Cite as:
Roseman, M. (1995). When is an object not an object? Proceedings of the 1995 Tcl/Tk Workshop, Toronto, July 6-8, Usenix Press.

When is an object not an object?

Mark Roseman
Department of Computer Science, University of Calgary
Calgary, Alberta, Canada T2N 1N4 (403) 220-3532
roseman@cpsc.ucalgary.ca

Abstract

This paper describes an approach to designing and building Tcl extensions that can be flexibly and dynamically changed using either Tcl or C. In particular, extensions having an object-based interface are considered. This extension approach seeks to avoid the "chasm" found in migrating code from Tcl to C as it matures by freely mixing Tcl and C to create an object's subcommands. The approach differs from traditional Tcl object frameworks in that it retains familiar mechanisms used to create new toplevel Tcl commands, and emphasizes extensions holding much of their data at the C level. A secondary goal is to illustrate how extension authors can encourage rich customization, by exposing object internals to change. To illustrate the technique, a simple data structure is extended to support sharing between multiple Tcl processes.

Introduction

The success of Tcl has been founded on its ease of extension, leading to a large number of high quality extensions. The core Tcl interpreter provides a single mechanism — the library function Tcl_CreateCommand — for extending the interpreter, whether directly to add a new command written in C, or indirectly, using "proc" to define a new command written in Tcl. While providing sufficient power to provide for extensions, the explosion of extension frameworks suggests the core mechanism does not necessarily make it easy to create the sorts of extensions developers want.

In particular, the core mechanism provides no support for the ubiquitous "object-oriented" style found in extensions such as Tk, where a single Tcl command supports many methods or subcommands. Many of the object oriented extension architectures — such as [incr Tcl] — have arisen to make it easier to add new subcommands to objects. These approaches tend to result in objects which are used like other Tcl objects, but creating these objects is fundamentally different from creating toplevel Tcl commands. These architectures also provide limited support for creating extensions that are implemeted primarily at the C level.

This paper describes a complementary approach, which leverages the familiarity and simplicity of the Tcl_CreateCommand approach to defining object oriented commands. While systems such as [incr Tcl] tend to emphasize extensions whose data and operations are primarily implemented at the Tcl level, the approach here is more suitable for extensions whose implementations are primarily at the C level. The goal here is to allow users of extensions — even C-based ones — to customize them as easily as they would [incr Tcl] extensions, using either C or Tcl, and in a manner analogous to creating toplevel Tcl commands. The lessons here are also applicable to [incr Tcl]-type extensions, suggesting how to design objects for easier customization.

This is not presented as an extension itself — the paper will argue why this is a poor idea — but rather as a set of guidelines or considerations to keep in mind when writing extensions. Keeping in mind that many people find "meta" to be a four letter word, the paper is heavily grounded in a particular extension which uses the techniques described here.

Terminology

A quick note on my use of the word "object" is appropriate here. As alluded to in the title, there are differing uses of this term. Throughout the paper, "object" and "object oriented" will normally refer to objects in the Tcl sense [4], such as found in Tk. One toplevel Tcl command (e.g. "button") creates an object, which results in the creation of another Tcl command (e.g. ".mybutton") having subcommands (e.g. "invoke") which perform operations on the object. I'm not assuming anything about an object-oriented system in the C++ or Smalltalk sense, where we have other properties such as inheritance.

Motivation

This section examines some relevant work to better understand the approach taken in this paper. Current Tcl object frameworks are briefly examined to highlight their strengths and weaknesses. A case is made that the transition from Tcl to C code is far too difficult and heavyweight at the moment. Finally, some work in computational reflection suggests some promising strategies for exposing the internals of objects to customization.

Why Allow Users to Customize Extensions?

A basic tenet of this paper is that there are often cases where it is valuable to customize an extension. That no extension will suit every need should be obvious. Often however a user's needs can be better met by small changes or additions to an existing extension rather than coding from scratch. I would argue that extension authors can actively support this sort of customization, rather than leaving users with the often grim prospect of mucking in the extension's source code.

Here are a few examples of customization scenarios:

- Can I change how [incr Tcl] inherits objects?
- Can a certain data structure be made persistant?
- Dump a Tk canvas as a GIF rather than Postscript?
- Have Tcl-DP use a different transport protocol?
- Make Tk work on a Mac?

The traditional approach is all too often to either toss out or severely modify the extension's internals. The approach here suggests that with good design and open implementations, it may be possible to anticipate and provide for such future changes and new uses.

