14’th Annual Tcl/Tk conference
Sept. 24-28 2007
Bourbon Orleans Hotel, New Orleans, LA USA

Conference Proceedings

14’th Annual Tcl/Tk conference
Sept. 24-28 2007
Bourbon Orleans Hotel, New Orleans, LA USA

Invited Talks

The great (internal) Var reform of 2007

Miguel Sofer
Universidad Torcuato Di Tella

September 15, 2007

Abstract

The Var struct used as internal representation for Tcl's variables currently contains six pointers and 2 integers, or 32 bytes on a 32-bit platform. For variables in hashtables, be they namespace variables or array elements, a hash entry structure consuming a further minimum of 24 bytes (but typically 28) is also maintained.

These requirements reflect a history of the structure, and are far from optimal. We will explain the requirements that have to be satisfied, how they gave rise to this structure, and a way to thin them down considerably: compiled variables are reduced to 8 bytes (75% reduction), hashtable variables to a grand total of 24 bytes (60% reduction)[1]. Further performance advantages of the new implementation will also be described.

1 Introduction

Variables are among the most-often accessed structs in Tcl[2]. Their impact on Tcl's performance is undeniable[3], both in terms of runtime and memory footprint. The current Var struct[4] in Tcl, defined in tclInt.h and reproduced in Figure 1, is not optimal: it is too large and its cache-friendliness is further handicapped by an unlucky layout, cache-unfriendly for every r/w operation.

This paper describes the motivation and design decisions in the new variable code in Tcl8.5, implemented in Patch #1750051. Section 2 describes the definition of variables in current Tcl, section 3 the newly defined structs in 8.5. We describe the advantages of the new code in section 4, and its disadvantages in section 5. Section 6 describes related work that will not appear in Tcl8.5 and remains to be done in the future.

[1] A partial implementation is already committed to HEAD, a full implementation may or may not appear in Tcl8.6

[2] Commands may profit from a similar reform in the future

[3] tbd: precise measurements

[4] from Tcl8.0 up to Tcl8.5a6

2 The Var struct in Tcl8.x (x<5)

The Var struct currently requires 32 bytes[5] and accomodates in the single struct both storage types for variables: compiled locals and variables held in a hashtable (either namespace variables or array elements). The necessity of the different fields can be classified as follows

1. *flags, value*: the really necessary fields for normal read/write operation. The first maintains the current state and type of the variable, the second its current value.

2. *name*, *nsPtr*, *hPtr*: necessary so that compiled variables (*name*) and hashtable variables (*nsPtr*, *hPtr*) can find out their names. Knowing the name is required for error messages and trace processing. Note that at most two of these fields are non-NULL for any variable.

3. *refCount*, *hPtr*: needed for lifetime management of the struct, which is only really necessary for hashtable variables.

4. *searchPtr*, *tracePtr*: necessary to store searches and traces currently defined on this variable.

Furthermore, hashtable variables require the maintenance of a Tcl_HashEntry struct that is allocated separatedly and requires either 24 bytes (for variable names of up to three characters), 28 bytes (between 4 and 7 characters), 32 bytes (between 8 and 11 characters), and so on.

The claim of suboptimality is based on the following observations:

- the fields *name*, *nsPtr*, *hPtr*, *refCount*, *tracePtr*[6] and *searchPtr* are rarely accessed, only *flags* and *value* are needed for normal r/w operation

[5] all calculations done for 32-bit platforms as illustration

[6] *tracePtr* is actually tested against NULL at each r/w operation, but this can be avoided

```
typedef struct Var {
    union {
        Tcl_Obj *objPtr;        /* The variable's object value. Used for
                                 * scalar variables and array elements. */
        Tcl_HashTable *tablePtr;/* For array variables, this points to
                                 * information about the hash table used to
                                 * implement the associative array. Points to
                                 * ckalloc-ed data. */
        struct Var *linkPtr;    /* If this is a global variable being referred
                                 * to in a procedure, or a variable created by
                                 * "upvar", this field points to the
                                 * referenced variable's Var struct. */
    } value;
    char *name;                 /* NULL if the variable is in a hashtable,
                                 * otherwise points to the variable's name. It
                                 * is used, e.g., by TclLookupVar and "info
                                 * locals". The storage for the characters of
                                 * the name is not owned by the Var and must
                                 * not be freed when freeing the Var. */
    Namespace *nsPtr;          /* Points to the namespace that contains this
                                * variable or NULL if the variable is a local
                                * variable in a Tcl procedure. */
    Tcl_HashEntry *hPtr;       /* If variable is in a hashtable, either the
                                * hash table entry that refers to this
                                * variable or NULL if the variable has been
                                * detached from its hash table (e.g. an array
                                * is deleted, but some of its elements are
                                * still referred to in upvars). NULL if the
                                * variable is not in a hashtable. This is
                                * used to delete an variable from its
                                * hashtable if it is no longer needed. */
    int refCount;              /* Counts number of active uses of this
                                * variable, not including its entry in the
                                * call frame or the hash table: 1 for each
                                * additional variable whose linkPtr points
                                * here, 1 for each nested trace active on
                                * variable, and 1 if the variable is a
                                * namespace variable. This record can't be
                                * deleted until refCount becomes 0. */
    VarTrace *tracePtr;        /* First in list of all traces set for this
                                * variable. */
    ArraySearch *searchPtr;    /* First in list of all searches active for
                                * this variable, or NULL if none. */
    int flags;                 /* Miscellaneous bits of information about
                                * variable. See below for definitions. */
} Var;
```

Figure 1: The Var struct in Tcl8.x (x<5)

- the fields *tracePtr* and *searchPtr* are NULL most or all of the time for most variables.
- the normal r/w operations[7] access the fields *flags, tracePtr* and *value* (in that order): first the end of the struct, then the beginning.
- creating a new variable in a hashtable requires two separate calls to malloc() - one for the Var, one for the Tcl_HashEntry. As these two structs (can) have the same lifetime and are in 1-1 relationship they could be allocated together, reducing the necessary calls to malloc/free by 50% on variable creation and destruction.

3 The Var structs in Tcl8.5

The layout in memory and access modes for variables has been completely redesigned in Tcl8.5[8], with significant reductions in the required memory and better memory access patterns.

In order to do this, two different structs have been designed for variables: Var (Figure 2) for compiled local variables, and VarInHash (Figure 3) for variables kept in hashtables. The most frequent operations, reading and writing, are impervious to the difference as they only access the Var part; the difference is only relevant for operations related to the variable's lifetime management: creating a link to the variable and unsetting it.

The details are described in this section.

3.1 Removing *tracePtr* and *searchPtr*

As observed previously, the fields *tracePtr* and *searchPtr* are NULL most or all of the time for most variables. The first one is accessed at each variable r/w operation in order to determine if the variable is traced, so that the correct r/w procedure can be used. That is: the fact that these fields are NULL or not is part of the state of the variable. By defining new flag bits to record the complete state of the variable, the linked lists currently held at *tracePtr* and *searchPtr* can be moved elsewhere: only if the corresponding bits are set will their values be accessed.

Two special hash tables (hanging from the Interp struct) have been designed to hold these linked lists. The trace and search code has been modified to maintain the new flag bits.

As an added advantage, the state of the variable can now distinguish the type of trace. This means for instance that reading a variable that carries a trace on write can proceed at full speed, without traversing the list of traces to (fail to) find a possible read trace.

3.2 Compiled local variables: most fields removed

Each time a proc is invoked, an array of variables is allocated on the Tcl stack to hold the body's local variables. These variables are normally accessed by indexing into this array, much faster than an access by name[9].

The lifetime management of the required memory is fairly simple: it is reserved when the proc is invoked and returned when the proc returns. Compiled local variables have no use for the *refCount* and *hPtr* fields, they are gone.

The name of a local variable is unqualified, the *nsPtr* field is not needed. Furthermore, the *name* does not change at each invocation, and it can safely be held in either the Proc or ByteCode structs[10].

Losing this field also simplifies the initialisation of local variables during a proc's invocation. The new flag bits and semantics were designed so that a local variable has to be initialised to {0,NULL}[11], which can be done by a fast memset.

As a result compiled variables only use 8 bytes, as seen in Figure 2, for a 75% size reduction.

3.3 Variables in hashtables: thinned down and consolidated with the entry, Tcl_Obj keys

Namespace variables require knowledge of their namespace in order to reconstruct their fully qualified name. But they have a pointer *hPtr* to their hash table entry, which in turn has a pointer *tablePtr* to the namespace's hash table. We have chosen to store a pointer to the namespace right after the hash table[12], so that every variable can find its namespace without needing to store *nsPtr* in the struct. As *name* was always NULL for these variables, it is gone too.

The Var structure is now allocated together with its corresponding Tcl_HashEntry, which requires that their lifetimes be tied together.

[7] the modes of access in decreasing order of frequency are: read, write, create, destroy, create a link to it.

[8] after Tcl8.5a6

[9] this is among the main performance wins of bytecompiling

[10] in the current implementation it is held in both; this choice was made in order to minimise the changes that might impact extensions.

[11] the proc arguments obviously require a different initialisation. Furthermore, variable resolvers as defined e.g. by incrTcl still require special var-by-var processing

[12] a new TclVarHashTable struct has been defined with two fields: a Tcl_HashTable and a Namespace*

```
typedef struct Var {
    int flags;                  /* Miscellaneous bits of information about
                                 * variable. See below for definitions. */
    union {
        Tcl_Obj *objPtr;        /* The variable's object value. Used for
                                 * scalar variables and array elements. */
        TclVarHashTable *tablePtr;/* For array variables, this points to
                                 * information about the hash table used to
                                 * implement the associative array. Points to
                                 * ckalloc-ed data. */
        struct Var *linkPtr;    /* If this is a global variable being referred
                                 * to in a procedure, or a variable created by
                                 * "upvar", this field points to the
                                 * referenced variable's Var struct. */
    } value;
} Var;
```

Figure 2: The Var struct in Tcl8.5

```
typedef struct VarInHash {
    Var var;
    int refCount;         /* Counts number of active uses of this
                           * variable: 1 for the entry in the hash
                           * table, 1 for each additional variable whose
                           * linkPtr points here, 1 for each nested
                           * trace active on variable, and 1 if the
                           * variable is a namespace variable. This
                           * record can't be deleted until refCount
                           * becomes 0. */
    Tcl_HashEntry entry;  /* The hash table entry that refers to this
                           * variable. This is used to find the name of
                           * the variable and to delete it from its
                           * hashtable if it is no longer needed. It
                           * also holds the variable's name. */
} VarInHash;
```

Figure 3: VarInHash struct

The hash tables for variables now use Tcl_Obj keys, as opposed to the previous string keys[13], insuring that the size of the VarInHash struct does not depend on the length of the variable's name[14].

3.4 Simplified flag semantics

The reform provided the oportunity to simplify the flag semantics by removing some previously allowed possibilites:

- VAR_SCALAR removed: scalar is the default state of a variable, signaled by the absence of array or link bits. The previous scheme allowed for a variable to be neither scalar nor array nor link.
- VAR_UNDEFINED removed: a variable is undefined precisely when its value is NULL. The previous scheme allowed a non-NULL value (garbage)

4 What has been won

The variable reform is a big change, slightly traumatic (see next section). The reasons that make it worthwhile in the author's view include:

4.1 Reduced memory consumption for variables

Assuming variable names between 4 and 7 characters, 24 bytes per variable are saved:

	Bytes		
Type	Tcl8.4	Tcl8.5	Reduction
Local	32	8	24 (75%)
Namespace	60	36	24 (40%)
Array elem.	60	36	24 (40%)

The savings increase for longer variable (or array element) names.

4.2 Cache friendliness

- normal r/w access addresses the first two fields in the struct, in order
- the table of compiled locals is 75% smaller, reducing the data cache pressure for the bytecode engine
- joint allocation of variables and their hash table entries allows to eliminate one level of indirection in the variable's access: instead of following the entry's clientData (which hold a pointer to the Var), the Var pointer is computed from the entry's using a known offset.

4.3 Reduced impact of traces

Up to Tcl8.4 access to a traced variable was always slower: even if the current access mode was not itself traced, this could only be discovered by traversing the linked list of traces. The new code proceeds at full speed when there is no trace relevant for the current access mode

4.4 Faster creation and destruction of variables

Typically a single call to malloc() on creation, and a single call to free() on destruction

- Reuse the Tcl_Obj in the creation request as the hash table key: increase its reference count instead of allocating a new string
- Decrease the name's reference count on destroying the variable; if the name is shared, no additional calls to free()

4.5 Faster access to variables[15]

The Tcl_Obj keys allow for faster lookup: shared literals increase the probability of a very cheap test in the matching case, the fact that the string length is stored allows for faster failure in many cases. Better possibilities for caching of resolved variable names.

5 TANSTAAFL

5.1 Added complication in trace code

The trace code has to maintain the trace-related bits in the Var's flags, while previously it would just add/remove items from the front of the trace linked list

5.2 Added complication of variable code

More code is dependent on the flag values. For example, compiled local variables do not have a refCount field. All the code that manages the reference count of variables has

[13] the variable access code is as of this writing not yet fully optimised for this change

[14] further advantages are described below

[15] not yet fully optimised as of today

to check the storage class of the variable (a special flag bit) to determine if a r/w of the reference count is necessary.

5.3 A new hashtable type

The hash table used to store variables is defined via a new tclVarHashKeyType. However, the current implementation uses the standard Tcl hash tables with a custom key type.

5.4 Slower trace invocation

The invocation of variable traces is somewhat slower, as it involves a new search in a hash table. This is deemed to be more than compensated with the faster access to variables when the access mode itself is not traced.

5.5 Breaking "rogue" extensions

Extensions that include tclInt.h and interact directly with the core's variables, variable hash tables or bytecodes are broken. The code of incrTcl, XOTcl and tbcload has been adapted to the new core; it is not known if more extensions are impacted.

As a general rule, it is relatively straightforward to adapt an impacted extension to restore source compatibility.

Binary compatibility in the sense that "a previously compiled extension runs in a current core" is essentially impossible. It is possible[16] to create "binary compatible sources", in the sense that a newly compiled extension can run on both a Tcl8.4 and Tcl8.5 core. This has been done for the three extensions mentioned above.

It is to be stressed that normal extensions that only include tcl.h suffer no effects, and that most extensions that do use tclInt.h (including Tk!) are also immune.

6 Remaining (somewhat) related rewrites

There are other related modifications that may (or may not) occur in the future - either Tcl8.6 or Tcl9. Among them

6.1 Optimisation of variable lookup and caching of variable names

Variable lookup and the caching of variable names has not yet been optimised for the reformed code[17]. This optimisation will occur before the Tcl8.5 release. It is expected to provide significant speedups.

6.2 Specialized hash tables for variables

The reform described here uses Tcl's standard hash tables. These are versatile and performant, but impose some penalties on variables that could be avoided with specialized hash tables:

- the entries are "too big" for our purposes, three more fields could be eliminated for a further 33% reduction in the VarInHash size (Figure 4), from 36 to 24 bytes.
- the hash table code owes its versatility to its generous usage of indirect calls; coding specifically for variable hash tables would allow faster access

6.3 commandReform

The second most critical struct in Tcl is the command. The core has a sophisticated mechanism for caching command names, and trying to avoid renewed lookups[18]. However, lookups by name are still frequent.

A reform of the command lookup code similar to the one described in this paper will be tested. The memory footprint of commands likely being much smaller than that of variables, and command creation/destruction being rarer than the analogon for variables, the payoff in terms of memory management is unlikely to exist.

On the other hand, especially if a reform manages to also reduce the cost of verification of a cached pointer's validity, the pay off in terms of increased command dispatch performance could be sizable.

6.4 objReform

I lied: the most time critical struct in Tcl is not Var, it is Tcl_Obj. Modifying Tcl_Obj handling is however very difficult without breaking every extension out there. Some experimental attempts that could pay off handsomely (but may be infeasible in Tcl8.x) include:

- Improving the alignment of Tcl_Objs, using the padding to store small strings within the Tcl_Obj struct itself (see Patch 1772004). This reduces indirection as well as calls to malloc()/free()
- Usage of tagged pointers to Tcl_Objs within the bytecode engine to store "small" integers to reduce indirections

[16] with some nasty hacks

[17] and is clearly suboptimal for some access patterns, see Bug 1793601

[18] already improved in Tcl8.5 to reduce sharing of command name literals in different namespaces

```
typedef struct VarInHash {
    Var var;
    int refCount;               /* Counts number of active uses of this
                                 * variable: 1 for the entry in the hash
                                 * table, 1 for each additional variable whose
                                 * linkPtr points here, 1 for each nested
                                 * trace active on variable, and 1 if the
                                 * variable is a namespace variable. This
                                 * record can't be deleted until refCount
                                 * becomes 0. */
    TclVarHashTable *tablePtr;  /* Pointer to the table containing this
                                 * variable */
    Tcl_Obj *keyPtr;            /* Pointer to the object containing the
                                 * name of this variable. */
    struct Var *nextPtr;        /* Pointer to the next variable in this
                                 * same bucket (only necessary for certain
                                 * types of hashtables).
} VarInHash;
```

Figure 4: Future VarInHash struct

7 Conclusion

Internally there are two different kinds of Tcl variables: compiled locals which reside in an array, and the rest which live in hash tables. Tcl8 defined a common struct to describe both, with some fields that are only useful for one kind and are wasted in the other. Additionally, part of the variable's state is kept in extra fields that are NULL for most variables most of the time.

Different specialised structs for each kind, and a redesign that insures that the state is fully described by the flag values, permitted a very significant reduction in the memory footprint of variables. There is no space/time tradeoff involved as there are also speed gains (not of the same magnitude).

This major change respects all public interfaces; however, a few extensions that use internal apis lost both binary and source compatibility and required an adaptation.

A similar revision of other important structures and access patterns in Tcl will be explored. The expectation is that there is a possible payoff in terms of speed, but no major impact on the memory footprint is to be expected.

14’th Annual Tcl/Tk conference
Sept. 24-28 2007
Bourbon Orleans Hotel, New Orleans, LA USA

GUI and Gui Techniques

Building an Electronic Design Automation (EDA) tool using Tcl/Tk and object oriented programming.

John Hughes
Software Engineer
Mentor Graphics Corp.
Wilsonville, OR

14th Annual Tcl/Tk Conference
New Orleans, Louisiana
September, 2007

Abstract

This paper discusses how a powerful debugging, and design analysis, tool was created for the Design-For-Test (DFT) division at Mentor Graphics Corp. This tool is called DFTVisualizer and is a quantum leap forward from the old tool. The old tool, known as DFTInsight, will be covered briefly for comparison purposes.

This paper will describe the reasons why Tcl/Tk was chosen and how the Tcl code interacts with the C/C++ "kernel" code. It will also cover the object oriented techniques used to build the tool and the various packages used in the implementation.

Introduction & Outline

Several years ago we were presented with the problem of replacing our current debugging tool with something new and exciting. The old tool was basically a schematic viewer that would allow the user to view portions of their chip design that were flagged as having design rule errors.

After nearly a year of meetings and gathering requirements we had an “idea” of the tool we needed to build. It needed to encompass the capability of the old tool, but offer several additional debugging and analysis features. Many of those features already existed in the system, but in the form of textual reports. While those reports contained the desired information, they were often labor intensive to read and “digest”. Showing the data in a graphical form was a requirement.

The remainder of this paper will be organized as follows:

1. A brief history of the legacy DFTInsight tool and textual reports.
2. Requirements and the functional specification.
3. Language and widget toolkit considerations.
4. Design of the DFTVisualizer tool.
5. Conclusion with a brief description of the implementation status and its success or failure with our customers.

History

The Design-for-Test tools aid in creating test patterns to run on the Automatic Test Equipment (ATE) in the “fab” when chips are being produced. Some DFT tools also modify the chip design to contain extra circuitry, so the chip can test itself. This is called Built-in-Self-Test or BIST for short. Creating test patterns for the latest microprocessor is a task that is very time and memory consuming. This can take days, even using grid computing engines, so the user wants the debugging process to be intuitive, effective, and reduce the overall time spent debugging the design. The Mentor Graphics DFT products have utilized a tool called DFTInsight to examine the design via a schematic that represents a portion of the chip design containing a problem.

See figure 1.

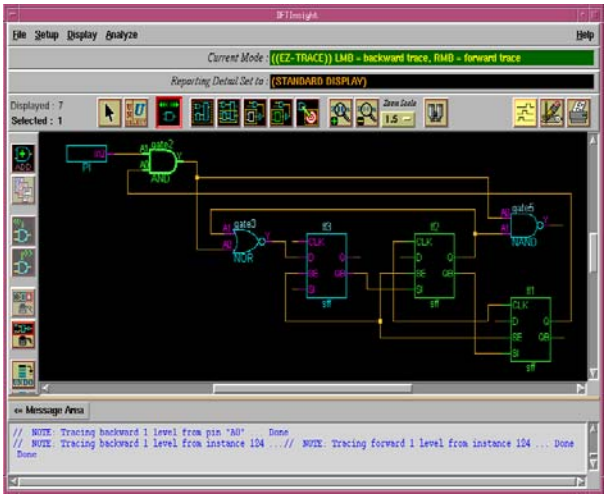


Figure 1. DFTInsight showing a DRC violation

Trouble shooting a DRC violation is only one of the problems that DFT customers encounter. There are other issues they need to analyze, like test coverage. Test coverage reports will tell the user what areas of the chip are testable and have test patterns generated for it. The user wants to see the coverage statistics for the entire chip and for the different design blocks used to build the chip. That way they can concentrate their efforts on the areas with the most problems. Traditionally this has been a textual report from the DFT products that the customer has to read. Often the customer would create scripts to produce more readable reports or convert them into CSV for importing into a spreadsheet program.

The next generation tool would evolve from strictly a debugging tool for design rule violations, to one that would allow graphical analysis of many textual reports. An example text report is shown below.

<ATPG> report statistics -hierarchy
... analyzing hierarchical statistics (Coll: 8248 faults, Full: 13282)

	# faults	UO	AU	% test
-top- (cm)	13282	0	34	16.07%
dir (txStatus)	176	0	0	23.86%
cfsm (cmFSM)	1138	0	0	11.51%
dec (decode)]	1142	0	22	17.81%
skipgen (skip_generate)	394	0	0	19.54%
lbmux (mux2to1b20)	86	0	0	0.00%
hpdech (hp_decode)	214	0	0	27.10%
hpdecl (hp_decode)	212	0	0	26.42%
dcx (detect_cx)	178	0	0	0.00%
dd (detect_data)	18	0	0	0.00%

Requirements and Functional Spec.

For approximately a year, a team met weekly to gather and discuss requirements for the new debug and analysis tool. This team was comprised of people from technical marketing, customer support, technical documentation, QA, and engineering.

Once the requirements were finalized (theoretically), work began on a functional specification document. This was largely authored by the engineering team and approved by customer support and technical marketing personnel. This document broke down each requirement into greater detail and how it would work in the new tool. One of the requirements was that there would be multiple windows, or views, of the data and those windows would be contained within a framework or parent window. These windows could be un-docked from the parent window and re-docked at a later time.

Another requirement is that this new environment needed to work with our legacy GUI and with our tool in no-GUI mode. The DFT tools can be run without a GUI in interactive mode or batch mode. In interactive mode, the user still needs to be able to bring up the new debugging and analysis environment by typing a command. This means that our command loop needs to work in conjunction with the TK event loop and not block each other. The DFT tools are single threaded applications, but do use grid distribution to run multiple processes. So, the master process is single threaded (legacy issue), but it spawns multiple slave processes on the grid.

Lastly, all of this needs to work in a Unix/Linux world because that is where the DFT tools are utilized.

Language & widget toolkit considerations

Several languages and widget toolkits were considered before coding started on the new tool. We knew that some of the code would be coded in C/C++ because that is what the “kernel” of our DFT products is written in. The DFTInsight tool, and most of our GUI, was written in Tcl/Tk, but we did consider the following options:

- Tcl/Tk
- Java
- Qt
- incrTcl/incrTk
- SWidgets (Mentor toolkit)
- mtiWidgets (Mentor toolkit)

incrTcl/incrTk with a C interface to the kernel code was ultimately selected and we also used the mtiWidgets toolkit. The reason that the mtiWidgets were selected is that they were already used in a popular Mentor product whose customers liked the GUI, and they were created with incrTcl/incrTk. Also, since we would be switching back-and-forth between C++ and Tcl code, incrTcl just seemed like a natural choice to do object oriented tasks in Tcl.

Design of the DFTVisualizer

The first thing we did once we started the design phase was to write a design document that detailed the different parts of the tool being built. The document contained the details of the data structures, classes, and API used by the framework and the "child" windows within the framework.

- What is the framework and what does it do?
    - The framework is a parent window that will contain all of the other windows in the tool. It also implements the API that is used to communicate between windows. This API is a singleton class written in incrTcl.
    - The framework also includes the tool bar, menu bar, and an MTI pane manager window [1].
    - The MTI pane manager window will contain the various debug & analysis windows, a transcript window, and a status bar.
    - The framework will handle the selection set across all windows.
    - It will also manage the drag-n-drop operations between windows.

- Framework API
    - addTextToTranscript
    - clearTranscript
    - showStatusBar
    - hideStatusBar
    - updateStatusBar
    - updatePreferences
    - start/endDragNDrop
    - methods for cross selection between windows.
    - ……. Many more

- Child Window API
    - showChildWindow
    - hideChildWindow
    - acceptDndObjects
    - dock
    - undock
    - addData
    - addObject
    - deleteObject
    - selectObject
    - …… many more

All of these API's are implemented via incrTcl methods. For example, here is a portion of the class signature for ChildWindow. A ChildWindow is one of the windows contained within the Framework window/class. For example, a window showing a hierarchical browser, a transcript window, a schematic viewer window, a waveform viewing window, etc.

```
itcl::class ChildWindow {

  constructor {frameHandle windowName windowFrame \
          windowDisplayName} {

    set _windowName $windowName
    set _frameHandle ""
    set _toolBar ""
    set _menuBar ""
    set _frameVars(child_frame) ""
    set _saveFocus ""

    ::ChildWindow::Create $frameHandle $windowFrame \
          $windowDisplayName

  }

  destructor {}

  # routine to have the child create itself.
  # i.e. Build the main window for the child
  private method Create {frameHandle childFrame childName}

  # get the name of the child window
  method getChildWindowName {}

  # get the displayed name of the child window
  method getChildWindowDisplayName {}

  # get the frame of the child window
  method getChildWindowFrame {}

  # tell the child to select/deselect something
```

```
    method notifySelection {selObject selectType {numOfSwitches "0"}  \
                {switches ""} }

    # tell the signal child window to select/deselect
    method notifySelectionToSignal {objectType objectName \
                dataType selectType}

    # add an object to the child window
    method addObject {object {numOfSwitches "0"} {switches }}

    # delete an object from the child
    method deleteObject {object {numOfSwitches "0"} {switches ""} }

    # add data to Child Window
    method addData {data}

    # delete data from Child Window
    method deleteData {data}


    # Class data members
    # handle/ptr back to framework
    protected variable _frameHandle;

    # generic "name" of a child Window
    protected variable _windowName;

    # displayed "name" of a child Window
    protected variable _displayName;

    protected variable _toolBar
    protected variable _menuBar

    protected variable _exportFormat
    protected variable _dlgToCmdFormat
    protected variable _saveFilePath
    protected variable _saveFileFormat

    protected variable _createDofilePath
    protected variable _replaceDofiletag

    protected variable _saveFocus

} ;# end ChildWindow class
```

See figure 2 for an example of the DFTVisualizer.

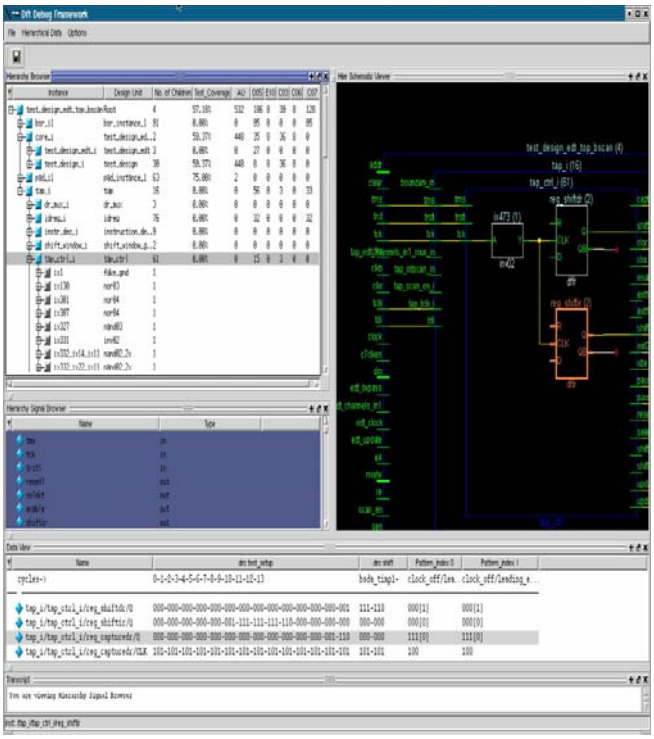


Figure 2. DFTVisualizer with multiple windows shown

See figure 3 for an example of the graphical representation of the textual report. This version will allow the user to browse through the design hierarchically and they can turn on/off the columns of data shown. This way the user can quickly navigate to the data they are most interested in seeing.

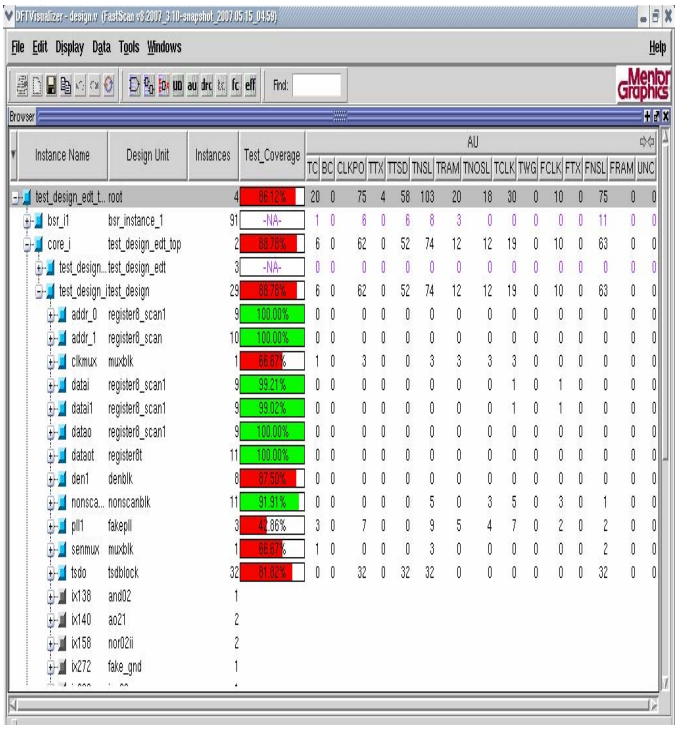


Figure 3. Graphical reporting of statistical data

Conclusion

The DFTVisualizer has many capabilities and is an order of magnitude better than its predecessor. The customers have given great feedback on the tool and have adopted it very quickly. We have many stories of customers saving days or weeks of debugging effort because of the DFTVisualizer.

Of course the customers have also been quick to ask for more features; some that we never even considered in our long term planning.

In general, the tool is solid and the customers are happy with it. The overall goal was to have a simple, easy to use, and clear process to debug problems in the customer design, and to present data in an easy to analyze format. Based on the customer response, we have achieved that goal.

Now we just need to add the other million features that the customers desire! ☺

Future Work

Currently there are 11 windows in the DFTVisualizer to aid with debugging and/or data analysis. One of the newer windows is a task manager that helps guide the user through common operations in the tool and will automatically open the appropriate windows for that task. Continuing to evolve the tool to provide more debugging and analysis options are a high priority, but looking at a higher level, this environment may eventually become a replacement for our current GUI. Streamlining task based operations will continue to be important and I expect that some project management features will be added too.

References

[1] Griffin, Brian S. The MTI Panemanager Widget 2-D Paned Window for user configurable U/I. Tcl/Tk 2005 (www.tcl.tk/community/tcl2005/abstracts/GUI/Panemanager.pdf)

Making Beautiful Graphs with Zplot

Remzi H. Arpaci-Dusseau
University of Wisconsin, Madison

Abstract

This paper introduces Zplot, a Tcl library for making two-dimensional data plots. Zplot provides a simple set of primitives that allow users to input and manipulate data, plot said data in a variety of formats, and decorate the resulting graphs with axes, labels, and other textual accents. Zplot then outputs encapsulated PostScript for ease of inclusion in technical documents.

1 Introduction

Over the past 20 years or so, I have used a variety of tools to generate data graphics for the various technical papers with which I have been involved. These tools left me despondent. They seemed incapable of producing all but the most basic of graphs. Many common graph types were not well supported (e.g., bar graphs). Simple data manipulations were forced into pre-processing steps, creating a clumsy tool chain. Manual manipulation on the resultant PostScript was often required to achieve the desired result.

Zplot is the fruit that was born of this frustration. Zplot is a pure Tcl library that allows the creation of two-dimensional data graphics in a flexible and powerful manner. Typical graphs are created with only a few lines of Tcl, and complex and intricate graphs can be produced from only tens of lines of code.

In this document, I describe Zplot. First, I give an overview of the tool and the basic primitives it provides. Then, I describe each of the basic routines in more detail, showing how they can be combined to produce a wide range of interesting graphs. Zplot drawing routines are all built upon a set of low-level PostScript-generating commands; these hide many of the details of generating correct PostScript from the rest of Zplot, boiling down most activities to simple drawing commands that place lines, shapes, and text on the drawing surface. I then conclude the paper with a few comments about Tcl, related and future work, and a final summary.

2 Overview

I now describe the basic primitives provided by Zplot. Let us start with a typical (if simple) graph as an example, and use this to drive the discussion of the different elements of Zplot. A typical graphing script might be written as follows (with the results of the graph shown in Figure 1).

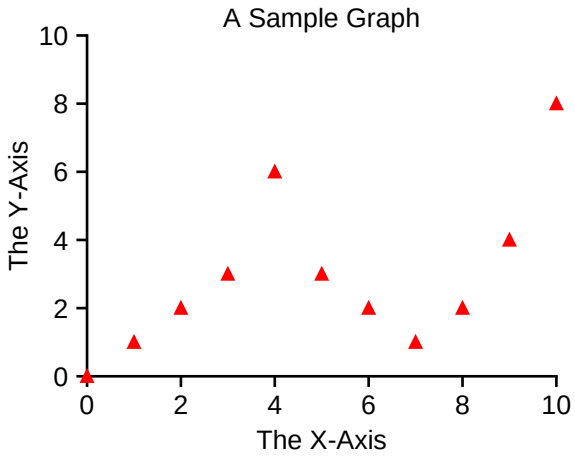


Figure 1: **An Example Graph.** *The most bare-boned of plots that one can make with Zplot.*

```
# input the library
source zplot.tcl
namespace import Zplot::*
# describe the drawing surface
PsCanvas -title "file.eps" -dimensions 300,200
# load some data
Table -table t -file "file.data"
# make a drawable region for a graph
Drawable -xrange 0,10 -yrange 0,20
# make some axes
AxesTicsLabels -title "A Sample Graph" \
  -xtitle "The X-Axis" -ytitle "The Y-Axis"
# plot the points
PlotPoints -table t -xfield x -yfield y \
  -style triangle -linecolor red
# finally, output the graph to a file
PsRender -file "file.eps"
```

In this example, the user creates a graph by first describing the drawing surface by calling `PsCanvas` and specifying its dimensions. Then, the user calls the `Table` routine to load data into Zplot, getting the data from a file (`file.data`). The user, wishing to plot the data, now creates a drawable region by calling the `Drawable` routine; doing so defines where on the canvas the drawable is, and also how to map data points onto the drawing surface (e.g., the range of x values and y values that map onto this drawable). With a drawable defined, the user can now call

one of a variety of plotting routines (e.g., `PlotPoints`) to plot the data onto the drawable. The plotting routines generally take a large number of arguments, enabling a wide variety of plots to be produced; in this case, the user chooses to draw a red triangle at each (x,y) point of the graph. Finally, the user adds some graphical and textual decorations to help clarify the graph (in this case, by simply calling the `AxesTicsLabels` routine), and then renders the PostScript to a file by calling PsRender. I now describe each of these primitives in more detail.

Note that each of these routines takes a large number of optional parameters. To find out what these are (without perusing the source code), one should simply call the routine and pass it the `-help` flag (or any bad flag); a useful error message about the routine and all of its parameters (including default values) will be printed.

2.1 Table

There are numerous routines available to users to input and manipulate data; these are known as the `Table*` routines. The most commonly used routine is the basic `Table` routine; usually, this routine is used to input a file and then plot its points. A typical file (such as `file.data` above) looks like this:

```
# x y
0 0
1 1
2 2
3 3
4 6
...
9 4
10 8
```

The first line contains the "schema" for the table, with names for each column; these names are subsequently used to refer to the data when manipulating it or drawing it to the screen.

One powerful routine is the `TableSelect` command; it allows one to perform a database-like selection over a table and put the results in a new table. Here is an example that selects data from table `t` with y-values above 5, and plots green circles around said points (the results of which are shown in Figure 2):

```
Table -table thi -columns x,y
TableSelect -from t -to thi -where {$y > 5}
PlotPoints -table thi -xfield x -yfield y \
  -style circle -linecolor green -size 4
```

There are a number of other useful table functions which are not covered here, mostly for manipulating and summarizing data. For example, `TableMath` can be used to perform a mathematical operation (or indeed, any valid Tcl expression) on a column of data. The routine `TableComputeMeanEtc` is useful for computing means and deviations over a column, and `TableBucketize` can be used to place data into bins. All of these primitives are built on lower-level table routines that access each row of a table and perform operations on its contents; thus, more complex operations on tables can be readily assembled by adventurous users.

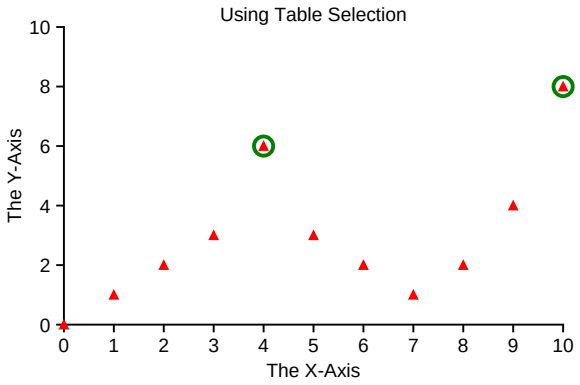


Figure 2: **Table Selection.** *The example uses a simple table selection to find y-values that are greater than 5. Then, these points are plotted as green circles.*

2.2 Drawable

The drawable is likely the most important abstraction that Zplot implements. A drawable is created by the `Drawable` command. Each drawable has a name; the default name is `default` and this default is used by all routines that expect a drawable unless otherwise specified. Here is the relevant portion of the example above rewritten to use the drawable name `foo` instead of the default:

```
Drawable -drawable foo -xrange 0,10 \
  -yrange 0,20
AxesTicsLabels -drawable foo \
  -title "A Sample Graph" \
  -xtitle "The X-Axis" -ytitle "The Y-Axis"
PlotPoints -drawable foo -table t \
  -xfield x -yfield y -style triangle \
  -linecolor red
```

The powerful aspect of a drawable is that it enables a user to place multiple (potentially overlapping) drawable regions onto the drawing surface. This feature can be used to implement a number of interesting graphs. For example, in Figure 3 (taken from [6]), two regions of the graph are of interest but hard to see due to their small size. Thus, one can create two additional drawables and plot closeups of the data in those regions:

These types of closeups are trivial to implement. Here is the code for one of them (the entire example, and many others, can be found on the Zplot web site):

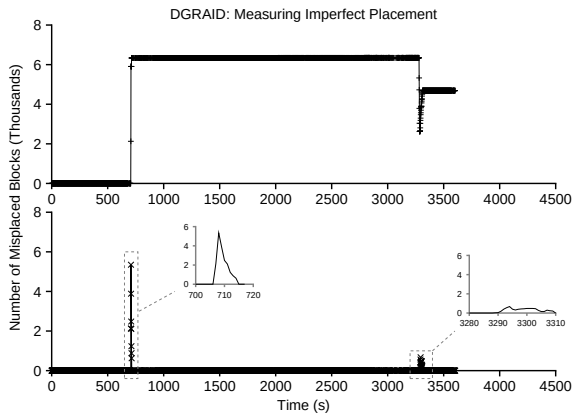


Figure 3: **Nested Plots.** *A plot from an earlier paper of ours is recreated. Two closeups are made in the lower graph, with only a few lines of Tcl code required.*

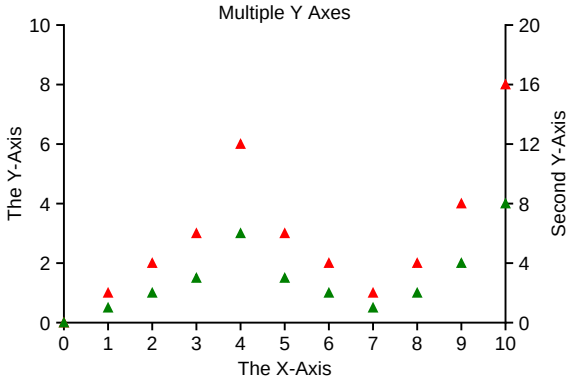


Figure 4: **Multiple Y Axes.** *The script creates two drawables, the right one with a y-range that is twice as high as the left one. The same data is plotted on both.*

```
Drawable -drawable copyc1 -coord 135,90 \
  -dimensions 40,40 -xrange 700,720 \
  -yrange 0,6000
Table -table copyc1 -columns c0,c1
TableSelect -from copy -to copyc1 \
  -where {($c0>=700) && ($c0<=720)}
AxesTicsLabels -drawable copyc1 \
  -xauto ,,10 -yauto ,,2000 \
  -linecolor gray -fontsize 6
PlotLines -drawable copyc1 -table copyc1 \
  -xfield c0 -yfield c1 -linewidth 0.25
```

This example also demonstrates a number of parameters that the `Drawable` routine can be passed. For example, a user can specify its exact position with the `coord` flag and its size with the `dimensions` parameter.

Multiple drawables can also be used to plot data with multiple y axes in a simple and straightforward manner. In this example, we plot the same data from the example above, except onto an overlapping drawable that maps the y range from 0 up to 20 (instead of 0 to 10). The code is below; the resulting graph (Figure 4) thus plots the same data twice, once in red (as relative to the left y-axis), and once in green (as relative to the right).

```
Drawable -drawable second -xrange 0,10 \
 -yrange 0,20 -width 230
AxesTicsLabels -drawable second -style y \
 -ytitle "Second Y-Axis" -labelstyle in \
 -yaxisposition 10 -yauto ,,4
PlotPoints -drawable second -table t \
 -xfield x -yfield y -style triangle \
 -linecolor green -fill t -fillcolor green
```

2.3 The Plot* Family

There are currently eight members in the `Plot*` family: `Heat`, `VerticalBars`, `HorizontalBars`, `VerticalIntervals`, `HorizontalIntervals`, `Points`, `Lines`, and `VerticalFill`. Most should be self explanatory from the name, and examples of each can be found in Figure 5.

