Objective-Tcl: An Object-Oriented Tcl Environment

Pedja Bogdanovich

TipTop Software
P.O. Box 30681
Bethesda, MD 20824
pedja@tiptop.com

Abstract

Objective-Tcl is an object-oriented extension to Tcl modeled after the Objective-C extension to the C language. The Objective-Tcl system facilitates sending messages to Objective-C objects from the interpreter without the need for any interface specification. The information stored in the Objective-C runtime system is used. In addition, Objective-Tcl facilitates defining classes and categories (collections of methods) in the Objective-Tcl interpreter. It is transparent to the runtime system if a method is implemented in Objective-C or in Objective-Tcl. In fact, a class can have a mixed implementation—certain methods can be implemented in Objective-C, while other methods can be implemented in Objective-Tcl. For Tcl, Objective-Tcl provides a seamless integration with the low-level compiled language (i.e., Objective-C), as well as a dynamic object-oriented environment which promotes more structured programming in Tcl. For Objective-C, Objective-Tcl provides an interactive development environment which promotes rapid prototyping.

1. Introduction

Tcl [19] is a popular *embeddable* and *extendable* interpretive (scripting) language. Embeddable means that the interpreter can be used as a subroutine package in a low-level language (e.g., C) application. This way the application is extended by providing an interactive interpreter which may be called as desired.

In addition, Tcl itself can be extended by registering commands implemented in a low-level compiled language (e.g., C or C++; we will simply use C in the rest of the paper to refer to the low-level language) with the Tcl interpreter. The interpreter calls back the C-implemented functions corresponding to the registered commands. Functions implemented in the low-level language which are available from the Tcl interpreter are commonly referred to as *C-callouts*.

Tcl's embeddability and extensibility are often employed to build a hybrid application. Parts of the application are implemented in a low-level compiled language (C) for greater efficiency, while other part are implemented in a high-level language (Tcl) as a set of scripts. The Tcl interpreter is typically used to provide high-level control over a program.

One of the problems with using the C-callouts from Tcl is that it is tedious and error-prone to implement them. The programmer must write an interface function for each C-function which is to be used within to Tcl. The interface function essentially is a parsing function—it converts each argument from its Tcl string representation to the appropriate type for the C function. In addition, the interface function must be registered as a Tcl procedure in order to be available from Tcl. There are systems to automatically generate these interface functions such as [4, 15]. Nevertheless, the integration of the low-level and high-level language is not seamless since this "glue" interface code has to be generated and maintained.

In this paper we present an extension to Tcl called *Objective-Tcl*. It solves the C-callout problem: no interface is needed at all. More generally, Objective-Tcl provides a dynamic object-oriented environment in Tcl. Objective-C objects can be sent messages from Objective-Tcl and classes and categories (collections of methods) can be defined in Objective-Tcl. There is no distinction between classes defined in Objective-C and Objective-Tcl. When a method implemented in Objective-Tcl is invoked from Objective-C, it is transparent to the the caller that the implementation is in Objective-Tcl.

Basically, we have added a set of new commands to Tcl to provide the desired functionality and interface

with the Objective-C runtime system. The most important of these commands are described below.

Why Objective-C?

Objective-C [6, 17] is a compiled object-oriented language. It provides an object-oriented paradigm very similar to Smalltalk [7]. Objective-C has a dynamic runtime system and Smalltalk-like method/message syntax. Whereas Smalltalk programs can be an order of magnitude slower [3] than conventional C programs, Objective-C is much faster than Smalltalk since it is compiled, and since not everything is an object, i.e., primitive C types are still used and thus the overhead of messaging is avoided for many simple operations.

Portability, good performance, widespread use, and the availability of an enormous amount of C code directly usable in Objective-C programs [11] are some of the reasons why Objective-C is attractive. In addition, Objective-C is the language of choice for the OpenStep initiative of NeXT Computer, Inc., Sun Microsystems, Hewlett Packard Company, and Digital Equipment Corporation. These vendors cover 68% of the Unix operating system market. In addition, OpenStep for Microsoft Windows NT and Windows 95 has been announced [16]. The main goal of the OpenStep initiative is "the creation of an open, portable standard for object-oriented computing" [18].

Objective-C is a *dynamic* object-oriented language. Dynamic object-oriented languages keep class information at runtime. Class information includes information about instance variables and methods. This is opposed to static object-oriented languages, such as C++, where this information is available only at compile time. This dynamic runtime information maintained by the Objective-C runtime system provides flexibility at the cost of a small performance penalty in method dispatching. It also makes it possible to seamlessly integrate Tcl with Objective-C.