Traditional Object Extensions

A number of object extensions have appeared over the last couple of years as a way to create object oriented Tcl commands. For reasons of brevity, I'll focus on Michael McLennan's excellent [incr Tcl] system as an exemplar of the approach [3].

The system is modeled after the style of objects found in C++, where object classes are defined, containing state information (fields or instance variables) and behaviors (methods or member functions). Classes can inherit (subclass) from other classes, allowing changes or extensions to be made to a class. Once a class is defined, instances of the class can be created containing the fields and methods described by the class. Classes, their fields and methods, are described by Tcl scripts.

The advantage to these frameworks is the added structure they add to Tcl programs, which is necessary for building larger programs. The frameworks add methods or subcommands through the class definition mechanism, which is somewhat more heavyweight than the traditional method of defining new Tcl commands. While still not preventing very interactive prototyping, it can be more difficult. As well, the frameworks do promote a different mental model of a program than found in standard Tcl.

Current Tcl object frameworks emphasize systems whose data and methods are usually stored at the Tcl level, though some, such as the newest version of [incr Tcl], support defining methods in C. For Tcl-based extensions particularly, they provide a rich means of customization, though implementing it through the "different" paradigm of classes and inheritance.

The Extension Chasm

The other main approach to extensions is of course to "roll your own" using the core mechanisms directly provided by Tcl — define toplevel commands at either the C or Tcl level. C level extensions tend to be better when there is a larger amount of possibly complex data or computation involved. Observations of Tcl development practice suggest that doing a C extension in this manner is particularly heavyweight. Extensions at this level have tended to be notoriously difficult to extend or customize — a point returned to shortly.

Additionally, C level extensions encapsulated in a toplevel "object oriented" command tend to be an "all or nothing" affair, where most, if not all of the extension is done in C at once. As many extensions seem to begin life as Tcl-based prototypes, it can be a daunting task to move from a Tcl-based implementation to a C based one, even when it becomes far too complex to realistically maintain in Tcl. Some of the "megawidget" extensions are good examples of this.

Ideally, one would like to be able to more incrementally move an extension from Tcl to C, as performance and other requirements dictate. It is completely reasonable that an extension — including one providing object oriented commands — could be implemented in a fluid mixture of both Tcl and C, using both as appropriate. This level of integration, analogous to the smooth mixture of C and Tcl toplevel commands, is not found in current extension writing practice.

Open Implementations

There has been a great deal of work in the computational reflection community on better understanding the role of abstraction in software development. This community has been concerned with meta-level architectures in highly dynamic programming languages such as CLOS and Dylan. This section describes some of the tenets held by that community, in the hopes of understanding how to create more flexible, extensible and dynamic extensions in Tcl. Material here draws heavily from Kiczales [2].

Fundamentally, we use abstraction to manage complexity in systems. Under an object-oriented metaphor, objects are the method for representing abstraction. Objects reduce complexity by allowing their clients to deal with only the "necessary" functionality of the object, while hiding the details of the implementation. Conventional views of abstraction hold that a client need not and cannot care about the object's implementation, just its interface.

The reflection community holds that this view of abstraction is not sufficient in practice. Kiczales uses the example of creating a view of a 100x100 cell spreadsheet object, by using the window system to create a sub-window for each cell. Though this is faithful to the abstraction provided by most, this solution is likely to be far too slow on most window systems. Important information about the design of a window system — that it is optimized for a relatively small number of windows — is hidden from the developer using the window system. The developer is forced to "work around" the abstraction to solve the problem, an all too common scenario. The abstraction fails by hiding relevant implementation internals behind its interface, making them inaccessible.

This is not to say that abstraction has no value, but instead that it is important to be able to "get inside" the abstraction when necessary, for performance or other reasons. Rather than defining only a single interface to an abstraction, *two* interfaces can be defined. The second, an *adjustment interface*, allows for examining and changing the internal workings of the object, which are hidden from view under a traditional view of abstraction. While the first interface allows the developer to ignore details of the implementation, the second interface provides a recourse in the case when abstraction breaks down.

Defining this adjustment interface is somewhat complementary to the process of actually implementing it. Object inheritance and subclassing is one method, but subclassing does not ensure a good adjustment interface. Computational reflection is founded on the premises that the design of the implementation should be as modular and well thought out as the design of the interface, and that an adjustment interface should be available to the abstraction's clients to permit examining and changing the implementation when necessary.