There is also a plotting function that takes an equation instead of a table: `PlotFunction`. This routine simply takes a function to evaluate and draws the result.

2.4 Axes, Tics, and Labels

A single complex routine supports the generation of axes, tic marks, and labels for a graph. It is (not surprisingly) called `AxesTicsLabels`. It has too many arguments to describe here in any detail. However, it is often quite simple to use. For example, to specify the title, label for the x-axis, and label for the y-axis, one simple do the following:

```
AxesTicsLabels -title "Title" \
  -xtitle "X-Axis" -ytitle "Y-Axis"
```

Internal algorithms compute reasonable locations for said labels (depending on whether tic marks are used, for example). Further, when the guesses are wrong, one can use a shift argument to move the text to a more appropriate location (*e.g.*, the `-titleshift` argument can be passed the value `3,0` to bump it 3 points to the right). Many of the other options deal with customizations such as font selection, rotation, color, and so forth.

2.5 Legend

Finally, Zplot provides support in most plotting routines for the addition of a legend. A given plot routine takes an optional `-legend` flag which indicates the name to be associated with the data. The user subsequently calls the `Legend` routine, to place the legend on the screen and control its appearance.

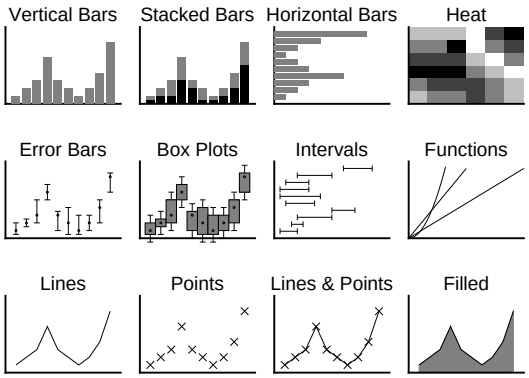


Figure 5: **Multiple Plot Types.** *This example plots a number of different plot types, as described in each title. Of course, many other variations are possible.*

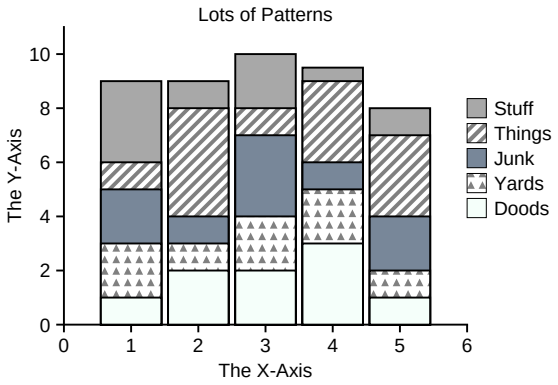


Figure 6: **Multiple Patterns.** *This example plots a number of different patterns in a set of stacked bars. As one can see, patterns such as diagonal lines and triangles can be used to fill a region. The example also includes a legend.*

3 PostScript Generation

Zplot is built on top of a number of underlying PostScript primitives, including basic lines, filled (or empty) shapes, and text. Each of these routines is used by the plotting routines and other entities that wish to create graphical or textual elements upon on the drawing surface. We now describe the primitives in turn.

```
PsLine -coord <x1,y1:x2,y2:...:xN,yN>
   -linecolor <color>
   -linewidth <width in pts>
   -linecap <0, 1, or 2>
   -linejoin <0, 1, or 2>
   -linedash <dash pattern>
   -closepath <true or false>
```

The `PsLine` primitive is passed a set of coordinates, some basic information about the line, and then produces a line that connects the coordinates in the resulting PostScript. All PostScript primitives take coordinates in PostScript "ems", each of which is 1/72nd of an inch. The `PsLine` primitive also takes additional arguments that allow the addition of an arrow to the end of the line; we omit these parameters for the sake of space.

```
PsBox -coord x1,y2:x2,y2
PsCircle -coord x,y -radius r
PsPolygon -coord x1,y1:...:xN,yN
  -linecolor
  -linedash
  -linecap
  -fill <true or false>
  -fillcolor <color of each element>
  -fillstyle <style>
  -fillsize <size of element in pattern>
  -fillskip <amount to skip between ...>
  -fillshift <+x,+y>
  -bgcolor <color behind pattern>
```

Each of these shape routines take a variety of arguments that describe their coordinates, and then all take three different sets of arguments that characterize the line around the shape (`-line*`), the fill of the shape (`-fill*`), and the background color behind the shape (-bgcolor). The line descriptors match those of `PsLine` above, and the background color is straightforward. Most interesting, then, is the variety and flexibility provided by the pattern descriptions.

The `-fill*` parameters allows users to specify a fill pattern for a region. The most important parameter is `-fillstyle`, which determines how the region is filled. Current styles that are supported include `solid, hline, vline, dline1, dline2, circle, square, triangle, utriangle`; more are added occasionally (when the author needs them). Each pattern takes two arguments to determine its contents: `-fillsize` and `-fillskip`. Within a given pattern, `-fillsize` determines the size of each element in the pattern, and `-fillskip` the space between each element. Figure 6 is a bar graph that demonstrates the use of some of these patterns.

```
PsText -coord <x,y>
  -text <the text to write on canvas>
  -font <font choice>
  -color <color>
  -rotate <angle of rotation>
  -anchor <how to anchor the text>
  -bgcolor <background color behind text>
  -bgborder <size of border around text>
```

The last primitive we describe is `PsText`, which draws text onto the screen. Most of its parameters are straightforward. However, the most crucial argument to understand is the `anchor`. This parameter describes how the

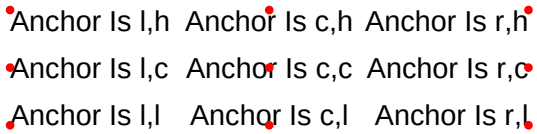


Figure 7: **Text Anchors.** *This example shows how to specify text anchors.*

text should be anchored relative to the coordinate that was passed to the routine. The parameter takes the form `xanchor,yanchor`, where xanchor specifies the anchoring of the text in the x direction (either `l` for left, `c` for center, or `r` for right), and yanchor the anchoring in the y direction (`l` for low, `c` for center, and `h` for high). Figure 7 shows the different possible anchors (the coordinates passed to the text drawing routine are highlighted with a red circle).

4 Commenting on Tcl

We now comment on a few aspects of Tcl that arose during the implementation of Zplot. We begin with performance issues, comment on namespaces and packages, and finally discuss error checking.

4.1 Performance

As floating point specialist William Kahan famously said, "The fast drives out the slow, even if the fast is wrong." Tcl is slow. Thus, Zplot is slow. If one tries to plots graphs with thousands of data points, one will have to wait, even on a modern processor. To show how slow, I present a rudimentary performance study of Zplot performance.

In the experiment, I simply timed how long it takes to produce a plot given an input file with 100,000 data points. The experiment was run upon a MacBook Pro laptop with 2.16 GHz Intel Core 2 Duo processors, 1 GB of RAM, and running Mac OS X 10.4.9. Five trials were run, and the input file fit comfortably into main memory (thus, no substantial I/O activity occurs during the experiment).

The average time to run Zplot over this large data file was 45.93 seconds (with very little variation). In comparison with other tools written in C, Zplot performance is many orders of magnitude slower (*e.g.*, plotting the same input file with gnuplot is nearly instantaneous). It is true that Zplot was not written with optimized performance in mind, but it was not written to be horrifically slow, either. It is simply the case that building clean Tcl programs with many nested subroutine calls leads to poor performance.

Ironically, John Ousterhout's paper [5] points out many reasons that operating system performance does not scale with processor performance; analogous arguments can be made about Tcl. Although processors have improved greatly in the past 10 years, Tcl remains slow in both a relative and absolute sense. It is this author's opinion that this performance flaw is one major reason Tcl has not become more broadly accepted.

4.2 Namespaces and Packages

Tcl namespaces are a simple and powerful feature; as a long-time Tcl user, they have been a welcome addition. Somehow, I do not find myself using Tcl packages; instead, I just create a single large Tcl file from the various source files of Zplot, and `source` said file to use Zplot. Primitive? Certainly. And yet somehow I prefer it to the current package creation system.

4.3 Error Checking

I found myself cursing the lack of assistance for error checking in Tcl. For example, when a user calls a routine and accidentally passes text instead of a numeric value to a particular routine, if one is not careful, some kind of Tcl error message will get printed and the program aborted – not very user-friendly.

To cope with this problem, I wrote a generic argument parsing package that performed type checking and other type-specific checks on a per-argument basis. Internally, most user-callable routines begin with a declaration as follows:

```
proc Table {args} {
  set default {
   {"table"   "default" \
      "isString 1" "name to call table"}
   {"file"    ""        \
      "isFile 1"   "file to read from"}
   {"separator" ""      "isString 1" \
      "if empty, whitespace; \
       otherwise, whatever is specified"}
  }
  ArgsProcess Table default args use \
  "Create a table. If '-file' is specified, \
   load the table from a file. Otherwise, \
   '-columns' must be specified and give a \
   comma-separated list of columns in the \
   table (e.g., '-columns x,y,mean')."
  ...
```

For each argument, a routine is specified that is used to perform whatever checks are relevant. For example, for the `-table` parameter above, the routine `isString` is called to ensure that the table name is a string (a primitive perform of dynamic type-checking). Defaults are also specified in case the user does not specify a given argument (*e.g.*, `-table` will default to the `default` table). When a problem occurs, an error message prints out each parameter, its default value, the info string per parameter (*e.g.*, `name to call the table`), and the overall description of the function as specified in the call to `ArgsProcess`. As mentioned above, one can call most routines with a bad flag to obtain said information.

5 Related Work

Much of the frustration I spoke of earlier was with a tool known as gnuplot [7]. Gnuplot provides excellent support for simple line graphs and scatter plots, as well as numerous other graph types. However, its lack of reasonable support for bar charts was one of the main driving forces behind Zplot. However, I should note that the PostScript produced by gnuplot was clear and easy to read, sparking my interest in that language, and thus (indirectly) making Zplot possible. Great PostScript resources, for those who are interested, are the blue book, red book, and (to some extent), the green book [1, 3, 2]; all are available online.

As I demonstrated Zplot to others, many people referred me to Ploticus [4], which is a more powerful and complete tool than gnuplot and is capable of producing a large variety of interesting graph types. Many of the features found in Zplot are also found in ploticus (e.g., a ploticus "area" is akin to a Zplot Drawable), and I often found myself downloading examples from the Ploticus web page to see if Zplot could easily do what Ploticus already does. Indeed, at one point I even considered dropping Zplot development and simply adding a few features to Ploticus that I found lacking (*e.g.*, bar graphs with a variety of pretty patterns). However, one look at the Ploticus source code convinced me that I might be on the right path (or, at least, a different path). Ploticus is comprised of over 60,000 lines of C code. Zplot, in contrast, is less than 5,000 lines of Tcl; although not always the prettiest code, certainly quite a bit simpler. This comparison is certainly a bit unfair, as Zplot is not as powerful as Ploticus, but I feel quite positive that it will never be nearly as large or complex, a testimony to the power of a higher-level language such as Tcl.

6 Future Work

Zplot is incomplete in a number of ways. For example, although the PostScript it generates is simple, it is often inefficient (*i.e.*, the resultant PostScript is larger than it need be). Some simple optimizations would noticeably reduce the size of the resultant PostScript files.

Error reporting has improved throughout the course of Zplot's development, but could always be better. The development of a more powerful argument processing package (as described above) helped a great deal, but there are still some cases where a user could trigger an internal assertion to fail and thus will see a stack trace telling them where something went wrong. Better error reporting remains something I plan to look into.

Finally, there are a host of features which would be useful. Better support for time and date formats would be of great benefit. More line styles, point styles, and fill patterns are always helpful. A facility to automate graph generation (much like the "prefabs" offered by ploticus) would probably be well-received.

7 Conclusions

In this paper, I have introduced Zplot, a pure Tcl package for drawing PostScript figures. Zplot provides a number of powerful but simple tools for making beautiful two-dimensional plots. In the course of building Zplot, I was again surprised by how slow Tcl is; however, its simplicity and power make programming in Tcl something unusual (to me) among its counterparts: fun.

If you are interested in Zplot, please visit: `www.zplot.org`.

Acknowledgments

The author thanks his colleagues at the University of Michigan for all of their support during the sabbatical year which made Zplot possible. The author also thanks his wife for the numerous discussions she was forced to have about Zplot, which she did gladly and gracefully, whether she wanted to or not. Finally, the author thanks his two daughters, Anna and Maddy, for looking at some of the resulting graphs and "oohing" and "ahhing" at the appropriate times.

References

[1] Adobe Systems Inc. PostScript Language Tutorial and Cookbook. www-cdf.fnal.gov/offline/PostScript/BLUEBOOK.PDF, 1985.

[2] Adobe Systems Inc. PostScript Language Program Design. www-cdf.fnal.gov/offline/PostScript/GREENBK.PDF, 1988.

[3] Adobe Systems Inc. PostScript Language Reference Manual. www-cdf.fnal.gov/offline/PostScript/PLRM2.pdf, 1990.

[4] Stephen C. Grubb. Ploticus. ploticus.sourceforge.net, 2007.

[5] John K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the 1990 USENIX Summer Technical Conference*, Anaheim, CA, June 1990.

[6] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 15–30, San Francisco, California, April 2004.

[7] Thomas Williams, Colin Kelley, Russell Lang, Dave Kotz, John Campbell, Gershon Elber, and Alexander Woo. Gnuplot. www.gnuplot.info, 2007.

Tcl/Tk Tools for EPICS Control Systems.

R. Fox
National Superconducting Cyclotron Laboratory[1]
Michigan State University
East Lansing, MI 48824-1321

***Abstract*—The Experimental Physics and Industrial Control System (EPICS) is a control system in wide use in the control systems of accelerator laboratories across the world as well as in large-scale particle physics experiments. This paper will describe a Tcl package that provides access to EPICS control systems and a set of widgets that allow user interfaces to EPICS systems to be easily constructed. The extension will be compared and contrasted with the et_wish EPICS aware extended wish, and a justification for choosing to write a new extension will be given.**

I. Introduction and Outline

The Experimental Physics and Industrial Control System (EPICS)[1] is a distributed control system that is heavily used in nuclear and high energy physics experiments and accelerators. Los Alamos National Laboratories and Argonne National Laboratories originally developed EPICS and the EPICS organization supports further development and international use.

The National Superconducting Cyclotron Laboratory at Michigan State University is the leading accelerator laboratory in unstable heavy ion research in the United States and one of the leaders in the world. Our accelerator and beam-line controls are built around the EPICS control system. Several facility experimental devices, such as the S800 spectrometer [2], also feature EPICS in their slow control paths.

Recently several factors pushed me to investigate the use of Tcl to produce applications that interface with the NSCL EPICS system:

1. In my role as the software lead for the data acquisition system, I was getting an increasing number of requests to interface the data taking system in a read-only manner with data that could be obtained from the EPICS system. These requests ran the gamut from on-line monitoring of EPICS system channels during experimental data taking to inclusion of time varying control system parameters in the main event flow.
2. The Gas Stopping Cell[3], an experimental system, which performs high precision mass and half-life measurements on unstable nuclei could be run more efficiently and more effectively if it had available to it a system that sequenced several data taking runs while making new controls settings for the beam-line and gas cell EPICS parameters between runs.
3. The accelerator controls development group at the NSCL, after several years of "Windows only" console subsystems was looking for ways to create portable console applications.
4. The accelerator operators were looking for ways to get faster turn-around for desired changes in console applications and new console application development.

The remainder of this paper is organized as follows:

- Section II will provide a brief structural summary of EPICS and how EPICS control systems are typically implemented in the field.
- Section III will describe past work on interfacing Tcl/Tk to EPICS, why we did not choose to use prior art and what our requirements and desirements for an EPICS interface package were. A discussion of how we would structure our software is given as well.
- Section IV breaks in to three sub-sections. The first describes the low-level compiled extension that provides Tcl/Tk applications with access to EPICS control system channels. The second describes a set of Tk mega widgets that can be used to meet some control system needs irrespective of the underlying control system. The third describes a set of "EPICS aware" mega widgets that can be used to quickly build control system applications in Tcl/Tk.
- Section V will describe the status of the software, its level of adoption amongst the various development groups at the NSCL, and availability for outside use.

[1] The National Science Foundation under grant number PHY0606007 funded this work.

II. INTRODUCING EPICS

EPICS is a distributed control system that was originally developed collaboratively at Los Alamos National Laboratories and Argonne National Laboratories in 1989 as an off-shoot of the Ground Test Accelerator (GTA) control system at Los Alamos National Labs. EPICS has been adopted to control over 30 accelerators world wide, several large detection systems, telescopes and is also in use in several commercial applications/industrial applications. [4].

In the initial versions of EPICS, work was allocated to I/O controllers (IOCs), and console systems. The IOC systems at the time were typically board level embedded products running the WindRiver vxWorks Software[5]. As i386 computing became increasingly powerful and cost-effective, EPICS IOC software has migrated to these systems and can run on Windows32, Linux, and Solaris86 operating systems. Furthermore, for smaller systems, the line between the IOC and console computer blurs since general-purpose computers are capable of running elements of both components.

A typical EPICS deployment is shown below in Figure 1:

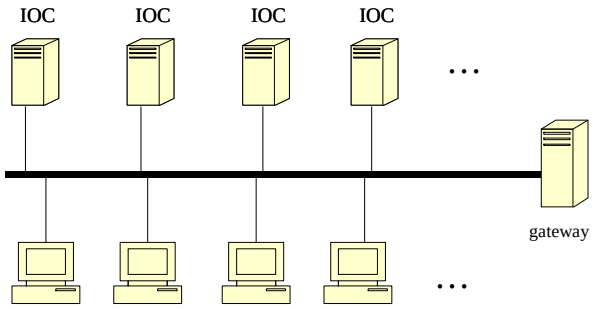


Figure 1 A typical EPICS Deployment.

IOC nodes are attached to the hardware either directly or, increasingly, via serial links and private subnets that they gateway on behalf of the EPICS channel access protocol. As more and more hardware interfaces are network capable, the IOC role is increasingly that of a protocol translator. Console systems run applications with which humans. The gateway system servers two purposes:

1. It is an access point that can determine which systems outside the EPICS control system are allowed to access EPICS channels and how.
2. It does broad/multi-cast traffic filtering. The EPICS channels (or process variables as they are called) are not listed in a centralized database. Instead a broadcast discovery protocol similar to ARP is used to locate the node that serves a specific process variable.

The EPICS process variable is stored as an IOC resident 'database record'. The name of the entry (e.g. ATHING) can typically be read to retrieve some hardware value. Descriptive information about ATHING may be found by reading other fields of the ATHING record. For example, the engineering units of ATHING are, by convention stored in ATHING.EGR.

The interesting thing about the EPICS channel access layer from the point of view of the console application is that there is no actual distinction, other than convention between accessing a process variable that represents hardware and a process variable that is some other field in the database record associated with that hardware.

The IOC software operates by cycling through database records calling handlers for each record that are intended to update the record's fields from the hardware and the hardware from the record's fields. Consider a simple example, a power supply. The power supply has a request voltage and an actual voltage. It an be turned on or off. It has a status that can describe its state that might be any of on, off, or interlocked. A record for this hypothetical power supply may have the structure shown in Table 1 below:

Table 1 A Sample EPICS database record.

Field	Meaning
PS1	Requested Voltage (write) Actual Voltage (read)
PS1.EGR	Engineering units of the requested voltage (read; returns "Volts").
PS1.STATUS	Status of the supply (read; returns "On", "Off" or "Interlocked").
PS1.REQ	Requested voltage (read only)
PS1.ON	Write 1 to turn on, 0 to turn off.
PS.TYPE	Type of record e.g. PSUPPLY

Note that by convention the name of the record is written to set the device and read to retrieve the actual value of the device. The database driven structure of EPICS provides several advantages.

1. Having created a record structure, and driver new instances of a power supply can be created by simply creating new database records and connecting them to the driver software (record fields not shown could provide actual hardware connection information to the driver, e.g. the serial port device the power supply was connected on, or a TCP/IP address).
2. Having described a power supply controller via a database record, only a new driver needs to be written to control a new type of power supply with similar application layer control characteristics.
3. Changing the hardware allocations of specific named devices is not a matter of changing software, but only

of changing the database and can be done while the system is running.

4. EPICS supports creating new devices by creating new database record types (structures), creating instances of them and device driver software to support them. Database records are described via a database meta-language that is used, in con junction with database definitions, to create record instances.

Each channel has a 'native data type', but all channels can be read as a string. This is a concept that is similar in nature to the dual ported Tcl_Obj used in the Tcl internals and API, however the 'native type' port is fixed and cannot be changed. Nonetheless, to some extent, software can be written that reads and controls EPICS channels that adhere to the Tcl EIAS (Everything Is A String) philosophy. It is also possible to obtain a process variable's 'native data type' and we will show in Section IV how we use that in the epics package to perform more accurate string conversions that EPICS itself does.

III. Tcl and Epics in the past

Research indicates two existing Tcl/Tk packages that support EPICS. These are ET[6], and IT[7]. These are both bundled in the EPICS caTCL extension. It turns out that IT is simply an extension of ET that can export data to the IDL data visualization and analysis tool[8]. I will therefore not discuss and analyze the strengths and weaknesses of IT as they are identical to ET with the additional requirement that IDL be available to make full use of its capabilities.

ET is delivered as an extended wish shell, et-wish. Et-wish provides the command **[pv]**. The **[pv]** command is an ensemble that allows Tcl applications to link Tcl variables to EPICS process variables, set process variables from EPICS channels and check the status of the connection between EPICS channels and the underlying application variables. There are some drawbacks however:

- Et is not a loadable package and requires a special shell; et-wish
- Et requires blt and internally uses its vector type.
- Et usage is not very Tclish in particular:
  - Tcl variables are type sensitive giving the impression that Everything Is Not a string
  - Tcl linked variables are not automatically updated by et-wish but must be manually updated and manually set.
  - Process variables themselves don't actually have a good object model. There's the PV command, and there are variables linked, there's no direct handle for a process variable that is being manipulated by the program.
- Et does not interface well with Tk, (because of the need to manually update linked variables)
- ET forces application designers to build widgets appropriate to control rather than providing a library of control widgets.

I felt the drawbacks of et-wish were sufficient to justify the effort required to build a new Epics interface to Tcl/Tk. Furthermore, since I already had epics channel access layer encapsulating classes, I felt I had a good leg up on that development process by interfacing these classes to Tcl through my Tcl++ partial encapsulation of the Tcl API.

The vision I had for Tcl/Tk support for EPICS is shown in the software-layering diagram below:

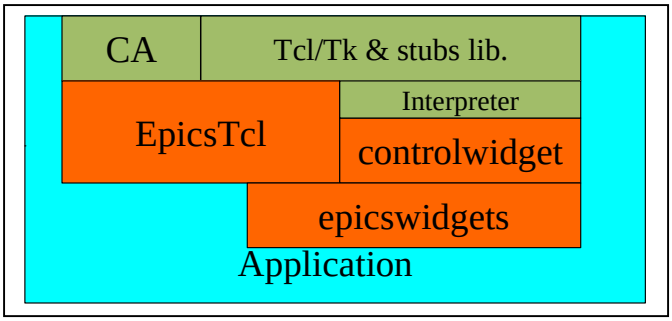


Table 2 below is a key to the boxes in the figure in figure 2.

Table 2 Key to figure 2.

Item	Meaning
CA	The EPICS Channel Access library.
Tcl/Tk&stubs lib	The Tcl API and the stubs library that provides a version independent front-end to it.
Epics/Tcl	A new loadable package that is stubs enabled providing Tcl-ish access to EPICS process variables.
Interpreter	A Tcl interpreter instance
Controlwidget	Pure Tcl widgets for arbitrary control applications.
Epicswidgets	EPICS aware mega widgets.
Application	A console application.

In the next section, we will describe the red components of this diagram.

IV. NSCL Support for Epics and Tcl/Tk

A. *The epics package*

The epics package is about 9000 LOC of C++ software, much of it (4300 LOC) the TCL++ wrapping of the Tcl API, and much of the rest (3000 LOC) a previously written C++ wrapping of the EPICS channel access layer (ca). The epics package (epicstcl for short) provides an object-oriented interface to EPICS process variables. This support is summarized in Figure 3 below:

The **epicschannel** command creates a new epics Process Variable object and a Tcl command that has the same name as the process variable. Operations on the process variable are performed via that command, which, as Figure 3 shows is an ensemble command.

```
epicschannel pvname
pvname get ?count?
pvname set value-list ?format?
pvname link tclVariableName
pvname unlink tcvVariableName
pvname listlinks ?pattern?
pvname updatetime
pvname values
pvname size
pvname delete
```

Figure 3 epicstcl Command summary.

Prior to describing how the package operates, I want to make a slight digression to describe some of the support epicstcl provides for 'programming in the large'. Programming in the large support considers the fact that almost certainly the same process variable will appear in different places on the same application simultaneously. This can lead to code sequences separated physically and temporally by a large distance like those shown below:

```
epicschannel achannel
achannel link achannelVariable1
…
epicschannel achannel
achannel link achannelVariable2

…
achannel delete
…
achannel delete
```

Which raise questions like:

1. What should the second **epicschannel** command on the same process variable as the first do?
2. What should the second **link** subcommand on the same process variable do?
3. What should **delete** do?
4. What should **unlink** do in the event a process variable is unlinked from its Tcl variable?

Good support for programming in the large requires that "the left hand not have to know what the right hand is doing" so that tight module and user interface coupling can be avoided. Therefore three design decisions were made to support programming in the large:

1. Process variable objects have a reference count and the epicstcl package internally maintains knowledge of the process variables that have been created. Duplicate process variables don't actually create another object, but instead increment the reference count. The **delete** operation similarly decrements the reference count and only deletes the underlying channel object/command when the reference count reaches zero.
2. The mapping of process variables to Tcl variables is one to many. That is more than one Tcl variable can be simultaneously linked to an epics process variable. Changes to the process variable are reflected in all linked Tcl variables, and a Tcl scripted change to any linked variable will cause a set to the underlying process variable (which eventually will cause a change in the value of the process variable that in turn will update the value of all the other linked Tcl variables).
3. Linked Tcl variables also have a reference count and epicstcl maintains internal knowledge of these links in a manner similar to the channel objects themselves. This supports a channel being linked to the same Tcl variable more than once.

The EPICS ca library provides 'channel access'. Ca allows access to EPICS process variables. In addition to allowing the application to poll the current values of a process variable, and to set new values, EPICS has an event model that supports notifying the application when an epics variable has "significantly changed". The significance of a change can be defined in the EPICS database records for a process variable.

EPICS performs this notification via threading, and the notification may occur in an arbitrary thread relative to the thread that requested the notification. It is therefore important to get the threading model right with respect to Tcl in order to avoid thread related failures in the Tcl interpreter.

Tcl/Tk supports an apartment-threading model. This model states that:

- A thread can have many interpreters.
- Each interpreter can for the most part be interacted with only in the thread that created it (each interpreter has only one thread).
- API Functions exist to post events to the event loop of an interpreter running in an arbitrary thread.

On the other hand, it is not possible to predict which application thread will receive an EPICS update notification. Therefore the epicstcl loadable package updates Tcl variables by posting an event to the interpreter that owns that variable rather than directly updating the variable itself.

Initial versions of the package always read the string version of the channel in keeping with Tcl's EIAS philosophy. Users discovered, however that EPICS's floating point to string representation conversion functions were inadequate, especially for process variables containing small values. For example, a beam current monitor that was displaying a few nano-amps of beam (e.g. $5x10^{-9}$ nA) would be converted to the string "0.000". Therefore, the epicstcl package reads each process variable in its native type. When the channel connection event is processed, a native-type to string converter is associated with the native data type.

The threading model of EPICS also leads to some interesting edge cases. Consider the script:

```
epicschannel achannel
achannel     delete
```

The first command expresses an interest in the EPICS process variable *achannel*. This:

1. Creates a new Channel object in the extension. The channel object requests an attachment to the process variable named *achannel*.
2. Creates the Tcl command **[achannnel]**
3. In a separate thread, EPICS will notify the Channel object that the process variable was successfully located and attached. This happens asynchronously.
4. Once the channel has been successfully attached, the channel object can express an interest in update notifications.
5. Update notifications can then proceed asynchronously and in an arbitrary thread.

The second command declares the application is no longer interested in the channel. This:

1. Detaches the channel from EPICS
2. Deletes the **[achannel]** command.
3. Deletes the channel object

The script shown will typically delete the channel object before the asynchronous notification that the channel has been connected and, often, prior to the actual connection itself. Thus care must be taken to cancel these notifications or to discard notifications for channels that have been already deleted.

Similarly each low-level channel object has associated with it a semaphore object (implemented on Unix-like systems as a pthreads semaphore and on Windows systems as a Critical Section) to ensure synchronization of internal data structures within the multiple threads that may be executing in an object. These are wrapped in objects that acquire the synchronization primitive on construction and release on destruction so that code of the form:

```
{
    CriticalRegion lock(id);
     ….
}
```

Will maintain the appropriate lock discipline even in the presence of C++ exceptions. Tcl semaphores are not used because this level of the code is intended for re-use in non-Tcl applications.

B. The controlwidget system independent widgets.

While EPICS is the dominant device control system at the NSCL, there are other control systems in simultaneous use. These include various small Labview systems as well as some ad-hoc systems for special purpose applications.

The Widget support for building console applications is therefore broken into two layers. The lowest layer provides some re-usable widgets that are independent of the control system. These operate very much like normal Tk widgets in the sense that they may have **–variable** options or **set/get** methods that some control system aware software can use to manipulate the widget appearance to correspond to the appearance of some control system parameter.

Snit[9] was used to create these widgets. I have had many pleasant experiences using Snit as a mega widget framework, and this project was no exception.

The following widgets were written:

- Led – An indicator that simulates a light.
- Meter – A vertically oriented rectangular meter.
- RadioMatrix – A rectangular array of radio buttons that can be used to chose one possibility from several.
- TypeNGo A type in widget coupled with a button that commits the value in the entry to the control system. The entry supports validation that is invoked when the button is clicked. This allows the application to be certain that a variable that expects a number gets a number e.g.

To give a sense for how these widgets work, Figure 4 below shows a test script for the meter widget. In this case, the 'control system' is just a proc that runs every 100ms and jitters the meter value.

C. *The controlwidget EPICS aware widgets*

The ultimate intent of our work is to make it easy to create control system applications for EPICS at the NSCL. To do this I have also written a set of EPICS aware widgets. In most cases, EPICS awareness means that these widgets have a **–channel** option that binds the widget to display/control a specific process variable in the EPICS control system.

The EPICS aware widgets have been implemented as a mix of **snit::widget** and **snit::widgetadaptor** 'classes'. Where possible, they are implemented on top of the widget set described in part B. of this section. For example, there is an **epicsMeter** widget. This is implemented in terms of the **meter** widget described in section B.

```
package require meter
namespace import controlwidget::*

set metervar      0.5

set jiggleMax     5
set jiggleAmount  0.1

meter .meter -variable metervar \
             -from -1.0 -to 1.0
pack .meter

proc jiggle ms {
    global metervar
    global jiggleCount
    global jiggleMax
    global jiggleAmount

    after $ms [list jiggle $ms]

    set jiggle [expr   \
          rand()*$jiggleAmount -  \
        $jiggleAmount/2.0]

    set metervar [expr $metervar + \
                $jiggle]


}

jiggle 100
```

Figure 4 Test script for meter.

The EPICS aware widgets that have been written include:

- **EpicsButton:** provides several types of epics aware buttons including a pair of buttons for e.g. on/off a single button that can toggle on/off states, and a button that can a process variable to an arbitrary value when clicked.
- **EpicsEnumeratedControl:** provides a wrapping of the **RadioMatrix** widget described in part B of this section.
- **Epicsgraph:** provides a wrapping of the BLT graph widget that allows one to graph the time evolution of one process variable against the time evolution of a second (see also **Epicsstripchart**)..
- **EpicsLabel, EpicsLabelWithUnits:** provides a read-only display of a process variable or a process variable with its engineering units.
- **EpicsLed:** an EPICS aware wrapping of the **Led** widget described B. above.
- **EpicsMeter:** an EPICS aware wrapping of the **meter** widget described above.
- **EpicsBCMMeter:** an EPICS aware wrapping of the **meter** widget along with range controls suitable for use with NSCL Beam current monitor devices.
- **EpicsPullDown:** an EPICS aware pull down menu that can present a set of choices for the value of a process variable.
- **EpicsSpinBox:** an EPICS aware spinbox.
- **EpicsTypeNGo:** an EPICS aware wrapping of the **typeNGo** widget.
- **EpicsStripChar:t** an EPICS aware wrapping of a BLT stripchart widget that allows time series data for epics process variables to be displayed.

V. Software status and level of Adoption

The software is currently stable at version 1.4-001. This version has been tested on Windows XP,2000 Linux and MAC OS-X. I have used this software routinely in my work providing EPICS interfaces to the experimenters. It is provided on the conference CD along with some installation instructions. See the software subdirectory of the CD subdirectory for this paper.

I have not been able to interest the NSCL controls group in this software. Instead they have embarked on an ambitious project to build portable user interfaces using Qt and C++. They estimate this to be a multi-year project, however in the meantime, laboratory administration has given the go-ahead to accelerator operators with software development experience to use this to develop their own user interface software and they have done so with great enthusiasm.

The epicstcl package and associated mega widgets have served as an enabling platform for the NSCL accelerator operators to get functioning control panels that meet their needs with better turn-around times than they have had in the past, and without waiting for the completion of the Qt/C++ application

framework described in the previous paragraph. The project has yielded benefits both for our experimental user group and for the operations program at the NSCL.

VI. References

[1] http://www.aps.anl.gov/epics/index.php
[2] http://www.nscl.msu.edu/tech/devices/s800
[3] http://www.springerlink.com/content/r3224712r7n17864/fulltext.pdf
[4] http://www.aps.anl.gov/epics/projects.php
[5] http://www.windriver.com/products/platforms/
[6] Bob Daly http://www.aps.anl.gov/epics/EpicsDocumentation/ExtensionsManuals/TclTk/et.tcltk.html
[7] Bob Daly http://www.aps.anl.gov/epics/EpicsDocumentation/ExtensionsManuals/TclTk/it.tcltk.html
[8] http://rsinc.com/idl/
[9] http://en.wikipedia.org/wiki/Snitsnit

14’th Annual Tcl/Tk conference
Sept. 24-28 2007
Bourbon Orleans Hotel, New Orleans, LA USA

TCL/Tk Develoment and Debugging tools

Xtrace: a high-level extension of Tcl-trace

Florian Murr
Siemens AG - Corporate Technology
Otto-Hahn-Ring 6
D-81739 Munich, Germany
+49 (0)89 636-44949
florian.murr@siemens.com

Manfred Burger
Siemens AG - Corporate Technology
Otto-Hahn-Ring 6
D-81739 Munich, Germany
+49 (0)89 636-43209
manfred.burger@siemens.com

ABSTRACT
In this paper two Tcl-packages written in pure XOTcl [3] are presented. The first "Xcom" is yet another socket-communication package; the second "Xtrace" uses Xcom to provide an observer command across multiple processes called "`xtrace`" that is modeled along the lines of the well-known Tcl "`trace`" command, but much more high-level.

Keywords
Xtrace, Tcl-trace, observer-pattern, distributed applications, distributed user interfaces, model based.

1. INTRODUCTION

Tcl has much of the famous "observer pattern" built into the language via the well-known Tcl-"`trace`" command.

The *observer-pattern* consists of some entity that might be accessed and any number of observers who want to be informed about these accesses.

Tcl- "*`trace add variable`*" provides observer-functionality at core language level in a slightly modified way:

Any trace-callback-command, i.e. "observer", does not only get informed, but can intercept any attempted access and decide whether to let it pass, to modify, or to block it. This reflects the very low-level and immediate nature of Tcl's trace command.

Being so low-level "`trace`" is best suited for extending Tcl with new commands or control constructs.

The observer pattern on the other hand is often used in a distributed context and is much more high-level in spirit.

If an observer resides in a different process than the observed variable, intercepting an attempted access is out of question. Even getting informed of every change of the observed variable is sometimes too demanding.

Within a distributed observer pattern, consistency matters more than immediate callbacks! - Therefore Xtrace favors high-level comfort and consistency over direct access.

Let's first have a look at the mindset that spawned Xtrace and than at a simple example.

2. BASIC SETTING

The basic distributed setting we had in mind when developing Xtrace consisted of:

- A "model" composed of XOTcl objects that reside at the server.
- Multiple "clients" that are observers-of / actors-on this model, residing in different processes or even on different machines.

The clients are able to change the model, through setting of variables, or through method calls and these changes get propagated back to the clients, assuring that all clients mirror the current state of the model. The process in which the observed variables reside is henceforth called the "*model*" (or "*server*"). [This notion is not completely correct, since xtrace allows in principle that the model- objects / variables are distributed among multiple processes, but we not yet made use of that possibility.]

The "*clients*" ("*observers*") are typically interested in state-transitions of the model. Such a transition of one consistent state to the next often involves the change of more than one variable. A high-level feature like xtrace will therefore propagate the cumulated delta after all the simple variable changes are complete. Fortunately Tcl's event-loop gives a handle to find moments when a consistent state should have been reached.

2.1 EXAMPLE 1

Consider some facility with several workplaces.

Appliances to be checked travel through these workplaces and get different tasks performed upon them. Every workplace has a "current appliance" variable and every appliance has a variable for every task to be performed.

So in our example the situation at the model might be something like this:

```
% ::workplace7 set currAppliance
::app12
% ::app12 set pTest
::task42
% ::task42 set values
{123 234 345}
% ::task42 set state
ok
% ::task42 set remarks
{}
% ::task42 set worker
::person773
% ::person773 set firstName
John
% …
```

Some monitor may show the “id”, “state” and “remarks” variables of the current task on the current appliance at that workplace.

With Xtrace one may make use of

```
obj xtrace add ?-soft? chain vars cmd
```

and code this in a single command-line:

```
::workplace7 xtrace add \
        {currAppliance pTest} \
        {values state remarks} \
        [list ::monitorCallback]
```

2.2 WELL-KNOWN OBJECTS

Xtrace works in the object-oriented setting of XOTcl and any use of it is directed to some Xtrace-participating XOTcl-object.

Objects participate in Xtrace, when they provide a “toModel” method, which returns the peer-object, to contact its model. [There are some classes that may be used, either as base-class or as “mixin” that provide default implementations for clients and server.]

Some of these objects should be known at coding-time to any client of an application, to kick-start Xtrace-communication. These are the so called “*well-known objects*”. In example 1 “::workplace7” is treated as such a well-known object.

These objects only “live” fully on the server, but the client at least knows that on the server there exists some object with this name. – Usually the client also knows the type (classes) of these objects. Other objects may not be known at coding-time, but get delivered to the client during runtime. The client then could use another “xtrace add” to listen to variables of those objects, too.

The client version of an (well-known) object is just a stub. It will only have those variables that get mentioned in “xtrace add”. These variables will be created as needed. The stubs usually do not have the functional implementations for methods either. Methods just send there call to the server.

[Actually there are some Meta-classes of Xtrace that provide these functionalities. If you use

- “instproc” the method is implemented on the server and the client,
- “instprocModel” the method exists only on the server,
- “instprocDist” the method gets distributed in the sense above.

But to get the idea, we usually abstract away such technicalities in this paper.]

The model version of a well-known object is thought to be the real object, be fully functional and have all the necessary variables and methods.

2.3 VARIABLES

All the chain- and vars- variables exist in the client and on the server. Just setting a variable (on the client or server) suffices to let the change automatically be distributed. Since communication takes place “after idle”, one may change many variables until the next consistent state is reached and not until then the dissemination of the cumulated changes starts.

2.3.1 “Chain” variables

The "*chain*" in example 1 is the list "`{currAppliance pTest}`". Such a chain consists of variable names. These variables are supposed to hold object-names as values or to be empty. The idea behind "*variable chains*" is that starting with an already known object (e.g. "::workplace7" which is known à priori) which has a variable whose name is the first element in the chain ("currAppliance"). This variable has some object-name as value, which holds the next variable and so on.

2.3.2 “Vars” variables

The “*vars*”-list contains names of variables of the last object in the chain. Normally these variables are XOTcl- class attributes of some application specific class. They are not restricted to specific value-types, as the chain-variables are. Our practical experience has lead us to quite often use “`dict`” values [2]. (We used the Tcl8.4 backport of “`dict`”.)

2.3.3 Slow clients

“`xtrace add`” has an optional flag “-soft”, that is intended for slow clients. The client may be considered slow, because the connection, or the machine of the client are slow, or because the client has some time consuming task to perform every time he gets informed by xtrace.

As a guiding example consider some rapid changing image, i.e. some pseudo-movie and clients with different speeds trying to be in sync, i.e. to show the same image. The variable “currImage” on the server changes with some predefined frame rate. Since the server may not show the image at all, but only hold the “currImage” variable, speed is no concern there. If a client wants to show the image, that may take some time (especially with Tk). Without precautions in this respect, every change would be sent to this client. In other words, messages arrive faster on the socket, then the client can get them off and the socket would be jammed.

Here the “-soft” option comes to the rescue.

If a variable gets traced “soft”, then the server just sends an “are-you-ready” message to the client, whenever the variable changes. – The client responds when he is idle. When this response arrives at the server, the up to then cumulated changes get sent to the client. On the up-side the client avoids clogging of his socket and stays in sync, on the down-side, he might miss some intermediate changes and skip some of the image to be shown.

2.4 CALLBACK INTERFACE

When the callback-command “*cmd*” of “`xtrace add`” gets called in the client the name of a “parcel”-object gets appended to the command. This is an object of type “XtrcParcel” which provides methods to comfortably extract all the information concerning the changes that have happened.

Sometimes one is not really interested in getting the callback, but is content when the variables in chain and vars are kept up-to-date. This behavior is achieved by giving an empty callback-command to “`xtrace add`”.

In either case, should any variable in "chain" or "vars" have its value changed, then the change will be distributed to the clients that are interested in this variable; i.e. all the clients that have an “xtrace” registered for this variable and those whose chain happens to contain the variable in question.

On the client all the variables that contribute to any of the registered xtraces get mirrored from the server and get updated with the current values before the callback command is called. Additionally there is an interface on “parcel” that allows retrieving the previous value of any of the variables in “chain” or “vars”, so that the transition “oldValue→newValue” for any of these variables is available during the callback call.

