Object-oriented Tcl OBJECTIVITY

CARSTEN ZERBST

Tcl is the IT industry's best kept secret - and very much alive. The object-oriented extension with the unwieldy name of [incr Tcl] that we are going to introduce at the start of our Tcl series, has contributed a great deal to Tcl's popularity.

With this article, Linux Magazine is starting an occasional series of reports on news from the Tcl world. We will be introducing tools and extensions that enable particularly simple or elegant Tcl solutions. As the development model of Tcl has changed recently, the first part will start with new additions and amendments. Following that, we will be introducing the object-oriented extension [incr Tcl].

For years the scripting language Tcl has been labelled "the IT industry's best kept secret". Although the Tcl programming community is growing, this self-deprecating description is quite accurate. While its competitors Perl or Python are on everyone's lips, Tcl ploughs on in obscurity without much fuss.

There are reasons for that. For one thing, there is still no equivalent to Perl-CPAN or Python.org, the search for Tcl extensions or documentation for a specific problem can thus be quite difficult. The 'The Story So Far' box examines the current state of affairs, how it came about, and what the near future is likely to bring.

The many changes in recent months have meant that many extensions and a lot of information have become even harder to find. The developers' information page is now at Active State. It contains many links to documentation, extensions and programs. Another option is the Tcl foundry at Sourceforge. For many everyday problems it's also worth having a look at the Tcl'ers Wiki. If you're still drawing a blank, the newsgroup *news://comp.lang.tcl* is the last resort for any questions on Tcl.

One old criticism of Tcl that keeps reappearing, and not just on the Tcl newsgroup, is its lack of object orientation. Fortunately, it is easily upgraded using an extension - the sophisticated and very popular OO-extension [incr Tcl], which is the subject of this month's article.

Doubleplus good: [incr Tcl]

With scripting languages you frequently wish for the blessings of object-oriented development: classes, inheritance, data encapsulation and so forth. Although Tk does already feel pretty object-oriented, Tcl does not support any OO features apart from namespaces. This is where the extension [incr Tcl] comes in. Not only is its name similar to C++, the Tcl command *incr* is the equivalent of the ++ in C, but [incr Tcl] aims to extend the base language with OO features, just like C++. This extension is available from Sourceforge, but it is also included in most Linux distributions.

Prior to use, the extension must be loaded into a *tclsh* or *wish*, this is done with the command *package require Itcl*. All [incr Tcl] commands are defined in the namespace *itcl*, you can either use their full names or import them into the current namespace. In the following text we will be describing programming with this extension using a type 1 font editor as an example.

Class-wise

Classes are the basic components of object-oriented programming. We will be assuming a certain degree of familiarity with the concept of classes. Before objects can be created and used, it is necessary to define a class. The assignment of classes determines which variables and methods exist, and what their tasks are. To define class variables it is sufficient to list their names with the command *variable*.

Methods are defined in a similar way to normal Tcl procedures: the name is followed by the input parameters and then by a body containing the actual source code. Within this body, class variables can be accessed directly. The variable *this*, containing a reference to the current object, is also available there. In addition, two special methods can be defined: constructor and destructor. They are invoked when objects are created or deleted. Their structure is again similar to that of normal procedures, but the destructor has no input variables.

Our example in Listing 1 shows the definition of the class point. Each point has an x- and a y-coordinate set by the constructor. Point objects can also be moved.

The classes defined with [incr Tcl] can be used in a similar way to Tk widgets. As you can see in Listing 2, the creation of a new object is invoked using *classname objectname ?parameter?*. The object name can either be specified explicitly or

Listing 1: Class definitions with methods and variables

```
package require Itcl
namespace import itcl::*
class Point {
 public variable x
 public variable y
 constructor {_x _y} {
   set x $_x
   set y $_y

puts "constructor: $this, $x:$y"
 }
 destructor {}
 public method move {dx dy} {
   set x [expr {$x + $dx}]
   set y [expr {$y + $dy}]
   return [list $x $y]
 }
}
```

assigned automatically using *#auto*. The object is now available as a new command under this name, with the syntax *objectname method ?arguments?*. The methods *cget*, *configure* and *isa* are available for each object.

The first two are used to return and set variables defined as *public*. The method *isa* checks whether an object is a member of a particular class. We also have the method *move* that we defined ourselves. The commands *delete object objectname* and *delete class classname* delete objects or classes.

Family ties

The outline of type 1 fonts consists of straight lines and curves. Each straight line is defined by two intersections, each curve by two intersections and two check points. In order to be able to distinguish between the two different point types, each one will be assigned its own class.

In the example in Listing 3 both classes inherit from the *point* class using the keyword *inherit*. The constructor passes the required variables to the base class. Additional variables and methods can be added in the definition of the derived classes - such as the method *coordinates* in the class *Intersection* in Listing 3.