Goals of this Work

The remainder of this paper presents an approach to building Tcl extensions having an object-oriented interface that are easily customized by their users. The approach illustrates two main points:

1. An extension whose core data and operations exist largely at the C-level can be changed at least as easily as one implemented using a Tcl-based object framework.
2. Careful design is necessary to ensure an easily customized extension; this applies to extensions written using the approach here as well as with other approaches.

In achieving these goals, the approach describes how to build extensions having the following properties:

1. Objects can be easily extended in either Tcl or C, by adding to or replacing existing methods.
2. Objects can be individually extended at runtime for greater customizing, i.e. extension is instance-based, not class-based.
3. The internals of objects are exposed and may be changed by users of an object, resulting in dramatic changes of behavior.
4. Extending objects should be done in a manner analogous to creating new toplevel commands.

Again, the focus of this work is a set of design principles for building customizable extensions that have an object-oriented interface — it is not itself an object-oriented extension. I believe the range of possibilities this approach applies to are far greater than could be encapsulated with a single extension.

An Example Scenario

To ground this approach in the concrete, the paper draws on an example found in GroupKit, a Tcl extension that helps developers create real-time groupware applications [5]. In real-time groupware, applications run across different machines, permitting, for example, a drawing program to be shared by users across a network, all contributing to a single drawing. In GroupKit, this is accomplished by having copies of the application running on each machine, exchanging messages with each other (using the Tcl-DP extension [6]).

GroupKit uses a data structure called an "environment" (for historical reasons) to keep track of a lot of information such as what users are active in a session and what they are doing. Environments are hierarchical data structures where any node can hold either a value or have other nodes as children. Nodes are referred to by a key, using a "." as a hierarchy delimiter, e.g. "users.5.name". Environments bear a strong resemblance to Extended Tcl's keyed lists, which served as the basis for the earlier implementations.

Environments are a very simple example of an object; they can be created and destroyed, and methods are provided to add, delete and inspect nodes in the environment. Table 1 summarizes some of the operations available in environments. In GroupKit, we wanted to be able to maintain this simplicity while permitting environments to be shared between GroupKit processes. Ideally, just by changing a node in the local copy of an environment, the change would be appropriately propagated to environments in the other processes. Under normal circumstances, application developers should not need to know how this occurs.

We knew from our earlier work [1] that it would not be sufficient to provide a single method for doing concurrency control or replication. A wide variety of choices are possible, and these can have dramatic effects

Operation	Description
gk_env *envName*	create an environment and its command
envName set *node value*	set the value of node in the environment to value
envName get *node*	get the value of a particular node; if the node has children, return a keyed list representing the structure of the subtree rooted at node
envName delete *node*	delete the indicated node, or subtree
envName keys ?*node*?	return the list of direct children of the given node
envName option ?*args*..?	get or set environment options, specified by key and value, the same as the environment's data
envName destroy	destroy the environment and its command

Table 1. Core operations on environments.

on the user interfaces of the highly interactive multi-user applications built with GroupKit. See Table 2 for examples of some of the customization possibilities. A better approach was to provide a core object that could be easily extended by either ourselves or application developers to support different strategies as needed for a particular application.

The Extension Approach

This section describes how to design and build these open and extensible Tcl objects, using the GroupKit environments as an example of one possible implementation.

Object Creation

A single toplevel Tcl command (e.g. "gk_env") is defined in C and registered with the Tcl interpreter. When invoked with an object name, this command performs the following operations:

1. Create and initialize any necessary internal core data structures.
2. Create a table of subcommands (e.g. a hash table) and fill in with a set of default operations.
3. Register a toplevel Tcl command to handle the object's instance command.

Defining Core Data and Operations

Despite the possibility for potentially radical change, a core set of data structures and operations are usually provided. This does two things. First, it defines the base level capabilities of the object, which will likely be shared by all or most extensions made to the object. Second, this provides a default implementation, hopefully suitable for use by a number of extensions.

For environments, the core data structure provided is a *n*-ary tree, where each node holds either a pointer to a value string or a pointer to a linked list of children, themselves nodes. A root tree is created, along with a node for holding a "data" subtree, and a node for holding an "option" subtree. Core operations including those necessary to get/set/delete values in either subtree. These are wrapped into "builtin" handlers for subcommands like "get" and "set" (which may be overridden, see below).