Should for example "currAppliance" be set to "{}" the values of "pTest" and its vars are no longer valid too. All changes in the chain gets distributed and the callbacks called. The clients also need no longer listen for changes on variables of the previous “pTest”. Or should another appliance, with some other “pTest”, take place in the “::workplace7”, the new corresponding values get distributed. This is all handled automatically in Xtrace.

3. SYNOPSIS

The following commands are provided by Xtrace:

```
obj xtrace add ?-soft? chain vars cmd
obj xtrace remove chain vars cmd
obj xtrace info chain vars cmd
```

“obj” may be any well-known XOTcl object for Xtrace or an object which is actually in some chain traced by xtrace.

“-soft” allows for slow clients.

"chain" is a list of variable names. The corresponding variables must contain an object name or be empty.

"vars" is a list of variable names, of the last object in the chain.

"cmd" is a Tcl or XOTcl callback command.

Further details of the meaning of the different arguments of “`xtrace add`” are described in chapter 2.

“`xtrace remove`” and “`xtrace info`” are most of all self–explanatory. We do not describe it here.

4. REMARKS

Xtrace goes especially well in combination with the "`dict`" command. Since a dict is a value, it may be the value of any of the "vars" in an xtrace. Therefore one can bundle values in a dict at the model, to facilitate observation.

The Tcl event loop has extraordinary power in synchronizing asynchronous calls! Since xtrace (interprocess-) communication takes only place when the process get "idle", some quasi-simultaneous changes from different clients get synchronized by the Tcl-event-loop. When the process becomes "idle" again, xtrace assumes that a consistent state has been reached, that gets distributed to the clients. - It is of course possible to program against this assumption, but with only moderate consideration it is possible to develop very stable and consistent distributed applications.

The xtrace interface is very high-level, since the user just focuses on the logical "chain of variables" and all the work of keeping the corresponding values up to date gets done by the xtrace package.

Another feature that makes Xtrace high-level is the allowance of slow clients. In that case Xtrace doesn't force the frequent changes upon the client (which would clog the communication socket), but informs that there has been some change and waits then to be requested by the client to send the current cumulated changes.

In very complex applications one even might consider not to have only one model-process, but to distribute the model over multiple machines. This should be no problem to xtrace, since every one of the "well-known" objects knows how to contact its model and these may well be in different processes for different objects.

5. COMPARISON OF “trace” AND “xtrace”

Xtrace is rather similar in spirit to the low-level Tcl-“`trace add variable`”. Here is a list of similarities and differences between "trace" and "xtrace:

- “trace” is low-level monitoring, intended to be able to extend Tcl with new infrastructure. That is exactly what xtrace uses Tcl-trace for!
- “xtrace” is intended for high-level observing. The new structures support monitoring of consistent changes of different objects and variables.
- "xtrace" is written in XOTcl, whereas "trace" is a Tcl core feature.
- "xtrace" allows to observe more than one variable in one call, it even allows so called “*variable-chains*” to be observed.
- "xtrace" does not call the callback command, the very moment the variable is accessed, but distributes the cumulated changes when the process becomes idle.
- "xtrace" specializes on "write"-access to variables. Read-only access of variables is not in its scope.
- "xtrace" may observe variables in objects in different processes.
- "xtrace" is bidirectional. Variables get mirrored in the observing client and changes get propagated in both directions.
- "xtrace" callback gets delivered a "parcel" that contains both: the old value and the new value of the variable.
- "xtrace" communicates using the "xcom"-package that uses "sockets" for communication.
- "xtrace" allows clients that are quite slow to participate. ("-soft" option)
- "xtrace" allows the callback to be empty. - Only the variable values keep getting synchronized.
- "xtrace" allows dynamically to extend the observed objects. The chain of variables need not exist.

6. THE “Xcom” PACKAGE

Xcom is yet another socket communication package, similar to the well known “comm” package [1].

Unfortunately "comm" had some limitations that lead us to reimplement its functionality using XOTcl. (Some conditions

dubbed "race-conditions" in comm are quite natural in xtrace and handled gracefully there.)

Xcom allows file-transfers to be triggered and a callback is executed once the file-transfer is complete.

The request for some file lets the called partner create a temporary file-server for this request and reply to the requesting-partner, the host and port of this file-server. The requesting-partner gets the file from the server and the callback command is evaluated. - This works in both directions, from Xcom-server to Xcom-client, or from client to server.

Xcom has some utility functions built-in for xtrace.

XOTcl allows to "mixin" classes dynamically, which makes it very easy to account for clients "speaking different languages". One client may speak "Tcl", just plain Tcl commands, another client might use "XML", say a Flash-UI using ActionScript-XMLsocket, still another client wants encrypted messages. Xtrace just mixes in the appropriate encoding classes. - As a result, Xtrace can communicate with each client in its preferred communication language.

7. REFERENCES

[1] Comm., http://tcllib.sourceforge.net/doc/comm.html

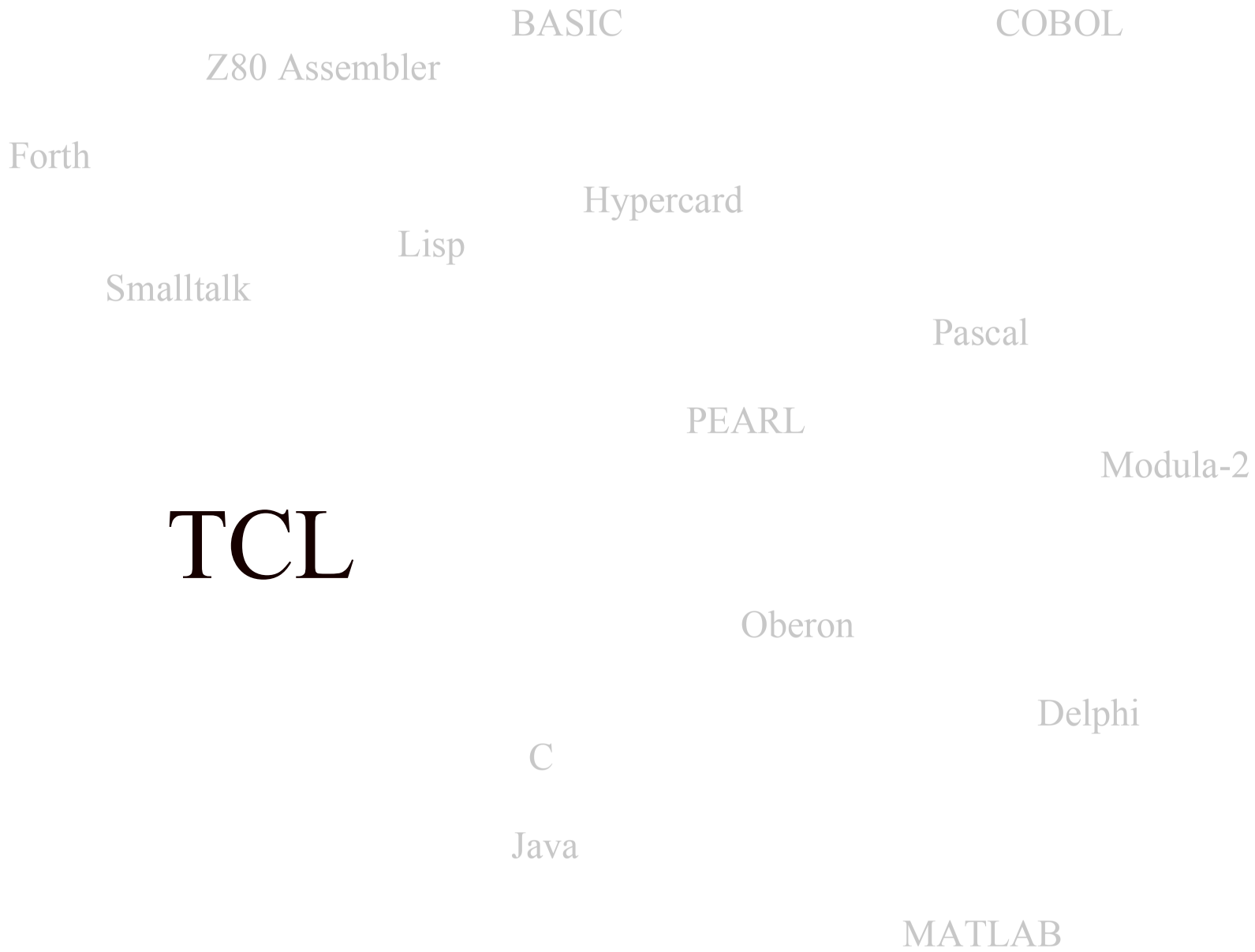
[2] dict, http://www.tcl.tk/man/tcl8.5/TclCmd/dict.htm

[3] XOTcl, http://www.xotcl.org/

Presenting eti

- How i learned to stop worrying and created megawidgets -

Axel Nagelschmidt http://axn.dyndns.org


BASIC
COBOL
Z80 Assembler
Forth
Hypercard
Lisp
Smalltalk
Pascal
PEARL
Modula-2
TCL
Oberon
Delphi
C
Java
MATLAB

How it all began ...

Started programming on a calculator TI-59

Built the Sinclair ZX-80 from a soldering kit after a holiday in England

Advanced to ZX-81, ZX Spectrum, Atari ST 520, Macintosh SE

Studied Computer Science and Mathematics in Erlangen

Worked for several institutes and clinical departments during years of study, designed PC Card for measuring tasks with DMA
Started business to assemble PCs according to customer requirements

Used CP/M, GEM, DOS, Windows, Mac OS, Minix, IRIX, Solaris, Linux, Mac OS X and other systems

Together with a friend founded a computer company positioned in trainings and consulting, mainly on java and OO techniques
Preparing course materials, administrating Solaris and Linux machines for classes and developers, first contact to TCL

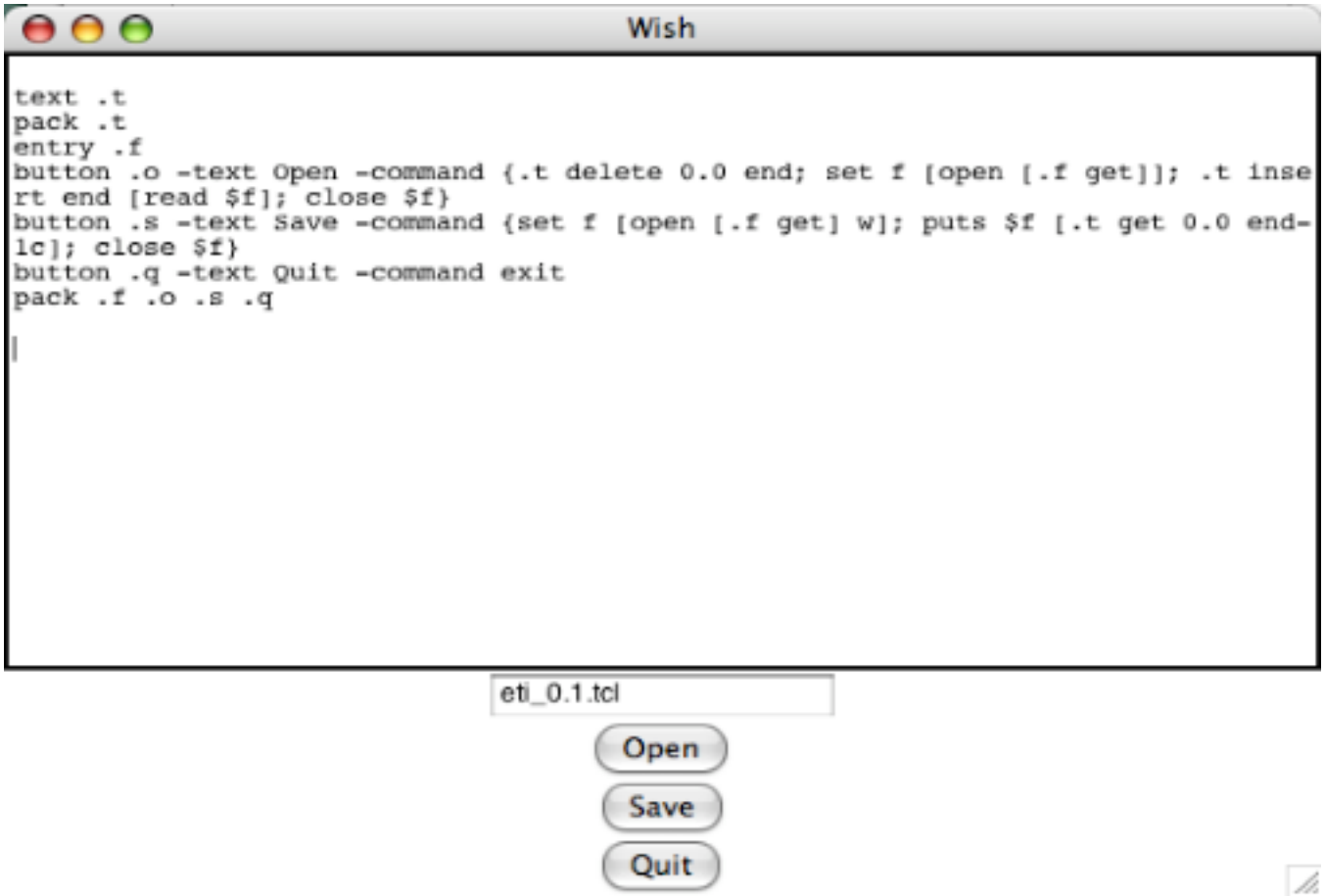
Working since 9 years in a company for biomedical technology, as developer, database developer and Solaris administrator

Today running Ubuntu on PCs, Mac OS X on several Macs (PPC), Solaris on some SUNs, being forced to use Win-XP at work

Until now met John Mc Carthy, Niklaus Wirth, Douglas Adams (SIGGRAPH Keynote in New Orleans 1996), happy to meet more experts here !

Writing a complete editor is possible after one month of learning TCL!

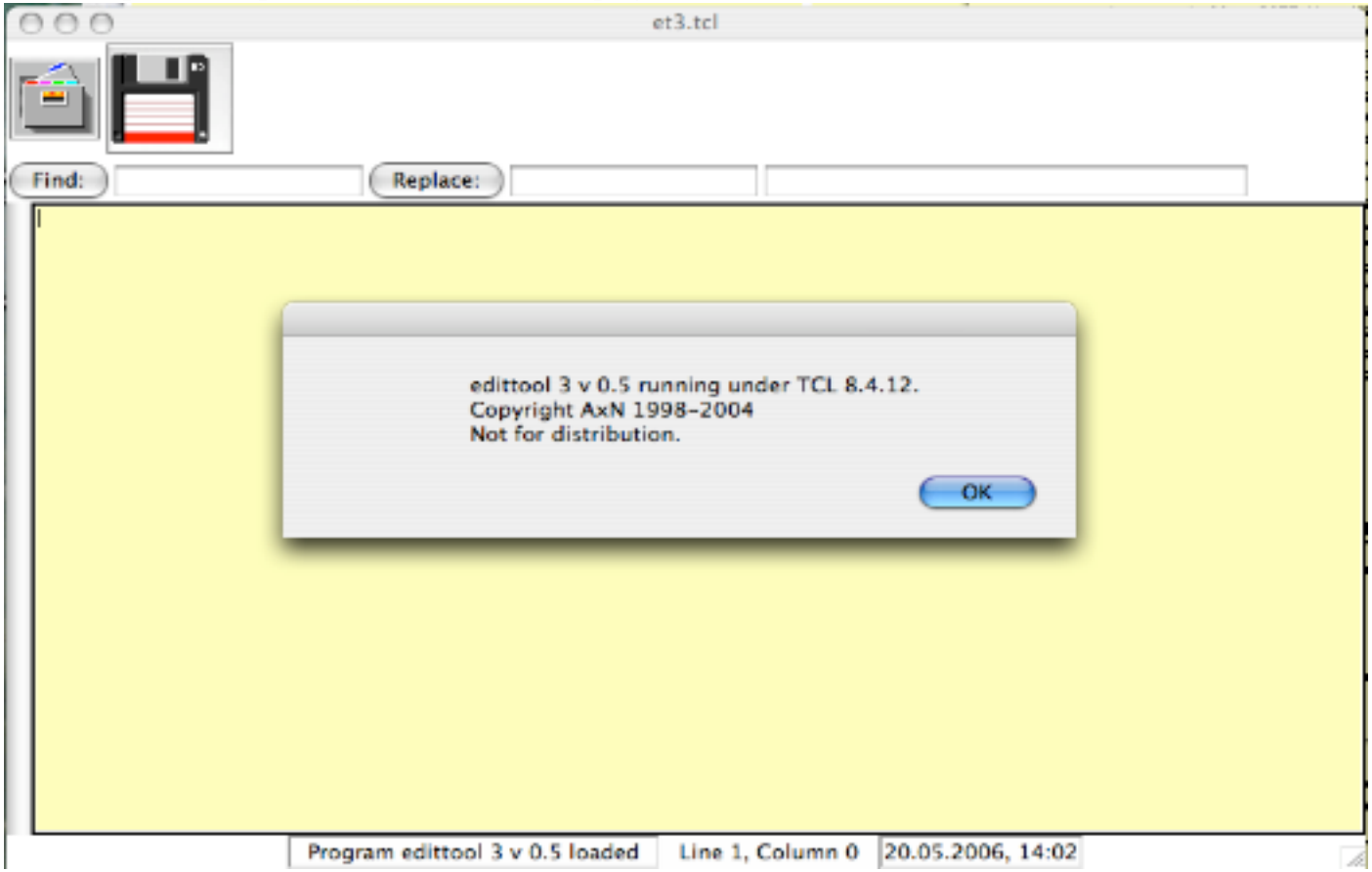
I wrote something like this around the year 1997:



eti 0.1, nearly feature complete

Even better version 0.3:

Has menubar (not shown here, screenshot from Mac)
Options to save / save as / open / close file, toolbar, find and replace
Options to change font family / size / background colour
Status bar with status info, cursor position and clock
Can mail it‘s content
Ported to Solaris (because no change necessary)
No widespread use, probably because never deployed



Started database programming with TCL and oratcl, network programming cross platform

Found TCL be able to connect to a database very easily

Created an online transaction processing application, processing tens of thousands messages per day from cardiac pacemakers

Needed to write a multiplexing front end to an SMS sender/receiver that allowed only one connect from a client

Added monitoring tools to watch processes, checks for availability of hosts, DB listeners, Webservers, disk space, load ...
Running several instances of tclhttpd on production and test servers to watch processing and resources, lightweight, easy to use

Using tequila to synchronize processes and have a look at process figures from several clients

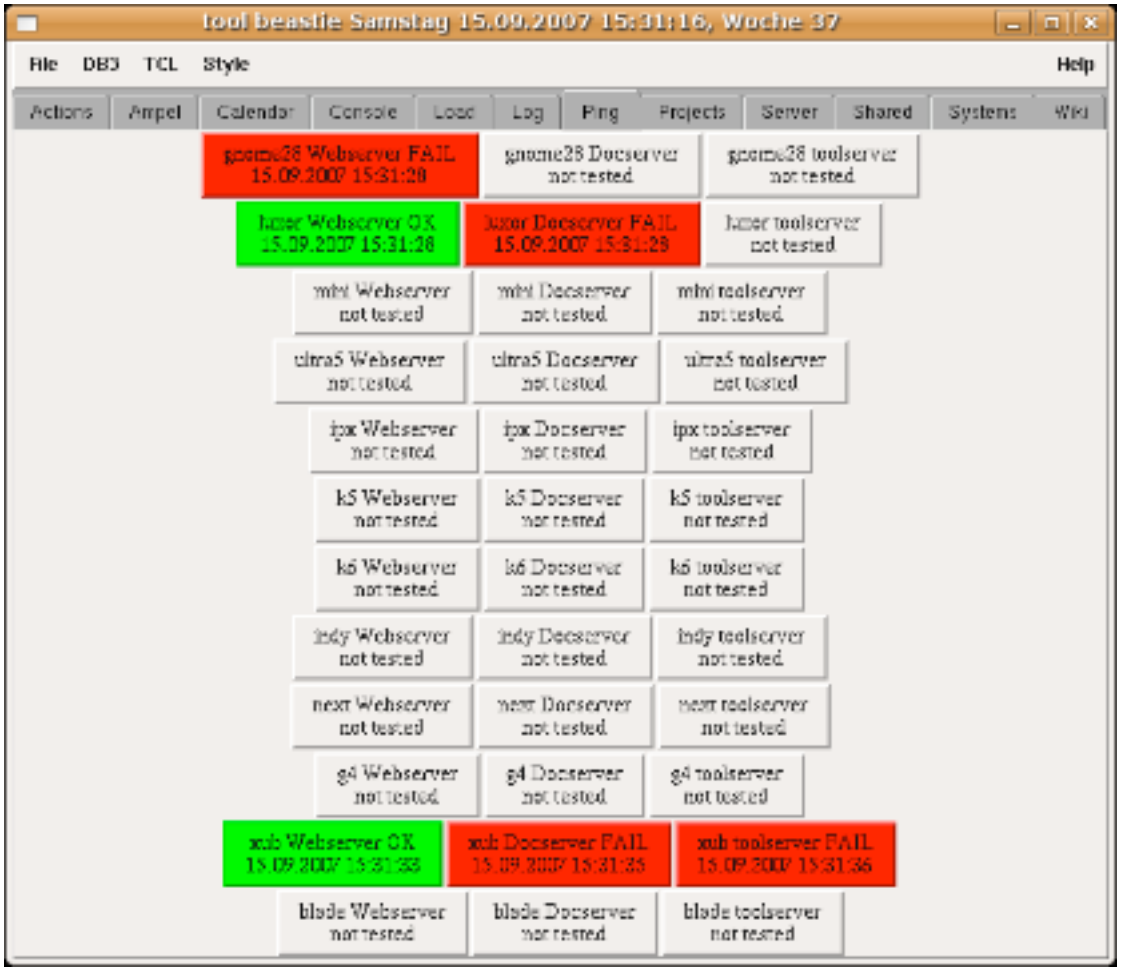
Also created more and more tools for me and collegues

- simple application
- single window
- some buttons and entries
- database connection
- monitoring tools
- text transformation
- database repair tasks
- XML parsing and creation of TCL and SQL scripts from description of data modules

Each of these sharing many properties with other tools, having it‘s own method for persistant storage like **.**files, ini-files etc.

A tool is born ...

Collected these into a common tabnotebook, without any interaction between the modules yet:



Usage of object orientation seemed to be a good idea:

- ◇ Have a mini editor for any kind of text. Derive HTML editor, TCL editor etc. from this.
- ◇ Have a set of classes needed for graphic programs, derive different graphical drawings / simulations from this.
- ◇ Specify a set of common used actions to include into the menu or a toolbar
- ◇ Have multiple instances of the same type, so can use a notebook with several tabs
- ◇ incr TCl offering classes and incr TK offering megawidgets like notebook seemed natural

But (1) ...

- ◇ incr TK seemed to be more complicated than necessary
- ◇ incr TK seemed to be slow because of code overhead

Solution I:

- ◇ create one superclass drawing the toolbar, the notebook and the status bar
- ◇ create one superclass in incr TCL (not TK!) for all instances (called etimodules)
- ◇ have each etimodul draw it‘s own frame
- ◇ inherit superclass etimodul to take care of common needed methods

But (2) ...

- ◇ creating multiple windows with notebooks needed another superclass
- ◆ inheritance for widgets was not really used
  - ◇ tcledit or diffedit could not really inherit the class miniedit, different count and layout of widgets
- ◇ learned about tile and increasing / ongoing support for Mac OS X

Solution II:

- ◇ recreated some widgets using tile components
- ◇ added menu to change style to any existing tile theme
- ◇ some usage of mkWidgets, because better creation of megawidgets was anticipated

But (3) ...

- ◇ mixing iwidgets with tile widgets was no good idea
- ◇ mkWidgets usable for megawidgets, but not really an OO system
- ◇ mkWidgets not being supported any longer?
- ◇ not much time to try something new

Then ...

- ◇ visited the 5th european TCL User meeting (and organized the 6th one)
- ◇ got more involved using tequila, thus making shared apps an easy possibility
- ◇ learned about integrating OpenGL to create apps using 3D graphics
- ◇ learned more about tile, the notebook api has some differences
- ◇ learned more on creating fullfeatured apps
- ◇ learned more on mixing text and graphics to create integrated learning experiences
- ◇ learned more on tablelist and it‘s integration into tile

Other influences happening in this time:

- ◇ More usage of Java and especially Eclipse at work
- ◇ Our company introducing Lotus Notes, me visiting a developers course
- ◇ Seeing that TCL can control much of windows through twapi
- ◇ .NET and mono appearing in widespread use, new IDEs anjuta and monodevelop
- ◇ Applications becoming more complex, each one developing their (own) makro language
- ◇ Seeing that TCL can integrate calls to graphic libraries like ImageMagic or OpenGL
- ◇ Understanding that integration in big consistent frameworks like Oberon or Squeak makes usage much easier
- ◇ Anticipating to learn that TCL can integrate calls to control OpenOffice.org
- ◇ Learning about snit and it‘s ongoing support and integration with TCL 8.5
- ◇ Writing more applications as helpers or for database access, needing a usable and robust framework to do so
- ◇ Expecting user interfaces to allow users much more choice and flexibility - ”graphical programming”
- ◇ Not wanting to learn a new P*-language every year
- ◇ Seeing that TCL is here to stay, easy to use, easy to teach or use as description language, cross platform

=> Decided to base framework and apps on TCL, integrating good usage examples into an usable IDE

(OK, did so before with Pascal, Oberon and Delphi ...)

Moving from incr TCL to snit

Topic	incr TCL	snit
Header	itcl::class <name>	snit::widget <name>
Inheritance vs. Delegation	inherit <upperclass>	delegate <method> to <otherclass>
Building widgets	frame $w, delivered as argument	set w $win, frame already created
Reference to instance	$this	$self
Calling own methods	<method>	$self <method>
declaring method	method <name> <args> <body>	method <name> <args> <body>
Destroying instance	itcl::class delete object $this	destroy $self
Overwrite class definition	no, only after deletion of class	yes, like the TCL way
Inherit methods	yes	no, but delegation helps

Rewrote about 5 major classes and the core of the framework in about 3 days.

Need to rewrite toolbar, and some handy megawidgets used from mkWidgets to snit.

First results seem to be very promising.

Skeleton of an etimodule:

```
snit::widget somepage {

        typecomponent super
        delegate method * to super
        typeconstructor {set super [etimodul %AUTO%]}

        variable w

        constructor args {

                # create the modules widgets
                # initialize variables

        }

        method start {} {
                # perform actions, start timers etc.
        }

        method onshow {} {
                # perform actions when module is shown again
        }

        method onhide {} {
                # perform actions when module is hidden
        }
```

```
	method f9 {} {
		# refresh, reload actions when requested to do so
	}

	method clone {} {
		# create a clone of the same type of module
	}

	method close {} {
		# stop timers, free resources, close this module
	}


	method whatever {} {
		# do something
	}

	# ......


}
```

All methods not implemented in a specific etimodule are called in the super class by delegation.

Some of them are empty, others do the easy and safe default action, e.g. clone, aboutme

A digression on network and monitoring:

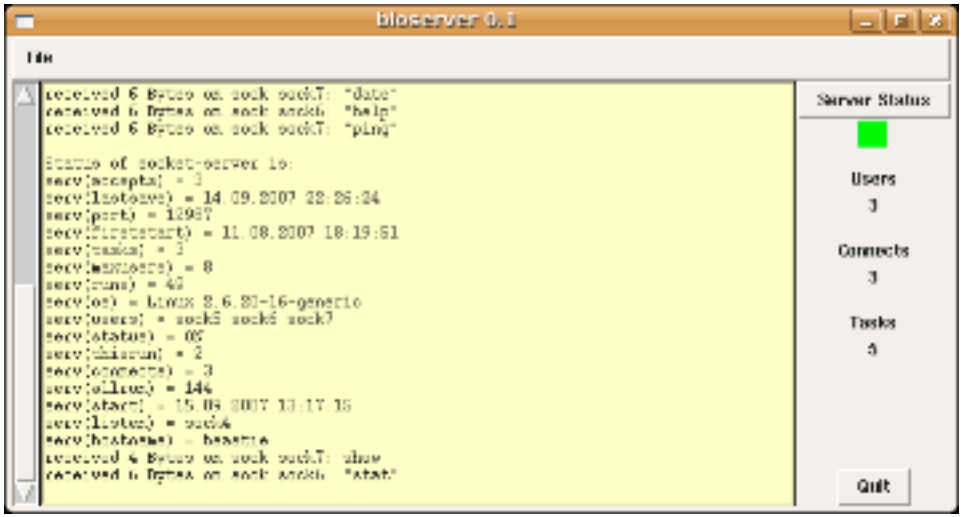
Needed a small TCP/IP connected client / server tool for cross platform monitoring and maintenance applications

Synchronous **reliable** remote calls to hosts or services that are down lead to sometimes intolerable delays in client tools

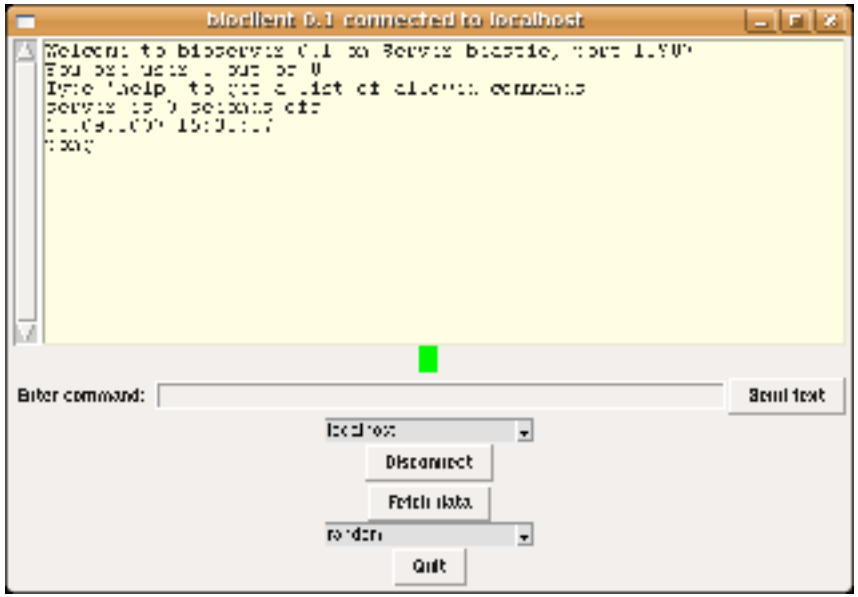
Until now we could live with tequila for process synchronization, carefully dividing into modules which may encounter delays and those that need short response times

Some problems remain, will explore more about this later. Maybe UDP can help?

Also using tclhttpd modules as application server, after integrating SQLite database into the webserver.



<==



Webclient with SQLite database should make deployment of application and upgrades very easy ...

```
#!/usr/local/bin/tclsh


# idea from my wiki 147, thanks to DKF
# stripped down, no interp, working and tested

package require Tk
package require http

proc http_source url {
        set token [http::geturl $url]
        set script [http::data $token]
        http::cleanup $token
        eval $script
}

http_source http://deployment.local/tclapps/initapp.tcl
```

(Remember the first version? 7 lines of code is all that is needed, the possibilities are endless ...)

Upgrading and migrating through tests is a reliable way ...

Also a good way to document bugs and show when they are fixed

- [x] `tcltest::test achterbahn test2 -body {math3::achterbahn 39} -result 34`
- [x] `tcltest::test fibonacci test1 -body {math3::fibonacci 14} -result 377`
- [x] `tcltest::test fibonacci test2 -body {math3::fibonacci 19} -result 4181`
- [x] `tcltest::test roman test1 -body {math3::roman 25} -result XXV`

Lately had to follow strict QM standard procedures, parts of TCL application now certified for medical usage in clinical studies

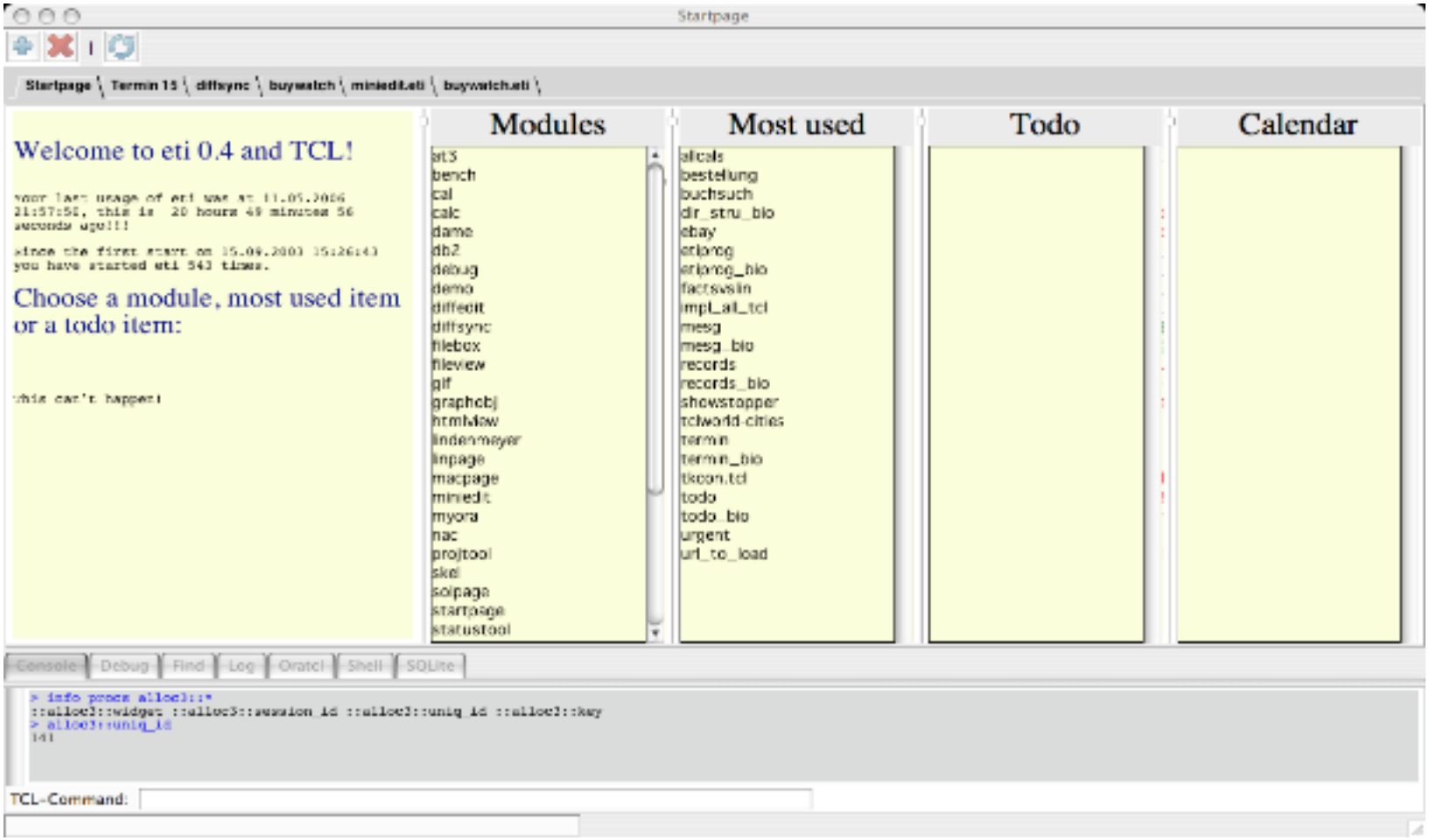
Requirement No.	Description of requirement
REQ-628-003.012	The new module shall be called routines3 and conform to the standard calls for etimodules. It shall be placed in the etilib2 module library.
REQ-628-003.016	The module shall have a method demo to start a small demo showing it's features.
REQ-628-003.018	The module shall have a method to reliably predict the weather for the next 20 days.

Test-driven development helps to structure requirements into code and to prove that all usecases have been considered:

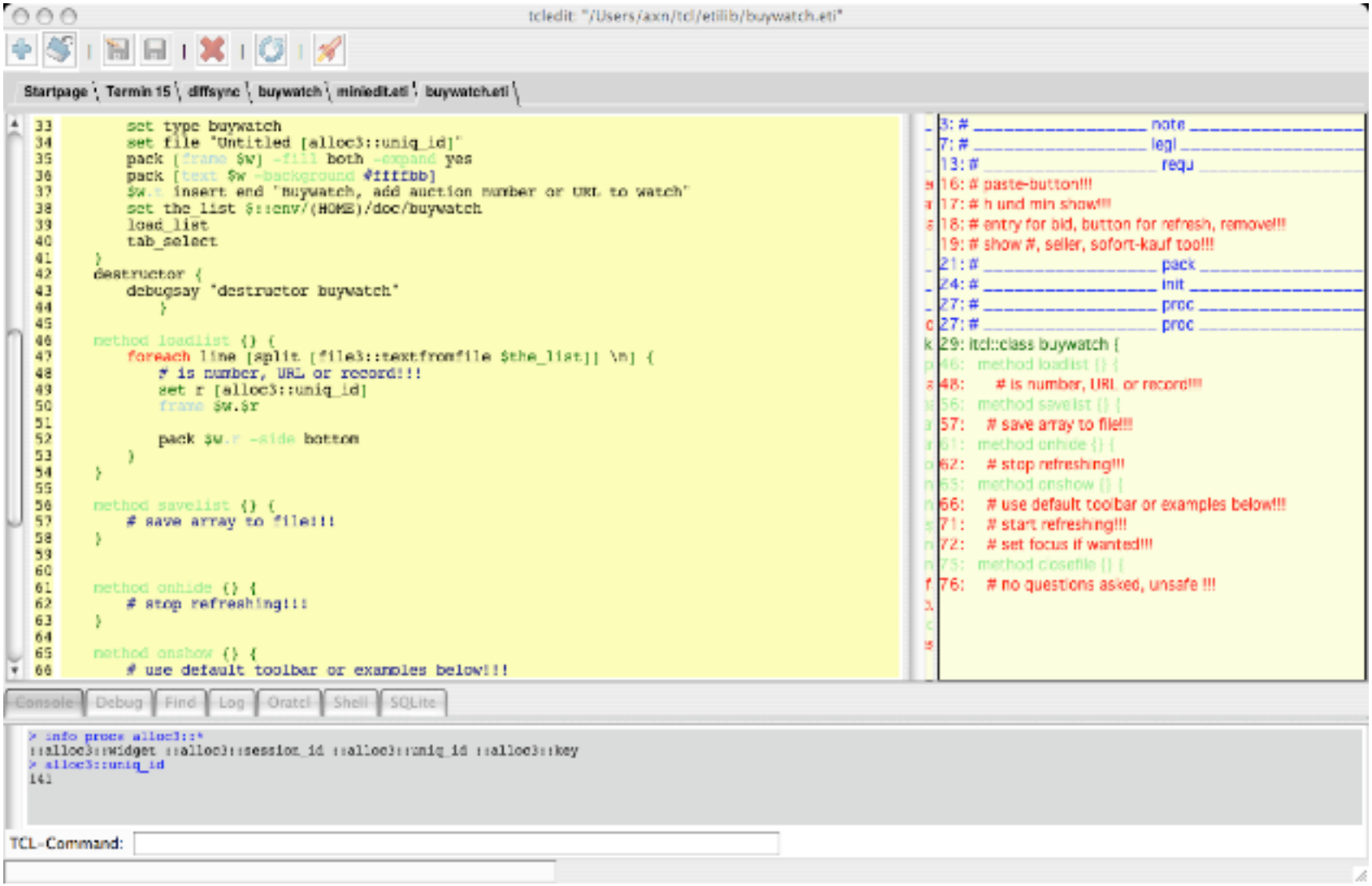
- [x] `tcltest::test routines3 exists -body {file exists $tclpath/etilib2/routines3.tcl} -result 1`
- [x] `tcltest::test routines3 demo -body {routines3 .a; expr {[lsearch [.a info methods] demo] >= 0}} -result 1`
- [ ] `tcltest::test routines3 test2 -body {# not so easy to write the test} -result {# the programmer will implement whatever he thinks is correct}`

Examples of etimodules:

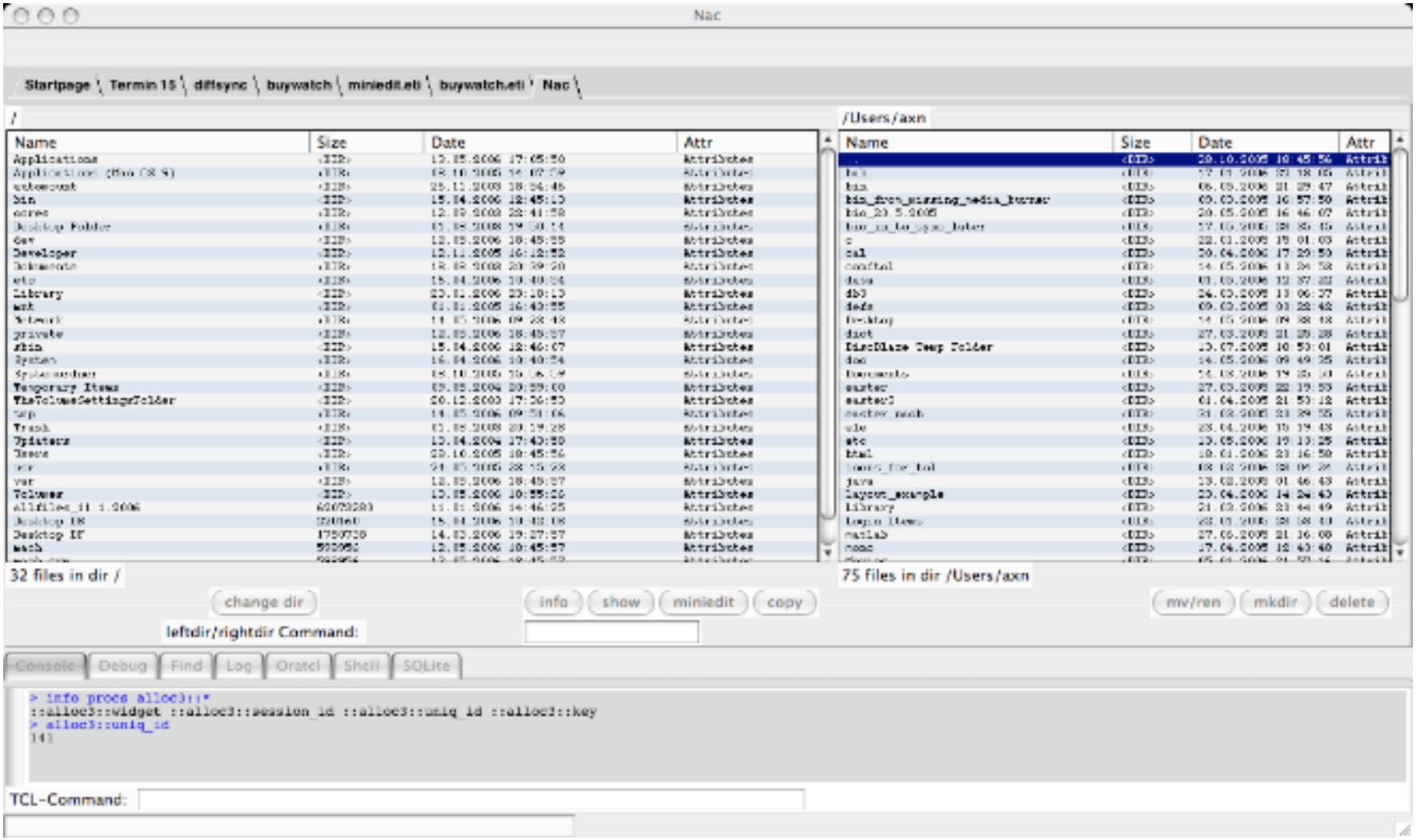
Startpage, allowing to start other modules, showing most used documents, todo and calendar
Older framework for helper applications (terminal, consoles for sqlite and oratcl) shown below



tcledit, editor with syntax colouring based on ctext
Allows working in projects and libraries, highlights requirements and proc / method headers



nac, (not another commander), because this is my main productivity tool
Using tablelist, but need more intelligent bindings for keyboard operation and cleaning of layout



Using a database in the application, using the application in the database ...