One of the problems associated with Objective-C is that the developer still has to go through the code-compile-link-debug development cycle. Even so, an object-oriented design coupled with the availability of powerful object kits significantly speeds development time (speedup of 5–10 times is typical for Objective-C development under NEXTSTEP compared to a procedural approach) [1]. Nevertheless, the lack of immediate feedback significantly distracts the programmer, instead of allowing him or her to concentrate on the task at hand. Immediate feedback is important for exploratory programming and rapid prototyping. From the Objective-C perspective, Objective-Tcl is designed to solve this problem by providing an interactive development environment which promotes rapid prototyping by providing the ability to access Objective-C objects from the interpreter, as well as the ability to dynamically create classes and categories (collections of methods) at runtime. (Although Smalltalk, SELF, and many other systems have interactive development environments, no such system exists for Objective-C.)

Why Tcl?

Nowadays, there are many interpretive languages available such as Python [25], Scheme [5], Rush [22], Perl [26], etc. We have chosen Tcl for the following reasons:

1. Tcl is simple; e.g., no data types—"everything is a string" type model.
2. It is easy to extend and embed within programs.
3. It is portable.
4. It is well supported, well-documented, and the source code is readable and understandable.
5. It is flexible enough and contains sufficient support needed (e.g., variable traces).
6. It is stable; most of the bugs have been eradicated by the current version v7.3.
7. It is *not* object-oriented, i.e., it does not enforce its view of an object-oriented world.
8. There are many extension packages available, such as *expect* [10], *Tcl-DP* [24], *TclX*, etc.

For example, whereas Python provides enough support to implement an object-oriented extension modeled after Objective-C (i.e., Objective-Python), Python is not appropriate since it imposes its own view of an object oriented world. Implementing Objective-Python would result in an inelegant object-oriented system with two object-oriented paradigms mixed in.

The most notable drawbacks of Tcl are:

1. There is no compiler (although this issue is being addressed [21]).
2. Tcl is relatively slow.
3. It uses dynamic scoping and there is no pass-by-reference (pass-by-name only).
4. Since there is only one namespace, Tcl does not scale well.

Other interpretive languages such as SCHEME [5] could have been used as the underlying interpreter. In our opinion, the advantages of Tcl outweigh the drawbacks. Nevertheless, our extension is implemented portably so that it could easily ported to be embedded in any sufficiently extendable interpretative language.

Related Work

`[incr tcl]` [13, 14] is a popular object-oriented extension to Tcl modeled after C++. It provides a structured programming environment by facilitating C++-like classes to be defined in Tcl. It does solve the namespace problem in Tcl. However, `[incr tcl]` does not address the issue of automatic C-callouts at all. There is no correspondence between classes and methods implemented in `[incr tcl]` and classes and methods in C++.

Object Tcl (Otcl) [23] is another object-oriented Tcl extension modeled after C++. Unlike `[incr tcl]`, Otcl does address the C-callout problem. Otcl supports binding C++ classes so that they can be used in Tcl. For each C++ class that is to be exported into Tcl, a class description in the Class Description Language (CDL) must be written, processed, and the resulting C++ file compiled and linked with the target application. It is possible to subclass C++ classes (which are exported this way) in Tcl. In short, Otcl provides integration of C++ and Tcl. However, due to the static nature of C++, the integration is not seamless. CDL files have to be written. Many of the C++ limitations show up. For example, if an Otcl class does not inherit from a C++ class, it cannot be passed into C++; messages that can be sent from C++ to an object defined in Otcl, are limited to the methods defined in a C++ subclass from which the Otcl class inherits.

CASTE [2] is a Tcl class system modeled after CLOS. It facilitates the creation and manipulation of objects, and provides an object-oriented class mechanism with the inheritance of slots and methods. It is another object-oriented extension to Tcl, suffering the same limitation with respect to the integration with the low-level language as `[incr tcl]` does.

Object Tcl [27] is another object-oriented extension to Tcl with a different object-oriented paradigm resembling CASTE and SELF in some aspects. Object Tcl does not address the problem of integration with a low-level language.

Dish [20] is a Tcl extension which facilitates messaging Fresco objects from Tcl. Fresco operations are specified in the CORBA IDL. Dish uses the IDL specification to automatically coerce method arguments from Tcl strings into appropriate types. The CORBA dynamic invocation mechanism is used. There is no support for defining classes.

The Tcl/Objective-C Library [12], distributed under GNU copyleft, provides the basic facility for sending messages to Objective-C objects from Tcl. The information stored in the Objective-C runtime system is used. Only a subset of primitive Objective-C types is supported. There is no support for defining classes.

Summary

The rest of this paper is organized as follows. The Objective-Tcl runtime system consists of the system for sending messages to objects from Objective-Tcl (described in section 2) and the system for defining classes in Objective-Tcl (described in section 3). The other runtime support is described in section 4. Finally, section 5 contains concluding remarks. Appendix A presents a brief overview of Objective-C.