Defining these data structures and primitive subcommands provides the necessary building blocks you'd need for extending environments. Fundamentally, these are the sorts of things any environment will want to do, though it might accomplish these operations in different ways. As well, this allows the possibility of extensions that replace the internal data representation if necessary, by building a new structure and specifying new primitives to replace the builtins.

Subcommand Dispatch

As mentioned above, when an object is created, one of the internal data structures it contains is a table of subcommands. The environment's instance command (analogous to a Tk widget command), searches through this table when an instance command is invoked to find a handler for the subcommand.

Handlers can consist of either a Tcl script or a C function. For the former, a Tcl command string is created by appending the script handler with the name of the environment as well as any additional arguments passed by the caller. The resulting command is then executed via Tcl_Eval. For a handler implemented as a C function, the function is called directly, passing the original arguments, and the environment as the clientData.

If a subcommand does not exist in the table, the subcommand dispatcher looks for the existence of a subcommand named "unknown" and will execute that if present. This allows for extensions such as the "implicit get/set" syntax extension described in Table 2. The subcommand dispatch process is illustrated by the following pseudo-code:

```
cmd = FindHashEntry(env->cmds, argv[1])
if (cmd==NULL)
  cmd = FindHashEntry(env->cmds, "unknown")
if (cmd==NULL) error
if (cmd->type==C_SUBCMD)
  cmd->func(clientData, interp, argc, argv)
else
  Tcl_Eval(interp,
           concat(cmd->script,argv))
```

Defining Tcl Subcommands

Subcommands are specified as normal Tcl scripts and then added to the object. For example, one subcommand that might be built is an "exists" subcommand for environments, which takes a single node key and returns a 1 if a node with that key exists in the environment, and a 0 if not. Assuming a "get" subcommand already exists (one implementation is supplied as a builtin), this subcommand could be defined as follows:

```
proc _gkenv_exists {env cmd key} {
    set result [{catch $env get $key}]
    if {$result==0} {
        return 1   # no error - exists
    } else {
        return 0   # error - doesn't exist
    }
}
```

The following code would be used to add this subcommand to an existing environment (e.g. "myEnv"):

```
myEnv command set exists "_gkenv_exists"
```

This command associates the "exists" subcommand with the Tcl script "_gk_env_exists". When invoked, e.g. via "myEnv exists foo", the _gk_env_exists proc will be called by the environment as follows:

```
_gkenv_exists myEnv exists foo
```

Defining C Subcommands

Subcommands specified in C are defined in exactly the same way that toplevel Tcl commands are defined. The argument list passed to the subcommand handler is copied verbatim from the argument list passed to the toplevel instance command. The clientData parameter holds a pointer to the object's internal representation (i.e. its core data structure described earlier).

We might define the "exists" subcommand from above instead in C as follows. Note that in this and other examples, error checking code has been omitted for purposes of clarity.

Concurrency control	
none (default) *	Changes (e.g. set, delete) affect only the local copy; changes are not reflected in copies of environments in other processes.
no concurrency *	Changes made locally are broadcast to other copies of the environment, but changes from multiple processes may arrive in different places in different orders, leading to inconsistent states [Note: for some groupware cases, this is perfectly acceptable].
centralized server *	All local changes are sent to a central copy of the environment, which serializes the changes (guaranteeing consistency) and sends them back to all copies, at which point changes take effect. [Note: can be substantial time lag depending on network].
locking	A portion of the environment must be locked before making changes, so a lock must be received before a change takes effect. [Note: potentially faster than centralized server if using the same part of the environment multiple times; locks can be implemented using many strategies].
optimistic locking	Like locking, but immediately make the change under the assumption you'll probably get a lock. [Note: must be able to deal later on with having the data revert back to its original value if the lock is denied.]
Notification	
none (default) *	No notification when the environment changes
global handler *	A global (application-wide) handler is called to notify the application that the environment has changed, potentially as a result of operations in a remote environment. [Note: in GroupKit, this uses the same mechanism used by other events].
binding table*	Bind event handlers to the environment directly, in a way similar to how events are bound to Tk widgets. The environment then deals with events directly.
Other	
implicit get/set *	A syntax extension whereby if the environment subcommand is not one of the recognized subcommands, it will attempt to map the command onto a get or set command, which is a very convenient shorthand and useful in prototyping, e.g. "env foo" maps to "env get foo" and "env foo bar" maps to "env set foo bar".
ignore errors *	An extension whereby operations like "get" or "keys" on non-existent nodes in the environment return an empty string rather than an error. [Note: potentially useful for prototyping, and in the case of GroupKit, a way to achieve backwards compatibility with some questionable earlier design choices].

