Interpreted C++, Object Oriented Tcl, What next?

Dean Sheehan (*deans@x.co.uk*)
IXI Visionware, Vision Park, Cambridge, CB4 4ZR, England.

Abstract

Tcl[1] is an interpreted high level language suitable for scripts, small scale systems, prototypes and embedding in larger applications. C++ is a powerful compiled language that provides support for object oriented programming and is suitable for building large complex systems. But what if you could move from C++ to Tcl and back again with the ease of an object reference and a dynamically bound function?

This paper describes an extension to Tcl, or an extension to C++ depending on your perspective, that makes it possible to:

- ✯ use object oriented programming concepts in Tcl
- ✯ inherit from C++ classes (with dynamic binding of methods) in Tcl
- ✯ instantiate C++ classes from Tcl
- ✯ invoke methods upon C++ objects from Tcl
- ✯ delete C++ objects from Tcl
- ✯ pass Tcl objects to C++ for method invocation and deletion.

The name of this extension (Tcl++ was rejected) is **Object Tcl**.

1 Introduction

Tcl was originally designed to be embedded in larger applications, implemented in C, to provide the user with a pragmatic means of controlling and customizing the application without resorting to application code changes. In recent years, the software industry has looked to object orientation as a means of managing the design, implementation and maintenance of today's complex systems. C++ has become the *de facto* standard for implementing such systems.

So what about complex object oriented systems requiring a command and customization language? Tcl is designed to be extensible by the addition of commands that are bound to C functions, but what if your system uses object orientation and C++? At first glance many people would say Tcl commands can be bound to C++ functions as before, but what about all the effort that went into producing your object oriented design? What about the cost of interfacing procedural C++ code with object oriented C++ code?

In an effort to resolve some of these issues, IXI Visionware has developed a Tcl extension that facilitates the use of Tcl as a command language for a system built using C++. This extension makes it possible to maintain the object concepts of the implementation in the command language.

This paper presents a brief description of the Object Tcl extension. Object Tcl is a tool command language for object oriented C++ applications, as Tcl is for C applications.

The structure of this paper is as follows: Section **2** describes the Tcl language additions, called Otcl. Section **3** describes the C++ binder responsible for supporting the use and re-use of C++ classes. Section **4** provides a complete example including Otcl code and C++ binding. Section **5** gives an overview of Object Tcl's implementation. Section **6** provides a comparison between Object Tcl and **[incr Tcl]**. Section **7** briefly introduces a recent extension to Object Tcl to support distributed programming. Section **8** describes the status and availability of Object Tcl. Section **9** presents our conclusions. Section **10** provides a bibliography.

This paper assumes a familiarity with object orientation, C++ and Tcl.

2 The Otcl Language

The Object Tcl extension augments the standard Tcl command set with commands for describing common object oriented concepts. These concepts include:

- classes
- objects
- instance methods
- class methods
- instance attributes
- class attributes.

This new language is called **Otcl**.

In Otcl, the description of a class is broken down into two parts, its interface and its implementation. A class's interface describes the external access to that class and objects of that class. A class's implementation describes the implementation of methods externally available (public methods) and any method used internally (private methods). A classes implementation also describes the internal state variables (private attributes[1]).

otclInterface *name ?-isA classList? interface*[2]

This command is used to describe the interface of an Otcl class, where *name* is the name of the new Otcl class.

The optional *-isA classList* argument can be used to specify a list of other classes that this new class will inherit from. An Otcl class may inherit from other Otcl classes and/or C++ classes that have been exported to Otcl (see section 3).

The *interface* argument is a script that may use the following commands to describe the classes interface:

constructor *argList*

> This command is used to describe the class constructor interface. Constructor methods are invoked automatically when new instances of the class are created. The *argList* argument is a list of formal arguments for the constructor.

method *name argList*

> The method command may be used to describe the interface of an instance method. Instance methods have names, *name*, and a list for formal arguments, *argList*. Instance method may be invoked upon instances of this class.

classMethod *name argList*

> The classMethod command may be used to describe the interface of a class method. Class methods have names, *name*, and a list of formal arguments, *argList*.

An example of an Otcl class interface is given below:

```
otclInterface Rectangle -isA Shape {
    constructor {width height x y}
    method getArea {}
    classMethod getTotalArea {}
}
```

This code segment describes the interface for a new class, the Rectangle class. This class inherits from an existing Shape class, it has a constructor that takes four parameters, an instance method called getArea and a class method called getTotalArea.

otclImplementation *name implementation*

This command is used to describe the implementation of a class, where *name* is the name of an Otcl class that has previously had its interface described.