2. Sending Messages to Objects

The Objective-Tcl runtime system facilitates messaging Objective-C objects from an instance of an Objective-Tcl interpreter. We have extended Tcl with a message "statement" which allows the programmer to send messages to Objective-C objects. The message statement syntax is similar to the syntax used in Objective-C:

Objective-C	Objective-Tcl
`[obj msg]`	`$obj msg`
`[obj msg:a1]`	`$obj msg: $a1`
`[obj msg:a1 with:a2]`	`$obj msg: $a1 with: $a2`

The only Tcl data type is a character string. When a message is sent from Objective-Tcl to an object, all Tcl arguments are converted into appropriate Objective-C types. The returned value is converted back into a Tcl string. If an object is returned, the runtime system takes care of registering the object with Objective-Tcl so that messages can be sent to the object.

Registering Objects

In Objective-C, objects are represented by `id` pointers. In Objective-Tcl, objects are represented by strings (object names). In order for an object to be messaged from Objective-Tcl, it must be registered with the Objective-Tcl runtime system. Registering

an object with the runtime system establishes a two-way mapping between the object's id and a unique string (Tcl object name) representing the object. In the example below, `@Object@000cb84` is the unique Tcl object name used to represent the newly allocated and initialized object. All class (and metaclass) objects are always registered. Each class object is simply represented by its class name.

As mentioned before, when a message is sent to an object in Objective-Tcl, if an object is returned as the result of the method invocation, it is automatically registered. On the other hand, when a registered object is deallocated, either from Objective-C or Objective-Tcl, it is automatically unregistered from the Objective-Tcl runtime system. This provides a degree of safety in Objective-Tcl, in that the runtime system will catch attempts to send messages from Tcl to deallocated objects, although, in general, there is no automatic memory-management (i.e., garbage-collection) in Objective-C.

Here is an example ('%' is the Objective-Tcl prompt):

```
% set obj [[Object alloc] init]
@Object@000cb848
% $obj class
Object
% $obj respondsTo: isEqual:
1
% $obj free
nil
% $obj class
invalid command or object name "@Object@000cb848"
```

In our implementation, objects being messaged can be either local or distributed (remote) objects [17]. Distributed objects can live within the same process, or, more importantly, within another process, possibly running on a different host machine with a different architecture and a different operating system.

3. Defining Classes and Categories

The other side of the Objective-Tcl runtime system consists of the mechanism and language extensions for defining classes and categories in the Objective-Tcl interpreter. Here too, the design is guided by the one-to-one correspondence principle—class, category, and method definitions are equivalent to their Objective-C counterparts. When an object is being messaged at runtime, it is transparent (and irrelevant) both to Objective-C and to Objective-Tcl whether the method is implemented in Objective-C or in Objective-Tcl.

This functionality is provided by the following four new Tcl commands:

class: The `class` command is used to declare and define a new class object:

```
class clsname superclass ivars methods
```

The `class` command corresponds to a set of `@interface/@implementation` declarations in Objective-C. `ivars` is the list of instance variables (types and names), and `methods` is the list of method definitions for the class being defined. See the method command below. An example of the class command is given in figure 1.

category: The `category` command is used to add and/or override methods of an already-defined class:

```
category clsname categoryname methods
```

An example of the `category` command is given in figure 2, where methods are added to the `Object` class.

method: The `method` command is valid only within class and category `methods` lists, and it defines a method implementation. (Unlike in Objective-C, methods need not be declared.)

```
method methodname+args... body
method rettype methodname+args... body
```

If the return type is `id`, it can be omitted. The `methodname+args` is a list consisting of method name components and arguments (type and name) following the notation used in Objective-C. Instance method names begin with a '`-`', while class (factory) methods begin with a '`+`'. If the argument type is `id`, it can be omitted.

When a method is invoked, variables `self` and `_cmd`, as well as all the instance variables of the receiving object are accessible as local variables within the method body.

super: Just like in Objective-C, super refers to the object that performs the method. Unlike `$self` which is a local variable, `super` is a term that substitutes for `$self` only as the receiver in a message statement. Sending a message to super invokes a method defined farther up the inheritance hierarchy.

```
super message...
```

Messages to super are only allowed in a method body.

A sample Objective-Tcl session is given in figure 3.

It could be said that Objective-Tcl introduces data types into Tcl. Types are only used in method prototypes for return and argument types and in instance-variable declarations. All primitive C-types except

Objective-Tcl

```
class MyObject Object {
  {int anInt}
  # id type can be omitted.
  aDelegate
  {STR aStr}
  {double aDouble}
  {SEL action}
} {

  method -setDelegate: d {
    set aDelegate $d
    return $self
  }

  method -setStringValue: {STR s} {
    set aStr $s
    return $self
  }

  method STR -stringValue {
    return $aStr
  }

  method -free {
    # Deallocate aStr
    set aStr 0x0
    return [super free]
   }

  method -setAction: {SEL a} {
    set action $a
    return $self
  }

  method -setInt: {int i} andDouble: {double d} {
    set anInt $i
    set aDouble $d
    return $self
  }

  method void -sendActionToDelegate {
    if [$aDelegate respondsTo: $action] {
      $aDelegate perform: $action with: $self
      # If $action takes one (id) arg:
      # $aDelegate $action $self
      # would work too.
    }
  }
}
```