Many possibilities with metakit, starkits etc. exist

Coming from Oracle, i found SQLite the most easy method to use a lightweight though very powerful database in my apps

Already had a personal wiki inside of eti, using structure of this to migrate the project into the database, shown in demo now

```
sqlite>
sqlite> .tables
addr   cal    code   prefs  todo   wiki
sqlite> .schema wiki
CREATE TABLE wiki (id id, item string, text string, first date, count integer, last date);
sqlite> .schema code
CREATE TABLE code (id id, type string, value string, name string, first date, count number, last date);
sqlite> select distinct type from code;
app lib module
sqlite>
```

Still using cvs in filesystem, but will move versioning into database soon, have code archive on other servers in net.

There is still more to come and more todo ...

TCL 8.5 should be ready soon (or may be it is already?)

Check tcl modules (see TIP #189) and convert my libraries to new format

API of etimodules has to be frozen at some point for deployment

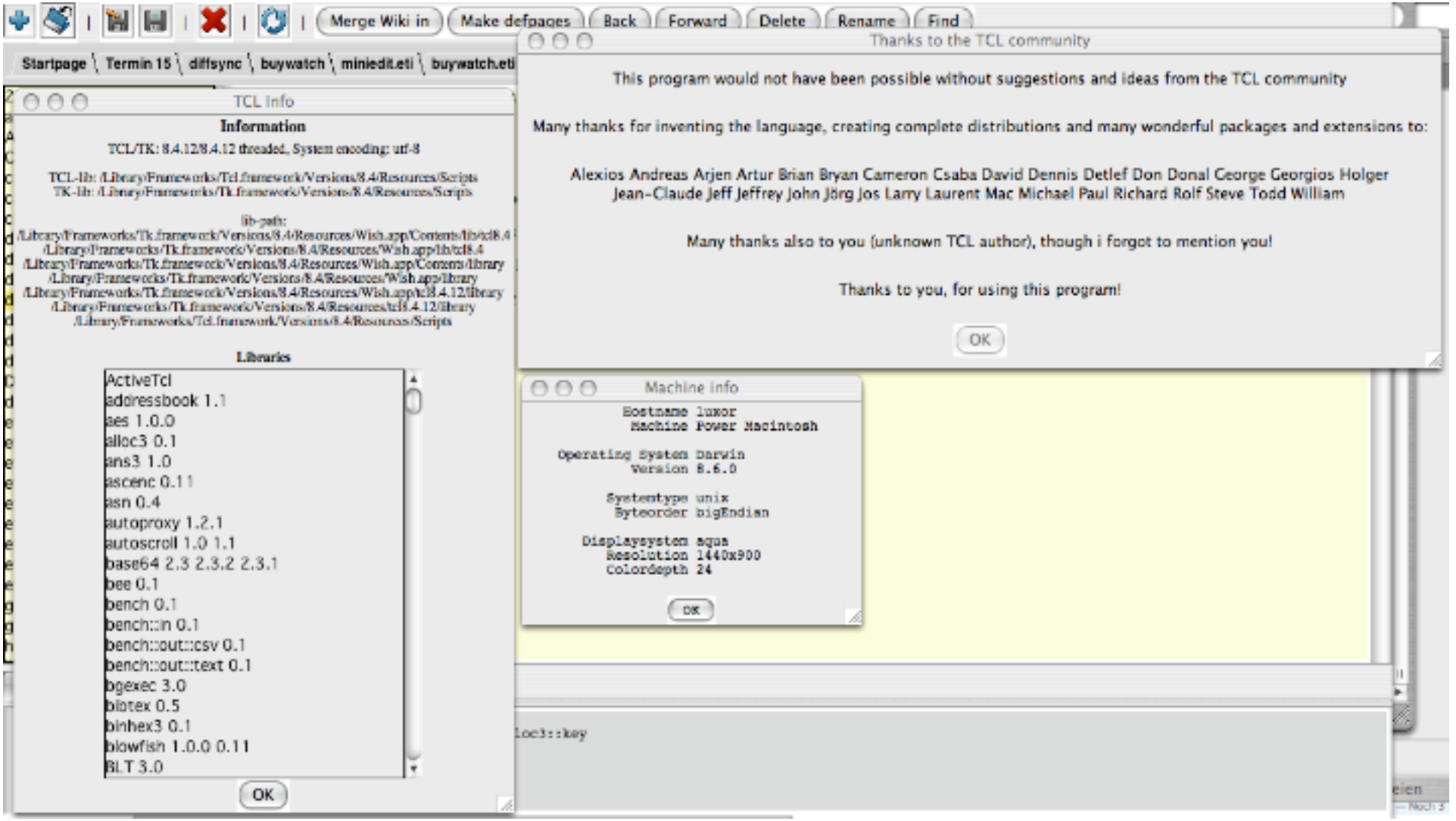
Update webserver and have code repository there

Solve some problems with timeout waits on TCP/IP

Have a single application framework for easier deployment of apps, perhaps using freewrap?

Follow development of Eclipse, Lotus and the Komodo Project ...

info, to show information about the application
Thanks again, this is really a tool made possible by the community!



14’th Annual Tcl/Tk conference
Sept. 24-28 2007
Bourbon Orleans Hotel, New Orleans, LA USA

Tcl as Glue

Tcl in the Middle

Michael A. Cleverly
Intermountain Healthcare
4646 W Lakepark Blvd
Salt Lake City, Utah 84120
michael.cleverly@intermountainmail.org

September 27, 2007

Abstract

Tcl has long been recognized as an excellent language to glue existing components together to create new applications. Tcl is just as useful when interjected into the middle of functioning N-tier "enterprise" systems.

SockSpy[1] is probably the best known[2] example of a "man in the middle" Tcl application. What may not be as widely appreciated, however, is that Tcl's strong TCP sockets and event-drive I/O make construction of custom "man in the middle solutions" (or proxies) quite straightforward.

These custom solutions solve real business problems often at a fraction of the cost of other potential solutions. This paper will look at a handful of examples where Tcl has been employed in this manner at Intermountain Healthcare.

1 About Intermountain Healthcare

Intermountain Healthcare[3] is a non-profit integrated healthcare delivery system headquartered in Salt Lake City, Utah and serving communities throughout Utah and southeastern Idaho. Intermountain employs over 26,000 people throughout its system of hospitals, clinics and healthplans.

Founded in the 1970s, Intermountain has been pioneering the use of information technology in healthcare since that time. Intermountain is a recognized leader in its uses of information technology in healthcare[4]. In 2005 Intermountain began a 10-year partnership with GE Healthcare to collaborate and jointly design and develop new electronic health record technologies[5][6].

2 Facilitating a corporate rebranding initiative

Prior to November 2005 Intermountain Healthcare was known as "Intermountain Health Care," or more often simply IHC. A major corporate rebranding initiative was launched in the end of that month to remove the space between the words "health" and "care" and eliminate references to IHC as an acronym.[7]. Several months later the health plans division

formerly known as “IHC Health Plans” changed its name to SelectHealth[8] completing the corporate rebranding initiative.

Prior to the rebranding, Intermountain's principal public web site had been powered by Expressroom, a proprietary Java & XML based Content Management System (CMS). Expressroom has had a rocky history, being acquired and then sold by a series of different owners[9]. In short Expressroom was viewed internally as a deprecated technology that needed to be replaced going forward.

When the rebranding initiative was announced internally, the various departments responsible for content realized they would need to modify substantial amounts of content to rid all URLs of any reference to the IHC acronym.

The old home page was located at http://www.ihc.com/xp/ihc/; the new home page was to be located at http://intermountainhealthcare.org/xp/public/.

While standard web technologies (such as Apache's mod_rewrite[10]) could be used to redirect incoming links to legacy URLs two problems would have remained:

1. Extra overhead from absolute links within the existing content that would incur extra redirection overhead
2. The user's browser would still show the now verboten acronym when a user moused over a link

Since the departments responsible for the content did not have the man power to change all of the Expressroom content prior to the date of the public announcement, Intermountain's Enterprise Web Operations team adapted the opensource AOLserver[11] web server to act as a “rebranding proxy.”

For those unfamiliar with the particulars of AOLserver's Tcl API[12], it is (trivial) to delegate the handling of portions of the URL space to specified Tcl procedures (which are invoked once for each request).

```
ns_register_proc GET  / rebranding-proxy
ns_register_proc HEAD / rebranding-proxy
ns_register_proc POST / rebranding-proxy
```

In the above example, any URL requested at or under / (i.e., the whole site) will be processed by a Tcl procedure named “rebranding-proxy.”

The algorithm employed by our rebranding-proxy is quite straightforward:

1. Get the requested URL using *ns_conn url*
2. Use *regsub* to munge the new-style URL (/xp/public) into the old form Expressroom uses (/xp/ihc)
3. Make a connection to the Expressroom application server and request the munged URL
4. If the MIME content type returned by Expressroom matches text/* then use *string map* to fix up any embedded links using the old-style URLs by translating them to the new format
5. Return the response to the user passing along the same HTTP response code and most of the same headers received from Expressroom

It is worth noting that not all response headers should be returned from Expressroom to the user. Specifically, if we've altered any URLs within the HTML (or CSS or Javascript, etc.) then the Content-Length header will absolutely need to be recalculated.

3 A restricted authorization proxy for static content

Intermountain Healthcare has standardized (for the time being) on Vignette's Portal (VAP) and Content Management (VCM) products for audience-focused intranet and extranet dynamic portals[13].

In this model, portlets running within the VAP application server consume dynamic content from the VCM database and render it to HTML. The VAP application servers are front-ended by multiple Apache web servers. Static VCM content, for performance reasons, is pushed out directly to the web servers and served by Apache.

As long as the static content was benign images or content meant for public (unauthenticated) consumption, this approach to managing static content was acceptable. Over time as portal adoption grew, some business units wanted to begin publishing static content that needed to be restricted more tightly than merely through the obscurity of its URL.

Based on the success of using AOLserver as a "rebranding proxy" Intermountain's Enterprise Web Operations team constructed a similar proxy, dubbed "portal-rproxy" to restrict access to static content to logged in authenticated users. Rather than having Apache retrieve files directly from the local filesystem we used mod_proxy[14] to point Apache to an AOLserver instance running on a non-privileged port bound to the local loopback interface.

Users who are authenticated to VAP will have a session id cookie. This cookie is only transmited over SSL between the user and the web server to prevent eavesdropping attacks. Because the SSL connection has already been terminated by Apache, *portal-rproxy* has access to it in cleartext. We leverage this fact to call forward to the VAP application server to see if the session is still valid and active; only if it is, do we serve up the static content from the file system (using AOLserver's *ns_returnfile* API call).

To make it possible for system administrators to change access restrictions without needing to modify Tcl code, we created two configuration files that are read at runtime: *noauth.conf* and *restrict-per-site.conf*.

The *noauth.conf* file contains a list of regular expression patterns. Blank lines, lines made up of only whitespace and lines where the first non-whitespace character is a # (i.e., comments) are ignored. Such a file would look like:

```
# Allow all access to .css stylesheets
\.css$

# Logos, etc. for the public areas of the portal
^/Public/Images/
```

The *restrict-per-site.conf* file contains URL patterns and a list of portal subsite(s) the logged in user must have access to in order to retrieve the content. A hypothetical entry in such a file would look like:

```
restrict-url "^/Surgery/Schedules/" to physicians
```

For performance, the results of a succesful check of the validity of a session id can be cached for a few minutes (to reduce the overhead of repeatedly checking while retrieving multiple static assets referenced from a single portal page). We recommend keeping this cache window fairly low (no more than several minutes) to limit the amount of time static content could be refreshed after a user has logged off.

As with any web application that uses session cookies if an attacker can obtain the session cookie (i.e., via an Cross Site Scripting (XSS) attack[15]) they effectively become the user.

Our portal-rproxy neither widens this risk (as compared to having the application server handle the static content) nor does it mitigate it any.

4 Pseudo source-NAT'ing with tcpsymlinks

The basic skeleton of a functioning man in the middle proxy in Tcl can easily be written to fit on a single printed page[16]. For example:

```
socket -server accept listeningPort

proc accept {client addr port} {
  if {[catch {socket -async destHost destPort} server]} then {
        shutdown $client
    } else {
     fconfigure $client -blocking 0 -buffering none -translation binary
     fconfigure $server -blocking 0 -buffering none -translation binary
      fileevent  $client readable [list glue $client $server]
      fileevent  $server readable [list glue $server $client]
    }
}

proc glue {src dst} {
    if {[catch {puts -nonewline $dst [read $src]}] ||
        [eof $src] || [eof $dst]} then {
        shutdown $src $dst
    }
}

proc shutdown {args} {
    foreach sock $args {catch {close $sock}}
}

# Enter the event loop
vwait forever
```

Just as a symbolic link (symlink) in a file system serves as "a special type of file that serves as a reference to another file"[17], we introduce the notion of a "tcpsymlink" which is just a listening port on a particular IP address that proxies traffic to another address and port.

Our *tcpsymlinkd* daemon looks in a *ports/* directory for files named either *description.port* or *description.port.interface*. The *description* portion of the filename serves merely as documentation for those administering the server the daemon is running on. The daemon listens on the specified port on either all interfaces or the one specified.

Each *ports/* file is expected to contain one line containing the hostname (or IP address) followed by a space and a port number. When a new connection comes in a new outgoing connection is made to this location and the two sockets are "glued" together in much the same way as the code skeleton above shows.

The daemon polls periodically (typically every fifteen seconds) to see if any of the *ports/* configuration files have changed, been deleted, or added. Changes only affect future connections; existing connections are not disturbed. If a *ports/* file has been deleted, the listening

socket is closed preventing future connections. Likewise if a new *ports/* file has been created, a new listening socket will be opened.

On occasions when the daemon needs to listen on a prvileged port, it must be started as root. In these cases it is recommended to drop root privileges as soon as the initial listening sockets are opened. Either the TclX extension[18] or a small C extension (such as the one included with TclHttpd[19] or one written using Critcl[20]) can be used to *setuid* to a non-privileged user.

5 A Tcl web server with a One Track Mind

OTM[21] is a web server written in Tcl that answers all requests in exactly the same way. This turns out to actually be a useful feature, especially when combined with tcpsymlinks.

Instead of having a web server configured to front-end an application server, we instead have the web server talk to a tcpsymlink which in turn talks to the application server. When it comes time to do periodic scheduled maintenance, a helpful downtime message can be served up using OTM and the tcpsymlink can be temporarily repointed away from the application server. When the downtime is over, the tcpsymlink can be changed back. All of this can be done without changing any of the configuration settings of either the web server or the application server and is potentially less error prone.

6 Front-ending an existing system with SSL

Another man in the middle use where Tcl shines is the ease with which existing applications can be extended to support SSL connections using the TLS extension[22].

In the simplest case, an application that already calls Tcl's native *socket* command need only call *::tls::socket* instead, possibly specifying some additional SSL-specific configuration options.

One recent implementation of an SSL-enabled proxy at Intermountain dealt with a new radiology image viewer. Several hundred physicians (not directly employed by Intermountain but who have admitting rights at Intermountain hospitals) and their offices and clinics have hardware VPN tunnels that allow them to access certain components of Intermountain's Electronic Medical Record (EMR) software.

The existing EMR system, written in Java, added significant overhead to the transmission of images because it would download the entire image from the radiology system, buffering it in memory before it would send any data back to the web server (which would only then begin to transmit the data back to the end user's browser). This added a roughly 10x penalty compared to directly accessing the radiology system.

Ideally the images would be transfered via an SSL connection over the hardware tunnel. Although the hardware VPN provides encryption for the data as it traverses the public internet, at the other end of the VPN tunnel the traffic is no longer protected to any eavesdroppers on the local LAN. End-to-end SSL encryption thus provides additional security against eavesdropping.

Using the existing SSL-enabled Apache web servers that front-end the EMR to proxy the radiology images provided somewhat better performance than having the EMR handle the

images directly, but the speed was still 2x to 3x slower than retrieving them directly from the radiology system (without SSL).

Since the VPN tunnels require configuration on both ends of the tunnel, adding a new address (say that of the radiology system itself) would require coordination between both Intermountain engineers and the (often contracted) IT staff that the affiliated physicians employ to manage their computer equipment. Coordinating and implementing such a change could easily have consumed 500-man hours of labor.

Instead, using Tcl code basically equivalent to the man in the middle skeleton shown previously, with *socket -server* replaced with *tls::socket -server* running on the same web servers on a high port (so no VPN tunnels needed to be reconfigured) affiliates were able to access the radiology image viewer using SSL. The TLS package was compiled to take advantage of the hardware SSL acceleration cards already present in the web servers. The net result is that there is no noticeable difference between requesting an image through our SSL-enabling Tcl proxy compared to accessing the radiology system (non-SSL) directly.

7 Deterministic load balancing

Our final example of using Tcl as a man in the middle proxy involves deterministic load balancing. If we have a pool of N application servers we choose an application server to route to by taking the final octet of the requester's IP address mod N and using the corresponding application server. For example:

```
set servers [list host1 host2 host3 host4]
set octet   [lindex [split $ip_addr .] end]
set choice  [expr {$octet % [llength $servers]}]
set backend [lindex $servers $choice]
```

Typically an application server vendor will supply a "plugin" for various web servers[23][24]. The job of the plug-in is to spread the load across the various backend application server instances. Since these plug-ins are traditionally proprietary and closed source, their algorithmic decision making process is somewhat opaque.

At Intermountain, we are phasing out our use of the WebLogic plug-in, replacing it with a Tcl man in the middle proxy we call "wlpr-proxy" (short for "WebLogic Plugin Replacement Proxy"). With the vendor's plug-in, active users would occasionally be redirected to a different application server instance for no apparent reason. This was very frustrating for end users (clinicians hate to have to repeat entry of lengthy patient notes merely because some piece of software routed them to the wrong destination). Since the problem was intermittent and not reproducible on demand, it posed a frustrating challenge for both quality assurance (QA) and support personnel alike.

WebLogic's plug-in can be configured to log debug data. On a busy web server, however, all the debug data from various connections quickly becomes intermingled and is nearly impossible to disentangle. Thus, a design requirement for our wlpr-proxy replacement was the ability to log each connection individually. We chose to have the capability of logging all client request headers (for GET, HEAD and POST requests) and server response headers (for GET and HEAD requests). For privacy reasons, we do not log any form data that the user POSTs or any of the servers response headers to the POST request.

The ability to reconstruct the precise sequence of end user requests and responses has proven to be a useful resource for both QA testers and developers. Several instances of obscure

corner-case bugs have already been identified via the enhanced logging that wlpr-proxy provides. In the past, because we lacked visibility, some of these bugs would have slipped through testing and into production.

8 Two caveat to keep in mind

When writing line-oriented proxies with Tcl versions prior to 8.5, it helps to keep in mind the potential Denial-of-Service (DoS) condition discussed on the Tcl'ers Wiki by Donald Porter and George Peter Staplin[25]. In non-blocking I/O mode a readable fileevent will trigger when new data is available on a channel even if an entire line is not available.

A malicious user could send excessively long lines (without ever sending a newline) forcing Tcl to eventually exhaust all of its available memory, panic and abend.

With the inclusion of TIP #287[26] in Tcl 8.5, a new subcommand of *chan pending* can be used to introspect how much buffered data is available to be read and enforce application-specific limits appropriately. For those interested (and some may not be since this problem is largely theoretical and rarely seen in the wild), prior to Tcl 8.5 several different mitigation techniques are possible:

1. Write a small C extension to expose the existing Tcl_InputBuffered and use that to introspect the amount of unread data
2. Set an event some number of seconds into the future using *after* and take some action (i.e., using *read* instead of *gets* or aborting the connection) if a complete line has not been read by then
3. Rewrite your application to use *read* instead of *gets*

The second caveat to keep in mind is that it is worth remembering to call *fblocked*: "The fblocked command returns 1 if the most recent input operation on channelId returned less information than requested because all available input was exhausted[27]."

If you are writing a proxy that inspects HTTP requests the end of the client's request headers is signified by a blank line[28]. When *gets* returns a blank line it could be because the line was blank (end of request headers) or there wasn't a complete line in the buffer. Calling *fblocked* (or in Tcl 8.5 *chan blocked*) allows the program to distinguish these two cases and avoid a failure to parse subsequent request headers.

For an example showing the use of both *chan blocked* and *chan pending* together, see this[29] December 2006 thread on the comp.lang.tcl newsgroup.

9 Not just IPv4 and TCP

Although the Tcl core itself only supports TCP IPv4 sockets various extensions exist which provide support for other protocols.

- TclUDP[30] provides UDP sockets and is available for both Windows and Unix systems
- IOCPSOCK[31] is a Windows extension providing faster IPv4 TCP sockets as well as IPv6 TCP and IrDA sockets
- Ceptcl[32] is a Unix-centric extension providing UDP, IPv6 and raw IP sockets
- hping3[33] is a low-level packet assembler scriptable with Tcl

10 Conclusion

We have seen that custom application-specific proxies can be quite easily written in Tcl. These Tcl solutions solve real business problems. Because of Tcl's powerful event-driven I/O model, Tcl solutions tend to be small and fairly easy to reason about.

References

[1] Poindexter, Tom, Keith Vetter, and Don Libes. "SockSpy." <http://sourceforge.net/projects/sockspy/>

[2] Laird, Cameron. "Sockspy Knows TCP/IP" Sys Admin December 2002. <http://www.samag.com/documents/s=7732/sam0212b/0212b.htm>

[3] About Intermountain: Serving Our Communities. Intermountain Healthcare. <http://intermountainhealthcare.org/xp/public/about-intermountain/>

[4] Intermountain Healthcare. "Report to the Community 2006." <http://intermountainhealthcare.org/xp/public/documents/corp/annualreport.pdf>

[5] Kozek, Andrea. "GE Healthcare & Intermountain Health Care To Provide Wide-Reaching IT System." 17 February 2005. <http://www.gehealthcare.com/company/pressroom/releases/pr_release_10225.html>

[6] Cowley, Daron. "GE Healthcare & IHC establish new research center to develop electronic health record technologies." 6 July 2005. <http://intermountainhealthcare.org/xp/public/about-intermountain/news/article26.xml>

[7] Intermountain Communications. "Intermountain Healthcare updates logo." 29 November 2005. <http://intermountainhealthcare.org/xp/public/about-intermountain/news/article6.xml>

[8] Intermountain Communications. "IHC Health Plans has a new name--SelectHealth." 3 April 2006 <http://intermountainhealthcare.org/xp/public/about-intermountain/news/article10.xml>

[9] "Expressroom Lives On..." CMS Watch. 21 July 2003. <http://www.cmswatch.com/Trends/224-Expressroom-Lives-On...>

[10] "Module mod_rewrite URL Rewriting Engine." The Apache Software Foundation. <http://httpd.apache.org/docs/1.3/mod/mod_rewrite.html>

[11] Davidson, Jim. "Tcl in AOL Digital City: The Architecture of a Multithreaded High-Performance Web Site." 16 February 2000. <http://www.aolserver.com/docs/intro/tcl2k/html/>

[12] AOLserver Tcl API Reference. <http://www.aolserver.com/docs/devel/tcl/api/>

[13] Smith, Ryan. "Intermountain Healthcare: Audience-Focused Dynamic Portals." 24 October 2006. <http://www.vignettevillage.com/Austin/ConferenceSessionDetails.html>

[14] "Apache module mod_proxy." The Apache Software Foundation. <http://httpd.apache.org/docs/1.3/mod/mod_proxy.html>

[15] Fogie, Seth, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D. Petkov. Cross Site Scripting Attacks: XSS Exploits and Defenses. Syngress, 2007.

[16] Cleverly, Michael A. "The skeleton of a man in the middle." Weblog Entry. Cleverly Blogged. 12 March 2007. <http://blog.cleverly.com/permalinks/285.html>

[17] "Symbolic link." Wikipedia: The Free Encyclopedia. 17 August 2007. <http://en.wikipedia.org/wiki/Symbolic_Link>

[18] Lehenbauer, Karl et al. "TclX." <http://tclx.sourceforge.net/>

[19] Welch, Brent. "TclHttpd." <http://tclhttpd.sourceforge.net/>

[20] Landers, Steve. "Access C library functions using Critcl." Tcl'ers Wiki. 5 April 2004. <http://wiki.tcl.tk/11227>
[21] Cleverly, Michael A. "OTM: A web server with a *One Track Mind*." Weblog Entry. Cleverly Blogged. 26 June 2005. <http://blog.cleverly.com/permalinks/158.html>
[22] Newman, Matt et al. "TLS extension." <http://tls.sourceforge.net>
[23] "Using Web Server Plug-Ins with WebLogic Server." BEA WebLogic Server 8.1 Documentation. 2003. <http://e-docs.bea.com/wls/docs81/plugins/>
[24] Cocasse, Sharad and Makarand Kulkarni. "Understanding the WebSphere Application Server Web server plug-in." October 2003. <http://download.boulder.ibm.com/ibmdl/pub/software/dw/wes/pdf/WASWebserverplug-in.pdf>
[25] Staplin, George Peter and Don Porter. "Using gets with a socket is a BAD IDEA." Tcl'ers Wiki. 24 October 2001. <http://wiki.tcl.tk/1183>
[26] Cleverly, Michael A., Donal K. Fellows, ed. "TIP #287: Add commands for Determining Size of Buffered Data." Tcl Improvement Proposal. 26 October 2006. <http://www.tcl.tk/cgi-bin/tct/tip/287.html>
[27] "fblocked manual page." <http://www.tcl.tk/man/tcl8.4/TclCmd/fblocked.htm>
[28] Fielding, et al. "Hypertext Transfer Protocol--HTTP/1.1." RFC 2616. June 1999. <http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html#sec5>
[29] Cleverly, Michael A. "Re: TIP #287: Add a Commands [sic] for Determining Size of Buffered Data." comp.lang.tcl newsgroup. 14 December 2006. <http://groups.google.com/group/comp.lang.tcl/msg/e5e4d9cf8842a092>
[30] Thoyts, Pat and Xiaotao Wu. "TclUDP." <http://tcludp.sourceforge.net/>
[31] Gravereaux, David. "IOCPSOCK." <http://iocpsock.sourceforge.net/>
[32] Cassoff, Stuart. "Ceptcl." <http://www3.sympatico.ca/stuart.cassoff/software/>
[33] Sanfilippo, Salvatore. "Hping3." <http://wiki.hping.org/>

An ODE solver for Tcl: old Fortran in a new interface

Kevin B. Kenny
Computational Biology Laboratory, GE Global Research Center, Niskayuna, NY
kennykb@research.ge.com

Abstract

LSODAR is a Fortran subroutine for integration of ordinary differential equations that has been actively maintained for over twenty years and gives high-quality results for a variety of problems. This paper demonstrates how such a "dusty deck" can be adapted to interface cleanly to a modern high-level language like Tcl, be made safe to use in the presence of threads and recursion, and take advantage of the symbolic computation capabilities available to Tcl. The result is an ODE solver for Tcl that is production-ready, at the expenditure of considerably less effort than it would take to develop comparable capabilities targeted specifically at Tcl. It presents a few tricks that should be handy in connecting other legacy Fortran applications to Tcl.

1. Introduction

Fortran has for decades been the first choice of language for scientific programming. Because of this, a great many Fortran codes are available for solving numeric problems, such as statistical analysis, root finding, and the integration of differential equations. Many of the best are available free of charge, having been developed by US Government researchers at taxpayer expense. These "treasures of the national labs" deserve to be more widely known outside the Fortran community, but the tremendous difference in style between Fortran and more modern languages inhibits their broader acceptance. This paper describes how one such code, LSODAR (Livermore Solver for Ordinary Differential equations with Automatic method selection and Root-finding) (4) was adapted to use in Tcl.

Among the issues that have to be addressed for this code are adapting the long parameter lists of the Fortran calls to something more Tcl-friendly, managing dynamically allocated memory, dealing with Fortran callbacks (callbacks to EXTERNAL functions) at the Tcl level, and dealing with issues like recursion and multithreading.

The combination is arguably greater than the sum of the parts, because Tcl's ability to perform symbolic calculations can be exploited in the combined code. The combined Fortran-Tcl system has the ability to use Tcl's symbolic capabilities to perform symbolic differentiation to exploit backward-differencing formulas.

2. Background

The motivation for this work was the observation that a number of the author's colleagues were performing numerical simulations of ordinary differential equations (ODE's) in the context of cell biology(1), pharmacokinetics(5), and other biological systems. The systems tended to divide into two groups. The first group was accepted by the biologist users, but generally was specialized for a single narrow problem, and often had tremendous issues of numerical stability (because of the use of ad hoc solvers for the ODE's). The second took care with the numerical analysis, often by using well-tested third party codes, and often offered considerably more generality, but tended to be

Fortran programs usable only by their authors. Clearly, some sort of middle ground was needed.

At the same time, a number of others were working on developing the interface between Tcl and Fortran. Having already realized Tcl's utility for ad hoc scientific calculations(8), Arjen Markus had developed a rudimentary interface for Fortran programs to evaluate Tcl scripts(7). Working together with Jean-Claude Wippler and Steve Landers, he also explored a "proof of concept" of an automatic interface generator whereby Fortran subprograms could be integrated into Tcl(6). The idea of integrating Tcl and Fortran was certainly in the air.

These two developments combined to motivate the development of a Tcl interface to a general-purpose Fortran ODE solver, LSODAR (Livermore Solver for Ordinary Differential equations [Automatic method selection with Rootfinding])(4). LSODAR was chosen because it is well qualified (it has been well maintained for twenty years!), and because it requires fairly little "care and feeding" in the area of solution method selection (Many other libraries require the user to recognize the realms in which their ODE's are stiff and non-stiff and choose methods accordingly, risking horrible numerical instabilities if the wrong methods are chosen.) Moreover, LSODAR offers a root-finding capability, which was essential to stop the simulation at bifurcation points in the mixed strategy (ODE's plus discrete event simulation) that some of the author's colleagues were pursuing.

3. The Tcl interface to LSODAR

The first approach to connecting LSODAR with Tcl was to do a fairly straightforward translation of the application programming interface. It became obvious fairly quickly, however, that this approach was not going to succeed – or at the very least, was not going to produce a calling sequence that was anywhere near to idiomatic Tcl. If we examine Figure 1, it becomes quickly obvious why this is.

```
      SUBROUTINE DLSODAR (F, NEQ, Y, T, TOUT, ITOL, RTOL, ATOL, ITASK,
     1             ISTATE, IOPT, RWORK, LRW, IWORK, LIW, JAC, JT,
     2             G, NG, JROOT)
      EXTERNAL F, JAC, G
      INTEGER NEQ, ITOL, ITASK, ISTATE, IOPT, LRW, IWORK, LIW, JT,
     1   NG, JROOT
      DOUBLE PRECISION Y, T, TOUT, RTOL, ATOL, RWORK
      DIMENSION NEQ(*), Y(*), RTOL(*), ATOL(*), RWORK(LRW), IWORK(LIW),
     1   JROOT(NG)
```

Figure 1. Calling sequence for LSODAR.

First, there is a long list of parameters – twenty or so. It is difficult to imagine any Tcl command accepting that many positional parameters. Moreover, a number of the parameters (NEQ, LRW, LIW, and NG) simply serve to give the lengths of arrays. In addition, only the arrays ATOL, RTOL,Y and JROOT are actually used by the caller,

either before or after the call[1]. RWORK, IWORK and G are simply working storage provided by the caller because Fortran 77 has no dynamic memory allocation.

A more awkward problem is that three of the parameters are EXTERNAL – which, in Fortran, means that they are pointers to Fortran subprograms that will be invoked by the called routine. These callbacks need to be adapted so that they can be delivered into Tcl. Clearly, at least a small development effort was needed to make a module that would present a Tcl-friendly interface and support calls back and forth between Tcl and Fortran.

A. Choice of compiler

Since Tcl's object level API's are all in C, and the interfaces between C and Fortran are not terribly portable among compilers, the decision was made to have all the code in C. For this reason, the first thing done was to take all the Fortran code needed for LSODAR and pass it through 'f2c' – a Fortran compiler that has C code as output(3). This C code, plus the C source code for four builtin functions from the associated runtime library, provided a code base that was independent of the Fortran compiler on any target system; it is simply included in the source code of the distributed module. It was tested on one Linux system by comparing the output of the LSODAR example programs compiled with 'f2c' and the output from the Fortran versions compiled with GNU 'g77'(2); output was identical.

B. Designing a Tcl interface

Rather than having a large number of positional parameters, the Tcl interface uses keyword-value syntax, with sensibly chosen defaults. Figure 2 shows a trivial example, where a simple harmonic oscillator is modeled for an eighth-cycle of its operation.

```
set solver [odesolv::odesolv -atol 1e-6 -rtol 1e-6 -indvar theta \
             -- s {$c} c {-$s}]
set s 0.0
set c 1.0
set theta 0.0
set troubles [$solver run [expr {atan(1.0)}]]
puts [list $theta $c $s]
rename $solver {}
```

Figure 2. Trivial example of use of the solver

The program in Figure 2 is read: "Create an ODE solver instance, and store it in variable 'solver'. The independent variable for the system of ODE's is ϑ, and the results are expected to be computed to either six decimal places (an absolute tolerance of 10^{-6}) or to six significant figures (a relative tolerance of 10^{-6}). The system to be solved is:

$$\begin{aligned} \frac{ds}{dt} &= c \\ \frac{dc}{dt} &= -s \end{aligned} .$$

Initial values are $\vartheta = 0, s = 0, c = 1$, and results are requested for $\vartheta = \pi / 4$."

[1] Not quite true – the IWORK and RWORK arrays also contain further optional parameters for the call. We are dealing with a truly Byzantine interface here!

Unsurprisingly, when run, the program prints that at the end, s and c are both within 10^{-6} of $\sqrt{2}$.

Note that the right-hand sides of the system are Tcl expressions: they are actually evaluated by making Tcl_ExprDoubleObj callbacks back into the interpreter where the solver instance was created.

C. Connecting Tcl and Fortran

The solver constructor command is fairly straightforward; it simply builds a C data structure that contains all the parameters (plus sensible defaults for any named parameters that have been omitted). It then constructs a Tcl command that has this data structure as its ClientData, and returns the name of the constructed command. The client data structure also includes pointers to the arrays that the Fortran code uses.

The instance command supports three subcommands, 'run', 'continue' and 'status.' The 'run' command starts an integration; the 'continue' command continues an integration already in progress. The 'status' command simply returns a dictionary containing a group of the internal variables that the integrator uses, reporting on things like the integration method in use, the number of steps taken so far, the size of the last successful step and the next step to be attempted.

The 'run' and 'continue' commands invoke the Fortran code in fairly straightforward fashion. The only thing that is at all unusual is that the callbacks F (which computes the vector of derivatives, JAC (which computes the Jacobian matrix of F) and G (which computes the algebraic equations whose roots are to be found) are sent back to C code within the interface module for further processing. The callback for F, for instance, is shown in Figure 3.

```
static void
SolverInstEvalDeriv(
    integer* neq,          /* Number of equations, but also
                            * instance pointer */
    doublereal* t,          /* Value of the independent variable */
    doublereal* y,         /* Values of the model variables */
    doublereal* ydot)             /* Values of the derivatives */
{
    Solver* solverPtr = (Solver*) neq;
    Tcl_Interp* interp = solverPtr->interp;
    int systemc;
    Tcl_Obj** systemv;
    int i;
    int status = TCL_OK;
    status = CopyOutModelVariables(solverPtr, t, y, &systemc, &systemv);

    /* Evaluate the derivatives */
    for (i = 0; (status == TCL_OK) && (i < *neq); ++i) {
       status = Tcl_ExprDoubleObj(interp, systemv[2 *i + 1], ydot + i);
    }
    if (status != TCL_OK) {
       longjmp(solverPtr->errorExit, 1);
    }
}
```

Figure 3. Callback that evaluates the derivatives in the solver

In the figure, we see first a horrible kludge. The parameter, ‘neq’, which gives the number of ODE’s in the system, is also the first member of the client data structure. Casting it back to the client data lets the evaluator find all the other information it needs. Believe it or not, this hack is actually documented in the manual for LSODAR.

The ‘CopyOutModelVariables’ call copies the Fortran state vector to Tcl variables in the current scope. Following this, each of the ‘neq’ derivatives is evaluated in Tcl. If an error occurs, the solver is aborted (by the expedient of longjmp’ing around the Fortran code, since LSODAR does not provide for an error exit); otherwise, we return to the Fortran to continue solving the ODE’s.

4. Troubles that needed fixing

This level of interface served for quite some time, until its user base expanded a little. The first real trouble came from unexpected recursion. A user had created a Tk GUI that allowed running multiple systems at once. For some reason, one of the derivatives was evaluated by an expression that contained an invocation of a Tcl procedure, which in turn contained an [update]. The event loop was invoked, and dispatched the solver for another system of equations, and the code crashed.

The problem was that the code generated by ‘f2c’ contained static data in several labeled COMMON blocks. This static storage had to be made local to a single instance of the solver. This proved to be surprisingly easy.

First, every file that designated a given labeled COMMON block has a ‘struct’ declaration simultaneously laying it out and defining it, as shown in Figure 4. A #define giving the name to be used to refer to the block follows immediately.

```
struct {
    doublereal rowns[209], ccmax, el0, h__, hmin, hmxi, hu, rc, tn, uround;
    integer init, mxstep, mxhnil, nhnil, nslast, nyh, iowns[6], icf, ierpj,
           iersl, jcur, jstart, kflag, l, lyh, lewt, lacor, lsavf, lwm, liwm,
            meth, miter, maxord, maxcor, msbp, mxncf, n, nq, nst, nfe, nje,
           nqu;
} dls001_;

#define dls001_1 dls001_
```

Figure 4. Machinery of ‘f2c’ COMMON blocks.

Each subprogram that appears in the file begins with a left brace ‘{‘ on a line by itself. This is the only place where ‘f2c’ generates such a line.

A simple Tcl script is able to rewrite the declaration to the code shown in Figure 5.

```
typedef struct {
    doublereal rowns[209], ccmax, el0, h__, hmin, hmxi, hu, rc, tn, uround;
    integer init, mxstep, mxhnil, nhnil, nslast, nyh, iowns[6], icf, ierpj,
           iersl, jcur, jstart, kflag, l, lyh, lewt, lacor, lsavf, lwm, liwm,
            meth, miter, maxord, maxcor, msbp, mxncf, n, nq, nst, nfe, nje,
           nqu;
} dls001_;
extern Tcl_ThreadDataKey Odesolvdls001_Key;
```

Figure 5. Rewritten COMMON block interface.

It can then add a codeburst to the head of every procedure to get the current locations of the blocks, as shown in Figure 6.

```
    dls001_ *dls001_Ptr = (dls001_ *)
        OdesolvGetCommon("dls001_", &Odesolvdls001_Key, sizeof(dls001_));
    dlsa01_ *dlsa01_Ptr = (dlsa01_ *)
        OdesolvGetCommon("dlsa01_", &Odesolvdlsa01_Key, sizeof(dlsa01_));
    dlsr01_ *dlsr01_Ptr = (dlsr01_ *)
        OdesolvGetCommon("dlsr01_", &Odesolvdlsr01_Key, sizeof(dlsr01_));
```

Figure 6. Retrieving COMMON block addresses at the head of procedures

(OdesolvGetCommon is a thin layer around Tcl_GetThreadData that calls Tcl_Panic if the thread data have not been initialized.)

When a solver instance is started via the ‘run’ or ‘continue’ subcommands, it saves any existing thread data, sets the thread data to point to instance-specific copies of the COMMON blocks, and enters the Fortran code. In this way, the usage of COMMON blocks is made safe for multiple instances of the solver, multiple interpreters, and even multiple threads. Since this change was made, no further mystery crashes have been observed.

5. Further convenience

One optional parameter to LSODAR is a subroutine that evaluates the Jacobian of the derivatives (that is, the matrix of second derivatives). Supplying this subroutine can accelerate convergence, and in the case of very stiff systems of equations, may be necessary for numerical stability.

Even in Tcl, writing code to evaluate the Jacobian may be onerous. Since most of the uses of the package so far have involved fairly simple expressions for the derivatives (in the sense that they do not require command substitutions or arrays, but make do with Tcl’s built-in functions and scalar variables), it looked feasible to address the problem with symbolic differentiation.

The ‘odesolv’ package looks for a ‘math::symdiff’ package. If it finds it, and the user has not supplied a Jacobian matrix, it invokes ‘math::symdiff::jacobian’ to try to develop one symbolically. If it succeeds, it invokes the Fortran code with the symbolic Jacobian; if it fails, it invokes the Fortran code with no Jacobian supplied and hopes for the best.

Figure 7 shows an example of computing a symbolic Jacobian for a system of equations.

```
% foreach row [math::symdiff::jacobian {
    y1 {-0.04 * $y1 + 10000. * $y2 * $y3}
    y2 {0.04 * $y1 - 10000. * $y2 * $y3 - 3.0e-7 * $y2 * $y2}
    y3 {3.0e-7 * $y2 * $y2}
}] {puts $row}
-0.04 {(10000.0 * $y3)} {(10000. * $y2)}
0.04 {-(((10000.0 * $y3) + ((3.0e-7 * $y2) + (3e-007 * $y2))))} {-((10000. *
$y2))}
0.0 {((3.0e-7 * $y2) + (3e-007 * $y2))} 0.0
```

Figure 7. Calculating derivatives symbolically

The details of the 'symdiff' package are described in a companion paper.