Table 2. Potential extensions of environments. Extensions marked with a "*" have been implemented to date.

```
int GkEnv_ExistsCmd(ClientData clientData,
    Tcl_Interp* interp, int argc,
    char *argv[])
{
  char *newArgs[3]; int result;
  Environment *env =
    (Environment*)clientData;
  Subcommand *subcmd =
        FindSubcommand(env, "get");
  newArgs[0] = argv[0]; newArgs[1] = "get";
  newArgs[2] = argv[2];
  result = ExecSubcommand(env, interp,
              subcmd, 3, newArgs);
  if (result==0)
    Tcl_SetResult(interp, "1", TCL_STATIC);
  else
    Tcl_SetResult(interp, "0", TCL_STATIC);
  return TCL_OK;
}
```

The subcommand would be registered with the environment as follows:

```
Env_AddSubcommand(env, "exists",
       GkEnv_ExistsCmd,NULL);
```

Note that this command simply locates the "get" subcommand in the environment and executes it, completely analogous to the Tcl version. This has the advantage of working even if the implementation of the underlying data structure changes, requiring only the "get" subcommand be reimplemented if the data structure changes. This is generally preferable. However, the "get" operation can be expensive, particularly when retrieving a large subtree which must be converted into a keyed list representation. The following implementation for "exists" could be substituted that uses an internal procedure to get the node of a tree, but would require reimplementation if the underlying data structure changed:

```
int GkEnv_ExistsCmd(ClientData clientData,
    Tcl_Interp* interp, int argc,
    char *argv[])
{
  Environment *env =
     (Environment*)clientData;
  EnvNode *node =
      Env_FindNode(env,argv[2]);
  if (node!=NULL)
    Tcl_SetResult(interp, "1", TCL_STATIC);
  else
    Tcl_SetResult(interp, "0", TCL_STATIC);
  return TCL_OK;
}
```

Manipulating Subcommands

One of the key features of these objects is the ability to manipulate the available list of subcommands. This can be achieved at the C level by changing the Tcl hash table holding the commands (via Tcl library functions, or wrappers like Env_AddSubcommand). At the Tcl level, a "command" subcommand is provided, described in Table 3.

envName command set *subcmd proc* Set the subcommand handler for *subcmd* to *proc*
envName command get *subcmd* Return the Tcl script for the subcommand *subcmd*, or <builtin> for subcommand handlers written in C
envName command delete *subcmd* Remove the subcommand *subcmd* from the object
envName command list List the subcommands in the object
envName command rename *oldcmd newcmd* Register the same subcommand handler for *newcmd* as is registered for *oldcmd*

Table 3. Operation of "command" subcommand.

The "command rename" subcommand is especially useful, since it allows you to wrap an exiting subcommand to provide additional functionality. This is equivalent to extending an inherited method in a full object oriented system. For example, to generate an event when a node is deleted from the environment, the following code could be used:

```
myEnv command rename delete _olddelete
myEnv command set delete notifyDelete

proc notifyDelete {env cmd node} {
    $env _olddelete $node
    generateDeleteEvent $node
}
```

Having the convention of an underscore to preface "internal" commands is useful. Objects can also inspect their own commands to generate new names via "command list". This allows such commands to be composed, as will be illustrated in the later section on packaging objects and extensions.

Note that since objects can manipulate even their built-in commands it is possible to "lock" an object (preventing changes at the Tcl level) with the following:

```
myEnv command delete command
```

Packaging an Object and Default Extensions

It is useful to package together an object and a set of optional extensions. For example, in GroupKit we provide a command called "gk_newenv" which invokes "gk_env" internally. Depending on options passed to "gk_newenv", different commands are added to the environment. For example, application developers can

specify “-notify” or “-share” (or both) for an environment. Typically, sets of changes are packaged together. This section walks through an example.