The *implementation* argument is a script that may use the following commands to describe the implementation of the class:

constructor *argList parentList body*

> This command is used to describe the implementation of the class constructor method. The *argList* is the list of formal arguments and the *body* is the Tcl script for the behaviour of the constructor. The *parentList* is a list of scripts that should be used to pass arguments to the constructors of any inherited classes. Each script must start with the name of the inherited class and then list the argument expressions in a similar manner to command invocation in Tcl.

1. All attributes in Otcl are private and therefore they cannot be accessed directly from outside the class, only via public methods.
2. The notation used for command descriptions is the same as that used in [1].

destructor *body*

The destructor command is used to describe the behavior of the class's destructor method. The destructor is invoked automatically when instances of this class are deleted. Destructor methods cannot take any arguments.

method *name argList body*

The method command is used to describe the implementation of an instance method. Instance methods are named, *name*, take a list of formal arguments, *argList*, and have a script describing the behavior of the method, *body*.

Instance method bodies may access class and instance attributes using the $ substitution syntax and pass them to standard Tcl commands like set and incr. Instance method bodies, including the bodies of the constructor and destructor methods, may also access an implicit instance attribute called this. The this attribute is a reference to the current object.

classMethod *name argList body*

The classMethod command is used to describe the implementation of a class method. Class methods are named, *name*, take a list of formal arguments, *argList*, and have a script describing the behavior of the method, *body*.

Class method bodies may access class attributes using the $ substitution syntax and pass them to standard Tcl commands like set and incr.

attribute *name ?value?*

The attribute command is used to describe an instance attribute. Instance attributes are named and may take an optional value for use in initializing the attribute on the creation of a new instance of the class.

classAttribute *name value*

The classAttribute command is used to describe class attributes. Class attributes are named and must also be supplied with a value to be initialized with when the class description is completed.

An example of an Otcl class implementation is given below:

```
otclImplementation Rectangle {
    constructor {w h x y} {{Shape $x $y}} {
        set width $w
        set height $h
        incr totalArea [$this getArea]
    }
    destructor {
        incr totalArea -[$this getArea]
    }
    method getArea {} {
        return [expr $width * $height]
    }
    classMethod getTotalArea {} {
        return $totalArea
    }
    # This method is private
    method giveRatio {} {
        return [expr $with / $height]
    }
    attribute width
    attribute height
    attribute anchored()  {{right ""}
                           {left ""}
                           {top ""}
                           {bottom ""}}
    classAttribute totalArea 0
}
```

This code segment describes a possible implementation of the Rectangle class with the interface described earlier. The code segment also demonstrates Tcl arrays as instance attributes, the anchored attribute. Tcl arrays may also be used as class attributes.

The implementation of a class may describe methods, both instance and class, that do not appear in the interface. These methods are private and may only be called from within other methods of the class. The giveRatio method in the code segment above is an example of a private method.

Class methods may be called by using the name of the class as a command with the first parameter being the name of the class method to invoke. Arguments to the class method may also be supplied as additional parameters to the command. For example:

```
puts "Total area is [Rectangle getTotalArea]"
```

This code segment invokes the getTotalArea class method upon the Rectangle class.

Instances (objects) of Otcl classes, or exported C++ classes (see section 3), can be created using the otclNew command. This command takes the name of the class as its first parameter and any remaining parameters are passed to the constructor of the object being created. The otclNew command returns a reference to the new object. The returned reference may be passed as a parameter to other commands and used to invoke methods upon the object. For example:

```
set aRect [otclNew Rectangle 100 150 0 0]
```

This code segment creates an instance of the Rectangle class, a Rectangle object. The constructor for the Rectangle class takes four parameters and these are supplied in the call to otclNew. The result of otclNew, the object reference, is assigned to the aRect variable.

Instance methods may be invoked using an object reference as a command with the first argument being the name of the instance method to invoke. Once again arguments to the method may also be supplied as additional arguments to the command. For example:

```
set area [$aRect getArea]
```

This code segment invokes the getArea instance method upon the object referred to by the variable aRect. The result of this method is assigned to the area variable.