6. Results

It is customary in an experience study like this one to present comparison results with other techniques for achieving the same goals. In this case, it is difficult, because it is hard to find other attempts at a production-quality ODE solver in Tcl.[2] Certainly, any attempt to compare the 'odesolv' package with the ODE solver in Tcllib is comparing apples with oranges: 'odesolv' offers automatic switching between Adams-Moulton and Gear backward differencing, while Tcllib does only fourth-order Runge-Kutta; 'odesolv' has adaptive stepsize control and error estimation, while the Tcllib method lacks these features; and so on. Nevertheless, it is possible at least to set up benchmark problems with the two systems.

The first problem, a nonstiff system, was the simple harmonic oscillator of Figure 2. Integration was carried out starting at $\vartheta = 0$ and continuing to $\vartheta = 13\pi/6$, a little more than one cycle. The second problem was the extremely stiff set of equations $\frac{du}{dt} = 998u + 1998v \quad \frac{dv}{dt} = -999u - 1999v$, carried out for $0 \le t \le 16$.
For the stiff solution, LSODAR chose the Adams-Moulton solver for the initial nonstiff settling time until t = 0.017, and then switched to Gear's backward difference formula. Table 1 shows the performance comparison.

Table 1 . Performance comparisons of Tcllib's integrator and 'odesolv'.

Problem	Nonstiff		Stiff	
Metric	Tcllib	Odesolv	Tcllib	Odesolv
Number of evaluations of derivatives	256	194	32768	472
Number of evaluations of Jacobian	N/A	5	N/A	21
Run time (ms)	4.13	0.55	284.00	2.03

The results show that to achieve the same accuracy of results 'odesolv' took roughly the same number of iterations in the nonstiff case, but arrived at the answer nearly an order of magnitude faster than tcllib's integrator. Tcllib's integrator proved unsuitable for the

[2] It may be possible to achieve this functionality using Tcl with SCILAB; I have not yet succeeded at this, but admittedly haven't tried very hard.

stiff problem. For fewer than 16384 iterations of the Runge-Kutta step, its numeric stability was so catastrophic that it crashed with a floating-point overflow rather than returning results. By contrast, 'odesolv' was, as expected, quite well behaved, switching to a backward difference formula and needing two orders of magnitude less time (and comparably fewer evaluations of the derivatives).

7. Conclusions

This work demonstrates a comprehensive reworking of a legacy Fortran interface into idiomatic Tcl. It shows how integrating high-quality scientific codes into Tcl can enhance the capabilities of both. It presents a few interesting techniques for integrating Fortran and Tcl; in particular, anchoring client data to the address of a Fortran parameter, replacing COMMON blocks with thread-specific data are both new techniques (as far as the author is aware). Finally, it demonstrates that combining Tcl's ability for symbolic computation and for reconfiguration yields a whole that is greater than the sum of the parts.

References

1) Beer, A.J., Haubner, R., Goebel, M., Luderschmidt, S., Spilker, M.E., Wester, H.J., Weber, W.A., Schwaiger, M. "Biodistribution and pharmacokinetics of the αVβ3-selective tracer ^{18}F-galacto-RGD in cancer patients." *J Nucl Med.* 46:8 (August, 2005), pp.1333-41.
2) Burley, James Craig. *Using and porting GNU Fortran.* Version 3.4.6. Cambridge, Mass: Free Software Foundation, 2006.
3) Feldman, S.I., Gay, David M., Maimone, Mark W., Schryer, N. L. "A Fortran-to-C converter." Computing Science Technical Report 149, Murray Hill, N.J.: AT&T Bell Laboratories, March, 1995. Reprint available at http://www.netlib.org/f2c/f2c.pdf.
4) Hindmarsh, Alan C. "ODEPACK: A systematized collection of ODE solvers." In *Scientific Computing* (R.S. Stepelman, *et al., eds.*) (Volume 1 of IMACS Transactions on Scientific Computation) Amsterdam: North-Holland, 1983, pp. 55-64. Reprint available at http://www.llnl.gov/CASC/nsde/pubs/u88007.pdf.
5) Kiehl, Thomas R., Mattheyses, Robert M., Simmons, Melvin K. "Hybrid simulation of cellular behavior." *Bioinformatics* 20:3 (March 2004), pp. 316-322.
6) Landers, Steve, and Wippler, Jean-Claude. "CriTcl - beyond Stubs and compilers." Proc. Ninth Intl. Tcl/Tk Conference, Vancouver, B.C. (September 2002). Reprint available at http://aspn.activestate.com/ASPN/Tcl/TclConferencePapers2002/Tcl2002papers/wippler-critcl/critcl.pdf.
7) Markus, Arjen. "Combining Fortran and scripting languages." ACM SIGPLAN Fortran Forum 21:3 (2002), pp. 10-18.
8) Markus, Arjen. "Doing mathematics with Tcl." Proc. Third European Tcl/Tk User Meeting. Munich (June, 2002). Reprint available at http://www.t-ide.com/tcl2002e/mathematics.pdf.

14’th Annual Tcl/Tk conference
Sept. 24-28 2007
Bourbon Orleans Hotel, New Orleans, LA USA

Tcl Applications

Anatomy of a Large Application: Architectural Patterns and Solutions

William H. Duquette
Jet Propulsion Laboratory, California Institute of Technology
William.H.Duquette@jpl.nasa.gov

ABSTRACT

JNEM, the Joint Non-kinetic Effects Model, is a large simulation application. Written almost entirely in Tcl/Tk, it makes architectural use of the Snit object system and the SQLite3 database engine. This paper addresses a number of architectural patterns and solutions that have been found useful during the two-plus years of JNEM development. Patterns include the three-layer package architecture (application, domain, and utility); Snit types as application modules; saving and restoring application state; the database-backed objects; SQLite3 as an application memory debugger; and a generalization of the scrollbar/scrollable pattern.

1. Joint Non-kinetic Effects Model

The Joint Non-kinetic Effects Model (JNEM) is a military training simulation that participates in a federation of simulations used to train military commanders. The federation is called the Joint Land Component Constructive Training Capability (JLCCTC) Multi-Resolution Federation (MRF). JNEM's role as a federate in the federation is to model the responses of the civilian population to force activities, up to and including actual combat, thus adding non-kinetic effects to the kinetic effects modeled by the battlefield simulation. JNEM is written primarily in Tcl/Tk 8.4, with a small amount of code in C/C++.

This paper describes certain architectural patterns and solutions used in the implementation of JNEM; it does not address specifics of the JNEM implementation, simulation model, or data formats. The techniques described here can be adapted to any large application/system with similar requirements.

2. The Three-Layer Package Architecture

Like many large application development efforts, JNEM is not really a single application; rather it is a system containing many applications of various sizes, of which the JNEM simulation proper is the largest. The system also includes control GUIs (the JNEM Console), simulators designed to represent other federates during development, and a number of ancillary tools of varying degrees of complexity, both GUI and non-GUI. As a result, there is opportunity for significant code sharing between the various components of the system.

The cleanest mechanism for sharing code between applications is the use of well-written reusable code libraries. Reusability, however, is context-dependent. Every coding effort is based on assumptions as to the context in which the resulting software will be used. This context then determines the arena in which the software is reusable.

The wider the context in which the code is to be reused, the more general and flexible (and hence the more complicated and costly) the code needs to be. The narrower the context, the more assumptions the author of the code may make and the more specific (and hence simpler and less costly) the code may be.

It is important to target code to the proper context. If a library module is written for too broad a context, it will be more complex than necessary, and thus more costly to implement and maintain. The increased complexity is often reflected in the module's interface; this in turn increases the complexity of all clients of the module, thus increasing the cost of using the module.

If the library module is written for too narrow a context, then it may not be possible to take advantage of unforeseen opportunities for reuse that will arise. When this happens, either the library module needs to be generalized—if there is time—or the coder is likely to be forced to engage in "bad reuse": copying the library code into a new module with small changes.

Deciding the appropriate context for a new module of code is to a certain extent an art, and depends on a clear vision of what the project's needs will be in the future. However, we have found that the following three-layer package architecture greatly simplifies the decision:

Application Layer
Domain Layer
Utility Layer

Each layer represents a particular context and has its own particular constraints; generality increases from top to bottom. The layers are defined in detail below; once they have been defined, we will discuss how code is structured within each layer.

2.1 The Utility Layer

The utility layer represents the widest context in the system. Code written for this layer should make few assumptions about the environment in which it runs, and, in particular, should avoid placing requirements on the application. For example, JNEM's utility layer includes a pair of comm(n)-based communications modules, commclient(n) and commserver(n). Among their duties is the logging of client connects, disconnects, and other message traffic. However, they write to a log file only when configured to do so; otherwise they would be suitable for large, long-running applications but not for short *ad hoc* utility scripts.

These considerations place the following constraints on modules in the utility layer:

Modules should not touch the global namespace. All utility layer code should be defined in package namespaces, with public names exported.

Modules without state should usually export a single ensemble command. This was determined to be a beneficial pattern by Roseman [3], and minimizes the risk of name collisions if names are imported into the global namespace. There are exceptions; frequently-used utility commands can be implemented as normal commands. But generally speaking, a family of related commands should be implemented as an ensemble. For example, JNEM's family of matrix operations are implemented as a single ensemble, mat(n).

Modules with state should be implemented as object instances. JNEM's logging facility, for example, is defined as a logger(n) object type, thus allowing an application to create multiple logger(n) objects and log to multiple files simultaneously.

Object instances must be wired together explicitly. If a commserver(n) object is to log communications traffic, for example, it must be given the name of a logger(n) object. It cannot simply assume that there is a logger(n) with a well-known name.

JNEM's utility layer includes the following kinds of modules:

- String and list handling
- Date and time
- Interprocess communication
- Math and geometry
- Control structures
- Data types
- Logging
- General purpose megawidgets

A module that contains no domain-specific knowledge should be included in this layer on one of two conditions: if it is definitely needed by two or more applications within the system, or if it has potential for reuse and little extra work is required to make it a library module. The logger(n) module is an example of the first case; string and list handling routines are examples of the second.

JNEM is not a cross-platform application; it specifically targets Linux/X-Windows. If it were to be made cross-platform, platform details would be handled in this layer as well.

2.2 The Domain Layer

The domain layer contains code intended for reuse in multiple applications in our particular application domain. For example, JNEM defines a module called sqldatabase(n), a Snit-based wrapper for SQLite3 database objects. Among other features, the wrapper defines the JNEM database schemas. This information is clearly specific to our application domain, and hence does not belong in the utility layer; on the other hand, we wish to be able to write a variety of applications, from the JNEM simulation proper down to simple scripts, that access JNEM database files. Consequently this code belongs in the domain layer, along with domain-specific data types, file format definitions and parsers, multi-application simulation modules, and domain-specific multi-application GUI components. As with the utility layer, code written for this layer should avoid placing requirements on the application.

Domain layer code generally operates under the same constraints as utility layer code; however, it will include domain-specific knowledge and will be more likely to implement specific policies. JNEM's utility layer, for example, includes a module called sqlib(n), which contains routines for querying SQLite3 schemas and for formatting the results of SQLite3 queries. This module makes no assumptions about the SQLite3 database upon which it operates, and defines no policies for how the database is used. The domain-layer module sqldatabase(n), on the other hand, not only defines the database schema but also implements a document-like open/save transaction policy.

A module containing domain-specific knowledge should be defined in this layer on either of two conditions: if it is definitely needed by two or more applications within the system, or if it has potential for reuse and little extra work is required to make it a library module.

It sometimes happens that a module requiring domain-specific infrastructure would also be useful in programs based solely on the utility layer. In this case, one needs to consider generalizing the module and its interface so that the module itself contains no dependencies on the domain layer, but can be configured to make use of domain-specific modules when available.

2.3 The Application Layer

The application layer contains code that resides in a single application. Properly speaking, then, there is no system-wide application layer; each application within the system has its own. Code written for this layer may access any desired modules from the two lower layers, and may embody any assumptions that simplify the implementation. The application may use the global namespace freely, and many objects in the application layer will have well-known names. For example, the application may define a global logger(n) object called `::log`, and all application modules may use it freely by that name. No "wiring together" of objects is required.

We have found the following guidelines to be helpful.

Define public names in the global namespace. We are used to not polluting the global namespace in our code, so as to avoid name collisions. But if the application itself cannot use the global namespace, who can?

Define private names in namespaces. A module's private internal commands should be defined in a module-specific namespace, so as to avoid inadvertant inter-module name collisions.

Prefer singletons to object types with multiple instances. Application modules will frequently contain state. Unless multiple instances of that state are definitely required, however, modules should be implemented as singleton objects with well-known names. This simplifies the code, and makes it easier to call modules from other modules without explicit wiring.

Implement singletons as ensemble commands. This gives each singleton module a single well-known name, and allows it to be treated as an object by other modules. The benefits of using ensemble commands for module public interfaces are discussed in [3].

2.4 Internal Architecture

Since the utility and domain layers consist of library modules, it will come as no surprise that each is implemented as a collection of packages. In principle, each module in these layers could be implemented as a separate package. For simplicity, however, JNEM groups the modules into a small number of packages, as shown below, although each module has its own man page:

	GUI	Non-GUI
Application Layer	jnem_*app*(n)	
Domain Layer	simgui(n)	simlib(n)
Utility Layer	gui(n)	util(n)

In use, the primary distinction is between GUI and non-GUI code, as requiring the Tk package in a non-GUI application has undesirable effects. Consequently, the utility and domain layers are each defined as a pair of library packages, one GUI and one non-GUI, each of which contains a number of modules.

Two architectures are used at the application layer: small applications, such as command-line utilities, are generally implemented as simple scripts which load the packages they need from the domain and utility layers. Larger applications, such as the JNEM simulation proper, are implemented as a short loader script which loads an application package, jnem_*app*(n). This architecture allows the application to contain arbitrarily many modules in an easily managed way. In JNEM, all such applications share a single loader script, jnem(1). Thus, the command

```
$ jnem sim
```

causes the jnem(1) script to load the jnem_sim(n) package and invoke its main entry point. By convention, every jnem_*app*(n) package contains at least one module, `app.tcl`, which defines a singleton object called `::app`. The application's main entry point is then `::app init`.

3. Snit Types as Application Modules

As stated above, an application module should present its public interface as an ensemble command defined in the global namespace, while the module's internal code should reside in a module-specific namespace. A `snit::type` definition serves this purpose admirably well:

- The type's name, defined in the global namespace, is the ensemble command.
- The type's type methods are the ensemble command's subcommands.
- All of the type's code and variables naturally reside in the type's namespace.

In addition, ensemble subcommands can be delegated to component objects or to other application modules, and ensemble subcommands can themselves be ensembles with subcommands of their own.

The one necessity is to ensure that the `snit::type` cannot create instances; otherwise, a mistyped subcommand will be treated as the name of an instance to be created, with mystifying results. The skeleton for such a module looks like this:

```
snit::type mymodule {
    pragma -hasinstances   no

    typevariable myvariable

    typemethod mysubcommand {args} {
        ...
    }
    ...
}
```

The `::mymodule` command is global, but all of the module's code then resides in the `::mymodule::` namespace.

The type's standard "info" and "destroy" methods can also be disabled, allowing them to be redefined by the module if the author so desires.

3.1 Library Ensembles

Ensemble commands implemented in the utility or domain layers can also be implemented in this way, placing the command in the package namespace. JNEM's matrix manipulation module, mat(n), a module of package util(n), is implemented something like this:

```
namespace export ::util::mat

snit::type ::util::mat {
    pragma -hasinstances   no

    typeconstructor { namespace import ::util::* }

    typemethod add {mat1 mat2} { ... }
    ...
}
```

The ensemble command is defined in the `::util` namespace, as are util(n) commands. Note that the ensemble's code resides in the `::util::mat` namespace, and consequently cannot see other commands defined in `::util`; hence, the ensemble must import them as shown. This is a nuisance, as it constrains the order in which util(n)'s modules are loaded. If JNEM were using Tcl 8.5 and Snit 2.1, this would not be necessary; in Snit 2.1, Snit types automatically add their parent namespace to their namespace path.

4. Saving and Restoring Simulation State

Training exercises can run twenty-four hours a day for five days or more. If a simulation in the training federation should crash, it is vital not only to get it running again as soon as possible, but also to get it running again with the same state it had prior to the crash (possibly adjusted slightly so as to avoid crashing again). Now, the state of a simulation is a complex thing, and the only way to recreate it from scratch is to re-run it with the same inputs and random draws. After more than a few hours, though, the time involved in "running up" the simulation from scratch is prohibitive. Consequently, every simulation in the federation is periodically asked to save its state so that the federation state can be restored later. Such a saved state is called a *checkpoint*, and saving simulation state is referred to as *checkpointing the simulation*.

In one sense, this is no different than the requirement on any document-centric application—a word processor, say, or a

spreadsheet. There are, however, two distinctions of note. First, the state data for a complex simulation can be of great variety and extent. In a word processor, each document is likely to be represented as an object of some type, probably managing a hierarchy of lower-level objects; and saving the document to disk is a natural operation on the document object. In a complex simulation, there are likely to be many, many objects and ancillary data to be saved, and there might not be any obvious organizing principle corresponding to the notion of a "document". Second, the requirement to load and save state is implicit in the notion of a document-centric application; it is not similarly implicit in the notion of a simulation. History shows that checkpointing is an architectural issue, and that it can be very difficult to implement if it isn't taken into account from the beginning.

Code written to save and restore complex application state to and from disk can be extremely fragile. First, all relevant state must be identified; if any state variables are omitted, it will not be possible to restore the simulation's state precisely. We will refer to this set of checkpointed state variables as the application's *persistent state*. Second, the routines which read the checkpoint must exactly mirror the routines that write it out, or efforts to restore will fail. An error in either one renders the checkpoint useless; and naturally, it's a common error (especially in applications for which checkpointing is an afterthought) to update the reader and forget to update the writer, or *vice versa*. The solution is to provide a framework for saving and restoring arbitrary state variables, and then register all state variables with the framework.

In most languages, it would also be necessary to register the type of each state variable. In Tcl, where everything is a string, defining such a framework is nearly trivial. JNEM follows a few simple patterns which make saving and restoring simulation state both easy to implement and robust in the face of change.

JNEM's internal state is of four kinds:

- Persistent state stored in the *run-time database* (RDB)
- Persistent state stored in Tcl variables which are mirrored in the RDB.
- Persistent state stored in Tcl variables which are not mirrored in the RDB.
- Non-persistent state stored in Tcl variables.

Only the first three kinds need to be included in a simulation checkpoint; however, the latter three kinds need to be clearly commented and distinguished in the code.

4.1 Saving the Run-time Database

The run-time database, or RDB, is an SQLite3 database. Much of JNEM's simulation data is stored there, particularly its knowledge of the various entities in the simulated world. Checkpointing the data stored in the RDB is, of course, trivial—JNEM creates a checkpoint simply by committing all current updates and making a copy of the database file. The data is restored by copying and opening the checkpoint file as a new RDB.

The checkpoint file is thus an SQLite3 database file. If all persistent state were stored in the RDB, no further work would need to be done. Inevitably, for performance or ease of implementation, some data will be stored only in memory. Some state data, notably object instance data, is stored in memory and automatically mirrored on change in the RDB; this makes it easy to perform queries on objects and also to produce reports. Such data is automatically checkpointed. Other state data in Tcl variables gets copied to and from the database on checkpoint and restore, as discussed in the following section.

4.2 Saving In-Memory Application State

The simulation consists of a number of modules, each of which has its own set of Tcl variables, some of which contain persistent data and some of which do not. Each module that has Tcl variables containing non-mirrored persistent state is registered with the checkpoint management module. This registry is simply a hard-coded list of module names; it is updated by hand as modules are added and deleted.

Each such checkpointable module is required to have the two following subcommands:

module `checkpoint`
: Returns the module's non-mirrored persistent state as a single Tcl value. This is usually a dictionary, and frequently a dictionary of dictionaries.

module `recover` *state*
: Restores the module's state to *state*, which must be a value returned by the module's `checkpoint` subcommand.

When a checkpoint is to be saved, the checkpoint manager simply asks each checkpointable module for its state, and stores it under the module's name in an RDB table called `checkpoint`. which has two string-valued columns, `component` and `data`. When saving a checkpoint, the application retrieves the state for each checkpointable module and stashes the module's name and state data into the `checkpoint` table. When restoring from a checkpoint, the application retrieves the module names and state values from the restored RDB file and asks each module to recover itself.

The preceding discussion uses the term "module", but some of the checkpointed "modules" are actually instances of `snit::type`s. What they all have in common is that they all have well-known global names, are created at simulation start, and persist for the life of the simulation. (Transient objects, on the other hand, mirror their state to the RDB.) Thus, the code is extremely simple; the only maintenance that is ever needed at the application level is to add and delete names from the list of checkpointable "modules" as the simulation's implementation changes during the course of development.

4.3 Saving In-Memory Module State

The previous section explained how the application saves and restores the persistent state of its modules to and from the RDB using each module's `checkpoint` and `recover` subcommands. This section explains how each module structures its internal data, and how the `checkpoint` and `recover` subcommands are usually implemented.

First, each checkpointable object is either a singleton implemented as a `snit::type`, as described above, or an

instance of a `snit::type`. Either way, the object's variables are grouped and clearly labeled as to whether they are checkpointed or not. Second, almost all checkpointed data is stored in one or more Tcl arrays. Some data is array-oriented by nature; and the remaining scalar data is stored in an array simply to make checkpointing convenient. The `checkpoint` and `recover` subcommands can then be implemented like this:

```
typemethod checkpoint {} {
    list \
        array1 [array get array1] \
        array2 [array get array2]
}

typemethod restore {checkpoint} {
    foreach {name value} $checkpoint {
        array unset $name
        array set $name $value
    }
}
```

In a few cases an object might have component objects with persistent state; this complicates the code slightly. But in each case, the pattern is the same: the object's state is checkpointed as a dictionary of named values and recovered accordingly.

The beautiful thing about this mechanism is that it is only necessary to touch the checkpoint/recover code when a checkpointed array or component is added to or removed from an object. Since checkpointable scalar variables are grouped into an array, any number of new scalar variables can be trivially added and correctly checkpointed just by defining them in that (carefully labeled) array.

In short, simply by storing data in arrays and adopting a simple convention, we get trouble-free saves and restores of in-memory data with almost no maintenance overhead. It just works.

5. Database-Backed Objects

The word "object" being oversubscribed, we will adopt the following terminology for this section. An *object* is a standard Tcl object: a command with subcommands. A *singleton* is an object defined as a `snit::type` with type methods. An *instance* is an object defined as an instance of a `snit::type`. An *entity* is a simulated thing with associated data that might or might not have an associated *object*.

In JNEM, for example, a ground unit—a platoon, say—is an entity. Ground unit data is received from the federation, and each unit's data is stored as a row in the RDB's `units` table, thus allowing units to be queried in interesting ways. Since units have little behavior within JNEM itself, there is no Tcl object associated with each unit.

Fixed site entities, e.g., power plants and hospitals, do have significant behavior within JNEM, and hence have significant amounts of associated code and data. In JNEM v1, all entity data was stored in the RDB, and the related code for a particular entity type resided within one more singletons within the application, singletons whose primary task was something else. There were three advantages to this approach:

- Entity data could be easily queried.
- Entity data was trivially included in saved checkpoints.
- Entities by their nature are transient, and the housekeeping nuisance of saving and restoring transient instances was thereby avoided.

There were also disadvantages:

- Code related to an entity type was split between different modules, wherever the entities were used; there was no central place for it to live.
- All changes to entity data required explicit SQL updates to the RDB, thus making the code uglier and more verbose.

(A note on using an SQLite3 database for an application's record data: it is sometimes reasonable to write an API for updating records in the database, but it is rarely reasonable to write an API for querying records. Such an API is never as expressive as SQL SELECT statements, and it is *much* faster to do a SELECT and allow SQLite3 to iterate over the selected entity data than it is to do a `foreach` over entity IDs and then query the RDB for each entity's data.)

In JNEM v2, it soon became clear that the entity-related code was becoming unmanageably ugly, and that some entities could benefit from being represented as Tcl objects, i.e., as Snit instances. Now, saving and restoring transient instances is a nuisance: on a restore, one must destroy all existing transient instances, and then recreate the saved instances. There are two aspects to this problem: creating the restored objects with the correct names, so that they can be used by other code, and creating the restored objects with the correct data. Moreover, we wished to make this change *without* losing the benefits of storing entity data in the RDB. The result was the "databased-backed objects" pattern. The details of the pattern are as follows.

Each entity type is mapped to a particular RDB table; individual entities are represented as rows in the table. Each entity has a unique ID, which is the table's primary key. Entities are always addressed by their unique ID, not by any object name. Thus, the object names are arbitrary.

The first step is to make it possible to efficiently retrieve a Tcl object for a given entity, given only the entity's ID. At any given time, an entity might or might not have an associated Tcl object. Any routine that needs to access an entity as a Tcl object requests such an object, giving the entity's ID. The object is created, if it does not already exist, and is cached for later use. By convention, the routine requesting the object may only presume that the returned object name is valid until the routine itself returns. The routine may use the object name freely, and may pass it to other routines; however, only the entity ID should be saved in data structures. Entity object names are *not* persistent. If the state of the simulation changes, i.e., if state is restored from checkpoint, then the content of the RDB is presumed to be different. The cache of entity objects is cleared, and all such objects are destroyed. By this means, objects are created on demand and then retrieved as needed.

This part of the pattern is implemented by a singleton associated with the entity type; fixed site entities, for example, are managed

by the `site` singleton. The `site` singleton provides the following method (among others):

`site get` *`id`* `?-create?`
: Returns an object instance for the fixed site entity with the given *id*. Unless `-create` is specified, the entity must already exist in the RDB.

Thus, any routine that needs to access a fixed site's behavior will know the site's ID and will call `site get` to retrieve the instance. Any routine which merely wishes to query one or more fixed sites will query the RDB directly.

The entity objects are, perhaps surprisingly, not instances of the `site` type, but rather instances of the `siteType` type. As noted previously, defining a `snit::type` with both type and instance methods can lead to perplexing bugs if the type is called with a misspelled type method. (Note that `snit::widget`s are more or less immune to this problem, as a misspelled type method name is unlikely to look like a widget name.)

The second step is to tie an entity object's data to the entity's row in the RDB. When an entity object is created, it retrieves the entity's row from the RDB, and stores it in an instance variable, an array called `info`. This allows the object's methods to read this data without the cost of accessing the database. The entity object then provides at least the following methods:

`$entity set` *`name value`*
: Sets the value of entity field *name*, updating both the `info` array and the related RDB row.

`$entity get` *`name`*
: Retrieves the value of entity field *name* from the `info` array.

The rules that ensure consistency between the data stored in the RDB and the data mirrored in the entity objects are as follows:

- Only type methods of the entity singleton and methods of the entity instance type are allowed to use SQL queries to modify the contents of the entity type's table in the RDB.
- Other modules must use the entity singleton to affect entities as a group, and entity objects to update fields of individual entities. They may use the `set` method directly, or use other entity instance methods which call `set` indirectly.
- If the entity singleton updates existing entities, then the singleton's instance cache must be cleared. We will not try to update all existing entity instances immediately; instead, we will recreate them on demand.

This pattern preserves the advantages of saving entity data in the RDB while allowing entity code to be well-structured and easily maintained, at minimal cost in code complexity.

6. SQLite3 as a Memory Debugger

In conventional C or C++ programming, a perennial difficulty is finding out precisely what's going on in the program's memory, especially in environments where an external debugger is difficult or impossible to use. Tcl/Tk eases this problem by its very nature; with a small amount of work, any application can pop up a "console" window with a command line, so that the developer can query Tcl variables via Tcl commands. Even so, navigating complex data structures can still be tedious, especially when data is stored in Snit instance variables.

As noted above, JNEM uses its "Runtime Database", or RDB, as a structured memory store. The user is allowed to enter SQL "SELECT" queries for the RDB from the JNEM Console's command line interface; the query results are returned in tabular format. To the extent that the application's data is stored in the RDB, then, that data is easily browsed using arbitrary queries—which has the effect of turning SQLite3 into a memory debugger without peer. Any desired data or condition can be queried; and if desired, triggers can be used to trap particular conditions as they occur. It would be difficult to overstate the convenience of this feature.

7. The Scrollbar/Scrollable Pattern

The scrollbar/scrollable pattern is familiar to every Tk programmer; adding scrollbars to a text widget to create a simple text editor is a standard introductory example, but scrollbars can also scroll canvas widgets, frames, and listboxes. Scrollbar A and scrollable widget B have a symbiotic relationship such that if A's position is changed, B's viewport updates to match, and *vice versa*. At the lines of code level, A and B are wired together by a pair of callback/subcommand protocols:

- When A's position is changed, A's `-command` callback calls B's `yview` subcommand to update B's viewport.
- When B's viewport is changed, B's `-yscrollcommand` callback calls A's `set` subcommand.
- Both A and B are smart enough not to do anything if the subcommand tells them to do what they are already doing.
- A and B are kept in sync through all changes, whether triggered by the user or by the application.

The beautiful thing about this pair of callback/subcommand protocols is that it is defined by the arguments passed by the callback to the subcommand, not by either the callback or the subcommand name. This is the feature that allows the text widget to use two scrollbars at once, using the same protocols. Only the developer who glues A and B together needs to know the callback and subcommand names.

During JNEM v2 development, we realized that, properly abstracted, this pattern can apply to pairs of other kinds of widgets, as a "meta-pattern" if you will. For example, JNEM's log browser includes a search box called the "finder". Entering a search string in the box causes matching log entries to be highlighted in the body of the browser; further, the search box displays a count of the number of matches, and buttons to scroll the log browser from one match to another. If the content of the log browser is updated, then the search box must update its search.

Our initial implementations of the finder and log browser were rather ugly; we kept trying to make the log browser subordinate to the finder, and we kept having synchronization problems. The difficulty was that updates were received by both the finder and the log browser, and the log browser's updates were being mishandled, precisely because the finder was not involved. Pondering the Scrollbar/Scrollable pattern, we realized that neither the finder nor the log browser should be in charge; rather,

the two widgets must be peers, each keeping itself synchronized with the other. This could be called the Finder/Findable pattern; it has the same nature as the Scrollbar/Scrollable pattern. Given a finder A and a log browser B, A and B are wired together by a pair of callback/subcommand protocols:

- When the contents of A's search box is changed, whether by the user or by the application, A's `-findcmd` callback calls B's `find` subcommand, passing it the target string and search type (exact match, regular expression, etc.). B does the search and highlights the matches. Note that `find` is an ensemble subcommand with subcommands of its own.
- When B's search results change, B's `-foundcmd` callback calls A's `found` subcommand, passing it the number of matches found and the index of the match currently displayed. A updates its appearances, displaying the index and count, and enabling or disabling its arrow buttons appropriately.
- If matches were found, then A's `-findcmd` can call B's `find` command to navigate through the found matches.

JNEM has several "findable" widgets that will happily work together with the Finder widget.

The meta-pattern can be expressed as follows: given two objects, A and B, each of which must remain synchronized with the other in the face of updates to either one, glue the two objects together by means of a pair of callbacks and subcommands, such that A's callback calls B's subcommand, and vice-versa. Define the protocol in terms of the required arguments of each subcommand **only**, as it allows B to communicate with multiple A's at the same time, much as the text widget communicates with multiple scrollbars.

8. REFERENCES

[1] Duquette, William H., "Snit's Not Incr Tcl", http://www.wjduquette.com/snit.

[2] Tcl Manual, http://www.tcl.tk/man/tcl8.4/TclLib/Eval.htm.

[3] Roseman, Mark, "Ten Years of Rapid Development", 9th Tcl/Tk Conference, http://www.markroseman.com/pubs/tenyears.pdf.

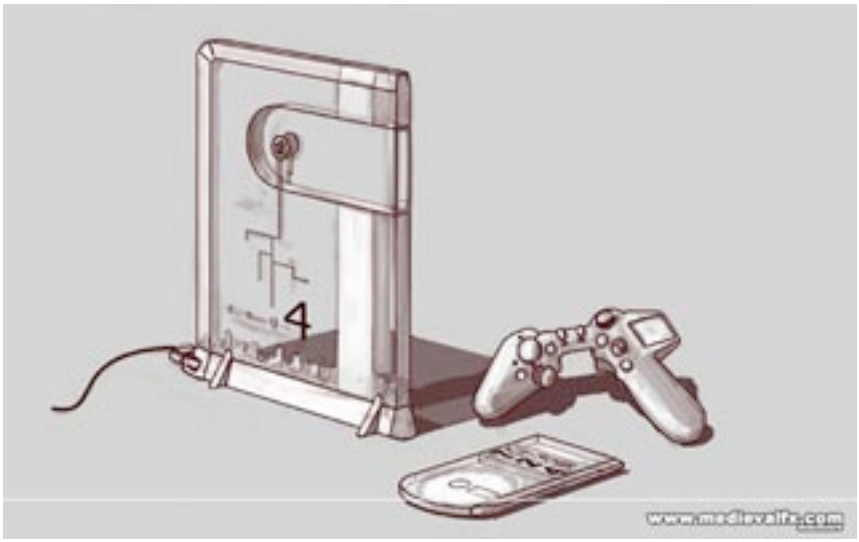
[4] Duquette, William H., "Type Definition Objects", 12th Tcl/Tk Conference.

9. ACKNOWLEDGEMENTS

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, during the development of the Joint Non-kinetic Effects Model (JNEM) for the U.S. Army Program Executive Office – Simulation, Training and Instrumentation (PEO STRI) in Orlando, Florida.

14’th Annual Tcl/Tk conference
Sept. 24-28 2007
Bourbon Orleans Hotel, New Orleans, LA USA

Tcl Does Data(bases)



TDIF

TCL Data Interface Framework

Presented to the 14th Annual Tcl Developer Conference by:

Sean Deely Woods

TABLE OF CONTENTS

TABLES AND FIGURES

INTRODUCTION

WHAT IS TDIF TRYING TO SOLVE?

TDIF is a uniform methodology for accessing external data within Tcl.

Its genesis stems from a need on my part to have the same software run under multiple platforms with different drivers to access MySQL. One distro of linux would ship with tclmysql. Another would ship with mysqltcl. Windows ships with tclodbc. And then there was the change in mysqltcl's interface between mysql_ and mysql::.

At the same time sqlite was coming into it's own, and I found myself replacing some MySql applications with it. I was also having to occasionally dump data out of MS Sql and MS Access. Being a lazy programmer, I developed a shorthand for all of the various database interactions I needed, and then wrote a suite of tools to convert that shorthand to the native interface of the storage engine.

TDIF is my attempt to adapt my own techniques, developed over time, into a formal interface.

TDIF PHILOSOPHY

TDIF caters to two different audiences.

One audience is simply looking for a consistent way in which to feed bare statements into a database engine. They want abstraction only so far as to ensure no matter what driver they use to access XSql, they get data returned in a predictable format. We will call this BMI: Bare Metal Interface.

This audience can pretty much skip to the chapter on Usage, most of this paper describes the full blown TDIF. Though the chapter on the Implementation may be an interesting read for those who wish to add new drivers.

The second audience does a lot of work accessing data from multiple data sources at once. For them, optimal database statements are a secondary concern. Their principle need is to transparently load and save data regardless of the database engine used. If indeed, the data is coming from a database at all. This interface uses all of the capabilities of TDIF: Tcl Data Interface Framework. TBMI is still available for the occasional optimization, or function that TDIF does not provide for.

In TDIF tables, records, and columns are abstracted into containers, elements, and properties (though I still call them 'fields' or 'columns' out of habit). This interface reduces all storage engines to a port in which to drop key/value lists.

I use TDIF as a framework on which to build much more complex systems. Reducing everything to key/value lists means that a webserver application no longer needs to worry about formulating queries. A user migration script can simply dump a list from one engine to another. An application writer can switch storage engines as a customization rather than a full-blown port.

TDIF CONCEPTS

Node

A node is simply an object within TDIF this is otherwise not defined. Structurally, tdif.node is a class that contains common operating elements of all the other object types. Method names use "node" because most of the time TDIF does not know or care what type of object is referenced.

Containers

While it us useful to think of a container as a data table, don't wrap yourself up in it. A data table is only on kind of container in TDIF. Standalone files are another type. A bank of instruments could be another container. In short, a container is any object in TDIF that acts as a grouping for other objects.

Containers can, in fact, contain other containers. Connectors are treated as a type of container. Instead of spawning database record objects, they spawn container objects for tables and property handling objects for columns.

A container can also spread it's data over connectors, or multiple tables within a connector, or conceivably multiple tables over multiple connectors. The point is, a developer simply writes and reads data to the container object. The container object works out all of the details on where data is stored and how.

Connectors

Connectors are a special kind of container that goes out and directly communicates with an outside data store. Other containers use Connectors to interact with the outside.

Database connectors spawn off other container objects. File connectors spawn off the records they contained with the file. Undoubtedly even more exotic arrangements will pop up as we go along.

The important takeway is that a connector is a container that has a few extra methods for interacting with the data store. Sql connectors also provide some creature comfort utilities for containers to formulate queries with.

Elements

Elements are objects that take data stored in records, and make them come alive within a TDIF application. Unlike containers, they do not contain other objects. Though that can link to other objects. Elements can also have data that spans multiple containers. (Though I suppose these should rightly be called **compounds**.) An element can also be present in more than one container. The key thing is that it deals with all of the data storage issue. All the application writer has to do is read and write key/value lists.

Properties

Properties are a special kind of node that controls how data being interacted with is formatted and validated. Think of them as objects that represent the columns in a database. If a container or an element want to generate a form, they pass the current value for each field to the property object, and it returns an Html input snippet. If the application wants to check if a new value is in a valid range, it can ask the property.

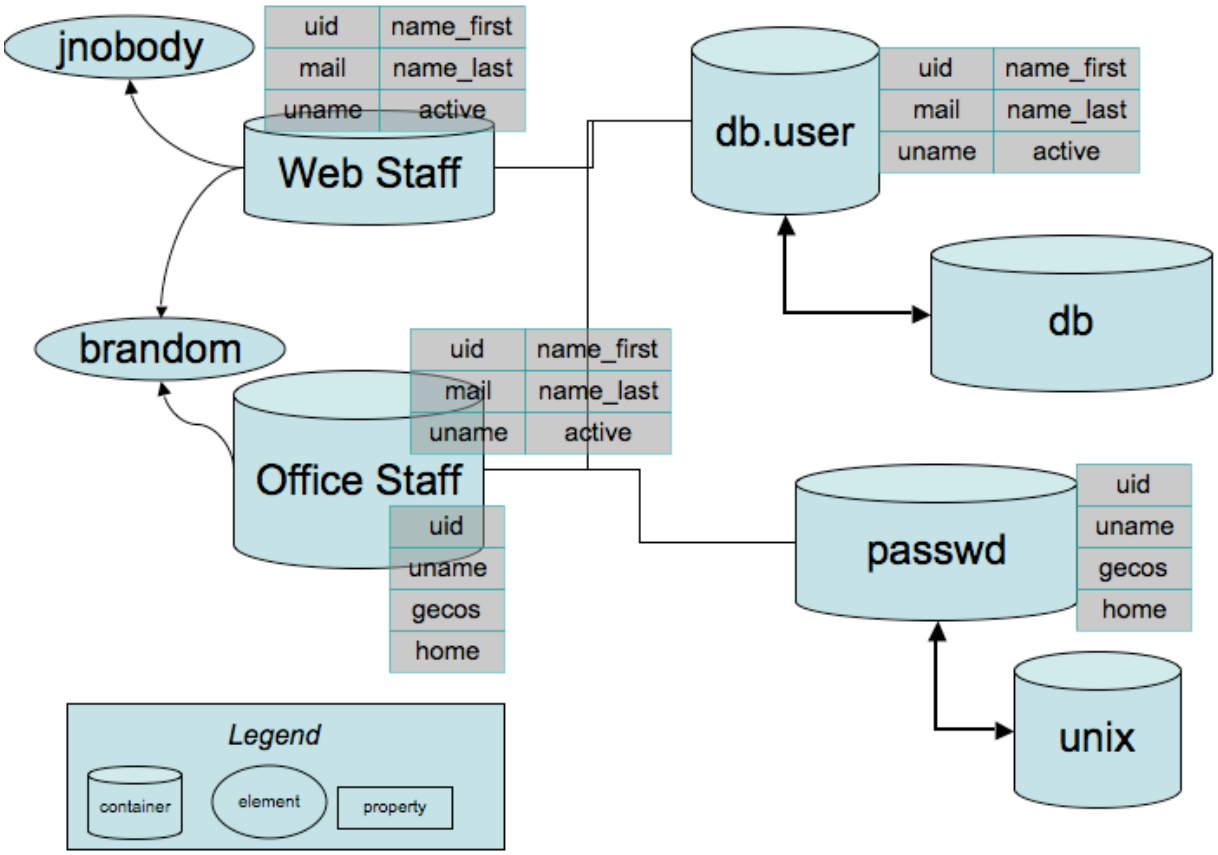
Putting it all together

This diagram shows all of the different concepts in action:

Figure 1 - TDIF in Action

The above example is a system in which we have different types of people. One is our office staff, who in addition to an entry in the online stafflist, also have an entry in the password table in unix. We also have frontline staff who come and go, but have no need (nor would we wish to grant them access to) our Unix system. A frontline staff element has all of the properties from the **db.user** table. Office staff have the properties from the **passwd** table in addition to the **db.user** table. Our element understands how changes to a field on one system need to reflect changes that take place in another.

For instance, if we change someone's last name on the staff directory, we also need to update the GECOS field in Unix if they are to be displayed properly. And of course some admins would find that writing directly to the *passwd* file to be a bit repugnant. So the **passwd** object makes all of it's updates through the *usermod* function. Also, no engineer worth his salt would put his security system at the mercy of his website, so the system call is done on the webserver's behalf by an another program. How the **passwd** container does its job is immaterial to us. That is the joy of abstraction.



USAGE

Let us building on our hypothetical staff directory/unix security bridge. We have a user "Betty Random." Betty marries a man with the surname "Stochiastic." Betty, being an old fashioned kind of

gal, changes her name to Betty Stochiastic. Of only to avoid writer's cramp from trying to sign Random-Stochiastic on checks. Because this sort of thing happens a lot in any vibrant organization, we want to design a simple way for our non-technical users in HR to be able to regularly update both the staff list and the name that appears on our unix hosted file server and domain controller.