So far we are lacking the ability to represent the objects on canvas. Normally this method would be placed in the point class. In order to demonstrate multiple inheritance we have defined a special class *draw* in the example.

As you can see from Checkpoint, multiple inheritance is also pretty simple to use. However, it is not altogether without problems. Just ask yourself the question: "What happens if two base classes each contain a method with the same name?" In [incr Tcl] the method from the first base class on the inherit list is used, which is not necessarily what you want. Another problem that can occur with multiple inheritance is diamond inheritance (see Figure 1). This case is not supported in [incr Tcl].

A type 1 font does not only consist of individual points, however, and we are still missing outlines. This is an opportunity to introduce another feature of object-oriented programming - delegation.

This describes a process where a class assigns tasks to objects in another class. In our case a straight line is defined by two intersections and does not have to concern itself with representation and data storage. Whenever the line requires intersection coordinates it can request them from the intersection objects. The necessary intersections are stored in the Line class (Listing 4).

Mine!

As part of data encapsulation a class should be able to determine who can access its variables and methods. [incr Tcl] supports three different levels of access:

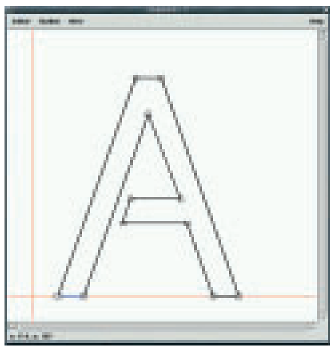
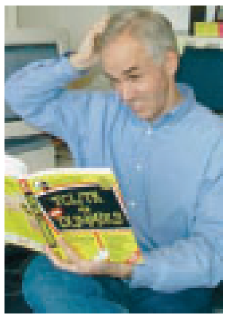


Figure 2: The type 1 editor is based on the program excerpts in this article.



John Ousterhout, father of Tcl. Still seems to be having problems with "Tcl/Tk for Dummies".

The story so far

John Ousterhout, the spiritual father of the language, was also its main developer for many years. He designed its basics while a professor at Berkeley University, before leaving for Sun Microsystems with a team of developers. There, the team created the bytecode compiler and took the first steps towards Unicode. Before internationalisation was completed with Tcl 8.1, Ousterhout left Sun to set up his own company, Scriptics. Scriptics' aim was to sell development tools (Tcl Pro) and professional support for Tcl. This was meant to finance further development of the language, amongst other things. Releases 8.1 to 8.3 were developed under the auspices of Scriptics. However, sales of Tcl Pro were apparently not sufficient to finance full-time Tcl developers.

This is the classic dilemma of open source companies, Python have had similar problems recently. Scriptics was renamed Ajuba Solutions, its new emphasis being the sale of B2B (business to business) products for integrating databases into the Internet using XML. As a result, Ajuba's developers seem to have got to know XML too well - the whole company was taken over by Interwoven, a large manufacturer of B2B software, who was not interested in developing Tcl any further.

So what is the situation for Tcl about half a year later? Ousterhout had already moved the home of the Tcl sources to Sourceforge in the days of Ajuba. Apart from the interpreter, you can find about 200 other projects there that develope Tcl tools or extensions, including the Tcl-Pro-Suite tools. Since Interwoven did not want to market Scriptics' crown jewels, they generously issued them as Open Source.

However, the issue of where the sources were stored was only part of the problem. It was also necessary to transfer development control from the hands of a benevolent dictator to the development community. The solution was the Tcl Core Team (TCT). Tcl users decided on the Internet who should be a member of the TCT and thereby determined the direction of future development. Development suggestions (mostly including the solution) are submitted as Tcl Improvement Proposals (TIP). After a discussion on the public mailing list TCT members vote on the proposals.

Thanks to this process, development has speeded up significantly. However, the loss of full-time Tcl developers paid for by Scriptics has, of course, left a large gap. This is where a company called Active State stepped in. They are already known to users of Perl and Python for support and porting, with Tcl they now offer all of the three major scripting languages. In the meantime, Active State has also picked up two prominent Tcl developers, namely Jeffrey Hobbs and Andreas Kupries.

Active State will be offering a "batteries included" distribution in the foreseeable future, putting an end to the constant need to search for suitable extensions. At the moment they are working on Tcl version 8.4, a new alpha release of which is meant to be available on Sourceforge by the time this issue goes to print. On the TCT mailing list and in the newsgroup comp.lang.tcl there are already first rumblings about Tcl 9.0. Overall, we can expect a lot of activity.