Objective-C

```
@interface MyObject : Object
{
  int anInt;
  id aDelegate;
  char *aStr;
  double aDouble;
  SEL action;
}

- setDelegate:d;
- delegate;
- setStringValue:(const char *)s;
- (const char *)stringValue;
- free;
- setAction:(SEL)a;
- setInt:(int)i andDouble:(double)d;
- (void)sendActionToDelegate;
@end

@implementation MyObject
- setDelegate:d
{ delegate=d; return self; }

- setStringValue:(const char *)s
{ if(aStr!=s) {
    if(aStr) free(aStr);
    aStr=NULL;
    if(s) {
      aStr=malloc(strlen(s)+1);
      strcpy(aStr,s);
  } }
  return self;
}

- (const char *)stringValue
{ return aStr; }

- free
{ if(aStr) free(aStr); return [super free]; }

- setAction:(SEL)a
{ action=a; return self; }

- setInt:(int)i andDouble:(double)d
{ anInt=i; aDouble=d; return self; }

- (void)sendActionToDelegate
{ if([delegate respondsTo:action])
    [delegate perform:action with:self]; }
@end
```

Figure 1: Example of a class definition in Objective-Tcl and Objective-C. This example demonstrates the Objective-Tcl syntax. In particular it shows how instance variable and method argument types are declared and used.

```
category Object MyMethods {
  # Just like in ObjC, every method has two hidden arguments: self and _cmd.
  method STR +hello { return "Class method: self=$self, method=$_cmd" }
  method STR -hello { return "Instance method: self=$self, method=$_cmd" }
}
```

Figure 2: Example of a Objective-Tcl category definition. Note that methods implemented in Objective-Tcl are added to the Object class which is implemented in Objective-C.

```
% set m [[MyObject alloc] init]
@MyObject@000861b4
% $m setStringValue: "Hello World!"
@MyObject@000861b4
% $m stringValue
Hello World!
% $m isKindOf: Object
1
% $m hello
Instance method: self=@MyObject@000861b4, method=hello
% Object hello
Class method: self=Object, method=hello
```

Figure 3: Sample session using the above Objective-Tcl-defined class and category.

unions and bitfields are supported. This includes structures and arrays.

4. Other Runtime Support

Multi-Interpreter Support

Tcl is not thread-safe. As a consequence, Objective-Tcl is not thread-safe. However, Objective-Tcl allows multiple interpreters to be used within a single single-threaded process. When there is more than one interpreter, the following issues arise:

First, consider defining a new class (or category) in an interpreter. The new class/category is then available with every other interpreter instance as well as from Objective-C, since there is *one* common Objective-C+Objective-Tcl runtime system. This is unlike Tcl procedure definitions whose scope is the interpreter instance in which they are defined.

Second, when there is more than one interpreter instance running and a method implemented in Objective-Tcl is invoked from Objective-C, it is not clear which interpreter instance the method should be executed in. The Objective-Tcl system allows the user to associate an interpreter instance with individual objects and classes. The user can also specify the *default* interpreter. The *current* interpreter is the interpreter from which the most recent message is sent. In our implementation, the method is executed in:

1. the interpreter associated with the receiving object, if non-NULL; otherwise:
2. the interpreter associated with the receiving object's class, if non-NULL; otherwise:
3. the *current* interpreter, if non-NULL; otherwise:
4. the *default* interpreter, if non-NULL; otherwise:
5. an exception is raised.

Error System

If a method implemented in Objective-Tcl generates an error when invoked, an exception is raised. If the method is invoked from Objective-Tcl, the appropriate Tcl error code will be returned as the result of the invocation.

Debugging Support

The Objective-Tcl system includes the Tcl debugger [9] which may be used to debug interpreted code. We have enhanced the debugger so that if a Tcl error occurs and there is no `catch` in progress, the debugger is automatically invoked at the innermost (deepest) frame level. This usually allows the user to fix the problem and continue execution of the program. (The default Tcl behavior is to return an error code at the outermost frame level.)

As noted in [9], some problems associated with the current implementation of the debugger are that: (1) there is not full support for multiple debuggers, and (2) there is no line number and filename support. As a consequence it is not possible to build a graphical interface for the debugger.

With respect to the compiled code, the Objective-Tcl system (under NEXTSTEP) provides support for *attaching* the GNU debugger `gdb` to a program that is already running. For example, this functionality is used so that, if the program crashes, the program is automatically attached to `gdb`. This way, the user may be able to fix the problem and continue with the program execution.

Dynamic Loading and Autoloading

In general, Objective-Tcl programs are mixtures of compiled and interpreted code. In addition to class libraries which are linked into the main executable and the interpreted Objective-Tcl class/category

definitions, separately compiled Objective-C code can be loaded at runtime. The `objtcl_load` command loads the specified modules and links the compiled classes and categories from the modules into the runtime system.

```
objtcl_load module ?module ...?
```

The modules are object files (i.e., compiled Objective-C classes and categories) with the relocation information preserved. Note that this is unlike the dynamic Tcl loading system described in [8] which is used to load Tcl extensions such as *expect*, *TclX*, etc.