A snippet of TDIF code to perform this update would look like this:

```
# Create the connector objects
tao::create_object maildb [lappend $dblogin class ::tdif::connector::mysqltcl]
tao::create_object passwd [class unixEtcPasswd]

# Create the staff directory container, and link it
tao::object_create staff [class staff_directory dbCon maildb passwdCon passwd]
tdif Container staff staff

# Spawn an object to handle brandom's account
set uObj [tdif spawn_object staff brandom]
$uObj displayName
Betty Random
$uObj Get
uid 1337 gecos {Betty Random} name_last Random name_first Betty
uname brandom active 1 mail brandom@fi.edu
$uObj Set {name_last Stochastic}
# Note that the name field has ALSO been updated
$uObj Get
uid 1337 gecos {Betty Stochastic} name_last Stochastic name_first Betty
uname brandom active 1 mail brandom@fi.edu
# Send changes back up to the containers. In the process, destroy the object
$uObj renew
passwd nodeGetField brandom gecos
Betty Stochastic
```

Example 1 - Using TDIF

The example above glosses some important details:

- How did we write the classes that drive this example?
- How does the object decide what fields go to which container?
- Which methods are standard TDIF, and what have my examples created?
- What does that "tdif Container" step do?
- What is that "spawn_object" step?
- Why does the table have the name **staff** in some places and **staffObj** in others?

All of these questions, and more will be answered in the course of this paper. We will begin with TCL interface, and work our way into creating new objects and finally creating new connectors.

Bare Metal

Many developers are not going to want TDIF to handle their table objects, nor provide transparent updates to things. What they do want is a consistent face with which to communicate to their database of choice. For these users we provide a subset of functions called the "BMI", the Bare Metal Interface. Essentially if you have the classes for the connectors already defined, you can call them into being and be on your merry way without all this talk about containers, elements, properties and whathaveyou.

What you do get is a consistent interface for any database the Tcl has a driver for, and someone has taken the time to code a TDIF wrapper. Here is a simple example of BMI in action:

```
# Call the connector object into being
tao::class maildb [class tdif.connector.mysqltcl \
      db_user smtpd db_pass password database maildb db_host localhost]
# Cry havoc an let loose the dogs of war
maildb query_flat "select email from user where username=‘swoods‘"
swoods@fi.edu
maildb cmnd "update user set name 'Evil Twin Skippy‘ where username=‘swoods‘"
```

Example 2 - Bare Metal Use if TDIF

Every TDIF connector exposes the following methods. They assume you know the native language of the database. Results are returned as a list, or a flat list. Doesn't sound all that thrilling, but it's actually the first feature that led me down the road to developing TDIF in the first place:

Method	Description
cmnd *statement*	Send a statement to the database that does not return data
query *statement*	Send a statement to the database the returns data, and format it as a list. Each row is one list item
query_flat *statement*	Send a statement to the database the returns data, and format it as a flattened list. Each column is one list item. Useful for feeding into foreach statements.
describe *table*	Using whatever means available, returns a dict that lists all columns and indexes for a table object
sqlfix *string*	Escape out any special characters in a string that could be interpreted as part of the statement
sqlprep *string*	Format a string complete with quotes to mark it as a value to be inserted as a column

Table 1 - Bare Metal Connector Methods

Connector objects are free to add additional methods to exploit the special features of their particular database engine, provided they do not overwrite one of the currently meaningful keywords. Keep in mind, connectors ARE containers, so in addition to the above 6, so all of the Container keywords also apply. See the appendix for a complete (as of the date of this writing) list of methods.

By convention, if your connector contains other sub-storage elements, container operations interact with meta-data, tables, columns, etc. If your connector goes right to a table, or some other flat data storage, container operations interact with its records.

containers

Containers are a corpus of data. Developers use them to provide a consistent interface to data tables and other logical storage units. Containers do not technically store data. They simply pass the messages between the nodes that use the data and the engine where the information is stored. They are also responsible for calling **element** and **property** nodes into being, as well as cleaning them up when the container is destroyed.

Some containers need to speak through a connector object. Others are tied directly to the data store. The difference is immaterial to the programmer. Their job is essentially to read and write key/value lists to a node identified by the nodeId.

A container in action (note we are re-using some of the objects from example 2):

```
# Create the table object using a class that reads the schema from the connector
::tao::object_create staffObj [class sqltable.static.autodetect \
    table maildb.user sqlObj maildb]
# Associate staff as a tdif container
tdif Container staff staffObj
staffObj nodeGetField 1104 name
{Evil Twin Skippy}
# Send an update through the container
staffObj nodeSet 1104 {name {Sean Woods}}
# See that the change has actually gone through
maildb query_flat "select name from user where username='swoods'"
{Sean Woods}
staffObj nodeGetField 1104 name
{Sean Woods}
```

Example 3 – Basic container usage

Containers expose the following methods to developers:

Method	Description
nodeSet *nodeId keyValueList*	Update set contents of *nodeId* with the values in *keyValueList*
nodeGet *nodeId*	Return the entire contents of *nodeId* as a key/value list
nodeGetField *nodeId fieldName*	Return the value of *field* stored in *nodeId*
nodeGetFields *nodeId fieldList*	Return the value of each *field* in *fieldList* stored in *nodeId* as key/value list
nodeDelete *nodeId*	Delete the node stored at *nodeId*
nodeClass *nodeId*	Return the class that defines how the node at *nodeId* should be generated
nodeAlias *string*	Return the address of a node that is associated with the alternate identification of *string*
train *nodeId*	Return a dict that is used by the object manager to create a spawned node object. Must include a value for **class**, **globalName**, **node_id**, and **containerObj**
tdifAttach	Method called when a container is associated with tdif.
/container	If this container is a child of another container, return the object handle of the parent.
/node *nodeId*	Return a node object to represent the data in *nodeId*
/property *propertyName*	Return an object handle for the **property** node that validates and represents the field **propertyName**.
spawnedNodes	Return a list of all nodes this container has spawned

Table 2 – Basic Container Exposed Methods

SQL Containers add a few additional methods.

Method	Description
/sql	Return the connector object of the Container
tdifAttach	Method called when a container is associated with tdif. SQL table objects use this method to populate the connector object with meta-data
schema	Return a dict with sub-elements that define the columns, indexes, and primary key of the table
nodeNext	Return the next available *nodeId*
nodeMatch *keyValueList ?like?*	Return a list of *nodeIds* that match the properties given by *keyValueList*. If a 1 is given for "like", the system matches using a pattern match instead of literal value.

Table 3 – SQL Container Exposed Methods

Container Handles

One may ask, why the manual step of running *tdif Container*? Truth be told, there is nothing keeping you the developer from adding that step to an object's constructor. The reason I include it in the example is to show that **tdif** is designed to play well with other object systems. It also is a nice way for me to mention the global record addressing system that **tdif** uses.

Suppose you wish to link records in two different containers. Yes, yes, if they are on the same SQL system you can create foreign keys and whatnot. But in script, say we have an entry in MySQL we wish to have associated with an entry in Sqlite. There is no realistic way to bind these things natively in the data store, so **tdif** has to do it in script.

To facilitate this kind of linking, **tdif** associates each container with a handle. During setup, the handle is associated with the object that is the actual container. A container can answer to many handles, but each handle can only be associates with on container at a time.

When we link records within the same container, we simply list the record ID numbers. (Or whatever the primary key is.) However, if we link to a element that is in another container, we add the handle of the other container. Because this link is stored in the actually data backend, and because we have no idea what the actual object that defines the container will be called in future implementations, we refer to it by this made up handle.

What handle a container answers to is entirely the responsibility of the developer to maintain. Though one convention I've developed for my webserver is to associate **users** with whatever container stores my user list, and **groups** with the grouplist. In my access control system, I can simply store links to **users-1104** and rest assured that however we are accessing the user list, as long as I keep a global number paired with an employee id, I'll be referring to user 1104, aka swoods, better known as me.

I've used this system for 6 years, and through countless technology changes. It works, even if the object systems change, tables move, and data technologies change. In my case the user list has moved from tfi.staff to mailbd.user, and will probably be an entry in LDAP pretty soon.

The other reason is to uncouple container objects from the same container system used to define TDIF. Because all of the interactions take place through object methods, there is nothing to keep a container (or any other object for that matter) defined in SNIT or XoTcl from working with TDIF. (TDIF itself is written in Tao). So not only could **users-1104** refer to database element in a completely different data store, the class that defines it could be written in an entirely different object system.

So why the convention of internally links be just the id, and external links be the container-nodeId? Because handles can change. If I decide to start referring to **users** as **staff**, how will I maintain all of the links that have already been created? What if a container is both **users** and **staff**? You start having to check all names for the container past, present, and possibly future just to find simple links between records. Not a happy state of affairs.

It doesn't hurt that a lot of db admins also just store record numbers in indexes, so this convention lets me rip link data direct from the database.

Elements

Elements are packets of data brought to life as objects. While they express the greatest variety of all objects in TDIF, they have the simplest interface. Elements are almost never created directly. They are spawned by their containers.

Here is a short example of elements in action. Again, this example builds on the previous example and makes use of the containers and connectors already created:

```
# Spawn an object
set recordObj [staffObj /node 1104]
$recordObj globalName
staff-1104
$recordObj Get name
{Sean Woods}
$recordObj Set {name {Evil Twin Skippy}}
$recordObj renew
staffObj nodeGetField 1104 name
{Evil Twin Skippy}
```

Example 4 - Basic element usage

In the above example, we spawn off an element object, write a value to it, and then save the changes back to the container. **renew** is an inside joke with me. It uploads changes and then calls the **carosel** method, that puts the object in line to be destroyed on the next pass of the garbage collector. (It's a reference to the movie *Logan's Run*.)

Why does upload lead to the destruction of a element object? It's the only way to consistently handle elements that change class based on their contents. At least in my experience.

Take a work order, for instance. It starts off as a problem report. After it is reviewed, it becomes a work order. That work order is then assigned to someone. Once the person assigned the work order has completed the task, the work order is marked completed. Once the supervisor is satisfied that a work order has been completed, it is closed.

In each phase of it's life cycle, the element responds differently to stimulus. While one *could* simply devise a grossly complicated all-seeing/all-dancing class that deals with all of the states through conditional statements... it's been my experience that these systems are a gory mess. Much better to define each state as a different class of element, and then use inheritence to keep the exploit the similarities.

To be a TDIF element, an object must respond to the following methods:

Method	Description
trashRecord	Signal to delete this node's data from the parent container on destruction of the object
carosel	Signal to the garbage collector to destroy the object on the next pass
renew	Upload the internal representation of the object to the container, and then **carosel**
NodeId	Return the 'primary key' of the element in the container
globalName	Return the fully qualifed name of this object: *containerHandle-nodeId*
Get *?field?*	Return the complete contents of the element as a key value list if not field is given OR Return the value of a specific property of the element
Set *keyValueList*	Update the internal representation of the object with the values in *keyValueList*
Input *keyValueList*	Validate/Format all of the values in a keyValueList (Does not change the internal state). Throws an error if a value is out of bounds. On success it returns a key/value list with values reformatted for the native store.
/property *propertyName*	Return an object handle for the **property** node that validates and represents the field **propertyName**. Note, this object will be the same whether you call /property from an **element** or it's **container**
/container	Return the object handle of this elements container

Table 4 – Basic Element Exposed Methods

Properties

Properties are the oddest part of the TDIF implementation. Early in the design, it became clear that it was useful to think of columns as objects unto themselves. Rather than store data, they were the voice of reason that understood all of the validation rules for the column. The column object knew what the max size for a field was. It would throw an error if we tried to store an integer in a string field, or vice verse. When I implemented an HTML display engine, the column object generated the individual user interface elements for each field, be it an input box, a dropdown menu, etc.

Because it is handy to have an **element** call for a **property**, we provide a mechanism for it to do so. Likewise, if we don't yet have an **element**, but we want to create a form to generate one, it becomes useful to call up a **property** from the **container**. Thus, both object types can call up **property** objects on demand.

While **properties** do not store element data, the *do* store their own information. Tidbits like the size and description for the field. This information is stored in the **connector** object. On nodes that are both the connector and the container, it's assumed the developer will be able to figure out which nodes are records and which are properties.

Containers have the option to generate all of their **properties** during *tdifAttach*, or to simply call them into being in response to a */property* call. Most elements redirect */*property calls to their container for this reason. **elements** are also welcome to generate their own properties. The key is that the object spelled out should be consistent whether the call is made to the container or the element, and that the the first thing the */*property method should do is check to see if a property exists before creating a new one.

Properties are not subject to the same garbage collection as elements. It is assumed that they will stick around until their container is destroyed.

Here is an example of property object usage. Again, this build on the objects we've created in previous examples.

```
# Spawn an object
set nameObj [$recordObj /property name]
$nameObj Display [$recordObj Get name]
Evil Twin Skippy
$nameObj htmlEntry [$recordObj Get name]
<input name=maildb.user.name value="Evil Twin Skippy" size=40>
$nameObj Input "An Increadibly Long String that is sure to exceed the size …"
An Increadibly Long String that is sure t
```

Example 5 – Basic property usage

Method	Description
element	Return the fully qualified path of this column (for use in forms and database statements)
Label	Return the description of the property
Display *value*	Render *value* into human readable form for the current display engine
Export *value*	Render *value* into human readable plain text
Input *value*	Check the validity of *value*, and convert an inputted value into native format of the storage engine
Entry *value*	Return a string, block, or script that defines a UI input form for this property
Search value	Return a string, block, or script that defines a UI search form for this property
Hidden *value*	Return a string, block, or script that defines a hidden UI element form for this property
Dump	Return the value entered into the form generated by Entry in a tk window
Update *value*	Update the display widget in tk window with the new *value*
Options	For boolean and select fields, return a list of valid options

Table 5 – Basic Property Exposed Methods

EXTENDING TDIF

The reference implementation for TDIF is written in my own object system, Tao. Tao currently requires either Tcl 8.5, or Tcl 8.4 with either the Dict or sqlite package. There is no earth shattering technology in Tao that would prevent TDIF from being re-written in any other object system.

TDIF also does not care what object system a node (be it a connector, container, element, or property) is written in. It will be perfectly happy passing data back and forth through an incr Tcl, Tao, snit, or plain old namespace. Simple implement the method as outlined in the usage section.

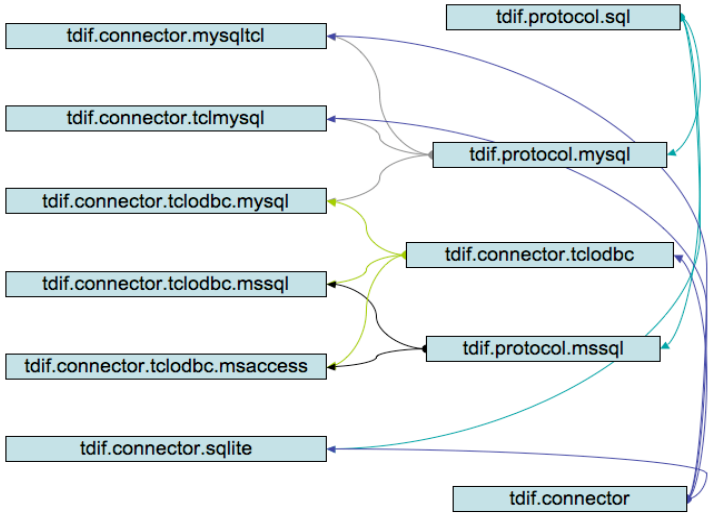
Writing in Tao would only be required to extend or enhance the existing library of connectors, containers, elements, and properties.

CONNECTORS

Connectors written in Tao have two parts:

- Data Language Interface – Translates between the TDIF and the specific SQL dialect of the database server.
- Data Driver Interface – Translates between TDIF and the Tcl interface of the database connection driver.

Why two? Basically the dividing line between the Protocol and the Connector is pretty blurry. To the right is a simplified relationship between the connectofs in TDIF.



TclOdbc, TclMysql and MySqlTcl all have a radically different connection driver. They are a raw pipe into MySql, though.

TclODBC doesn't just talk to MySql. It can talk to everything from Microsoft SQL, to access, to .txt files. It can also talk to PostGres, MySql, BerleyDBs, and so on. While the driver is identical, the language and features available depend on what type of database you are connecting to.

Then you have Sqlite, which is it's own language, its own driver, and it's pointless to try to separate the two. In other cases, our "driver" is really a raw socket talking to a relay. Once you abstract, it's really hard to think of connectors as monolithic things anymore.

Because we have no control over the data languages, and it will take some years for the driver writers to catch up with a standard interface, we need to accept their diversity as a fact of life. Each of system has its own capabilities, hindrances, and design panache. Sometimes a function a function we are looking for is in the language.

And then we have LDAP. Which I briefly tried to integrate, but frankly don't understand enough about to sensibly formulate policy. It did send me down the path of abstracting out database command statements though.

WRITING CONNECTORS

Essentially a connector needs to implement every method that a container object would need to call to access data. This includes abstracting out common queries and language constructs. Why? Well

some language provides a very nice way to INSERT OR REPLACE. Which saves a lot of work on updates, since we don't have to check for the existence of a record. On others, we check first, if a record exists we INSERT, otherwise we UPDATE. And, depending on your SQL dialect, INSERT and UPDATE have very different syntaxes.

In other cases we have features like full-text searching. It's nice if you have it. But many database systems don't. So we have to fake it. I've prepared a table of all of the methods that my Containers call and my Connectors implement. There will undoubtedly be more as we move along.

Method	Description
stmt_insert {keylist valuelist dtable}	Return an insert statment
stmt_exists {keylist dtable}	Return a 1 if the record in keylist exists in table, 0 otherwise
stmt_replace {keylist valuelist dtable}	Formulate an INSERT OR REPLACE where it's available, Otherwise call stmt_exists and return an INSERT statement or an UPDATE statement as appropriate
stmt_select {keylist fieldlist dtable}	Return a select statement
stmt_update {keylist valuelist dtable}	Return an update statement
stmt_where {keylist {forbid_null 0}}	Return a where statement, optionally yell if we try to enter a null value
stmt_delete {keylist dtable}	Return a DELETE statement
table_qualify {rawtable}	Convert a **db.table** to just **table** on the braindead systems that don't handle it properly. In the systems that DO handle **db.table**, make sure we add **db** if we just give **table**. Essentially we always want to call a table by the same name
column_qualify {rawcolumn {table {}}}	A similar process to table_qualify for columns. On systems with no concept of alternate databases, make the reference: **table.column**. On those that do, make it **db.table.column**. And by the way, make it relative to *table*
table_exists table	Return 1 if a table exists, 0 if it doesn't. And use the best mechanism you have.
searchStmtPrim {var op val}	Return a comparison statement in the native form for **op** the database. Valid **op**s in TDIF are: match notlike isnot = > < Also convert **null** and **now** in the val field to either a native representation, or some tcl calculated value. Not too useful for SQL, but essential when we go to add non-sql data languages
searchStmtJoin {joinop stmtl}	Connect several conditions created by searchStmtPrim into a compound conditional. Again, SQL we all know how to do. This is primarily for other languages.
searchStmtNest stmtl	Nest a compound statement into a single conditional. (In SQL we use parentheses. Other systems do it differently)
table_create {table schemaDict}	Create a table, if it doesn't already exist, to the spec defined in schemaDict.
stmt_create_column {field infoDict}	Return a column creation statement for a column named *field* using the information in *infoDict* as a guide.
stmt_create_index {idxtable idxname infoDict}	Return a statement that will create and index of *idxname* on *idxtable* according to the specs in *infoDict*
stmt_create_table {table columns indexes {table_type {}}}	Generate a statement that will create a table with the columns defined by the **columns** dict, the **indexes** dict, and of type **table_type**. At present, *temporary* is the only alternate type TDIF understands.
searchFullText {value {columns {}}}	Generate a fulltext search statement. If fulltext is not available, fake it.

Table 6 - Connector Language Abstraction Methods

Writing Containers

Containers would seem to be the hard part. There are so many, with so many different shapes. From a TDIF perspective, though, they are quite simple. And if you looked at the section on writing connectors, you'll see that most of the heavy lifting has actually been abstracted out to the connector.

Sql connectors are available with the stock release of TDIF, and all that is really required to customize them is to supply the proper schema. In most cases, I either dump data from the connector

object, or I provide a hard-coded dict to return through the schema method. There really isn't any magic going on behind the scenes.

A default container stores information as an in-memory dict. SQL containers redirect the nodeGet and nodeSet commands to pull data from the connector instead. And even there, they don't build sql statements. They farm that out to the connector.

The only trick is keeping track of what you spawn off in the way of **element** and **property** nodes, so you remember to destroy them when you destroy the container. Now containers *do* get to be monstrously complex. But the complexity is all driven by the application. The needs of TDIF are simple and few.

Writing Elements

Elements are only as complicated as your application. Beyond the basic methods outlined in the usage chapter, there are not magic interactions. You only need to be aware that garbage collection does take place, and *elements* are not permanent

Writing Properties

Like elements, the methods that TDIF provides are minimal for inter-operability. How they are used, and what additional methods you chose to implement are your own.

Writing in different object systems

TDIF is blind to what object system you are actually using, assuming you are using an object system at all. TDIF does make a few small requests if you do decide to strike out on your own and write objects that interact with TDIF in a system other than Tao:

Pay the Ferryman

If you write elements in your own object system, implement a "**scythe**" method for the garbage collector to call. **scythe** should return a procname to evaluate which will destroy your object. The garbage collector calls: "[$object scythe] $object" internally. If your object system has an ensemble command "object destroy $object", just define a proc that takes care of the details. For instance:

```
proc itclScythe objname {
  ::itcl::delete object $objname
}
```

Example 6 – Scythe Implementation for [Incr Tcl]

When a temporary object (especially elements) come into being, register with the garbage collector *thatanos.* All that's required to add an object to it's accounting is a call to *thanatos alloc $object*. Once thanatos has it allocated, it will delete temporary objects whenever you call: *thanatos cleanup*.

By convention, if you want to prevent the garbage collector from destroying an object, return an empty list when **scythe** is called.

For web-engines, I call thanatos cleanup at the end of every page view. TK based systems should probably call it after a data entry screen closes. Thanatos will only delete objects that have been "kissed." Tao calls *thanatos kiss $object* at the end of the *carosel* method.

Thantos also destroys objects if they have been idle for 60 seconds. To mark an object as not idle, call *thantos touch $object*. This will grant it another 60 second lease on life. To change the lease length for all objects, set a new value for ::thanatos::kill_time. Kill_time takes integer values, in seconds.

The complete interface to *thanatos* is:

Method	Description
alloc *object*	Add an object to automatic garbage collection
free *object*	Remove an object from automatic garbage collection
touch *object*	Extend an object's lease on life
kiss *object*	Trigger an object's destruction on the next call to *cleanup*
knock	Check to see if any objects are set to be destroyed. (If zero, cleanup is not called)
cleanup	Run through and dispatch any object that has been *kissed* or whose lease has expired

Table 7 – Garbage Collector Interface

Thanatos expects to see the following methods in any object it interacts with:

Method	Description
scythe	Proc to call to destroy an object

Table 8 – Methods to expose for Garbage Collector

I would also like to point out that "thanatos kiss" should not be called during an object's destructor. Odd's are if the destructor has been called, the object has already been kissed. What you may want to do is "thantos free" the object. This will remove if from garbage collection no matter if it self destructs or is removed through another mechanism.

Stay with the times

TDIF includes a built in schedular object *chronos*. Chronos is called periodically to run tasks. Rather than invent your own periodic task use TDIF's. Chronos requires no exposed methods in objects.

The MySql connector objects presented a troublesome problem for me because the idle/reconnect feature in the drivers is broken, wounded, or missing. So I had to implement my own in script, thus the genesis of Chronos. To use chronos:

```
set jobId [::chronos::JobCreate name "Test Job  " \
                            interval 60 script "puts {Hi There}"]
Hi There
Hi There
….
  ::chronos::JobKill $jobId
```

Example 7 – cronos usage

Format for schema dicts

Containers report their schema as a dict. The table_create function in the connector objects ALSO takes data in as a dict. What is the format for this dict? The dict has four main entries: **column, index, primary_key,** and **type**.

```
staffObj schema
primary_key uid
column {
   uid {type intkey}
   username {type char length 32 required 1}
   name {type string desc {Full Name}}
   name_last {type string desc {First Name}}
   name_first {type string desc {Last Name}}
   mail {type char length 64 required 1 desc {Email Address}}
   active {type boolean options {1 0} states {0 no 1 yes} default 1 \
      desc   {Display on Stafflist}}
   password {type password crypt sha1 length 64 sqltype char \
      desc {Password}}
}
index {
   uuname {type unique columns username}
   uemail {type unique columns email}
   maildx {type index columns {
     uid
     {username direction ascending}
     {email length 16 direction ascending}
   }
   nsearch {type fulltext columns {name name_last name_first}}
}
```

Example 8 – schema dict

Column

The column dict lists each column in a table in the format: *columnName {key value ...}*. This dict is used to generate **property** nodes. The table of values used by TDIF are as follows:

Property	Description
type	What type of property (see table)
sqltype	Native storage type (optional)
desc	Property/Column description
length	Length of the field (longer values are truncated on Input)
width	Display size of the field
default	The default value for the column
options	For enum and select types, the range of valid values
states	Mapping of stored values to human readable values
required	Allow null values
collate	Colation to use on column. Valid values: binary, nocase

Table 9 – TDIF Column properties

TDIF Type	MySql Representation	MSSql Representation	Sqlite Representation	Description
string	tinytext	char(*length*)	string	Generic String
int, integer	int	int	int	Integer
intkey	...	...	...	Auto Incrementing Primary Key (implementations differ between data engines)
char, varchar	varchar(*length*)	char(*length*)	string	String of fixed width. Note that the **collate** option will select char or varchar in MySql to emulate the collate functionally in sqlite.
text	bigtext	text	string	Large block of text
select	enum(*options)*	char(*length)*	string	Enumerated list of values
boolean	tinyint	tinyint	int	1/0 or Y/N or True/False value

Table 10 – TDIF Column Types

Index

The index portion simply lists all of the indexes, by name, as well as a type and columns affected.

Field	Description
type	Index Type: index, unique, fulltext
storage	Native storage engine to use (if supported, ignored otherwise). In MySql available values are: BTREE and HASH
columns	Dict describing the columns indexed

Table 11 – TDIF Index properties

Field	Description
length	Integer, length of the field to index
direction	Sort direction to index by this column. Valid values: asc, ascending, desc, descending
collate	Colation to use (if supported by database, ignored otherwise) For sytems

Table 12 – TDIF Index column properties

TDIF Type	MySql Representation	MSSql Representation	Sqlite Representation	Description
index	index	index	index	Generic index
unique	unique index	unique index	unique index	Uniquely constrained index
fulltext	fulltext index	*emulated*	*emulated*	Fulltext search index

Table 13 – TDIF Index Types

CONCLUDING REMARKS

TDIF is a work in progress. My intent is to at least start a conversation on how we in the Tcl community can begin adopting a formal way of interfacing with external data. Why is such a framework desirable?

Newcomer Friendly

For starters, it makes the system friendlier to newcomers. I won't pitch this as a major reason for adoption, but it is at least a nice side effect. Few can argue though that providing a consistent interface for all database connections will make documenting simpler.

Interoperability

A stronger reason than consistency for consistency's sake is that software developed for one system can be more easily integrated into another if they can at least agree on how to access the database. By abstracting most of the SQL statement generation, one also allows software originally developed for one environment to be turned around and used in another with little to any modification.

Optimization

We are not all SQL experts. By abstracting out common functions to a system that can be customized for the individual database engine, we can exploit performance enhancing tricks. Take for instance using the "INSERT OR REPLACE" statement in Sqlite, instead of the conventional two step process required for other databases.

Safety

Abstracting out SQL statement generation also allows us to catch common mistakes. Say a "DELETE FROM TABLE" or "UPDATE" with no WHERE condition. It also ensures that all of the data from Tcl is escaped properly, and won't cause a bizarre interaction if someone manages to input a value that just so happens to be the name of a column, and they JUST so happen to have not put the value in quotes.

It doesn't just work for SQL

By abstracting the concept of tables out to containers, we open ourselves to an entire world of data storage systems. We can write a handler that wraps around a flat file. We can transparently relay database calls to another machine.

Code can work in multiple operating environments

A case in point, I have libraries of scripts the run from a shell environment on various server as well as withing my tclhttpd base intranet. Depending on the computer, it may or may not have access to the mysqltcl package. If it does not, I create an object, with the same name, that takes all of the calls that *would* have gone to the mysqltcl connector, and instead relays them to a helper daemon running on the database server.

My software doesn't know, or care, which is which.

For all of these reasons, and more, I hope that TDIF will spark a conversation, if not a change in thinking, on how we treat outside data. Feel free to contact me with suggestions on how TDIF can be improved.

You can download a reference implementation of TDIF, as well as Tao (the object system it is written in) at:

http://www.etoyoc.com/tao

Further Reading

"The Tcl Architecture of Objects", Sean Woods: http://www.etoyoc.com/tao/

About the Author

Sean Deely Woods is the Senior Network Engineer at the Franklin Institute Science Museum in Philadelphia, PA. He has been writing in Tcl professionally since 1997. His claims to fame include a serial port driver under MacOS classic, and the famous "Hypnotoad" presentation at the 13th tcl conference.

Sean can be reached via email at: yoda@etoyoc.com

He maintains an independent website at: http://www.etoyoc.com/

He can be found on the Tclers chat under the pseudonym **hypnotoad**

Tclers Wiki entries by him are normally signed **SDW**

Credits:

Graphics and Art

Cover Graphic, "Game Console", Username MachZero, ConceptArt.org:
http://www.medievalfx.com/machzero/graphics/050420a.jpg

Fonts Used:

Text: Eurostile
Title: **MAINFRAME**
Headings: **MAINFRAME**, METRODF, Monoglyceride
Captions: *Aerovias Brasil NF*
Code: Monoco CE

Speed Tables - A High-Performance, Memory-Resident Database for Tcl

Karl Lehenbauer
Peter da Silva

Speed tables provides an interface for defining tables containing zero or more rows, with each row containing one or more fields. The speed table compiler reads a table definition and generates C code to create and manage corresponding structures, producing a set of C access routines and a C language extension for Tcl to create, access and manipulate those tables. It then compiles the extension, links it as a shared library, and makes it loadable on demand via Tcl's "package require" mechanism.

Speed tables are well-suited for applications for which this table/row/field abstraction is useful, with row counts from the dozens to the tens of millions, for which the performance requirements for access, search and/or update frequency exceed those of the available SQL database, and the application does not require "no transaction loss" behavior in the event of a crash.

In contrast to ad-hoc tables implemented with some combination of arrays, lists, upvar, namespaces, or even using Tcl 8.5's dicts, speed tables' memory footprint is far smaller and performance far higher when many rows are present.

Speed tables' search capabilities include indexed searches, results sorting, setting offsets and limits, specifying match expressions, and counting. A configurable searching engine, speed table searches bypass the Tcl interpreter on a row-by-row basis (except for processing matches), providing high performance. Speed tables support tab-separated reading and writing to files and TCP/IP sockets, and has a direct C interface to PostgreSQL. Examples are provided for importing SQL query results into a speed table as well as copying from a speed table to a database table, again bypassing the interpreter on a per-row basis.

Representing Complex Data Structures in Tcl

Tcl is not known for its ability to represent complex data structures. Yes, it has *lists* and *associative arrays* and, in Tcl 8.5, *dicts*. Yes, object-oriented extensions such as *Incr Tcl* provide ways to plug objects together to represent fairly complex data structures and yes, the *BLT toolkit*, among others, has provided certain more efficient ways to represent data (a vector data type, for instance) than available by default and, yes, it is possible to abuse *upvar* and *namespaces* as part of expressing the structure of, and methods of access for, your data.

There are, however, three typical problems with this approach:

1. It is memory-inefficient.

   Tables implemented using Tcl objects use at least an order of magnitude more memory than native C.

   For example, an integer, stored as a Tcl object, has the integer value and all the overhead of a Tcl object, 24 bytes minimum, routinely

more, and often way more. When constructing Tcl lists, there is an overhead to making those lists, and the list structures themselves consume memory, sometimes a surprising amount as Tcl tries to avoid allocating memory on the fly by often allocating more than you need, and sometimes much more than you need. [1]

Another drawback of Tcl arrays is that they store the field names (keys) along with each value, which is inherently necessary given their design but is yet another example of the inefficiency of this approach.

2. It is computationally inefficient.

   Constructing, managing and manipulating complicated structures out of lists, arrays, etc, is quite processor-intensive when compared to, for instance, a hand-coded C-based approach exploiting pointers, C structs, and the like.

3. It yields code that is clumsy and obtuse.

   Using a combination of *upvar* and *namespaces* and *lists* and *arrays* to represent a complex structure yields relatively opaque and inflexible ways of expressing and manipulating that structure, twisting the code and typically replicating little pieces of weird structure access drivel throughout the application, making the code hard to follow, teach, fix, enhance, and hand off.

Speed tables reads a structure definition and emits C code to create and manipulate tables of rows of that structure. We generate a full-fledged Tcl C extension that manages rows of fields as native C structs and emit subroutines for manipulating those rows in an efficient manner.

Memory efficiency is high because we have low per-row storage overhead beyond the size of the struct itself, fields are stored in native formats such as short integer, integer, float, double, bit, etc, and field names only occur once for a table type regardless of the number of tables created and the number of rows in those tables.

Computational efficiency is high because we are reasonably clever about storing and fetching those values, particularly when populating from lines of tab-separated data as well as PostgreSQL database query results, inserting into them by reading rows from a Tcl channel containing tab-separated data, writing them tab-separated, locating them, updating them, and counting them, as well as importing and exporting by other means.

Speed tables avoids executing Tcl code on a per row basis when a lot of rows need to be looked at. In particular when bulk inserting and bulk processing via search, Tcl essentially configures an execution engine that can operate on millions of rows of data without the Tcl interpreter's per-row involvement except, perhaps, for example, executing scripted code only

---

1 It is common to see ten or twenty times the space consumed by the data itself used up by the Tcl objects, lists, arrays, etc, used to hold them. Even on a modern machine, using 20 gigabytes of memory to store a gigabyte of data is at a minimum kind of gross and, at worst, renders the solution unusable.)

on the rows that match your search criteria.

Null Values

Speed tables maintains a "null value" bit per field, unless told not to, and provides an out-of-band way to distinguish between null values and non-null values, as is present in SQL databases... providing a ready bridge between those databases and speed tables.

Indexes

Speed tables supports defining skip list-based indexes on one or more fields in a row, providing multi-hundred-fold speed improvements for many searches. Fields that are not declared to be indexable do not have any code generated to check for the existence of indexes, etc, when they are changed, one of many of optimizations in place to make speed tables fast.

Speed Table Data Types

The following data types are available[2]:

- *boolean* - a single 0/1 bit
- *varstring* - a variable-length string
- *fixedstring* - a fixed-length string
- *short* - a short integer
- *int* - a machine native integer
- *long* - a machine native long
- *wide* - a 64-bit wide integer (Tcl Wide)
- *float* - a floating point number
- *double* - a double-precision floating point number
- mac - an ethernet MAC address
- *inet* - an internet IP address
- *tclobj* - a Tcl object

Fields are defined by the data type followed by the field name, for example...

```
double longitude
```

...to define a double-precision field named longitude.

Configurable Field Attributes

Field definitions can followed by one or more key-value pairs that define additional attributes about the field. Supported attributes include

- *indexed*

If “indexed” is specified with a “true” value, the code generated for the speed table will include support for generating, maintaining, and using a skip list index on the field being defined.

Indexed traversal can be performed in conjunction with the speed table’s search functions to accelerate searches and avoid sorting (since skip lists are sorted). This defaults to “indexed 0”, i.e. the field is not generated with index support.

- *notnull*

If notnull is specified as true, the code generated for the speed table will not have code for maintaining an out-of-band null/not-null status created for it, increasing the performance of ma-

2 Additional data types can be added, although over Speed Tables’ evolution that has become an increasingly complicated undertaking.

nipulating fields for which out-of-band null support is not needed. Defaults to “notnull 0”.

• *default*

If default is specified, the following value is defined as the default value and will be set into rows that are created when the field does not have a value assigned.

There is no default default value; however if no default value is defined and the field is declared as notnull, strings will default to empty and numbers will default to zero.

• *length*

Currently only valid for fixedstring fields, length specifies the length of the field in bytes. There is no default length; length must be specified for fixedstring fields.

• *unique*

If unique is specified with a true value, the field is defined as indexed, and an index has been created and is in existence for this field for the current table, a unique check will be performed on this field upon insertion into the speed table.

• *key*

If key is specified as true, this field will become the key for the table. There must be at most one “key” field, and it currently must be a varstring. (Any field type can be indexed but our Tcl-adapted hashtables require strings as indexes.) If no “key” field exists then the key will not be exposed as part of a row unless it is explicitly referenced with the name “_key”.

Special Fields

Named fields may not begin with an underscore (as these are reserved for speed table internals), but there are two special field names that may be used in any place where a field is specified:

• *_key*

If no field is specified as a key, this name can be used to reference the key for the row.

• *_dirty*

This is set whenever a field is modified, and may be explicitly cleared… for example when a table is saved to a TSV file in a search operation.

Example Speed Table Definition

```
package require speedtable

CExtension animinfo 1.1 {

      CTable animation_characters {
          varstring name indexed 1 unique 0
          varstring home
          varstring show indexed 1 unique 0
          varstring dad
          boolean alive default 1
          varstring gender default male
          int age
```

```
        int coolness
    }

}
```

- Speed tables are defined inside the code block of the *CExtension*.
- Executing this will generate table-specific C functions a Tcl C language extension named *Animinfo*, compile it along with support code and link it it into a shared library.
- Multiple speed tables can be defined in one CExtension definition.
- No matter how you capitalize it, the package name with be the first character of your C extension name capitalized and the rest mapped to lowercase.
- The name of the C extension follows the CExtension keyword, followed by a version number, and then a code body containing table definitions.

Loading Your Speed Table-Generated C Extension

After sourcing in the above definition, you can do a

```
package require Animinfo
```

or

```
package require Animinfo 1.1
```

and Tcl will load the extension and make it available.

It is not necessary to re-execute the CExtension definition to use it again, but it is always safe (and efficient) to do so -- we detect whether or not the C extension has been altered since the last time it was generated as a shared library, and avoid the compilation and linking phase when it isn't necessary.

Sourcing the above code body and doing a `package require Animinfo` will create one new command, *animation_characters*, corresponding to the defined table. We call this command a *meta table* or a *creator table*.

`animation_characters create t` creates a new object, **t**, that is a Tcl command that will manage and manipulate zero or more rows of the *animation_characters* table.

You can create additional instances of the table using the meta table's *create* method. All tables created from the same meta table operate independently of each other, although they share the meta table data structure that speed table implementation code uses to understand and operate on the tables.

You can also use **`set obj [animation_characters create #auto]`** to create a new instance of the table, without having to generate a unique name for it.

Basic Examples

All rows in a speed table have a unique key value, which normally resides outside of the table definition itself. The simplest way to create or modify a row is with the *set* operation. Performing a *set* on a speed table performs an update or insert:

```
t set shake \
  name "Master Shake" \
  show “Aqua Teen Hunger Force”
```

If there is no row in the table with "shake" as a key [3], this creates a new row in the speed table **t**, otherwise it updates the current value of the row having the key "shake" with a new name and show.

We can set other fields in the same row:

```
t set shake age 4 coolness -5
```

And increment them in one operation with "incr":

```
% t incr shake age 1 coolness -1
      5 -6
```

You can fetch a single value naturally with "get"...

```
if {[t get $key age] > 18} {...}
```

Or can get all the fields in the row, in the order they were defined in the ctable definition:

```
puts [t get shake]
{} {} {} {} {} 1 male 5 -6
```

Forgot what fields are available?

```
% t fields
id name home show dad alive gender
age coolness
```

You can get a key-value list of fields, suitable for passing to **`array set`**, using array_get:

```
array set data [t array_get shake]
puts "$data(name) $data(coolness)"
```

If a field's value is null then the field name and value will not be returned by *array_get*. So if a field can be null, you need to check for its existence using `info exists` before trying to use it or use *array_get_with_nulls*, which will always provide all the fields' values, substituting a null value string, and typically the empty string) when the value is null.

You can check if a key exists with *exists*:

```
t exists frylock
0
```

Or load a complete table from a file tab-separated data with *read_tabsep*:

```
set fp [open
animation_characters.tsv]
t read_tabsep $fp
close $fp
```

Search

Search is one of the most useful capabilities of speed tables. Let's use search to write all of the rows in the table to a save file:

```
set fp [open save.tsv]
t search -write_tabsep $fp
close $fp
```

Want to restrict the results to a certain set of fields? Use the `-fields` option followed by a list of the names of the fields you want.

```
t search -write_tabsep $fp \
    -fields {name show coolness}
```

Sometimes you might want to include the names of the fields as the first line...

```
t search -write_tabsep $fp \
    -fields {name show coolness} \
    -with_field_names 1
```

Let's find everyone who's on the Venture Brothers show who's over 20 years old, and execute code for each result:

```
t search \
    -compare {
        {= show "Venture Brothers}
```

3 The key for a row has a name, "_key", but it's not exposed implicitly in operations on the default list of fields. It is also possible to use the "key" attribute to make any single varstring

```
        {> age 20}
    } \
    -array data -code {
    parray data
    puts ""
}
```

Additional meta table methods

- *animation_characters null_value \\N* - which sets the default null value for all tables of this table type to, in this case, \N.
- *animation_characters method foo bar* - this will register a new method named *foo*, which will be available to all instances of the table. Invoking the foo method will cause the *bar* proc to be called with the arguments being the name of the table followed by whatever arguments were passed.

For example, if after executing **`animation_characters method foo bar`** and creating an instance of the *animation_characters* table named **t**, if you executed

```
t foo a b c d
```

then proc *bar* would be called with the arguments "*t a b c d*".

Where the table Is Built

The generated C source code, some copied .c and .h files, the compiled .o object file, and shared library are normally written in a directory called **`build`** underneath the directory that's current at the time the CExtension is sourced, unless a build path is specified. For example, after the "package require ctable" and outside of and prior to the CExtension definition, if you invoke

```
CTableBuildPath /tmp
```

...then those files will be generated in the **`/tmp`** directory.

Note that the specified build path is appended to the Tcl library search path variable, *auto_path*, if it isn't already in there.

Methods for Manipulating Speed Tables

The following built-in methods are available as arguments to each instance of a speed table:

> *get*, *set*, *array_get*, *array_get_with_nulls*, *exists*, *delete*, *count*, *foreach*, *type*, *import*, *import_postgres_result*, *export*, *fields*, *fieldtype*, *needs_quoting*, *names*, *reset*, *destroy*, *statistics*, *write_tabsep*, *read_tabsep*

For the examples, assume we have done **`cable_info create x.`**

• *set*

There are two ways to specify the fields to set:

```
x set key field value \
    ?field value...?
```

or

```
x set key keyValueList
```

The key is unique. It can be any string and is not normally a field of the table. The following commands are equivalent:

```
x set peter ip 127.0.0.1 \
    name "Peter da Silva" i 501
x set peter {
    ip 127.0.0.1
    name "Peter da Silva"
    i 501
}
```

Thus a natural way to pull an array into a speed table row is:

```
% x set key [array get dataArray]
```

• *fields*

"fields" returns a list of defined fields, in the order they were defined.