Listing 2: the interactive point class

```
% Point p1 10 10
p1
% p1 cget -x
10
% p1 configure -x 200
% p1 cget -x
200
% Point #auto 20 20
point0
% point0 move 10 20
20 30
% point0 isa Point
1
% point0 isa Oink
0
% delete object p1 point0
% delete class Point
```

Info

Developer's information page:
http://tcl.activestate.com
Tcl foundry at Sourceforge:
http://sourceforge.net/foundry/tcl-foundry
Tcl'ers Wiki:
http://mini.net/cgi-bin/wikit/0.html
Tcl newsgroup:
news://comp.lang.tcl
[incr Tcl] at Sourceforge:
http://sourceforge.net/projects/incrtcl/
Combat, a Tcl CORBA extension:
http://www.fpx.de/Combat/
The Tcl Tk homepage:
http://www.tcltk.org/
The type 1 editor:
http://www.tu-harburg.de/~skfcz/tcltk.html

■

The author

Carsten Zerbst is a member of staff at Hamburg Technical University. Apart from researching service integration on board ships he also investigates Tcl in all its forms.

- public: The method or variable is accessible to anyone.
- protected: Protected elements can only be used within the class itself or in classes that have been derived from it.
- private: Only accessible within the class itself.

Up to now, the coordinates of a point could be modified at will using *configure*, without any validation of the values being performed. Once the coordinates have been defined as *protected*, they can only be changed using the method *move*.

```
class Point {
  protected variable x
  protected variable y
  # ...
}
```

The derived classes can still access variables directly, without being dependant on specific methods. If the coordinates had been assigned a level of *private*, however, even the derived classes would not be able to access them directly.

All in all

[incr Tcl] is a simple way of defining classes including inheritance, in which variables and methods can be protected against access at different levels. Its features - including multiple inheritance - are modelled on C++. In fact, [incr Tcl] classes can even inherit from C++ classes. A particularly good example of this if the CORBA extension Combat by Frank Pilhofer.

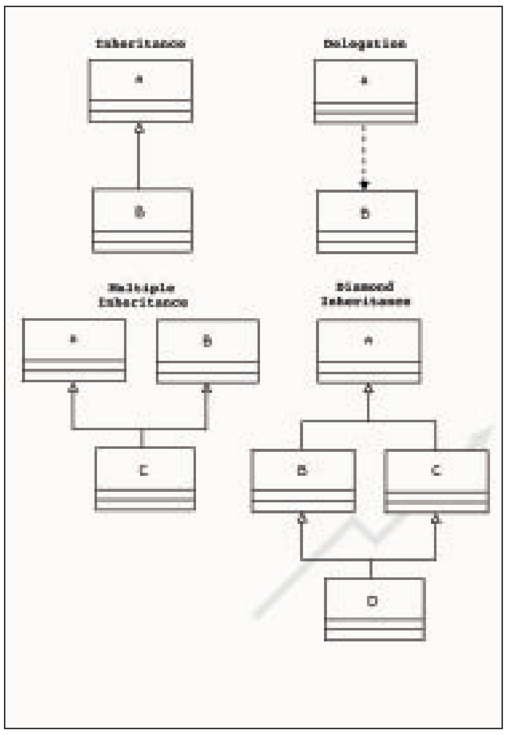


Figure 1: Valid and invalid relations between classes in [incr Tcl]. Inheritance, multiple inheritance and delegation are permitted, but diamond inheritance is not.

Using objects will be familiar to any user who has worked with Tk before. The features of [incr Tcl] can therefore be used without too much additional learning effort being required. Namespaces, which were first developed in [incr Tcl] have already found their way into normal Tcl several years ago, which is why we have not dealt with them here. Additional literature on [incr Tcl] can be found at *http://www.tcltk.org/*

The examples above, and an editor derived from them (Figure 2), can be found at *http://www.tu-harburg.de/~skfcz/tcltk.html* However, at the moment, there isn't any source text to represent the letters on canvas. We will be looking at the options of the canvas widget in the next instalment of Tcl. ■

Listing 3: Inheritance and multiple inheritance

```
class Draw{
 constructor {} {}
 destructor {}
 public method draw {} {
   if {[$this isa Checkpoint]} {
      # draw Checkpoint
   } elseif ...
 }
}
class Intersection {
 inherit Point
 constructor {_x _y} {
   Point::constructor $_x $_y
 } {
   # constructor for Intersection
 }
 public method coordinates
 {} {
   return [list $x $y]
 }
}
class
 Checkpoint {
 inherit Point Draw
 constructor {_x _y} {
   Point::constructor $_x $_y
   Draw::constructor
 } {
   # constructor for Checkpoint
 }
}
```

Listing 4: Delegation

```
class Line {
 private variable k1
 private variable k2
 constructor {_k1 _k2} {
     foreach k  {$_k1  $k2} {
        if  {![ $k  isa  nodal point]} {
            error “Node $k is not a nodal U
point!”
          }
     }
     set k1 $_k1
     set k2 $_k2
 }
}
```