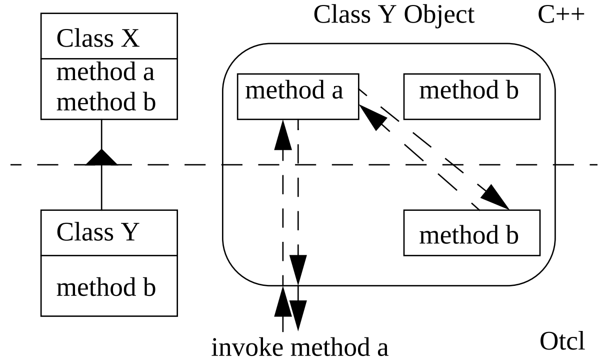
Objects may be deleted using the otclDelete command that take only one parameter, a reference to the object to be deleted.

```
otclDelete $aRect
```

This code segment deletes the Rectangle object referenced by the aRect variable.

All instance methods in Otcl are subject to dynamic binding, so invoking a method on an object will result in the execution of the named method in the most specific class in the inheritance hierarchy for that object. If the object is of a C++ class, or an Otcl class that inherits from a C++ class, it is quite possible that the dynamic binding will end up in the C++ domain. Furthermore, if a method is executed from within the C++ domain upon an object that is an instance of an Otcl class, derived from a C++ class, it is possible for the dynamic binding to execute a method that is in the Otcl domain. Figure 1 provides an illustrated example of this.

Figure - 1



Within instance method bodies an optional flag may be specified between the object reference and the method name to force the execution of the method in the named parent class. The flag must be of the form -*name* where *name* is the name of one of the inherited classes. This optional flag allows subclasses to provide a version of a method defined in one of its superclasses but still make use of the superclasses version in its implementation. For example, assuming Cube is a subclass of Shape that has a method rotate:

```
otclImplementation Cube {
    method rotate {degrees} {
        # perform some Cube specific
        # rotation things and then
        $this -Shape rotate $degrees
    }
}
```

3 The C++ Binding

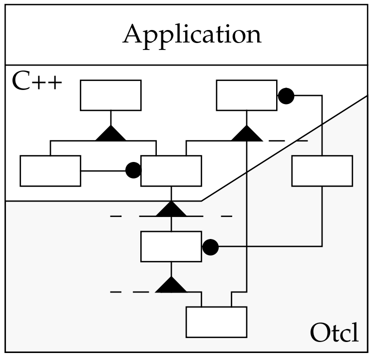
The Object Tcl extension provides a mechanism for binding an application's C++ classes into the Otcl language. The C++ binding makes it possible for Otcl scripts to:

- invoke static member functions of C++ classes
- create instances of C++ classes
- invoke instance member functions of C++ objects
- delete C++ objects.

Also, Otcl classes can inherit from C++ classes and redefine virtual member functions with Otcl instance methods. Additionally, C++ may invoke virtual functions upon Otcl objects that are of Otcl classes derived from C++ classes.

The Object Tcl C++ binding facility makes it possible to develop a C++ class framework that can be specialized in Otcl as illustrated in figure 2.

Figure - 2 (OMT [2])



Before an application's C++ classes can be utilized within Otcl, they must be described and exported to Object Tcl. Exporting C++ classes is performed by writing class descriptions in the Class Description Language (CDL), processing the CDL files using Object Tcl's cdl processor (cdl), compiling the generated source files and finally linking the resulting object files with the target application.

The output generated by the cdl processor consists of a C++ header file and source file for each CDL file. These files contain the declaration and definition for implementation classes that bind Otcl to the C++ application classes via multiple inheritance (see section 5).

CDL is based on Tcl with commands for describing C++ classes. The following commands are available in CDL:

pass ?option? *statement*

> Where *statement* is a string that is to be passed through to the generated output without any processing. This facility is commonly used to place #include directives or comments in the C++ code. The *option* parameter may be either -h or -s which can be used to specify whether the *statement* should only be passed though to the header file or source file respectively. If no *option* is specified, the *statement* will be passed through to both the header and source file.

class *name description*

> Where *name* is the name of a C++ class being described and *description* is a Tcl script that may use the following commands to describe the interface of the C++ class:

constructor *args*

> Where *args* is a Tcl script containing formal argument type commands (described below).

method *name* -(static | dynamic) *args rtn*

> Where *name* is the name of the C++ member function, *args* is a Tcl script containing formal argument type commands (described below) and *rtn* is a script containing a single return type command (described below). The -static and -dynamic flags are mutually exclusive and are used to indicate whether the member function should be statically or dynamically bound between the Otcl and C++ domain[1].

classMethod *name args rtn*

> Where *name* is the name of the C++ static member function, *args* is a Tcl script containing formal argument type commands (described below) and *rtn* is a script containing a single return type command (described below).