• *field*

"field" returns information about the field attributes. Since we ignore attributes we don't recognize, you can include your own key-value pairs and access them using this method: **`field getprop name`** returns the value of the *name* attribute**`. field properties`** returns a list of all attributes. **`field proplist`** will return the names and values of all the properties in the usual name-value format.

• *get*

Get fields. Get specified fields, or all fields if none are specified, returning them as a Tcl list.

```
% x get peter
127.0.0.1 {} {Peter da Silva} {} {} {} 501 {} {}
% x get peter ip name
127.0.0.1 {Peter da Silva}
```

• *array_get*

Get specified fields, or all fields if none are specified, in "array get" (key-value pair) format. Null fields will not be fetched.

```
% x array_get peter
ip 127.0.0.1 name {Peter da Silva} i 501
% x array_get peter ip name mac
ip 127.0.0.1 name {Peter da Silva}
```

• *array_get_with_nulls*

Get specified fields, or all fields, in "array get" format, including null fields.

• *exists*

Return 1 if the specified key exists, 0 otherwise.

```
% x exists peter
1
% x exists karl
0
```

• *delete*

Delete the specified row from the table. Returns 1 if the row existed, 0 if it did not.

```
% x delete karl
0
% x set karl
% x delete karl
1
% x delete karl
0
```

• *count*

Returns the number of rows in the table.

• *batch*

The batch command provides an efficient way to perform a series of ctable operations. It takes a list of ctable commands (without the ctable name) and returns a list of results. Each element in the result list is a list of the index of the result, and a list of two values: the Tcl result code (for example, 0 for TCL_OK, 1 for TCL_ERROR) and the result string (result or error string).

```
% x batch {{set dean age 17} {incr dean age 1} {incr brock age foo}}
{{1 {0 18}} {2 {1 {expected integer but got "foo" while converting age increment amount while processing key-value list}}}
```

Dean's age to 17 produced no result. Incrementing it returned the incre-

mented value (18), and trying to add ‘foo’ to Brock’s age produced an error.

Note that *errors in batched commands do not cause batch to return an error*. It is up to the caller to examine the result of the batch command to see what happened: “batch” will only return an error in the event of bad arguments such as an invalid “batch” list.

• *search*

Search for matching rows and take actions on them, with optional sorting.

Search is a powerful element of the speed tables tool that can be leveraged to do a number of the things traditionally done with database systems that incur much more overhead.

Search can perform brute-force multivariable searches on a speed table and take actions on matching records, without any scripting code running on an every-row basis.

On a modern 2006 Intel and AMD machines, speed table search can perform, for example, unanchored string match searches at a rate of sixteen million rows per CPU second (around 60 nanoseconds per row).

Search[4] has scads of options:

• `-sort sortArg`

Sort results based on the specified field or fields. To sort a field in descending order, put a dash in front of the field name.

• `-fields fieldList`

Restrict search *results*[5] to the specified fields, which (among other things) may produce a noticable performance boost..

• `-glob pattern`

Perform a glob-style comparison on the *key*, excluding the examination of rows not matching.

• `-countOnly 1`[6]

Counts matching rows but does not take any action based on the count.

• `-offset offset`

• `-limit limit`

Like the SQL “offset” and “limit” parameters, these limit the results to a section of the ctable. The results are not well-defined without -sort or -countOnly.

• `-write_tabsep channel`

Matching rows are written tab-separated to the file or socket (or postgresql database handle) "channel".

• `-with_field_names 1`

If you are doing -write_tabsep, `-with_field_names 1` will cause the first line emitted to be a tab-separated list of field names.

• `-compare list`

Perform a comparison to select rows.

---

4 Like berkeley ls.

5 Fields that are used for sorting and/or for comparison expressions do not need to be included in -fields in order to be examined.

6 All search parameters must have a value, so “-countOnly” requires the value “1”.

Compare expressions are specified as a list of lists. Each list consists of an operator and one or more arguments. Each expression is applied to each row in turn, and all expressions must match for the search to succeed.

Here's an example:

```
$speedTable search -compare {
      {> coolness 50}
      {> hipness 50}
} ...
```

In this case you're selecting every row where coolness is greater than 50 and hipness is greater than 50.

Most expressions are fairly easy to understand:

- `{false` *`field`*`}`
- `{true` *`field`*`}`
- `{null` *`field`*`}`
- `{notnull` *`field`*`}`

Comparisons are type-sensitive:

- `{< field value}`
- `{<= field value}`
- `{= field value}`
- `{!= field value}`
- `{>= field value}`
- `{> field value}`

String matching uses “glob” operations:

- `{match field expression}`
- `{match_case field expression}`
- `{notmatch field expression}`
- `{notmatch_case field expression}`

Range and are the most efficient when performed on indexed fields:

- `{range field low hi}`

List operations must be performed on indexed fields, and only one may be in a list. Yes, this is not optimal and will be changed in a future release:

- `{in field valueList}`

- `-code codeBody`

Run scripting code on matching rows, along with one or more of these options:

- `-key keyVar`

Make the key value of the matched row be available inside the code block as *keyVar*.

- `-get listVar`

The fields of the row are available in the variable *listVar*.

- `-array arrayName`
- `-array_with_nulls arrayName`

The fields are available as the array *arrayName*.

- `-array_get listVar`
- `-array_get_with_nulls listVar`

The fields are available in an “array get” format list in *listVar*.

Search examples:

Write everything in the table tab-separated to channel $channel

```
$speed table search
-write_tabsep $channel
```

Write everything in the table with coolness > 50 and hipness > 50:

```
$speed table search\
   -write_tabsep $channel
   -compare {
       {> coolness 50}
       {> hipness 50}
}
```

Run some code every everything in the table matching above:

```
$speed table search \
    -compare {{> coolness
50} {> hipness 50}} \
     -key key -array_get
data -code {
          puts "key -> $key,
data -> $data"
    }
```

- *incr*

  Increment the specified numeric values, returning a list of the new incremented values

  % `x incr $key a 4 b 5`

  ...will increment $key's a field by 4 and b field by 5, returning a list containing the new incremented values of a and b.

- *type*

  Return the "type" of the object, i.e. the name of the object-creating command that created it.

  % x type

  cable_info

- *key*

  Return the name of the “key” field in the ctable (usually _key).

- *makekey*

  Given a list of name-value pairs, return the key value.

- *methods*

  Return a list of defined methods (commands) that the ctable can handle. The speedtable API may include extensions (such as the ctable server) or implement ctable-compatible classes independently of ctables (for example, there’s a STAPI definition for pgsql), so it may be necessary to check whether the STAPI-compatible object that you are examining supports the commands you need.

- *store*

  x store keyval_list

  Stores the list in the ctable using the key defined in the ctable definition, or using an autoincremented numeric key compatible with read_tabsep if the table’s key field is not specified in the list.

- *import_postgres_result*

  `x import_postgres_result pgTclResultHandle`

  Given a *Pgtcl* result handle, *import_postgresql_result* will iterate over all of the result rows and create corresponding rows in the table. This is fast as it does not do any intermediate Tcl evaluation on a per-row basis.

```
set res [pg_exec $connection \
    "select * from mytable"]
if {[pg_result $res -status] ==
"PGRES_RESULT_OK"} {
  x import_postgres_result \
    $res
}
$res destroy
```

- *fieldtype*

  Return the data type of the named field, such as varstring.

- *needs_quoting*

  Given a field name, return 1 if it might need quoting. For example, varstrings and strings may need quoting, while integers, floats, IP addresses, MAC addresses, etc, do not.

- *names*

  Return a list of all of the keys in the table. This is fine for small tables but horribly inefficient for large tables; use *search* instead.

- *reset*

  Clear everything out of the table.

- *destroy*

  Delete all the rows in the table, free all of the memory, and destroy the object.

- *read_tabsep*

  Read tab-separated entries from a Tcl channel, with a list of fields specified, or all fields if none are specified.

```
set fp [open /tmp/output.tsv r]
x read_tabsep $fp
close $fp
```

  The first column is normally the key and is not included in the list of fields. So if you name five fields, for example, each row must contain six columns.

  Options:

  - *-glob pattern*

    If the key does not match, the row is not inserted.

  - *-nokeys*

    The first column is not a key column. If the table has a key field defined, and that column is in the fields being read, then it will be used. Otherwise an auto-incremented numeric key will be generated for each row and read_tabsep will return the last key generated.

    read_tabsep stops when it reaches end of file OR when it reads an empty line.

- *index*

  Index actually creates the index for fields with the indexed attribute. This is a separate operation because it is far more created the indices AFTER populating a large table.

  ```
  x index create foo 24
  ```

  Creates a skip list index on field "foo" and sets it to for an optimal size of 2^24 rows. The size value is optional. If there is already an index present on that field, does nothing.

  ```
  x index drop foo
  ```

  Drops the skip list on field "foo." If there is no such index, does nothing.

  ```
  x index count foo
  ```

  Returns a count of the skip list for field "foo".

  x index span foo

  Returns a list containing the lexically lowest entry and the lexically

highest entry in the index. If there are no rows in the table, an empty list is returned.

```
x index indexable
```

...returns a (potentially empty) list of all of the field names that can have indexes created for them.

```
x index indexed
```

...returns a (potentially empty) list of all of the field names in table x that current have an index in existence for them, meaning that index create has been invoked on that field.

Performance Importing PostgreSQL Results

On a 2 GHz AMD64 running FreeBSD, speed tables can import import about 200,000 10-element rows per CPU second, i.e. around 5 microseconds per row. Importing is slower if one or more fields has an index.

Speed Table Search Performance

An example of brute force searching that there isn't much getting around without adding fancy full-text search features is unanchored text search. Even in this case, with speed tables's fast string search algorithm and quick traversal during brute-force search, the authors have observed 60 nanoseconds per row, thus searching about sixteen million rows per CPU second on circa-2006 AMD64 machines.

Although many optimizations are being performed by the speed table compiler, further performance improvements can be made without introducing huge new complexities, perturbations, etc.

Indexed Searches

Many searches can be greatly accelerated through the use of indexes on appropriate fields. Please consult the documentation for which operators and under what conditions indexes cause searches to be accelerated.

Client-Server Speed Tables

Tables created with Speed Tables are, by default, local to the Tcl interpreter that created them.

Early in our work it became clear that we needed a client-server way to talk to Speed Tables that was highly compatible with accessing Speed Tables natively.

The simplicity and uniformity of the speed tables interface and the rigorous use of key-value pairs as arguments to search (requiring values in all cases) made it possible to implement a Speed Tables client and server in around 500 lines of Tcl code.

This implementation provides identical behavior for client-server speed tables as direct speed tables for *most speed table methods*.

Stored Procedures

There is a Tcl interpreter on the server side, pointing to the possibility of deploying server-side code to interact with Speed Tables[7], although there isn't any formal mechanism for creating and loading server-side code at this time.

---

7 Fairly analogous to stored procedures in a SQL database, Tcl code running on the server's interpreter could perform multiple speed table actions in one invocation, reducing client/server communications overhead and any delays associated with it.

Speed Tables' *register* method appears to be a natural fit for implementing an interface to row-oriented server-side code invoked from a client.

Speed Tables can be operated in safe interpreters if desired, as one part of a solution for running server-side code, should you choose to take it on.

Dedicated Speed Table Servers

Once you start considering using Speed Tables as a way to cache tens of millions of rows of data across many tables, if the application is large enough, you may want to consider having machines basically serve as dedicated Speed Table servers.

Take generic machines and stuff them with the max amount of RAM at your appropriate density/price threshold. Boot up your favorite Linux or BSD off of a small hard drive, thumb drive, or from the network. Start up your Speed Tables server processes, load them up with data, and start serving speed tables at far higher performance that traditional SQL databases. This is a lot stronger than *memcached* because memcached is basically just a directory of files. Here, at the very least, you can define fields that contain metadata and use search to look stuff up.

Speed Table URLs

```
sttp://foo.com/bar
sttp://foo.com:2345/bar
sttp://foo.com/bar/snap
sttp://foo.com:1234/bar/snap
stty://foo.com/bar?moreExtraStuff=
sure
```

The default speed table client/server port is 11111. It can be overridden as above. There's a host name, an optional port, an optional directory, a table name, and optional extra stuff. Currently the optional directory and optional extra stuff are parsed, but ignored.

Example Client Code

```
package require speedtable_client

remote_speedtable \
speedtable://127.0.0.1/dumbData t

t search -sort -coolness -limit 5
-key key -array_get_with_nulls
data -code {
     puts "$key -> $data"
}
```

Example Server Code

When registering a table on the server side, use a wildcard for the host:

```
package require speedtable_server
# create a ctable 't' here.
::speedtable_server::register \
speedtable://*/dumbData t
```

That's all there is to it. You have to allow the Tcl event loop to run, either by doing a *vwait* or by periodically calling *update* if your application is not event-loop driven, but as long as you do so, your app will be able to server out speedtables.

Performance

Performance of client-server speed tables is necessarily slower than that of native, local speed tables. Network round-trips and the Tcl interpreter being involved on both the client and server side for every method invoked on a remote speed table inevitably impacts performance.

That being said, a couple of techniques, batching and using searches in places of gets can have a dramatic

impact on client/server speed table performance.

Batching

Consider a case where you know you're going to set values in dozens to hundreds of rows in a table. You can batch up the sets into a single batch set command.

```
$remoteCtable set key1 var value
?var value...?
$remoteCtable set key2 var value
?var value...?
$remoteCtable set key3 var value
?var value...?

$remoteCtable batch {
    set key1 var value ?var
value...?
    set key2 var value ?var
value...?
    set key3 var value ?var
value...?
}
```

In the second example, all of the set commands are sent over in a single remote speed table command, processed as a single batch by the speed table server (with no Tcl interpreter involvement in processing on a per-command basis inside the batch). A list is returned comprising the results of all of the commands executed. (See the batch method for more details.)

Most speed table commands can be batched, except for the search methods (this is not checked, though, and the results are undefined). In particular, get, delete, and exists can be pretty useful.

Use "search" Instead of "get"

Another common use of speed tables is to retrieve values from rows in some kind of loop. Perhaps something like...

```
foreach key $listOfRows {
    set data [$t get $key]
    ...
}
```

In the above example, every "get" causes a network roundtrip to the speed table server handling that table. If we substitute a search for the above, we can get all the data for all the rows in a single roundtrip. The "in" compare method can be particularly useful for this...

```
$ctable search -compare {in key
$listOfRows} -array_get data {
    ....
}
```

Shared Memory Speed Tables

Client-server speed tables can take a fairly big performance hit, as a sizable amount of Tcl code gets executed to make the remote speed table behave like a local one.

While they're still pretty fast, server actions are inherently serialized because of the single-threaded access model afforded using standard Tcl fileevent actions within the Tcl event loop.

When the speed table resides on the same machine as the client, particularly in this era of relatively inexpensive multiprocessor systems, it would be valuable for a client to be able to access the speed table directly through shared memory, bypassing the client/server mechanism entirely.

Speed tables can use shared memory to accelerate concurrent access by multiple processes. The design objective was to provide a way for same-server clients to access the speed ta-

ble through shared memory while retaining the ability to build and use speed tables without using shared memory at all.

When a speed table is instantiated for use with shared memory, the entire table, all keys and indexes are stored in shared memory, and may be used when there is sufficient memory available.

Tricky synchronization issues surfaced quickly while development this. For instance, what should we do if a row gets changed or added while a search is being performed? We don't want to completely lock out access to the table during a search. Thus we have to really deal with database updates during searches, which raise referential integrity issues and garbage collection / dangling pointer issues. Many searches, such as ones involving results sorting, collecting a set of pointers to the rows that have matched. Those rows cannot be permitted to disappear behind search's back.

Also search tables were already in heavy production with tables containing tens of millions of rows. This work had to be rock solid or it wouldn't be usable.

To simplify the problem, we decided to funnel writes through the client/server mechanism and only allows reads and searches to occur through shared memory. In many cases all changes are handled by a single process anyway, and no updates need be sent from clients.

We take advantage of the skiplist code's ability to support lockless synchronization between processes sharing memory. Our approach is to maintain metadata about in-progress searches in shared memory and have a cycle number that increases as the database is updated. When a search begins, the client copies the current cycle number to a word in shared memory allocated for it by the server. As normal activity causes rows to be modified, updated, or deleted by the server, the cycle number they were modified on is stored in the row. If rows (or any other shared memory object, such as strings) are deleted, they are added to a garbage pool along with the current cycle, but not actually freed for reuse until the server garbage collects them on a later cycle.

If the client detects that a row it's examining has been modified since it started its search, it restarts the search operation. The server makes sure to update pointers within shared memory in an order such that the client will never step into a partially modified structure. This allows the whole operation to proceed without explicit locks, so long as pointer and cycle updates are atomic and ordered.

Garbage collection is performed by locating deleted memory elements that have a cycle number is lower than the cycle number of any client currently performing a search.

To use shared memory support, a new parameter was added to the "create" command, specifying that a table was shared as a *master* or a *reader*. This parameter is followed by a list of options that describe the size of the table and how it is built.

STAPI - the Speed Table API

STAPI creates the speedtables API, which is used for a variety of table-like

objects. This includes remote speed tables through ctable_server and SQL databases. There are two main sets of routines in STAPI, and they're not normally used together.

- *st_server*, a set of routines for automatically creating a speed table from an SQL table as a local cache for the table, or as a workspace to be used for preparing rows to be inserted into the table. It's normally used in a ctable_server task providing a local read-only cache for a remote database for many client processes.
- *st_client*, which provides the general interface for creating STAPI objects identified by URIs.

  The primary mechanism for using STAPI as a client is through **::stapi::connect**, which connects to a speed table server or other database providing a speed table interface via a URI.

  ```
  ::stapi::connect uri ?-name value...?
  ```

Only one option is normally required:

```
-key  col
```

Define the column used to generate the key.

If a key is not provided, some STAPI capabilities may not be available.

```
::stapi::register method \
transport_handler
```

register a transport method for ::stapi::connect

At this time the following methods have been defined in STAPI:

Using a ctable server via sttp (client/server)

`package require st_client`

sttp://[host:port]/[dir/]table[/stuff][?stuff]

Using a ctable server via sttp (shared memory)

`package require st_shared`

shared://port/[dir/]table[/stuff][?stuff]

Access a speed table server on localhost, using shared memory for the "*search*" method and the client-server speed table transfer protocol for all other methods.

An additional option is used:

```
-build directory
```

The ctable built by the server must be in auto_path, or in the directory defined by the "-build" option.

Using a PostgreSQL database directly

`package require st_client_pgtcl`

sql://connection/table[/col[:type]/col...][?param&param...]

Create a stapi interface directly to a PostgreSQL table

The connection part has not been implemented yet. It will be something like **[user[:password]]@[host:]database**

If no columns are listed, all columns of the table will be returned.

Parameters are name=value style, just like in HTTP. A key column should be defined. Pseudo-columns can be defined here, too, with parameters like **`-column=sql_code`** or **`_key=column_name.`**

Using an already opened speed table

`package require st_client`

If the URI is not URI format, it assumes it's an object that provides stapi semantics already... typically a ctable, an already-opened ctable_client connection, or the result of a previous call to ::stapi::connect. It queries the object using the `methods` command, and if necessary creates a wrapper around the ctable to implement the extra methods that STTP provides.

STDisplay - Display Functions for the web

STDisplay is derived from Rivet's DIODisplay, and the calling sequence is similar. For example:

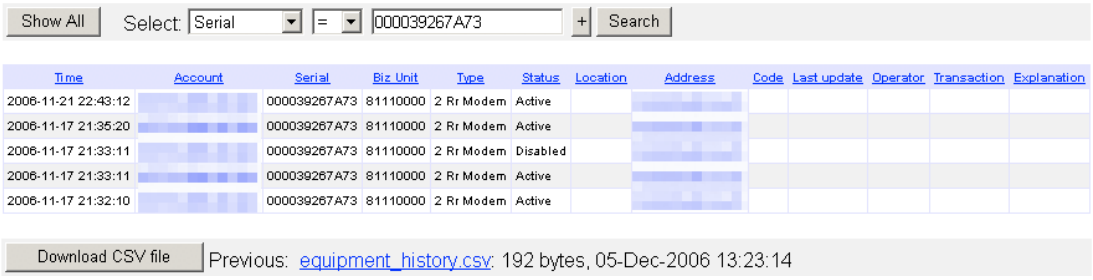
```
set display [
 ::STDisplay #auto -uri \
  sql:///history?_key=time
]

$display field time
$display field account
$display field serial
…
$display field explanation
$display show
```

Since STDisplay works with anything that can be exposed in the Speed Table API, it's an efficient and convenient mechanism to browse many kinds of database and database-like tables on the web.

Speedtable C-level implementation

You can interact with any speed table, regardless of its composition, from C, by making standardized calls via the speed table's methods (pointers to functions) and speed table's creator table structures.

Varstrings are *char* * pointers and a length. We allocate the space for whatever string is stored and store the address of that allocated space. We avoid malloc/frees when possible by reusing the space when values change and the string being store fits. Default values are represented with a



null pointer., while fixed-length strings are generated inline.

The null field bits and booleans are all generated together and should be stored efficiently by the compiler. We rely on the C compiler to do the right thing with regards to word-aligning fields as needed for efficiency.

You can examine the C code generated -- it's quite readable. If you didn't know better, you might think it was written by a person rather than a program.

How we invoke the compiler can be found in **gentable.tcl**. We currently only support FreeBSD and Mac OS X, and a general solution will likely involve producing a GNU configure.in script and running autoconf, configure, etc. We'd really appreciate some help on this.

License

The Speed Tables package is distributed under the same permissive Berkeley license that Tcl uses.

Obtaining Speed Tables

A SourceForge project has been requested for speed tables and should be available by September 20th, 2007.

http://sourceforge.net/projects/speedtables

Included with speed tables is a 65 page developer's manual. A test suite includes dozens of tests.

Request For Participation

Speed tables are working well, for the authors as least. An autoconf-style configuration system would be of great value. We'd also like a mechanism for defining C-based search comparison routines, and someone to use and construct examples for interfacing to C directly.

Speed tables could be a useful addition to other scripting languages as well, and of course additional documentation would always oblige, as well as manual translations into other langauges, testing for different locales, and so forth.

Summary

Speed tables powerfully extends Tcl's ability to access and manipulate data, providing unique capabilities, far higher performance, and greater memory density over traditional Tcl-based approaches. Its permissive license makes it feasible for use in a wide range of projects that need higher performance storage and searching than available from a SQL database or from ad hoc methods.

GEB: SQLite in Tcl/Tk in SQLite

Abstract

GEB is a Tcl/Tk program for displaying and manipulating SQLite databases. Each of its major functions is stored in an SQLite table. It has much of the functionality of the SQLite stand-alone executable, plus spreadsheet-like table display, nearly complete ALTER TABLE functions, SQLite version 2 to version 3 conversion, the ability to execute a table as either SQL or Tcl, and a few other functions I had a need for. The name GEB was chosen because using an SQLite database to store the program which displays and modifies the SQLite database itself seemed reminiscent of the "self reference at a higher level" which was a recurring theme in the book "Gödel Escher Bach: An Eternal Golden Braid." If you insist on interpreting it as an acronym, it could stand for "Gerry's Experimental Box," but that is really a backronym.

This paper contains a brief background and history of GEB, a discussion of its current capabilities, and a list of some possible future additions.

Background

GEB began with a desire to migrate some smallish database projects from proprietary s/w (MS Access and dBase n) to FOSS. SQLite caught my eye. Neither the command-line program (CLP) nor any of the free packages for it seemed to do everything I wanted, so I decided to roll my own. My project would have to run on Linux and Windows, and I was looking at both Perl and Tcl, with the former having the edge because of its being used a lot at work. I installed both language packages on both platforms. Tcl and SQLite worked perfectly on both systems. I had problems with the perl package on one of the platforms (no idea which one by now), so I went with Tcl.

My goal was to put all the functionality I needed into my package, so I wouldn't have to switch over to the stand-alone executable or some other program for some step in the middle of a process I was undertaking.

I have tried to be extremely cautious about data integrity. Algorithms have been kept as simple as possible, to make them easier to verify. In some cases where GEB generates SQL, the user is shown the SQL and is given the chance to decline to accept it.

History (showsqlite)

The first functions I needed were displaying the tables in a file and their fields, and the contents of a table. Figure 1 shows my first version, which used text widgets. The upper window shows one table per line, with the number of entries in blue, the table name in black, and the column names in red. Clicking on a table name lists the contents in the lower window.

The display format was not appropriate for editing the table contents, so a separate editing window was created, shown in Figure 2. It was opened using the Data – Edit menu.

Since I was always thinking of more columns I needed to add to some tables, an Alter Table function was needed. This window, shown in Figure 3, was brought up by clicking on the column names of a table. Columns could be added or deleted, changed in order, renamed, or copied. All changes were made by copying to a TEMP table (possibly with changes), deleting the original table, copying tot the original name (possibly with changes), and deleting the TEMP table. Since some kinds of changes are

incompatible with other kinds (for a single pass through the alteration), some functions are disabled and their buttons grayed out when others are started. If two "incompatible" functions are needed, the user is required to make two passes through the process. The generated SQL is shown, and the user can cancel without making the changes. The example in the figure shows the result of some Move Up/Down operations. The Add, Rename, and Copy operations are incompatible and their keys have been grayed out, but Delete is still available. The SQL listing shows everything but the Commit Transaction, which is not sent until the user accepts the action.

The main window displays, and the data edit window were clunky, even by my standards, so my first upgrade after basic functionality was achieved was to switch to tkTable-based display.

Upgraded Display (showtable)

The original impetus for using tkTable was looks—getting a spreadsheet-like display. The expected side-effect was getting a much nicer format for editing and entering data. An unexpected one pleasantly took care of a potential problem with large tables. The original text-based display was populated by reading the entire table into memory and inserting it into the text widget. This worked fine with the data sets I was using, so I never got around to coding something better, but it would not have scaled well. The tkTable widget, in the command data mode, calls a user-specified proc for any cell it needs to display (or update), so the size of the table is of little importance to the display speed. I have thought a little about speeding things up by reading the whole row when the first column is asked for, and then filling in the rest of the row from cached values, but I have not been able to convince myself that some race condition could not result in an updated value being missed—and besides, the current speed is acceptable. The main window now is as shown in Figure 4, which has the standard spreadsheet-like look.

Clicking on a table name now brings up a window like Figure 5, which not only looks better, but also allows updating and adding data.

When updating data, the default is to require confirmation for each cell changed, but there are options for read-only (no changes allowed) or making the change immediately, without confirmation. Also, the capability to extend the number of rows in the table can be set to Never, Confirmation required, or Always done. Finally, selected rows can be deleted.

Stand-alone Functions

This section covers capabilities of GEB that do not interact (at least very much) with the basic display and edit windows that have been discussed so far. They were added over an extended period of time, but will be covered together.

Execute SQL/Tcl

There is often a need to execute a single line of SQL or Tcl, so I created windows for those functions. The result is made a little more readable by allowing the user to specify the number of items per line. Figure 6 shows an example.

Run a Table

More than once I wrote and debugged some Tcl code to process some data in an SQLite file, and then could not find the code when I needed to run it again. I hate doing things twice. A lot. So i started adding tables to the file with a column named text for storing the code snippets.

Naturally, copying the data from these tables and pasting it into one of the Execute windows got old real fast, and so the ability to edit and execute the code was added. I was really worried about the code interfering with GEB, so I experimented with executing the Tcl code in either a safe interpreter or a special namespace. I also included the ability to execute SQL code. None of these special capabilities turned out to be very useful, and this capability has not been updated, but the lessons learned led to the Tcl storing and execution that made it possible to use the database file for storing the program itself.

Import/Export

The CLP falls short of my import/export needs in a few ways. First, in some cases my data has the desires column names as the first line, and in other cases it doesn't. Including the names is an option for both import and export, as is picking the delimiter. Second, in one data set all lines contained the “essential” columns but some also had some “optional” data. Therefore the import routine optionally allows some lines to be missing columns. It would have been easy enough to handle the data manually, but why spend a few minutes doing menial work if you can avoid it with an hour or two of programming?

There was also a separate capability to import a single cell from a text file, or export it, similar to the onecolumn method, but not using it.

Convert between Version 2 and Version 3

This work started during the SQLite Version 2 days, and for a long time I refrained from using any SQLite capabilities restricted to Version 3 so that my code would work with either file format, but I eventually gave in, because of V3's ability to do its own variable substitution.

Anyhow, functions were added to convert from V2 to V3 (because it was needed), and from V3 to V2 (for completeness, and to simplify checking.

Putting the Program in the Database File

When I read DRH's paper “SQLite and Tcl” I was intrigued by the concept of storing the whole program in the database file. I was not yet familiar with Tcl's handling of unknown procs, but the wiki showed how to use and extend that. I had, or at least thought I had, all the other pieces, so the basics were soon cobbled together.

I did not exactly implement DRH's writeup, since I wanted to reduce the number of tables. Therefore I grouped the supporting procs with the main proc for each function and put them in a single table with the name of the main proc.

This had two impacts: There could not be separate columns for the header and body of a proc, and the support procs could not be called from a proc stored in a different table, unless it was known to already have been loaded. The grouping of the procs was done properly, and so the second theoretical impact has never been a problem.

Two unexpected problems did arise, though.

I had never needed to work on more than one file at a time, so I had not implemented ATTACH capabilities. The program could look at just one file, and that file was now the one containing the program itself. No place for the file with the data. A significant rewrite was needed to handle attached files, with a bunch of one-dimensional arrays describing the database schema needing to be redone as two-dimensional arrays, with the attach name as the new index.

Secondly, the simple editing support the program had was no longer enough. I could no

longer do a global search for the name of a proc or variable and see every place it was used. So the scope of the search routine was expanded to cover all tables with a column named tcl.

Figure 7 shows the main editruntcl window, and Figure 8 the search window. The two are tightly integrated, with edits in one making the appropriate changes in the other. In fact, automatic propagation of changes is something I worked hard to achieve, since the display or use of obsolete data could lead to database corruption. The one deliberate exception is the tk_optionMenu widget in the editruntcl window, whose table list must be reset manually by clicking on the button next to it. This was because updating the list affects the display, and I wanted that under user control.

Program Organization

One goal was to put as much of the startup process in the database, so that the external bootstrap would be as small as possible. As can be seen in Figure 9, that effort met with success. The last two lines are optional, and one or the other is usually commented out, depending on the work being done. For program development the editruntcl line is left uncommented, so the main editing window will come up automatically. Otherwise an attachit line brings up a display of the desired data file.

The main_attach routine controls most of the startup process. It must read in and execute two procs to set up the array of table names needed by the extended unknown handler. Refactoring the functionality could reduce the number of procs needed to one, but such strictly esthetic improvements are low priority compared to improving functionality.

Extending the capability of the unknown handler is based on http://wiki.tcl.tk/2776. Whenever an unknown proc name is encountered, the ::GEB::tcltablelist(main) list is searched for that name preceded by main. (so that currently attached files are not used for auto loading). If such a table name is found, the table is loaded. If reading in the table defines a proc with the desired name, that new proc is executed with the appropriate arguments and its result is returned. If either of these conditions is not met the original processor for unknown procs is executed.

All other functions are initiated with menus or buttons.

Future Plans

The development process so far has been so pleasant that I look forward to adding further functionality. In the past it has often taken longer to decided what I wanted to do and how I wanted it to work than it took to implement it, and I fully expect that to be as true in the future.

Two things I am planning to implement soon are allowing wildcards (of the [string match] kind to searches, and some sort of version control, at least to the point of executing a stable version while editing and testing a development version of routines. Further down the road I may look at a GUIfied way of doing Full Text Searches, and setting up tkTable displays of views.

Potential Users and Licensing

GEB is a set of Tcl routines for displaying and manipulating SQLite databases, stored in an SQLite database. As such, it could conceivable appeal to a wide range of users.

At one extreme would be a Tcl developer who didn't care about SQLite. GEB could be used by such a developer, but other environments are more powerful in many ways.

At the other extreme is an SQLite user who doesn't care about Tcl and doesn't want to learn it. This use seems like a bit better fit, and there is less competition, but most would still find other programs more to their liking.

GEB is most likely to appeal to those in the middle: those with some SQLite data and some Tcl scripts for processing the data, who want to keep, use, and maintain them together.

I intend to release GEB to the public domain, since it would feel wrong for my contribution to have a more restrictive license than SQLite itself. Therefore, GEB may be used for any purpose whatsoever. Development is encouraged, and especially if the new code is also public domain.

Support is available—inquire within.