Dynamic loading allows, for example, to load a compiled class which is a subclass of an interpreted class. As mentioned before, Objective-Tcl is seamlessly integrated with Objective-C. It is not possible to distinguish if a class in the system was originally defined in Objective-C and compiled, or defined in Objective-Tcl! However, although it is transparent to the messaging system if a method body is compiled or interpreted, it is possible to ask if a method is implemented in Objective-Tcl and obtain the method body.

To further support dynamic class loading, the Tcl autoload mechanism is extended to support the loading of Objective-Tcl-defined classes *and* compiled classes (`.a`, `.O`, or `.so` files). If an undefined class is referenced, either from Objective-Tcl or from Objective-C, the appropriate Objective-Tcl class definition will be loaded using the `source` command, or an object module containing the compiled class will be linked with the runtime system using the `objtcl_load` command.

Object Persistence and Typed Streams

One of the important aspects of object-oriented programming is object persistence. In Objective-C, *Typed Streams* are used to archive objects. To support object archiving, commands for opening and closing typed streams, and commands for reading and writing primitive Objective-C types from and to typed streams are provided.

Specifically, the `TTObjTclInterp` class, which is an object wrapper around `Tcl_Interp*`, can be archived. When an interpreter instance is archived, the interpreter state is saved. In the current implementation, only globals and procedures are saved. Tcl stack frame, variable traces, etc. are not saved. When an interpreter object is dearchived from a typed stream, it is restored to its original state.

Interactors and Shells

The Objective-Tcl system introduces the concept of *interactors*. An interactor is an object which "asks" an interpreter to evaluate some Objective-Tcl text; i.e., an interactor invokes an `eval` method of an Objective-Tcl interpreter object. On the other side, interactors typically take user keystrokes, pack keystrokes into text strings to be evaluated, and present results of evaluation back to the user.

Currently, two interactors are implemented: the *TTY* and the *AppKit* interactor. The TTY interactor simply reads text from the standard input, evaluates it, and prints evaluation results. The *local* Objective-Tcl shell, `objtclsh` which is the equivalent of `tclsh`, simply creates an instance of Objective-Tcl interpreter and invokes a TTY interactor on it.

The AppKit interactor is a graphical interactor which opens a window and runs from the application event loop, which is appropriate for graphical programs.

The *remote* (distributed) Objective-Tcl shell, called `dosh`, uses the TTY interactor just like `objtclsh`. However, whereas `objtclsh` creates an interpreter, `dosh` connects to an interpreter within an already-running application and uses the TTY interactor to send evaluation requests to the interpreter within the host application.

The capability to automatically export an Objective-Tcl interpreter as a distributed object server is provided in the system. This allows one application (e.g., such as `dosh`) to connect to an interpreter within another already-running application, and directly interact with the objects within the remote application through the interpreter.

InterfaceBuilder Support

For NEXTSTEP systems, Objective-Tcl provides InterfaceBuilder (IB) support. IB is a NEXTSTEP/OpenStep "visual programming" tool. It facilitates building an interface graphically from already available (compiled) objects, rather than writing Objective-C code. With IB, objects can be dragged from palettes directly into the application being built. Once there, an object can be modified in ways that are specific to its class. Objects are hooked together so that they can communicate with one another. For example, a button can be connected to the window it appears in, so that when the button is clicked, the window closes.