The standard formal argument type and return type commands are given in Table 1:

Table - 1

Argument	Return
int	int
float	float
double	double
str	str
obptr *className*	obptr *className*
obref *className*	obref ?-new? *className*
void	void

The obptr and obref argument and return commands support object passing between the Otcl and C++ domains. All of these commands take an additional argument for the name of the actual class expected by this member function for the purpose of type coercion. The obref return argument also accepts an optional flag,

1. In general, C++ virtual functions should be specified as -dynamic. In cases where it isn't expected for any Otcl subclasses to redefine the function, the -static specifier may be used thus avoiding a small performance overhead.

-new, to indicate that the object returned by the member functions should be copied into a new object on the heap.

The CDL processor has been implemented using object oriented programming concepts. This facilitates the quick and easy addition of new classes supporting additional argument and return types.

The cdl processor takes a file containing CDL descriptions and generates either a C++ header file or a C++ source file depending on a command line argument. Once the application's object files have been compiled and the CDL files processed[1], only the link phase is required to make the application's C++ classes usable from Otcl[2].

The C++ binding provided by Object Tcl has the following restrictions:

> Object Tcl does not support method overloading. This means that the CDL description for a C++ class that uses overloading must describe only one, or none, of the overloaded methods. Method overloading also includes constructor overloading. The overloading restriction stems from the fact that Tcl is not typed and therefore argument types cannot be used to reduce the set of possible methods down to one.
>
> Currently, Object Tcl does not support operators. Object Tcl cannot take advantage of operators on exported C++ classes.

All of these restrictions can be worked around without massaging your object oriented design too much.

4 Complete Example

This section provides a complete, although quite useless, example that demonstrates the Otcl language, C++ binding and dynamic method binding crossing the C++ and Otcl domains.

1. It is possible to add *Makefile* rules that take CDL files, with a specific suffix like ".cdl", and generate intermediate C++ files that are then compiled to object files automatically.
2. The C++ generated by the **cdl** processor utilises C++ static object constructors to register the C++ classes with no alteration of the application or Otcl code, therefore the applications *main* function must be compiled by a C++ compiler and the application must be linked by a C++ linker.

4.1 File A.H

```
class A
{
public:
    // Constructor member function
    A (A *next);

    // Destructor member function
    virtual ~A ();

    // Virtual member function
    virtual void doIt (void);

    // Member function
    void doItNext (void);

    // Member function
    void setNext (A *obj);
private:
    // Member variable
    A *next;
};
```

4.2 File A.C

```
#include <iostream.h>
#include "A.H"
A::A (A *n)
{
    next = n;
    cout << "A::constructed" << endl;
}
A::~A ()
{
    cout << "A::destructed" << endl;
}
void A::doIt (void)
{
    cout << "A::doIt" << endl;
}
void A::setNext (A *n)
{
    next = n;
}
void A::doItNext (void)
{
    if (next != NULL)
    {
        // "doIt" is dynamically bound
        // so it could be
        // the "doIt" of class A or
        // a subclass
```

```
        next->doIt();
    }
}
```

4.3 File A_cdl.cdl

```
pass {
// Generated from A_cdl.cdl
#include "A.H"
}

class A {
    constructor {obptr A}

    # "doIt" requires dynamic binding
    method doIt -dynamic {void} {void}

    method doItNext -static {void} {void}

    # The 'A' parameter to the obref type
    # indicates the actual type of the
    # parameter expected by the
    # "setNext" method.
    method setNext -static {obptr A} {void}
}
```

4.4 File B.otcl

```
otclInterface B -isA A {
    constructor {next value}

    # Redefine the "doIt" method
    # available in class A
    method doIt {}
}

otclImplementation B {

    # The constructor method passes on its first
    # argument up to the constructor of the 'A'
    # parent class.
    constructor {n v} {{A $n}} {
        set value $v
        puts "B::constructed with value $value"
    }

    destructor {
        puts "B::destructed"
    }

    # The new version of the "doIt" method.
    method doIt {} {
        puts "B::doIt, value is $value, calling A::doIt"
        # Invoke the "doIt" method but make sure it
        # is the version of the 'A' parent class
        # and not this version that would be chosen
        # by default using dynamic binding.
        $self -A doIt
    }

    attribute value
}
```

