for example, which

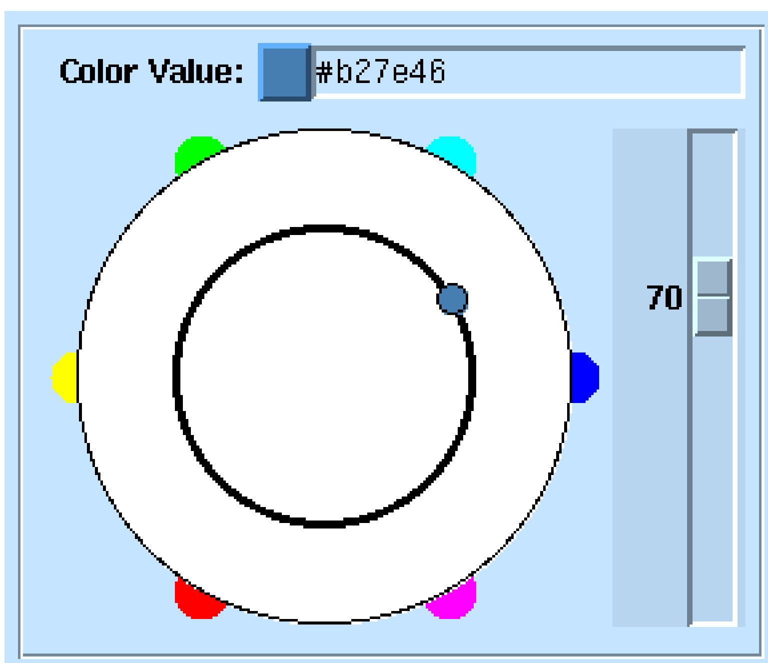

FIGURE 7 – **ColorEditor** mega-widget implemented using **[incr Tcl]**.

contains a **SelectBox** at its core, and adds a title and a filter entry around it. The filter entry restricts the display to items matching a string pattern. A simple application illustrating these new listbox widgets is shown in Figure 8.

With no C code and very little Tcl code, I have created an array of useful widgets. **[incr Tcl]** supports a clean separation between the main application and the special-purpose code that drives these widgets. I believe that these same principles can be applied in a variety of different contexts, allowing programmers to encapsulate many pieces that join together in their application.

CONCLUSIONS

The object-oriented paradigm has become popular because it supports the construction of complex applications. **[incr Tcl]** represents a simple set of object-oriented extensions designed to help programmers write applications that are easier to maintain. Instead of relying on a single global namespace, programmers can package related procedures together with their shared data into a class definition. These procedures provide a well-defined interface for manipulating the data that is otherwise hidden from the outside world. Classes can inherit characteristics from one another, allowing for code re-use through specialization. Each class has transparent access to its own local namespace, but can also access the namespace of inherited classes and the global namespace, when needed. Together, these facilities should provide a solid foundation for writing better applications with Tcl.

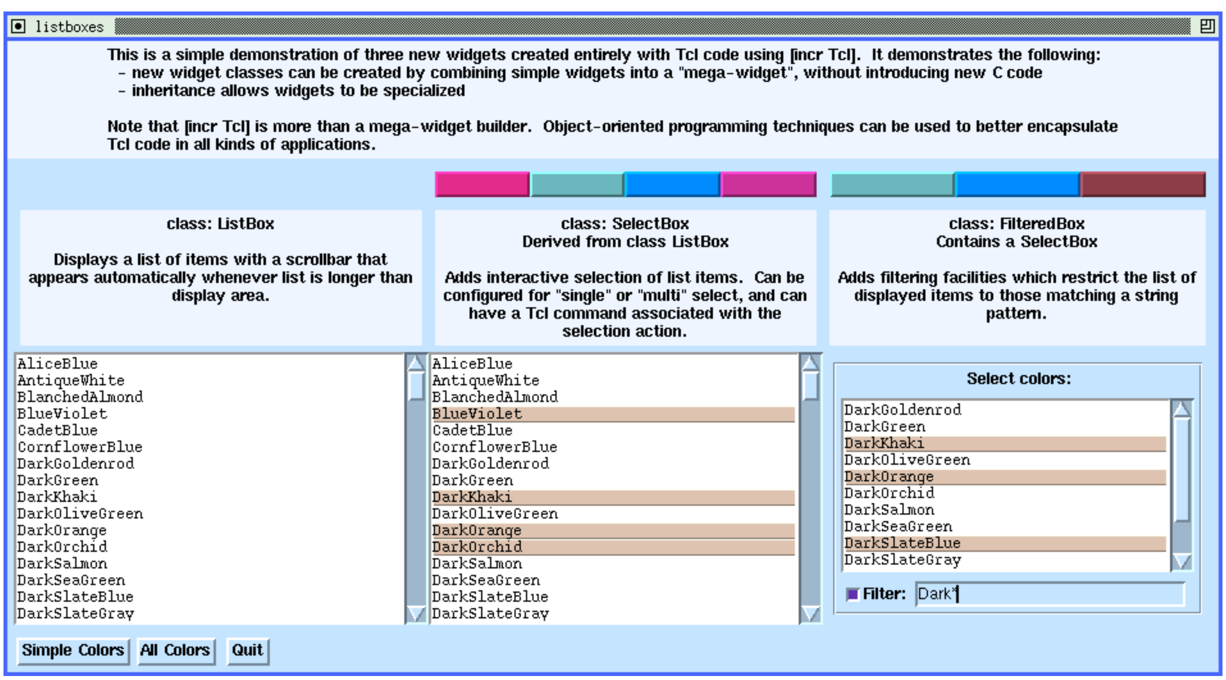

FIGURE 8 – A series of listbox mega-widgets implemented using **[incr Tcl]**.