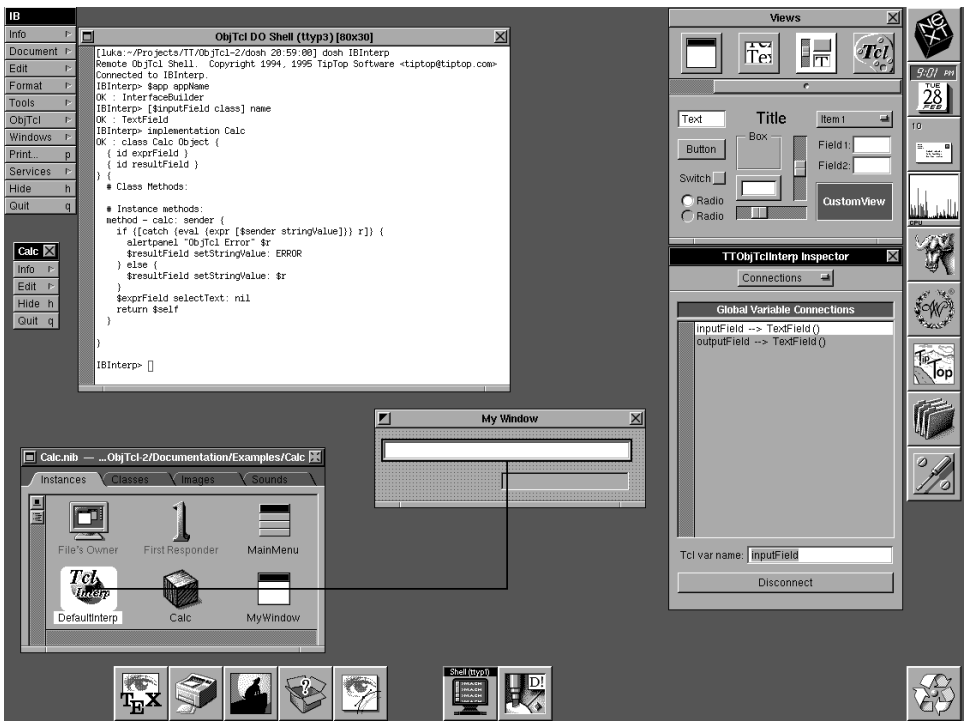

Figure 4: Sample Interface Builder session. Global variable `inputField` is connected to the text field object. The text window is running `dosh`, a simple shell process connected to the Objective-Tcl interpreter *within* the Interface Builder.

In many ways, using IB to create an application is much like using a graphics editor to create a drawing. However, when one builds an application with IB, one is interacting with the actual (compiled) programming code that will be executed when the application runs on its own. The objects manipulated in IB *are* the objects that will appear in the working version of the application. The application being built can be tested (run) within IB even before a single line of code is written.

The result of using IB is a file which contains archived versions of the objects assembled for the application, information about connections between these objects, and other information. When the application begins running, these objects and associated information are unarchived from one or more of those archive files.

IB is designed to be used with a compiled language, i.e., Objective-C. Objective-Tcl's IB support consists of several method categories which facilitate Objective-Tcl classes being defined and used directly within IB.

In addition, a palette containing an Objective-Tcl interpreter object is provided. An interpreter object can be added to an application by simply dragging it from the Objective-Tcl palette. Once an interpreter object is added into the application, one can connect any object in the application as an Objective-Tcl global variable in the interpreter. At runtime, the connected variables will point to the corresponding objects.

The IB support provided in Objective-Tcl augments the power of IB. It allows rapid prototyping and exploratory programming by giving immediate feedback for all programming actions. Classes and categories can be defined and tested interactively within IB, without the need for (re)compilation.

Object Browser

ObjectBrowser (OB) is a complimentary graphical programming tool to InterfaceBuilder. OB is used to view objects in many different ways. For example, methods, instance variables, and other object specific information of an object can be viewed and edited. OB is useful for development of Objective-C/Tcl programs. It allows the programmer to easily investigate the structure and methods of various objects and classes.

To a user, OB appears as a hierarchical browsing tool, similar to the file system browser. Each node

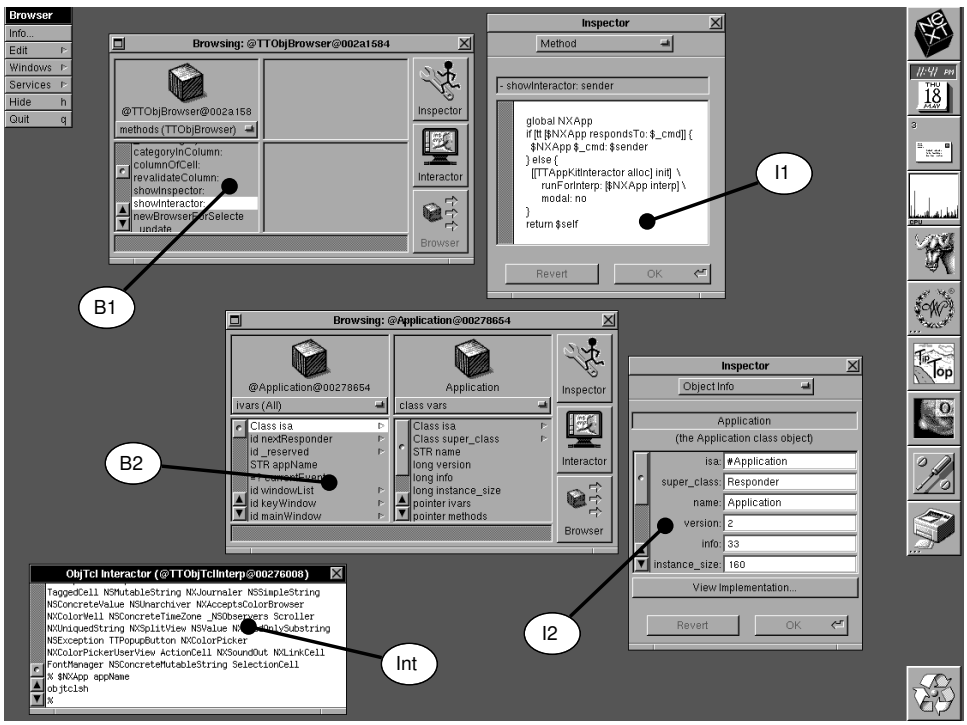

Figure 5: Object Browser example. Method `-showInspector:`, which is a subnode of a `TTObjBrowser` object, is selected in the browser window (B1). The corresponding inspector (I1) is showing the method body which can be changed directly in the inspector. The browser window (B2) is showing instance variables of an `Application` object in the first column. The `isa` instance variable is selected; it points to the `Application` class object in the second column. The second column and the corresponding inspector (I2) are showing the variables of the Application class objects. The window (Int) is an AppKit interactor used to type in commands to the Objective-Tcl interpreter.