4.5 Build

To build the example code into a new version of tclsh, assuming the Otcl and Tcl libraries have been compiled, perform the following:

```
system% CC -c A.C
system% $(OTCL)/cdl -h A_cdl.cdl A_cdl.H
system% $(OTCL/cdl -s A_cdl.cdl A_cdl.C
system% CC -I$(OTCL) -I$(TCL) -c A_cdl.C
system% CC -o examplesh -L$(OTCL) -L$(TCL)
A.o A_cdl.o $(OTCL)/tclAppInit.o
$(OTCL)/tclMain.o -lotcl -ltcl -lm
system %
```

4.6 Test.tcl

```
set a [otclNew A ""]
$a doIt
source B.otcl
set b [otclNew B "" 5]
$b doIt
$a setNext $b
$a doItNext
otclDelete $a
otclDelete $b
```

4.7 Example

To execute the example:

```
system% ./examplesh
% source Test.tcl
A::constructed
A::doIt
A::constructed
B::constructed with value 5
B::doIt, value is 5, calling A::doIt
A::doIt
B::doIt, value is 5, calling A::doIt
A::doIt
A::destructed
```

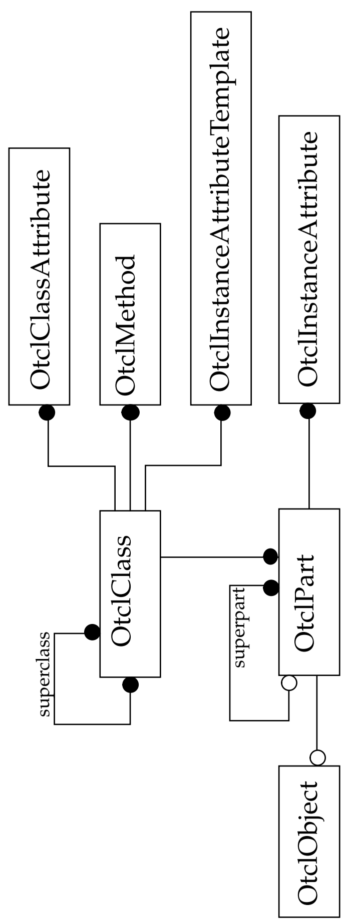
A::destructed
B::destructed
% **exit**

5 Inside Object Tcl

There are effectively two problems to be solved in Object Tcl: How do we extend Tcl to provide support for object oriented programming and how can we get this object oriented Tcl (Otcl) to bind easily with C++. The implementation of Otcl tackles both of these problems at once using object oriented programming concepts within C++.

Figure 3 provides one possible object model for representing the common object oriented concepts. A class is described by a collection of methods and attributes and by the other classes it inherits from. Objects are instances of classes and each of the classes in the inheritance hierarchy manifest themselves in a part of the instantiated object.

Figure - 3 (OMT [2])



Each class in Otcl is modelled by an instance of the OtclClass class. Each OtclClass object is related to a collection of OtclClassAttribute's, OtclMethod's and OtclInstanceAttributeTemplate's. When an Otcl class is instantiated, an OtclPart object is instantiated for each class in the class hierarchy of the instantiated Otcl class. Each OtclPart references a collection of OtclInstanceAttributes that were created from the OtclInstanceAttributeTemplates of the appropriate OtclClass. Only the most specific OtclPart object is related to the OtclObject object, all other OtclPart's in the object are related to subparts. All access to the parts of the object are managed by the OtclObject.

The object model in figure 3 needs to be massaged a little to account for the C++ restriction that in order for an Otcl class to be passed into C++ for manipulation, it must truly inherit from an existing C++ class. This is because C++ is a typed language and inheritance means inheritance of type signature. At this point I must state that if an Otcl class doesn't inherit from a C++ class, it cannot be passed into C++. This is not a restriction of the extension but a cornerstone of typed object oriented languages. You cannot manipulate what you do not understand! The new object model is given in figure 4.

The cdl processor (see section 3) generates C++ code from descriptions of the C++ classes we wish to export to Otcl. For each class description, the cdl processor actually generates two C++ classes, one class inherits from OtclClassCpp and the other inherits from OtclPartCpp and the C++ class to be exported. The first class is responsible for instantiating an instance of the second class when a part is created for Otcl. The second class is responsible for binding dynamic functions that cross the domain from C++ to Otcl and vice versa.