The following Tcl procedure is the normal procedure invoked to create an environment. It parses through the arguments looking for “-notify” and “-share” flags, and picks out the name of the environment from the end of the argument list. It creates the environment, and then calls routines to add in notification and sharing if the appropriate flags are set.

```
proc gk_newenv {args} {
  set notify no; set share no
  foreach i $args {
    if {$i==“-notify”} {set notify yes}
    if {$i==“-share”} {set share yes}
  }
  set env [lindex $args \
       [expr [llength $args]-1]]
  gk_env $env

  if {$notify==“yes”} \
      {_gkenv_initNotify $env}
  if {$share==“yes”} \
      {_gkenv_initShare $env}
}
```

Next is the “notification” extension, which replaces the set and delete commands with routines that generate events in addition to making the requested changes. Note that this package also adds an internal “_notify” subcommand, which may be used by other extensions to do notifications if desired. These extensions can detect the presence of this subcommand by inspecting the list of available subcommands. It also allows new methods of notification to be devised, without repatching the set and delete commands.

```
proc _gkenv_initNotify env {
  $env command rename set _set
  $env command set set _gkenv_notifySet
  $env command rename delete _delete
  $env command set delete _gkenv_notifyDel
  $env command set _notify \
      _gkenv_genEvent
}

proc _gkenv_notifySet {env cmd node val} {
  $env _set $node $val
  $env _notify set $node
}

proc _gkenv_notifyDel {env cmd node} {
  $env _delete $node
  $env _notify delete $node
}

proc _gkenv_genEvent {env cmd event node} {
   # however notifications are done...
}
```

Next is the “sharing” extension, which implements a simple form of sharing that ignores concurrency control. Changes in a local copy of the environment are echoed in remote copies of the environment. The GroupKit command “gk_toAll” command is used to execute a command on all the GroupKit processes.

```
proc _gkenv_initShare env {
  $env command rename set _doset
  $env command set set _gkenv_shareSet
  $env command rename delete _dodelete
  $env command set delete _gkenv_shareDel
}

proc _gkenv_shareSet {env cmd node val} {
  gk_toAll $env _doset $node $val
}

proc _gkenv_shareDel {env cmd node} {
  gk_toAll $env _dodelete $node
}
```

The current version of GroupKit provides six different extensions to the core environment data structure, many of them reusing not only the implementation of the core environment, but also the implementation of other extensions. Due to the implementation structure of the core environments, the amount of code to implement each addition is trivial.

Defining packages of extensions in this way is similar to the use of “mixin” classes in C++. The advantage here is that to allow developers to freely intermix *n* different customizations of the basic object, there is no need to create 2^n instantiable classes. Extensions to basic objects are composed rather than inherited here. The approach here is arguably cleaner and easier to understand than multiple inheritance.

Conclusions

This paper has presented an approach to building Tcl object-oriented extensions that can be easily extended in either Tcl or C. The approach brings a level of customization usually found only in Tcl-based extensions to those implemented largely in C. The description of open implementations and the case study should suggest ways that extensions built using either approach can be made more customizable. Rather than relying on a different paradigm, the implementation of these extensions reflects both the familiar structural aspects and the high level of dynamic programming available when creating top level Tcl commands. This lightweight approach to customization is intended to complement the more heavyweight approach found in conventional Tcl object frameworks.

Ideas from computational reflection were evidenced in the approach, which exposes much of the internal object representation to inspection and change, yet does so in a

controlled way so that casual users of the objects need not be concerned with the internal complexity. The techniques described here were illustrated using GroupKit's environments, showing how a simple data structure was extended to support notification and data sharing between replicated processes.

The strength of Tcl is its simplicity of extension and customization; using the approach described here can help extension writers capture the same simplicity in their own work.

Acknowledgements

This paper benefited from feedback provided by Rob Kremer, Earle Lowe and Saul Greenberg, and discussions with Michael McLennan. Ted O'Grady and Paul Dourish, both of whom would cringe to find their names in a paper about Tcl, convinced me of the value of meta-level architectures. Financial support was provided by the National Science and Engineering Research Council and Intel Corporation.

References

1. Greenberg, S. and Marwood, D. (1994). *Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface.* Proceedings of CSCW '94.
2. Kiczales, G. (1992). *Towards a New Model of Abstraction in the Engineering of Software.* Proceedings of IMSA '92.
3. McLennan, M. (1993). *[incr Tcl] — Object-Oriented Programming in Tcl.* Proceedings of the 1993 Tcl workshop.
4. Ousterhout, J. (1994). *Tcl and the Tk Toolkit.* Addison-Wesley, pp. 283-284.
5. Roseman, M. (1993). *Tcl/Tk as a basis for groupware.* Proceedings of the 1993 Tcl workshop.
6. Smith, B., Rowe, L. and Yen, S. (1993). *Tcl Distributed Programming.* Proceedings of the 1993 Tcl workshop.