in the browser typically represents an object or some other piece of information (e.g., a method). Each node can have any number of subnodes. For example, when looking at an object, instance variables or methods appear as subnodes; when looking at a `List` object, contents of the list appear as subnodes; when looking at an interpreter object, procedures and global variables appear as subnodes; etc.

In addition, each object can have a number of *inspectors* associated with it. Inspectors allow browser nodes to be viewed in some specific way as well as to be edited. For example, to view an implementation of a method, the method is simply selected in the browser. It then can be edited in the method inspector. This is a convenient aid to development. OB is extensible. The user can provide object-specific browser nodes and custom inspectors for any object in the system.

OB is part of the standard Objective-Tcl library, and hence, it can be used from any application. It is dynamically loaded when needed.

OB was completely implemented in Objective-Tcl. However, for performance reasons, some of the most heavily used classes have been translated into Objective-C and compiled. OB is an excellent example of Objective-Tcl flexibility and utility. While designing OB, Objective-Tcl allowed us to easily experiment with many different prototypes, and to easily change design as development progressed.

5. Concluding Remarks

Objective-Tcl is an object-oriented environment for Tcl. The Objective-C runtime system which supports compiled objects is augmented with an object-oriented extension to Tcl modeled after Objective-C. The system provides the seamless integration of Objective-C and Tcl, both from Tcl and Objective-C viewpoints.

Our Objective-Tcl implementation is portable. It has been extensively used under NEXTSTEP for Motorola, Intel, HPPA, and SPARC architectures. Objective-Tcl also runs with the GNU Objective-C

runtime and has been tested and used under SunOS 4.1 (Sun SPARC) and Dec OSF/1 (Dec Alpha). Although Objective-Tcl has not yet been publicly released, it has been commercially used at a number of sites such as the Union Bank of Switzerland. Objective-Tcl with *expect* extensions is used as a script interpreter in *TipTop*, a commercial telecommunication and terminal emulation application for NEXTSTEP [28].

The main problem with the current Objective-Tcl system is that the compiler and the interpreter are in fact different languages. Since currently there is no Tcl compiler [21], it is not possible to compile Objective-Tcl code to obtain highly optimized compiled code. This implies that, if performance is critical, after a set of classes has been designed in Objective-Tcl, the classes must be translated into Objective-C and compiled. Tools to help do this translation are provided. However, since Tcl and Objective-Tcl are usually used to provide high-level control over a program, our real-world experience shows that this translation is rarely necessary.

Availability

Objective-Tcl is available from TipTop Software. See `http://www.tiptop.com/` for more information.

Acknowledgments

Thanks to Hadar Pedhazur, Gary Knott, Claudio Esperanca, Glenn Pearson, and Vojislav Lalich-Petrich for critiquing this work and providing suggestions that greatly enhanced the readability of the paper.

A. Overview of Objective-C

Objective-C is a layer on top of C [6, 17]. It supports classes and message passing paradigms similar to Smalltalk. Describing Objective-C in detail is outside the scope of this paper. Here we give only a brief overview of Objective-C, which is based on the Objective-C FAQ in `http://www.yahoo.com/Computers/Languages/Objective_C/`. The main characteristics of Objective-C are:

- It is compiled.
- It has a dynamic runtime system (objects are dynamically typed). Full type information (name and type information of methods and instance variables and type information of method arguments) is available at run time.
- Classes and methods can be added (dynamically loaded) at runtime.
- There is one root class Object from which all other classes inherit. (This is not quite true; other root classes can be defined, e.g., NSProxy in OpenStep.)
- Method/message syntax is similar to Smalltalk.
- Unlike in Smalltalk, there is no garbage collection. Instead, reference counting pseudo-garbage collection techniques are typically used.

Objective-C introduces a few keywords and syntactic constructs into C:

@interface *declares* a new class. It specifies class and superclass names, instance variables (types and names) and declares the method prototypes implemented by the class.

@implementation *defines* a class. The implementation is a collection of method definitions.

@category is a named collection of methods which are being added to an *existing* class. A category may redefine existing methods.

id is a predefined type. An `id`-type variable is a pointer to *some* object. The actual class of the object being pointed to is not known at compile time but it is known at run time.