Figure 5 shows part of the object model resulting from exporting the C++ class called Box to Otcl. There is a single instance of the Box_otclc class constructed at process start time using a C++ static object. This registers the C++ class with the Otcl language extensions and provides Otcl with a way of instantiating the necessary OtclPart subclass, Box_otclp, when instances of Box, or Otcl subclasses of Box, are created.

So, if CardboardBox was a subclass of Box, described in Otcl, instantiating a CardboardBox would result in the objects and relationship shown in figure 6. Remembering that Box_otclp is a true C++ subclass of Box, the binding into C++ can be handled by C++ virtual functions in the Box_otclp class.

Figure - 4 (OMT[2])

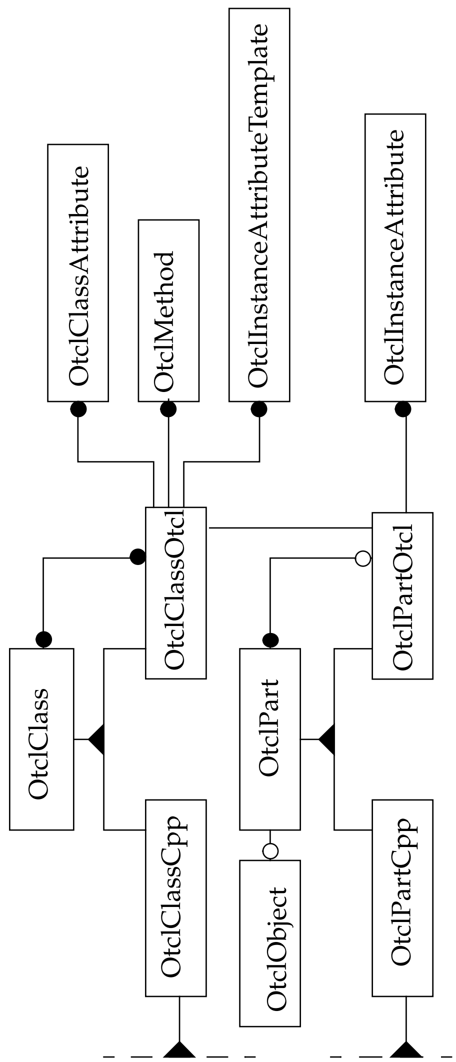
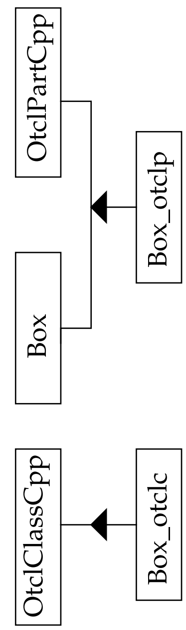


Figure - 5 (OMT [2])



6 [incr Tcl] Comparison

[incr Tcl] [3] is a Tcl extension that provides support for object oriented programming in Tcl. **[incr Tcl]** was not designed with object orientation as its objective but to support a more structured way of programming in Tcl. **[incr Tcl]** is especially directed at Tk[1] programming.

Although Object Tcl and **[incr Tcl]** have different objectives, they do overlap in the area of providing Tcl with support for object oriented programming and therefore a comparison is valid. The following comparison is based upon **[incr Tcl]** version 1.5 and Object Tcl b1.1. Both of these extensions are still evolving and therefore subject to change.

6.1 General

Both Object Tcl and **[incr Tcl]** provide support for:

- classes
- inheritance (single and multiple)
- constructor methods
- destructor methods
- instance methods
- class methods
- instance attributes
- class attributes

6.2 Access Control

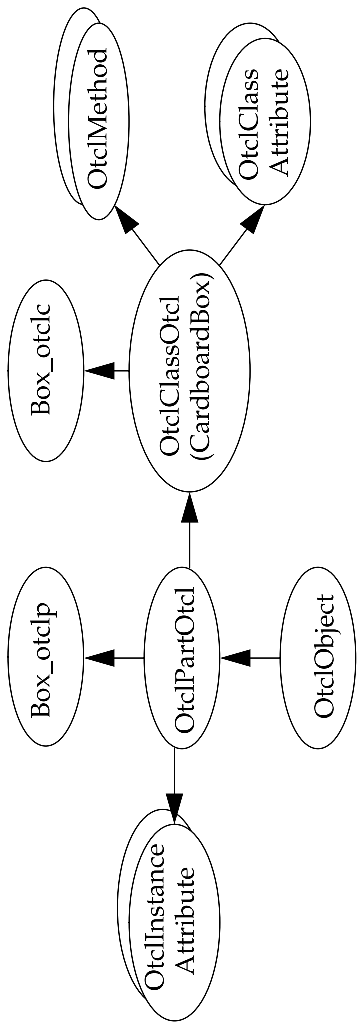
[incr Tcl] provides finer access control granularity over methods and attributes, allowing both to be public, protected (private but accessible from subclasses) and private.

In Object Tcl, methods can only be public or private and attributes can only be private.

6.3 Run Time Type Information

[incr Tcl] provides support for run time type information, such as querying the class of an object. Object Tcl does not provide any support for run time type information although the addition of full meta information is expected in the future.

Figure - 6



6.4 Config Parameter

[incr Tcl] has the config formal argument that is really directed at Tk programming so that **[incr Tcl]** objects can be used in the same manner as Tk widgets.

The config argument is similar to the args argument of Tcl in that it catches all trailing arguments to a method. The config argument differs from the args arguments in that it is parsed as a set of (-attribute value) pairs. These pairs are then used to set the value of an objects attributes. This allows clients of **[incr Tcl]** objects to say:

```
object configure -name Fred -size 100
```

Object Tcl does not provide any similar behavior although it can be built into Object Tcl classes at the user level.

6.5 Tk Binding

Both Object Tcl and **[incr Tcl]** can be used in Tk applications.

[incr Tcl] classes and objects can be managed in the same way as Tk widgets: Classes are created using the name of the class as a command, can be configured using a method called config that takes (attribute value) pairs and deleted by calling the delete instance method upon the object. The similarities between Tk widgets and **[incr Tcl]** classes/objects means that **[incr Tcl]** can be used to develop compound mega widgets that are used in the same way as Tk widgets.

It is believed, although not tried and tested, that Object Tcl could support the development of compound mega widgets with only a few minor modifications. These modifications may also support the development of compound mega widgets in C++ which may then be exported to Otcl.

6.6 C++ Binding

[incr Tcl] does not provide any support for executing C++ static member functions, instantiating C++ classes, invoking C++ member functions, deleting C++ instances or inheriting **[incr Tcl]** classes from C++ classes.

Object Tcl provides support for all of these with the addition of dynamic binding of methods between C++ and Otcl.

6.7 Arrays

Object Tcl supports Tcl arrays as class and instance attributes. Initial array values may be specified for array attributes.

[incr Tcl] does not support Tcl arrays.

6.8 Parent Construction

Object Tcl provides a mechanism for constructors of subclasses to pass parameters to the constructors of inherited classes.

No such mechanism has been found in **[incr Tcl]**.

6.9 Interface Separation

Object Tcl separates the interface of a class from its implementation thus reducing the chances of a class's clients relying on assumptions about its implementation.

[incr Tcl] provides no support for separating out a classes implementation.

6.10 Summary

In summary, Object Tcl and **[incr Tcl]** are comparable when comparing their OO abilities purely in the Tcl domain. There are stylistic differences between the two, notably Object Tcl forces increased encapsulation whereas **[incr Tcl]** allows you to open up a little more.

The main differences are when you bring in Tk and C++. **[incr Tcl]** was developed with Tk in mind whereas Object Tcl wasn't. Object Tcl can probably be used to implement mega widgets but not as easily as **[incr Tcl]**.

[incr Tcl] provides no support for integration with C++.

If you wish to use object oriented concepts purely in a Tcl\Tk domain then Object Tcl and **[incr Tcl]** are evenly matched. If you want to develop or use mega widgets then **[incr Tcl]** is probably a better choice. If you want to use and inherit from C++ classes then Object Tcl is the only way.

7 Distributed Object Tcl

A recent addition to the Object Tcl extension is support for distributed programming.

The purpose of ObjectTcl-DP, as it has been nicknamed, is to support the separation of a single process into multiple processes with minimal impact upon the application code.

ObjectTcl-DP provides the following additional commands:

otcl remoteClass *name address*

> This command registers the named class as a remote class available from the process listening on the specified TCP/IP address. The address must be of the form host:port.
>
> The result of this command is that class methods may be invoked upon the remote class and the otclNew command may be used to create instances of the remote class. The remote instances are actually in the remote process but the reference returned by otclNew is usable locally for instance method invocation and in the otclDelete command.

otcl oserver init *?port?*

> This command initializes the object server provided by Object Tcl-DP. Only objects created after this command may be referenced by external processes. The port parameter specifies the TCP/IP port number for this process to listen on. If no port is specified then this command will automatically select one and return the chosen port number.