`-`*message*; declares a method called *message*. The '`-`' indicates that the message can be sent to objects. A '`+`' instead indicates the message can be sent to class objects. A method can take arguments and return a value.

```
[obj message],
[obj message:arg1],
[obj message:arg1 with:arg2]
```

These statements are examples of sending messages to object `obj` (i.e., invoking the corresponding methods) with 0, 1, and 2 arguments respectively. The name of the message is called the *selector*. In this example, the selectors are `message`, `message:`, and `message:with:` respectively.

References

[1] Booz-Allen & Hamilton Inc. NeXTSTEP vs. other development environments, Jan. 1992.

[2] Michael S. Braverman. CASTE: A Class System for Tcl. In *Tcl/Tk Workshop Proceedings*, pages 39–44, Berkeley, California, June 1993.

[3] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *OOPSLA '91 Proceedings*, Phoenix, AZ, October 1991.

[4] Wayne A. Christopher. Writing Object-Oriented Tcl-based Systems Using Objectify. In *Tcl/Tk Workshop Proceedings*, pages 99–101, Berkeley, California, June 1993.

[5] William Clinger and Jonathan Rees. The Revised Report on the Algorithmic Language Scheme. *Lisp Pointers IV*, July–Sept 1991.

[6] Brad J. Cox and Andrew J. Novobilski. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, Massachusetts, 1991. ISBN: 0-201-54834-8.

[7] Adele Goldberg and David Robson. *Smalltalk-80: The Language.* Addison-Wesley, 1989. ISBN 0-201-13688-0.

[8] Kevin B. Kenny. Dynamic Loading for Tcl: (What became of it?). In *Tcl/Tk Workshop Proceedings*, 1994.

[9] Don Libes. A Debugger for Tcl Applications. In *Tcl/Tk Workshop Proceedings*, pages 3–19, Berkeley, California, June 1993.

[10] Don Libes. *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Applications.* O'Reilly and Associates, Inc., December 1994. ISBN: 1-56592-090-2.

[11] Christopher Lozinski. Why I need Objective-C. *Journal of Object-Oriented Programming*, pages 21–28, September 1991.

[12] Andrew McCallum. Tcl/Objective-C Interface Library. `ftp://ftp.cs.rochester.edu/pub/packages/objc/libtclobjc-1.0.tar.gz`.

[13] Michael J. McLennan. [incr tcl] – Object-Oriented Programming in TCL. In *Tcl/Tk Workshop Proceedings*, pages 31–38, Berkeley, California, June 1993.

[14] Michael J. McLennan. The New [incr tcl]: Objects, Mega-Widgets, Namespaces, and More. In *Tcl/Tk Workshop Proceedings*, Berkeley, California, July 1995.

[15] John Menges and Brian Ladd. Tcl/C++ Binding Made Easy. In *Tcl/Tk Workshop Proceedings*, 1994.

[16] NeXT Computer, Inc. NeXT Announces OpenStep for Windows NT and Windows 95. `http://www.next.com/Info/PR/Fin_OpenStep.022195.html`.

[17] NeXT Computer, Inc. *NEXTSTEP Object Oriented Programming and the Objective-C Language.* Addison-Wesley, Reading, Massachusetts, 1993. ISBN: 0-201-63251-9.

[18] NeXT Computer, Inc. and SUN Soft. OpenStep and Solaris white paper. `http://www.sun.com/sunergy/Papers/Papers10/openstep.solaris.wp.html`.

[19] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, Massachusetts, 1994. ISBN: 0-201-63337-X.

[20] Douglas Pan and Mark Linton. Dish: A Dynamic Invocation Shell for Fresco. In *Tcl/Tk Workshop Proceedings*, 1994.

[21] Adam Sah and Jon Blow. A Compiler for the Tcl Language. In *Tcl/Tk Workshop Proceedings*, pages 20–26, Berkeley, California, June 1993.

[22] Adam Sah, Jon Blow, and Brian Dennis. An Introduction to the Rush Language. In *Tcl/Tk Workshop Proceedings*, June 1994.

[23] Dean Sheehan. Interpreted C++, Object Oriented Tcl, What next? In *Tcl/Tk Workshop Proceedings*, Berkeley, California, July 1995.

[24] B. C. Smith, L. A. Rowe, and S. Yen. Tcl Distributed Programming. In *Tcl/Tk Workshop Proceedings*, pages 50–51, Berkeley, California, June 1993.

[25] Guido van Rossum. An Introduction to Python for UNIX/C Programmers. In *Proceedings of the NLUUG najaarsconferentie*, 1993.

[26] Larry Wall and Randal Schwartz. *Programming Perl.* O'Reilly and Associates, 1992. ISBN: 0-937175-64-1.

[27] David Wetherall and Christopher J. Lindblad. Extending Tcl for Dynamic Object-Oriented Programming. In *Tcl/Tk Workshop Proceedings*, Berkeley, California, July 1995.

[28] Rob Wyatt. Top Notch Communications. *NeXT In Line*, 1(3):21–22, Winter 1994.