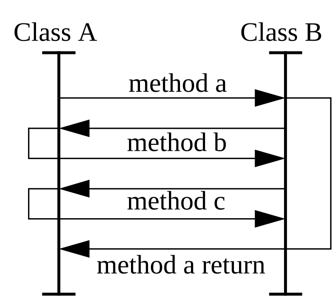
otcl oserver ready

> This command informs the object server that it should start processing external object requests. The object server must already have been initialized using the previous command.

The init and ready object server commands are currently under review along with the whole communication infrastructure to see if it is possible to layer Object Tcl-DP upon the facilities offered by Tcl-DP[4], TclX[5] and Tk. One possible problem with this approach is that Object Tcl is undergoing porting to Windows and thus relying on Tcl-DP or TclX extensions may cause problems. It is expected that the Object Tcl-DP services will be integrated with Tk's event loop at least.

A common calling sequence in object oriented systems is illustrated in figure 7. Unlike Tcl-DP [5], which provides support for RPC in processes with client/server relationships, distributed object systems generally require peer-to-peer communication. Object Tcl-DP has been developed with this in mind so that two object servers can continue a networked dialogue that relates to the same execution thread. If another client sends requests to either of the other two servers while they are performing their bi-directional method calls, the new client will be blocked.

Figure - 7



The following example takes the example described in Section 4 and makes it work over the network.

Process A

Process A is started on the machine with hostname essex.x.co.uk.

```
system% ./examplesh
% source B.tcl
% otcl oserver initialize 2000
2000
% otcl oserver ready
```

At this point process A blocks waiting for remote object requests.

Process B

Comment out the line in TestB.tcl that sources B.tcl and start a basic otclsh process on any machine on the network.

```
system% ./otclsh
% otcl remoteClass A essex.x.co.uk:2000
% otcl remoteClass B essex.x.co.uk:2000
% source TestB.tcl
```

Process A

Process A responds to the requests from process B with:

```
A::constructed
A::doIt
A::constructed
B::constructed with value 5
B::doIt, value is 5, calling A::doIt
A::doIt
B::doIt, value is 5, calling A::doIt
A::doIt
A::destructed
A::destructed
B::destructed
```

Process A then blocks pending further requests

Process B

Process B returns to the otclsh prompt.

8 Status & Availability

Object Tcl is currently at revision b1.1 and is available from IXI Visionware's WWW server at:

http://www.x.co.uk/devt/ObjectTcl/

These WWW pages provide more detailed documentation on the Object Tcl system as well as access to the source distribution.

The Object Tcl source distribution is also mirrored at many of the FTP sites favored by the Tcl community.

ObjectTcl-DP will be available in version b1.2 of the Object Tcl system expected in late June 95.

9 Conclusions

If Tcl and C++ could be made to bind perfectly without any discernible difference in paradigm or syntax then Tcl would be C++ and vice versa. It is said that power is derived from differences. Tcl and C++ are very different and both provide considerable power in their applicable domains. In some circumstances the power of both languages is required to provide the best possible solution.

Object Tcl smooths the transition between C++ and Tcl without reducing these languages to their common denominator. Powerful object oriented C++ applications can now be built that provide a Tcl frontier for customisation and control. Object oriented Tcl applications can have collections of their classes moved into C++ for higher performance or C++ classes can be moved down into Tcl for customisation. Hopefully the best of both worlds.

It is hoped that Object Tcl will broaden the domain of applications for both C++ and Tcl.

10 References

[1] John K. Ousterhout, “Tcl and the Tk Toolkit”, Addison-Wesly.

[2] Rumbaugh *et al*, “Object-Oriented Modelling and Design”, Prentice Hall.

[3] Michael J. McLennan, “[incr Tcl] - Object Oriented Programming in Tcl”, AT&T Bell Laboratories.

[4] Brian C. Smith (Dept of Computer Science, Cornell University) & Lawrence A. Rowe (Computer Science Division-EECS, University of California at Berkeley), “Tcl Distributed Programming (Tcl-DP)”.

[5] Karl Lehenbauer, Mark Diekhans, “Extended Tcl (TclX)”.