Gerald C. Snyder
mesmerizerfan@gmail.com

Figures

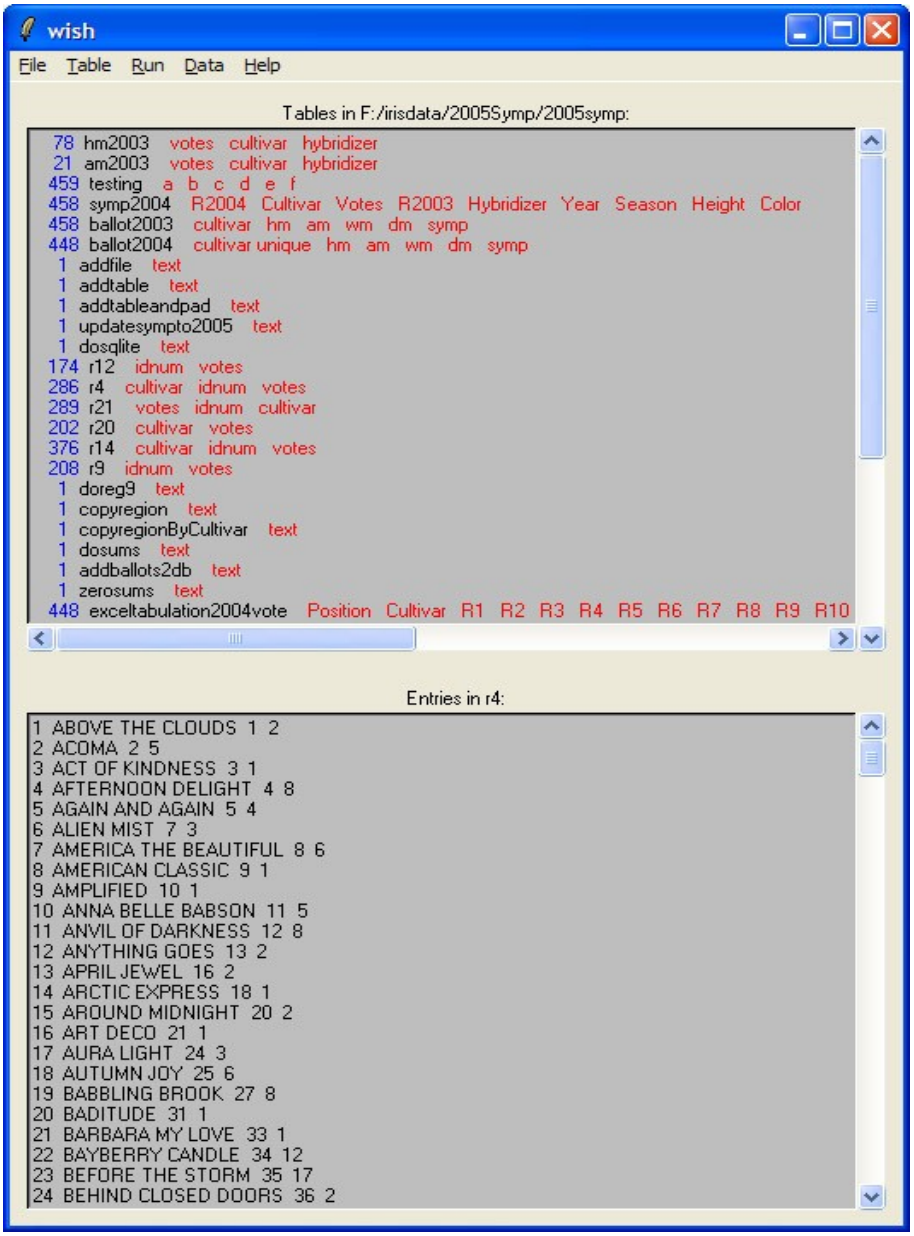


Figure 1. Pre-tkTable Main Display Window

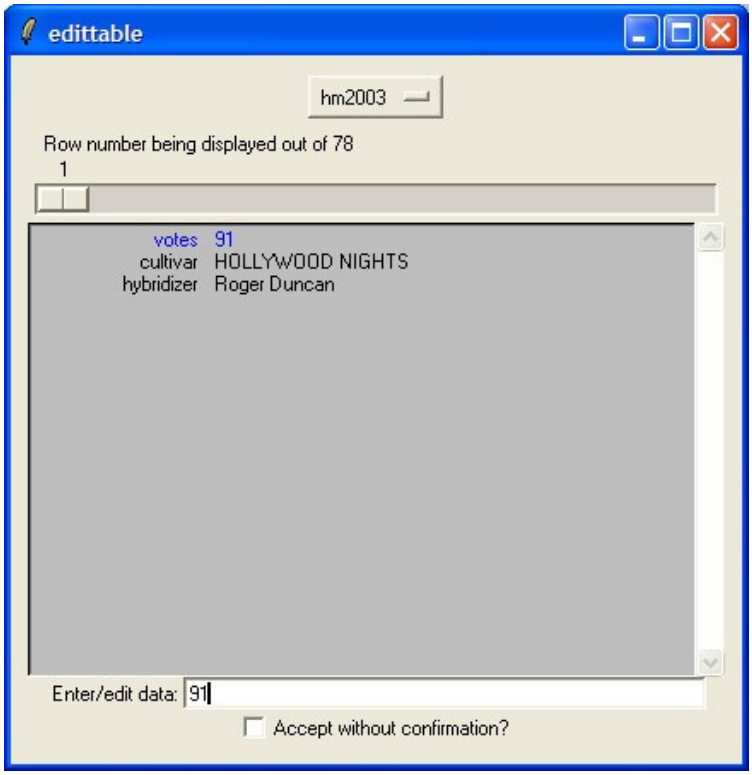


Figure 2. Pre-tkTable Edit Window

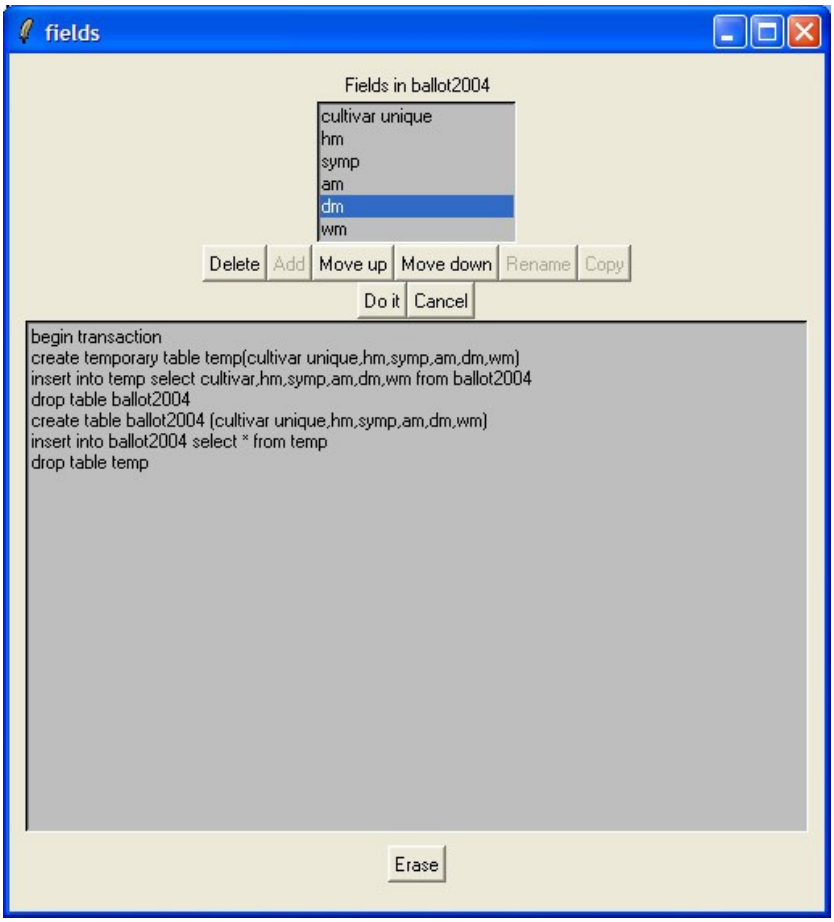


Figure 3. Alter Table Window

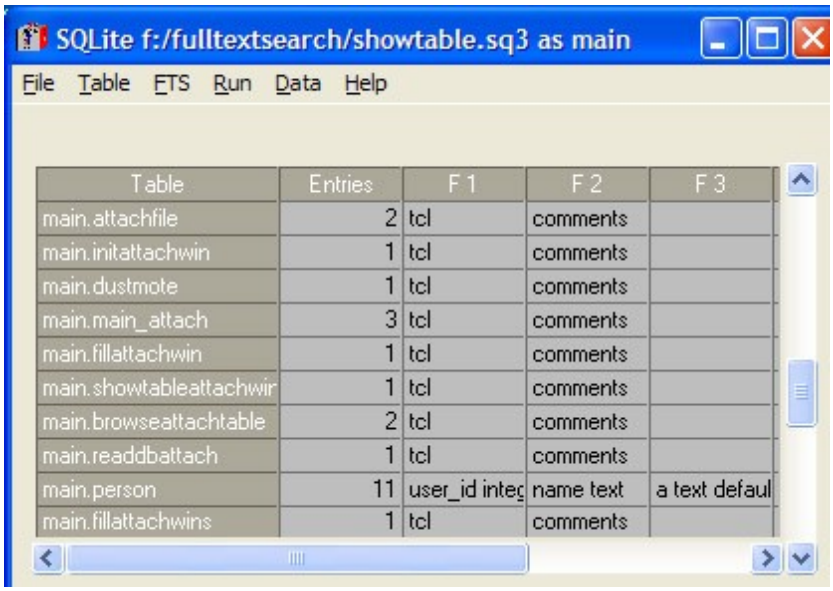


Figure 4. Main Window with tkTable

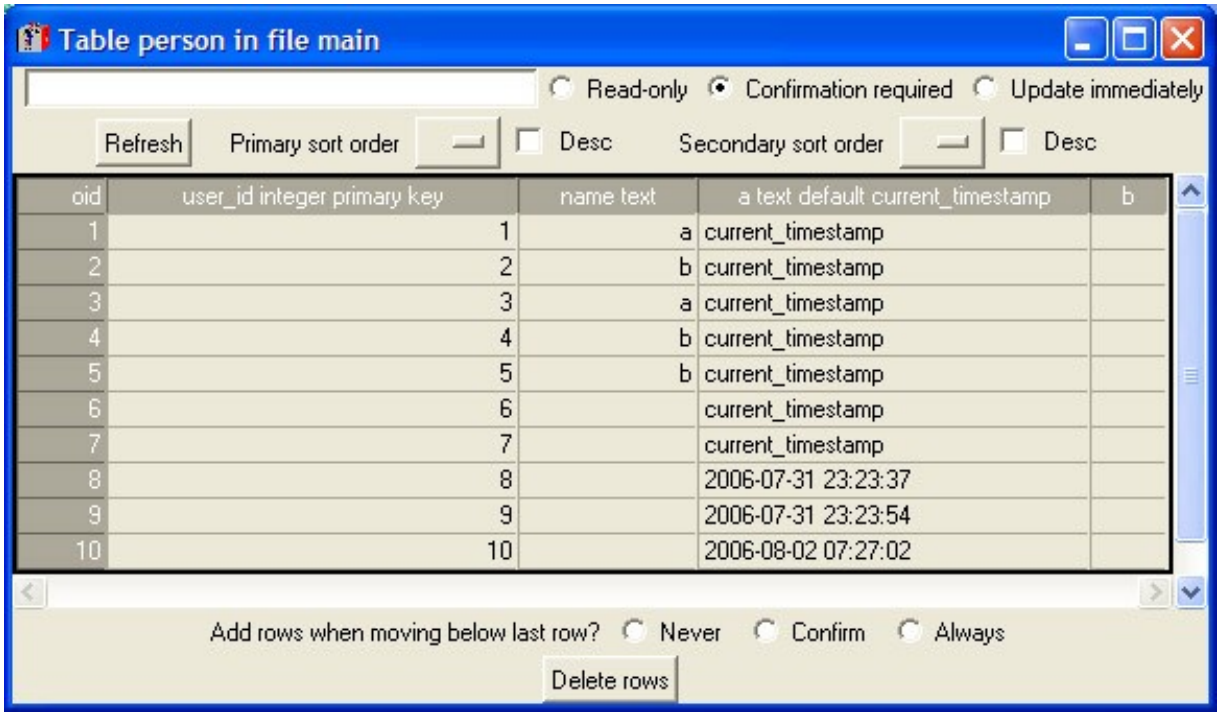


Figure 5. Table Display with tkTable

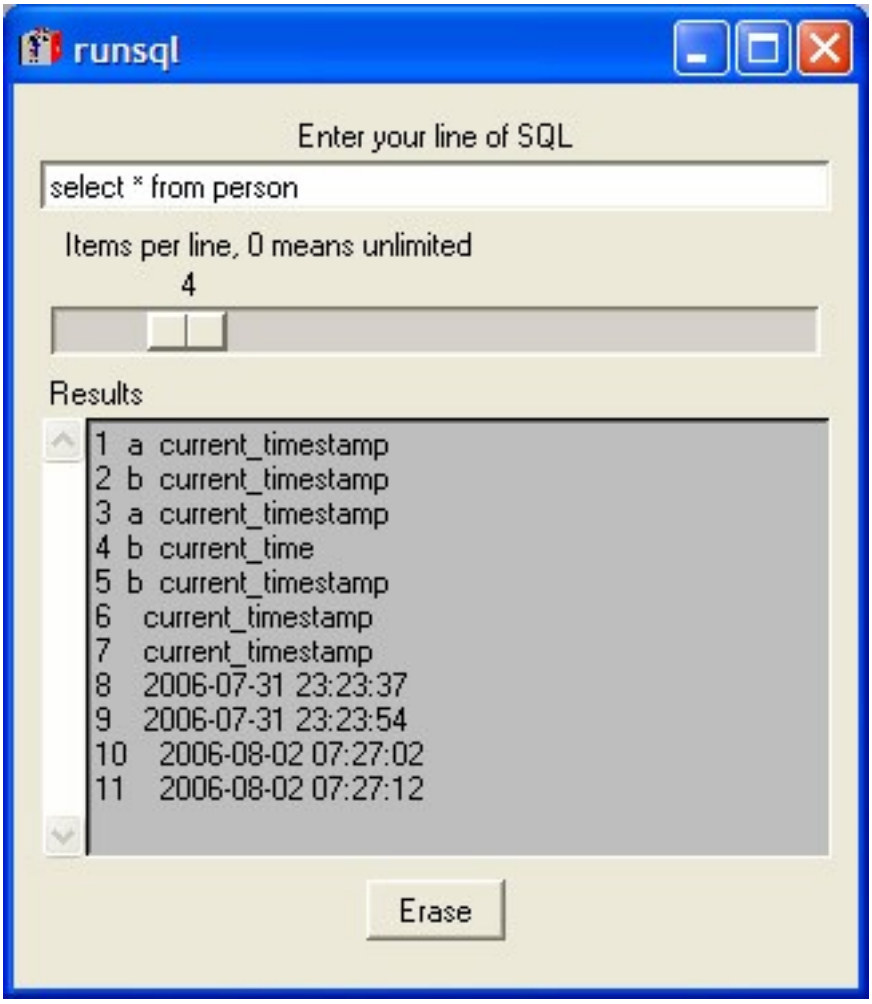


Figure 6. Window for runsql

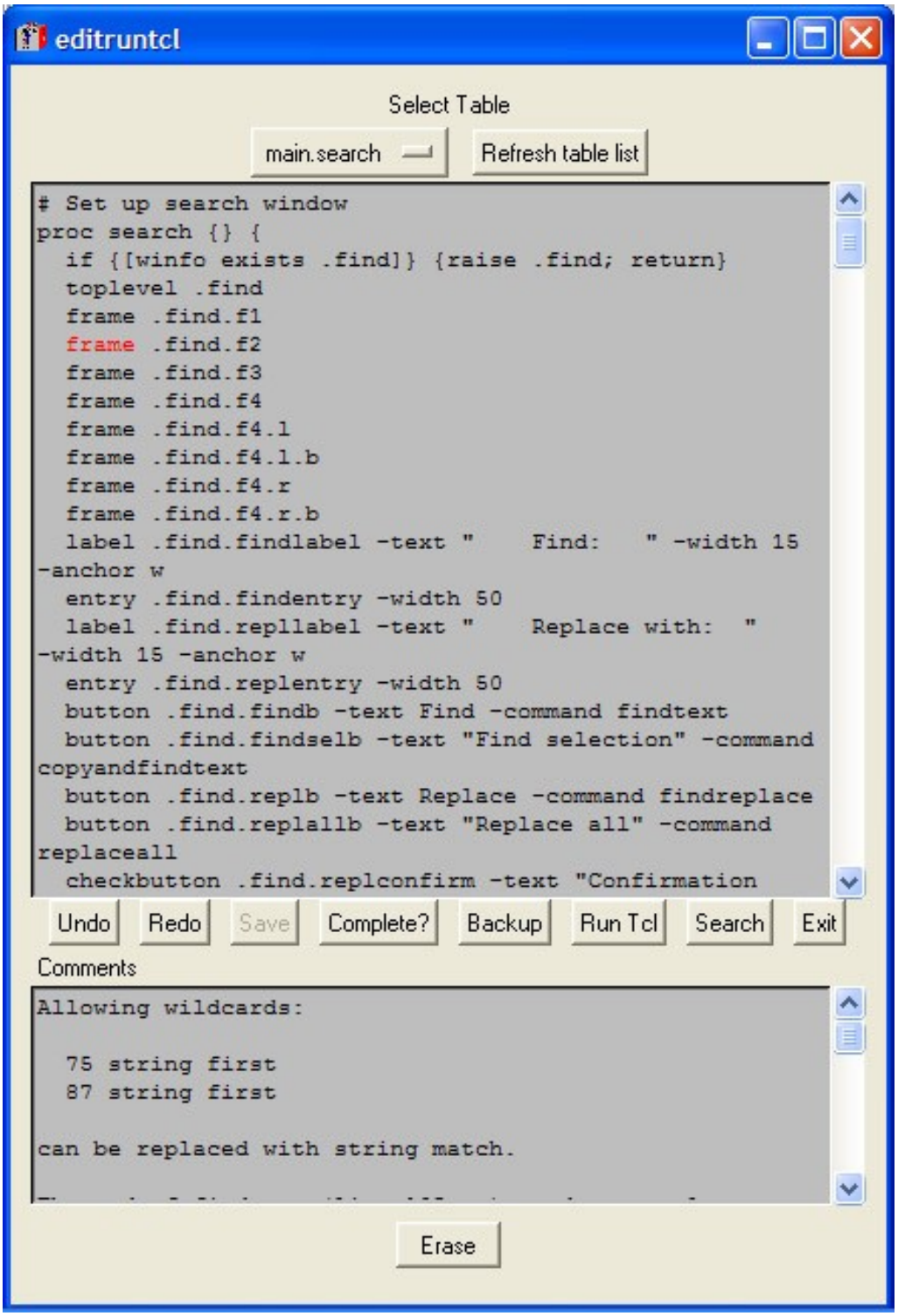


Figure 7. Window for editruntcl

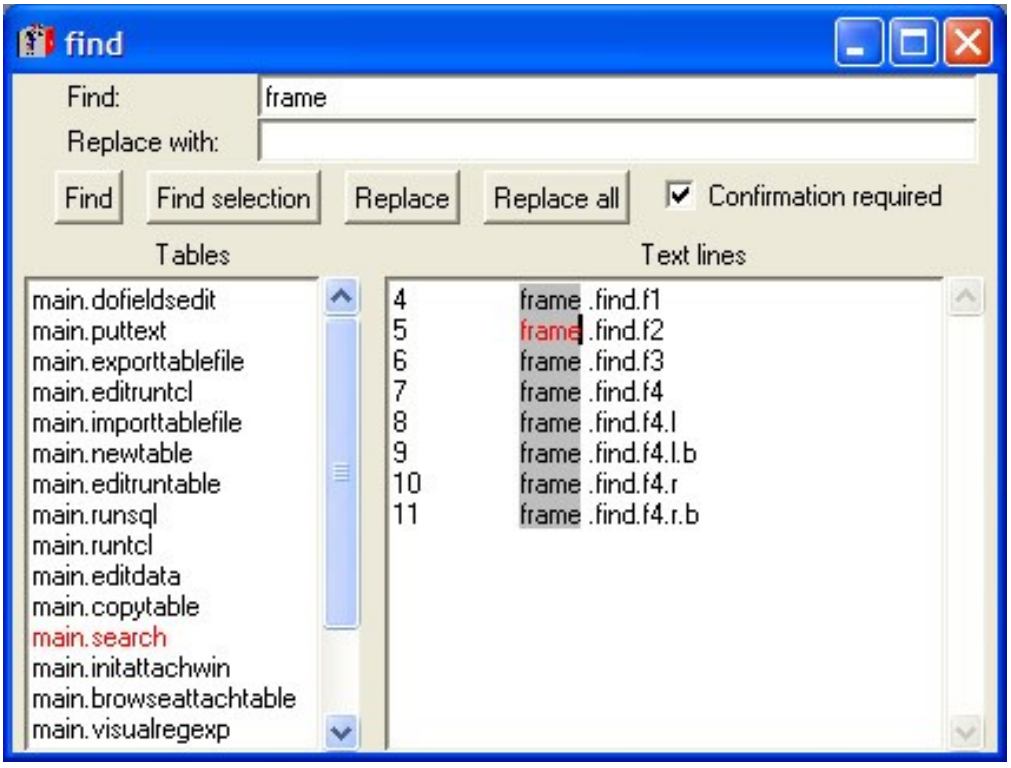


Figure 8. Search Window

```
package require Tk; package require Tktable; console show
# Create namespace for all "globals" and procs
namespace eval ::GEB {}
 load /sqlite/tclsqlite3.dll
 load /sqlite/tclsqlite.dll
set dbfile showtable.sq3
 sqlite3 sq $dbfile
 set ::GEB::attachfilename(main) $dbfile
proc evalsqlitetcl table {
  uplevel #0 [join [sq eval "select tcl from $table limit 1"]]
}
evalsqlitetcl main_attach
editruntcl
# attachit s2008 /irisdata/2008symp/2008symp.sq3
```

Figure 9. GEB Bootstrap

14’th Annual Tcl/Tk conference
Sept. 24-28 2007
Bourbon Orleans Hotel, New Orleans, LA USA

Tcl Does Web

Using and Providing Web Services in Tcl

Gerald W. Lester
TicketSwitch USA, LLC

Overview

- What are Web Services
- Available Packages
- Traditional Approaches
- Our Approach
- Examples
- Conclusion

What are Web Services

- W3C
  - XML Bassed
  - SOAP based
    - Web Services Description Language (WSDL)
- Other definitions exist
  - RESTful

Available Packages

- WebServices for Tcl
  - Server
    - TclHttpd
  - Client
    - Any Tcl Application
- http://members.cox.net/~gerald.lester/WebServicesForTcl.html

Available Packages

- Server
  - Tcl Web Services Toolkit (TWIST)
    - AOLserver
- Client
  - None yet, future goal
- http://code.google.com/p/twsdl/

Traditional Approaches

- Define WSDL by hand
- Generate documentation for service by hand
- Use tool to generate server abstract class definition
    - Handle your own XML
- Use tool to generate client stubs
    - Handle your own XML

Our Approach

- No knowledge of XML is required
  - Dictionaries used for data structures
- Attempt to make as much in the Tcl spirit as possible
  - Typing and constraints not enforced

Our Approach - Server Side

- Server generates WSDL and "man/help" page from definition of procedure and data structures
- Web Services are strongly typed
  - Uses literate programming style

Our Approach - Client Side

- Client side parses WSDL
  - Optionally generates stubs
  - Parsed WSDL can be saved to a file and reloaded
    - For when WSDL is not accessible
    - More efficient

Packages Used

- tDOM
- http
- log
- uri
- html
- dict

Current Status

- Version 1.x
  - Man/Help pages
- Tutorial (thanks Bryan Oakley)
  - http://www.tclscripting.com/articles/nov06/article1.html

Example

- Echo service
  - Two methods
    - SimpleEcho
      - Input: String to echo
      - Returns: String
- ComplexEcho
  - Inputs: String to echo
  - Returns:
    - Date/time
    - String

Client Example

```
package require WS::Client
##
## Get Definition of the offered services
##
::WS::Client::GetAndParseWsdl http://localhost:8015/service/wsExamples/wsdl

set testString "This is a test"
set inputs [list TestString $testString]
```

Client Example - Synchronous

```
puts stdout "Calling SimpleEcho via DoCalls!"
set results [::WS::Client::DoCall wsExamples SimpleEcho $inputs]
puts stdout "\t Received: {$results}"

puts stdout "Calling ComplexEcho via DoCalls!"
set results [::WS::Client::DoCall wsExamples ComplexEcho $inputs]
puts stdout "\t Received: {$results}"
```

Client Example - Stubs

```
##
## Generate stubs and use them for the calls
##
::WS::Client::CreateStubs wsExamples
puts stdout "Calling SimpleEcho via Stubs!"
set results [::wsExamples::SimpleEcho $testString]
puts stdout "\t Received: {$results}"
puts stdout "Calling ComplexEcho via Stubs!"
set results [::wsExamples::ComplexEcho $testString]
puts stdout "\t Received: {$results}"
```

Client Example - Asynchronous

```
##
## Define asynchronously callback routines
##
proc success {service operation result} {
    global waitVar
    puts stdout "A call to $operation of $service was successful and returned $result"
    set waitVar 1
}
proc hadError {service operation errorCode errorInfo} {
    global waitVar
    puts stdout "A call to $operation of $service was failed with {$errorCode}
{$errorInfo}"
    set waitVar 1
}
```

Client Example - Asynchronous

```
##
## Call asynchronously
##
set waitVar 0
puts stdout "Calling SimpleEcho via DoAsyncCall!"
::WS::Client::DoCall wsExamples SimpleEcho $inputs \
        [list success wsExamples SimpleEcho] \
        [list hadError wsExamples SimpleEcho]
vwait waitVar
puts stdout "Calling ComplexEcho via DoAsyncCall!"
::WS::Client::DoCall wsExamples ComplexEcho $inputs \
        [list success wsExamples SimpleEcho] \
        [list hadError wsExamples SimpleEcho]
vwait waitVarå
```

Client Example - Google API

```
package require WS::Client
package require dict

::WS::Client::GetAndParseWsdl http://api.google.com/GoogleSearch.wsdl

dict set args key "<your google license key here>"
dict set args q {site:tclscripting.com font}
dict set args start 0
dict set args maxResults 10
dict set args filter true
dict set args restrict {}
dict set args safeSearch false
dict set args lr {}
dict set args ie latin1
dict set args oe latin1

set result [::WS::Client::DoCall GoogleSearchService doGoogleSearch $args]

foreach item [dict get $result return resultElements item] {
    puts [dict get $item title]
    puts [dict get $item URL]
    puts ""
}
```

Server Example

```
package require WS::Server
package require WS::Utils

##
## Define the service
##
::WS::Server::Service \
    -service wsEchoExample \
    -description  {Echo Example - Tcl Web Services} \
    -host         $::Config(host):$::Config(port)
```

Defining Schema

```
##
## Define any special types
##
::WS::Utils::ServiceTypeDef Server wsEchoExample echoReply {
    echoBack      {type string}
    echoTS        {type dateTime}
}
```

Defining Operations

```
##
## Define the operations available
##
::WS::Server::ServiceProc \
    wsEchoExample \
    {SimpleEcho {type string comment {Requested Echo}}} \
    {
        TestString      {type string comment {The text to echo back}}
    } \
    {Echo a string back} {

    return [list SimpleEchoResult $TestString]
}

::WS::Server::ServiceProc \
    wsEchoExample \
    {ComplexEcho {type echoReply comment {Requested Echo -- text and timestamp}}} \
    {
        TestString      {type string comment {The text to echo back}}
    } \
    {Echo a string and a timestamp back} {

    set timeStamp [clock format [clock seconds] -format {%Y-%m-%dT%H:%M:%SZ} -gmt yes]
    return [list ComplexEchoResult [list echoBack $TestString echoTS $timeStamp]  ]
}
```

Demo of Auto-generated Pages

- Service Documentation
- WSDL

Known Problems

- Some “legal” XSchema very hard to parse
  - .NET generated with some tools
- Some Server implements expect certain “hardcoded” namespace prefixes instead.

Workarounds

- For most “difficult” XSchema minor edits allow it to be parsed
- No good workaround for “hardcoded” namespace problems

Conclusion

- Provides an easy way to provide Web Services from Tcl.
- Useable to call a good number of Web Services
  - May require minor modifications to WSDL
- Consider a work in progress

Questions?

14’th Annual Tcl/Tk conference
Sept. 24-28 2007
Bourbon Orleans Hotel, New Orleans, LA USA

Programming Techniques

OO for Tcl

or "How I Learned to Stop Worrying and Write the Code"

Donal Fellows
<donal.k.fellows@manchester.ac.uk>

For the past two years, I have been working on developing a new OO system for Tcl that is intended to serve as a basis for a wide range of OO styles. In this paper, I will describe and explain the current status of the work, discuss the issues involved in producing a high-performance flexible OO system, and describe a number of issues that have been encountered during work (with Arnulf Wiedemann) to build a version of [incr Tcl] on top of the core OO system.

As many of you know, I have been writing an object system for Tcl for a couple of years now. The intention was that this object system should focus on just the core task of making a fast method dispatch system as well as seeding the very heart of the inheritance hierarchy. This is in contrast to the other major object systems (e.g., [incr Tcl], XOTcl, Snit) that provide a much larger set of features, but at the cost of being far more complex. By sticking to the fundamentals, my code will be well placed to focus on how to be fast and readable, allowing the other OO systems to focus on "added value" such as collection management systems, rich application support, etc. Like that, it would allow us to have the power of the object-programming paradigm without enormous amounts of effort or tearing up the large number of existing scripts that depend on the features of the previously existing object systems.

This paper does not describe the detailed programming interface for the object system: I covered that previously[1]. Instead, it goes into more detail about the details of what makes for a fast and flexible object system. There has been a major change since I presented the initial proposal for this work two years ago though: after much discussion, I decided that any object system that goes into the core must have a substantial amount of practical "in-deployment" experience first. In order to gain this experience, I redesigned my object system to work as an extension package using the TEA build system. In addition, by doing this, I made it far easier for other people to work with the system during development. More eyeballs really do mean fewer bugs!

This has resulted in the TclOO extension, which you can use with any sufficiently recent version of an 8.5 core (i.e., after the sixth alpha release). It has documentation and a test suite, and I know that it builds and works correctly on both Windows and Linux. It even exports its own API via a stubs table, making it even easier to build your own extensions on top. There are down-sides to this though: how they have been dealt with is one of the topics of this paper.

[1] See my paper in the Tcl 2005 conference, or TIP #257, which was derived from it.

Flexibility and the Art of Code Writing

One of the major driving requirements of the TclOO package has been that it should be possible for third party code to extend it, and in as many different ways as possible. Thus, you can define not just new methods, but new *kinds* of methods and (currently experimental) new ways of invoking objects. However, doing this, especially for the long term, requires both the definition of structures (so that typing information can be provided in a sane fashion) and the rigorous hiding of the internal details of those structures that are private to the TclOO package itself.

The concealment of the internal details of private structures is relatively straightforward in practice: when those tokens even potentially pass outside the control of the package, I conceal their real types and they are just an abstract pointer[2]. I then provide a set of accessor functions to allow third-party code to extract the information within the real structures without exposing details that can change between versions.

Ensuring that public structures are future-proof is more complex. The structures that require this treatment are there to express the type of something, and instances of those structures will typically be compiled as constants in extension code. This binds the binary versions of those extensions inherently to the version of the API they use. Luckily, this problem has already been resolved in Tcl for structures such as the `Tcl_Filesystem` and `Tcl_ChannelType`, which would otherwise have the identical problem. These handle versioning by putting a version number directly into the structure: by knowing what version of the structure declaration the code was compiled against, the set of valid fields can be understood. This allows structures for purposes such as the definition of types of methods or metadata to be migrated into the future at minimal cost.

Another thing that has come from the TclOO work has been the way that some parts of the Tcl core are much easier to extend than before. For example, the Tcl `info` command is now an ensemble, as this allows the addition of new `info class` and `info object` subcommands in a simple fashion. The alternative would have been a special mechanism just for the `info` command itself, which would have required extensive testing instead of being just an application of a more general facility.

Inheriting Diamonds

One of the things that I wished to support was multiple inheritance, since that is a feature that is often very useful; e.g., a school bus is both a road vehicle and a passenger transportation device, and yet those superclasses are fundamentally distinct (compare with dump trucks and cruise ships!) And yet this opens up the way to a classic problem where you have a class that is a subclass of two other classes that define conflicting methods: the key to the problem being which method is “more important”? Since this can involve almost arbitrary amounts of additional confusing complexity, this problem is genuinely hard. (Arguably, this should not happen as method names should never clash like this, but method names model human language, and language is messy and imprecise.)

---

[2] In C, a pointer is abstract if it points to a type that the compiler does not know the size of. The classic example of an abstract pointer is `void*`, but pointers to a structure of unknown size are better in practice for many things, since they require an explicit cast to be converted to another type.

A study of the literature for dynamic object systems (static systems like C++ have other constraints that did not concern me) indicated that the best solution was to think in terms of first converting the inheritance graph (as viewed from a particular point) into a tree back to the object root, where any node may appear multiple times. Then you walk the tree "pre-order" to produce a traversal list, traversing the parents of each node in "natural" order (i.e., the order specified in the definition of the class). Finally, you remove every reference to any method on the list *except the last one*. The resulting list of methods (see Figure 1) turns out to be exactly what you want.

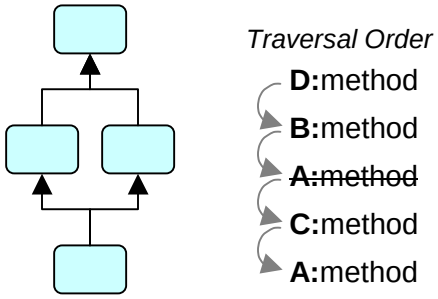


Figure 1: Diamond Inheritance Pattern

Well, almost. TclOO also supports mixins and filters, which add to the complexity. Mixins are classes that are added to objects; they are great for modelling roles and orthogonal behaviour, and come in the inheritance order before conventional classes (which model types better.) Filters are a way to decide whether to skip the evaluation of a method or perform some other kind of wrapping evaluation on a per-method basis, and are implemented as a list of method names to call before calling the real method. With both mixins and filters added, you have the model used by TclOO and XOTcl. (Other Tcl object systems typically have either simplifications of this model – [incr Tcl] is like this – or are done in a totally different way – the Self-modelled ones are in this category.)

Caching for Fun and Profit

The algorithm for calculating the method call chain described above is distinctly expensive, as you can imagine. The only way to get reasonable speed out of it is to be strict about using caching. And yes, TclOO uses caches a lot.

In particular, it caches method chains carefully so that the second time you call an object's method, the chain can be retrieved rapidly and the method dispatched in double-quick time. But care must be taken when doing such caches that the values retrieved from them are valid; if a class in the inheritance tree is modified, it can mean that all your assumptions about what the method chain looks like are wrong! Luckily, it turns out that it is easy to build a system for detecting potential problems that is also cheap. Just as with Tcl's bytecode engine, I use epoch counters. When an incompatible change happens, the appropriate epoch is updated – every object has its own epoch counter, but classes use a global one because they may be used outside themselves – and the code that retrieves values from the cache can just check two epochs (the object epoch and the global epoch) against the values saved when the chain was created. When both epochs match, the chain of implementations for that particular method name is correct and can be dispatched immediately.

Of course, if you have experience with the `Tcl_Obj` value system you might expect that the caches would be kept in the method name value itself. After all, that is exactly where Tcl's ensembles and functions like `Tcl_GetIndexFromObj` keep their caches. But this is not actually a safe thing to do, since we have per-object methods (and mixins and …) and without a strong classical object typing system I must keep the caches in the object itself and not the method name value. This is a significant difference between a subcommand dispatch scheme designed to support an ensemble and one for objects and their methods.

Getting [Incr]ementally Better

As mentioned earlier, one of the major aims of this work was to support the building of other object systems on top. This is a good thing to aim for since they have historically reached very deep into Tcl's innards in order to get speed, and that has left them inclined to be tightly bound to particular versions of Tcl. Not exactly the Stubs promise!

Instead, I have been working (with much prompting from Arnulf Wiedemann) on providing an API that allows these other extensions to build their style of methods on top of my core ones without having to pry deep inside my code. For the moment, this API is not public – I do not know yet whether the functions and structures involved are at all stable – but it is my intention to make it available. In particular, it allows for code to do things like adjusting the command resolution scheme specifically for the body of the method instead of by doing strange things with the overall command resolution system. This limits the effects and makes it easier to increase the performance. Other areas that have an internal extension point are the mechanisms for deciding how to implement a particular method call, for deciding the exact level of privacy enjoyed by a class method, and to allow classes to control the name of the namespaces of the objects they create.

The net result of this (and much work by Arnulf) is that it has proved possible to implement a new version of [incr Tcl] on top of the TclOO core and get it to sufficient quality where it passed the itcl test suite. One of the biggest gains is that this new version, currently known as itcl-ng, can do this without need to deal with direct allocation of structures that are in the Tcl core. This in turn makes it likely that future versions of itcl will be compliant with the broader Tcl Stubs promise: that a new minor version of Tcl will not force the rebuilding of extensions built against the old version.

Collecting Examples

But you would rather see code, right?

There are many features in the TclOO system that are of interest at the scripted level. Although it only defines two classes (being `oo::object`, the class of objects, and `oo::class`, the class of classes) these classes have many abilities, some of which I shall show off here. Firstly, let us define some simple collection classes.

Since we want to allow objects to be automatically deleted when they are no longer referenced, it greatly helps to start with a reference-counting infrastructure. The following class creates objects that maintain a reference count just like those for `Tcl_Obj` values, deleting themselves when the count drops below one.

```
oo::class create Refcountable {
    constructor {} {
        variable refcount 0
        next
    }

    method incrRefCount {} {
        variable count
        incr count
    }
    method decrRefCount {} {
        variable count
        if {[incr count -1] <= 0} {
            my delete
        }
    }
}
```

On top of this class, we then build some simple collection classes. This list class can have reference counted objects added to it, searched for in it, removed from it, and can also iterate over the list of objects. When the list is destroyed, it will automatically remove its references to its contents (possibly deleting them in turn, of course).

```
oo::class create List {
    superclass Refcountable
    constructor {} {
        variable list {}
        next
    }
    destructor {
        my foreach object {
            $object decrRefCount
        }
        next
    }

    method add args {
        variable list
        foreach object $args {
            lappend list $object
            $object incrRefCount
```

```
        }
    }
    method has object {
        variable list
        expr {$object in $list}
    }
    method remove object {
        variable list
        set idx [lsearch -exact $list $object]
        if {$idx >= 0} {
            set list [lreplace $list $idx $idx]
        }
        return
    }
    method foreach {var body} {
        variable list
        upvar 1 $var v
        foreach v $list {
            uplevel 1 $body
        }
    }
}
```

But as we all know, lists are not the only sort of collection. The other major kind is the map. This map class maintains a mapping (in a dictionary) from strings to objects. The objects are naturally reference counted. It supports methods to put (add or update) a mapping, get the object from a mapping, delete a mapping or list the keys in the mapping. Aside from the other features, one interesting thing to note here is the `Decr` method, which is hidden from use by things outside the class. This happens automatically when the method name does not start with a lower-case letter.

```
oo::class create Map {
    superclass Refcountable
    constructor {} {
        variable map {}
        next
    }
    destructor {
        variable map
        dict for $map {key object} {
            $object decrRefCount
```

```
        }
        next
    }

    method Decr key {
        variable map
        if {[dict exists $map $key]} {
            [dict get $map $key] decrRefCount
            return 1
        }
        return 0
    }
    method put {key object} {
        variable map
        $object incrRefCount
        my Decr $key
        dict set map $key $object
        return
    }
    method get {key} {
        variable map
        return [dict get $map $key]
    }
    method unset {key} {
        if {[my Decr $key]} {
            variable map
            dict unset map $key
        }
        return
    }
    method keys {} {
        variable map
        return [dict keys $map]
    }
}
```

As you can see, it is quite easy to build all the trappings of a conventional object system. Or at least it is if you do not permit renaming of objects with `rename`. When objects may be renamed, things get quite a bit more complex since you can no longer safely store the

object's name; instead, you need to use some kind of unique identifier that is never modified: the name of the object's private namespace serves this purpose well. This class also demonstrates how the object system can use other features of Tcl (in this case, namespaces and traces) to achieve its aims

```
oo::class create Renamable {
    superclass Refcountable
    constructor {
        variable ::objforname
        set objforname([namespace current]) [self]
        trace add command [self] rename \
                [namespace code {my Renamed}]
        next
    }
    destructor {
        variable ::objforname
        unset objforname([namespace current])
        next
    }
    method Renamed {from to op} {
        variable ::objforname
        set objforname([namespace current]) $to
    }
    method uid {} {
        return [namespace current]
    }
    method getFromUid {uid} {
        variable ::objforname
        return $objforname($uid)
    }
}
```

Updating the list and map classes to use this new class's features by storing the unique identifier values instead of the object names is left as an exercise for the reader.

Wrapping Widgets

One key rite of passage for an object system is integrating with Tk. Everyone wants to do it so they can make megawidgets and create other sorts of enhanced functionality. Here I demonstrate how to do this in a simple example using an entry widget:

```
oo::class create Entry {
```

```
    self.unexpose create
    constructor {widgetName args} {
        entry $widgetName {*}$args
        variable realName __$widgetName
        rename $widgetName $realName
        rename [self] $widgetName
        trace add command $realName delete \
                [namespace code {my delete ;#}]
    }
    method unknown {method args} {
        variable realName
        return [$realName $method {*}$args]
    }
    unexpose unknown
}
```

Note that I use the special unknown `method` here to direct any method invocations not otherwise known to the subcommands of the real widget. This, very much like Snit's delegation, makes it simple to override a method without having to maintain the whole list of subcommands (a traditional problem with [incr Tk]).

But we want to do something fancier with this new capability. We do this by Here's a new kind of entry widget that we can flash like a button:

```
oo::class create FlashEntry {
    superclass Entry
    method flash {{times 5}} {
        set bg [my cget -bg]
        set fg [my cget -fg]
        for {set i 0} {$i < $times} {} {
            my configure -bg $fg -fg $bg
            update idletasks
            after 200
            my configure -bg $bg -fg $fg
            update idletasks
            if {[incr i] < $times} {
                after 200
            }
        }
    }
}
```

Now we can use this like this, which (apart from the slightly different creation sequence) is now just like using a normal widget, except it has this extra capability:

```
FlashEntry new .e
pack .e
bind .e <Return> {%W flash}
```

Tackling Threads

As you might expect, the TclOO package is completely thread-safe. This means that we can use it with the Thread package with very little fuss. For example, here is a small thread pool manager that also looks after getting the results from the pool back and cleaning up after itself:

```
package require Thread
oo::class create Parallel {
    constructor {lambdaTerm $args} {
        variable term $lambdaTerm
        variable pool [tpool::create {*}$args]
        variable posted {}
    }
    destructor {
        variable pool
        variable posted
        if {[dict size $posted]} {
            my cancel
        }
        tpool::release $pool
    }
    method start {values} {
        variable term
        variable pool
        variable posted
        if {[dict size $posted]} {
            error "pool still busy"
        }
        variable results {}
        foreach v $values {
            dict set posted [tpool::post -nowait $pool \
                    [list apply $term $key]] $v
        }
    }
    method wait {} {
```

```
        variable pool
        variable posted
        variable results
        set done [tpool::wait $pool [dict keys $posted]]
        foreach j $done {
            dict set results [dict get $posted $j] \
                    [tpool::get $pool $j]
            dict unset posted $j
        }
        return [dict size $posted]
    }
    method cancel {} {
        variable pool
        variable posted
        variable results
        set left [tpool::cancel $pool [dict keys $posted]]
        foreach j $left {
            tpool::wait $pool $j
            dict set results [dict get $posted $j] \
                    [tpool::get $pool $j]
        }
        set posted {}
    }
    method results {} {
        variable results
        return $results
    }
}
```

The thread pool manager can be used to execute lambda terms on many values in parallel; for example, this simple example demonstrates how to compute Fibonacci numbers in a somewhat foolish fashion:

```
Parallel create Fib {x {fib $x}} –maxthreads 6 –initcmd {
    proc fib x {
        if {$x <= 2} {return 1}
        expr {[fib [incr x -1]] + [fib [incr x -1]]}
    }
}
```

```
Fib start {10 20 30 40 50 60 70 80 90 100 110 120 130 140}
while {[Fib wait]} {}
array set fibonacci [Fib results]
puts "got part way..."
Fib start {150 160 170 180 190 200 210 220 230 240 250 260}
while {[Fib wait]} {}
array set fibonacci [Fib results]
Fib delete

parray fibonacci
```

Working with WebServices

Of course, we can also do things with objects and WebServices. Indeed, this is how they are typically created in most of the rest of the WS community.

```
package require WS::Server
package require WS::Utils
oo::class create Service {
    self.unexport new
    constructor {args} {
        global Config
        variable ServName [self]
        ::WS::Server::Service -service [self] \
                -host $Config(host):$Config(port)
                {*}$args \
                -premonitor  [namespace code {my}] \
                -postmonitor [namespace code {my}] \
                -checkheader [namespace code {my CHECK}]
    }
    method PRE {service operation argList} {
    }
    method POST {service operation status results} {
    }
    method CHECK {
        service operation caller httpHeaders soapHeaders
```

```
    } {}


    # A Simple Utility Method
    method DateTime {instant} {
        clock format $instant –format {%Y-%m-%dT%H:%M:%SZ} \
                -gmt yes
    }
    # A Utility Method
    method type {name definition} {
        variable ServName
        ::WS::Utils::ServiceTypeDef Server $ServName \
                $name $definition
    }


    # Utility method
    method operation {nameInfo argList doc body} {
        variable ServName
        set args {}
        set name [lindex $nameInfo 0]
        oo::define [self] method $name $args $body
        set body2 [namespace code [list my $name]]
        foreach arg $argList {
            append body2 " $" [lindex $arg 0]
            lappend args [lindex $arg 0]
        }
        ::WS::Server::ServiceProc $ServName $nameInfo \
                $argList $doc $body2
    }


    # Utility method for producing operation results
    method Result args {
        set op [uplevel 1 {self method}]Result
        upvar 1 _RESULT_ result
        if {![info exists result]} {set result {}}
        if {![dict exists $result $op]} {
            dict set result $op {}
        }
        dict set result $op {*}$args
```

```
    }
}
```

A demonstration of how to use this code is naturally in order. This is adapting from the

```
Service create wsExamples \
        -description {Tcl Example Web Services}
wsExamples type echoReply {
    echoBack {type string}
    echoTS   {type dateTime}
}

wsExamples operation {
    SimpleEcho {type string comment {Requested Echo}}
} {
    {TestString {type string comment {Text to echo back}}}
} {Echos a string back} {
    my Result $TextString
}
wsExamples operation {
    ComplexEcho {type echoReply comment {Requested echo+ts}}
} {
    {TestString {type string comment {Text to echo back}}}
} {Echos a string back with a timestamp attached} {
    my Result echoBack $TestString
    my Result echoTS   [my DateTime [clock seconds]]
}
```

This example, based on the code on the Web Services for Tcl website, is already considerably simpler for the application of simple object technology. But deeper support should be possible in the future. After all, ideally a web service should not be significantly harder to write syntactically than a conventional Tcl namespace; there is more than enough other complexity to deal with!

Accelerating with Aspects

Another thing you can do with TclOO is create aspects. An aspect is a way of “cross-cutting” a program so that code does not need to deal with everything in one place. Instead, you can have each part be a specialist in what it does, perhaps by adding logging or persistence to some existing code that would otherwise need significant reengineering. For example, below we define a special class that is used for applying transparent caches to an object. This is great when you are dealing with methods that can take a long time to

execute because of computation, though care must be taken with it because it does not understand object internal state.

```
oo::class create cacheAspect {
    filter Memoize
    method Memoize args {
        # Do not filter the core method implementations
        if {[lindex [self target] 0] eq "::oo::object"} {
            return [next {*}$args]
        }

        # Check if the value is already in the cache
        my variable ValueCache
        set key [self target],$args
        if {[info exist ValueCache($key)]} {
            return $ValueCache($key)
        }

        # Compute value, insert into cache, and return it
        return [set ValueCache($key) [next {*}$args]]
    }
    method flushCache {} {
        my variable ValueCache
        unset ValueCache
        # Skip the cacheing
        return -level 2 ""
    }
}
```

You can then apply this to any object to add memoization to that object's methods by mixing the class in. For example:

```
oo::object create demo
oo::define demo {
    method compute {a b c} {
        after 3000 ;# Simulate deep thought
        return [expr {$a + $b * $c}]
    }
```

```
}
```

This object just does some simple calculations, but takes a long time over it.

```
puts [demo compute 1 2 3]      prints "7" after delay
puts [demo compute 1 2 3]      prints "7" after delay
puts [demo compute 1 2 3]      prints "7" after delay
```

Time to add that memoization!

```
oo::define demo mixin cacheAspect

puts [demo compute 1 2 3]      prints "7" after delay
puts [demo compute 1 2 3]      prints "7" instantly!
puts [demo compute 1 2 3]      prints "7" instantly!
puts [demo compute 4 5 6]      prints "34" after delay
puts [demo compute 4 5 6]      prints "34" instantly!
puts [demo compute 1 2 3]      prints "7" instantly!
```

If we change things, we need to flush the cache…

```
oo::define demo method compute {a b c} {
    after 3000
    return [expr {$a * $b + $c}]
}
puts [demo compute 1 2 3]      prints "7" instantly, wrongly!
demo flushCache
puts [demo compute 1 2 3]      prints "6" after delay, right!
puts [demo compute 1 2 3]      prints "6" instantly
```

And all this from just the application of a mixin and a filter. The `demo` object itself knows nothing at all about how to do caching, but we waved our magic wand and added the functionality after the fact. Can aspects make *your* programming tasks easier?

Future Directions

Thanks to the help I have received from many people (especially Arnulf Wiedemann), the TclOO package is almost ready for public release. The main thing left to do is to discover what features have I left out that are critical, and that is something which is best done by letting other people try to use and break it. As always, the code probably needs more work so that it goes faster. I also want to really encourage everyone to take my code and find cool ways to use it to do things that are relevant to you.

Symbolic differentiation in Tcl: reusing the Tcl parser for symbolic algebra

Kevin B. Kenny
Computational Biology Laboratory, GE Global Research Center, Niskayuna, NY
kennykb@research.ge.com

(Extended abstract)

Symbolic differentiation is one of the easier problems in symbolic algebra; it is often presented as a student exercise in artificial-intelligence courses. Even though it is easy, it remains useful (and often underutilized) for mathematical computations such as root-finding, minimization and maximization of functions, and solving ordinary differential equations.

The most time-consuming part of writing a symbolic differentiator, in many languages, is writing a parser for the expressions to be differentiated. Fortunately, Tcl, being an interpretive language, comes with a parser for expressions that is available at run time. While the parser is not normally exported to scripts, the parser interface that the instrumentor in TclPro uses allows for script access via an extension. One advantage to using the built-in parser is that the programmer can be certain that the language of expressions to be differentiated is exactly the language of expressions to be evaluated.

The differentiator begins with a Tcl expression whose derivative is to be found, and the variable with respect to which it is being differentiated. The first thing that it does is to se the 'parser' extension to parse the expression. It then rewrites the parse tree into a form that is suitable for evaluation as a Tcl command. For instance, the Tcl expression,

```
2 * sin($x) * cos($x)
```

would be rewritten into the Tcl command:

```
{operator *} \
    {{operator *} \
        {constant 2.} \
        {{operator sin} \
            {var x}}} \
    {{operator cos} \
        {var x}}
```

Various rewritings are then available by evaluating the command in various namespaces. In particular, the command:

```
namespace eval math::symdiff::differentiate \
    [linsert $parseTree 1 $varName]
```

differentiates the given expression with respect to the given variable. The result is a tree in the same form: second and higher derivatives can be obtained by the same method.

The differentiator itself is fairly stupid. It includes the rules for finding the derivatives of sums, differences, products, quotients and powers. It also has the basic rules for differentiating the built-in functions, and a number of these rules also invoke common

code for the Chain Rule. A typical rule, in fact an unusually complex one, is the one for the two-argument arc-tangent function:

```
proc {math::symdiff::differentiate::operator atan2} {var f g} {
    set df [eval [linsert $f 1 $var]]
    set dg [eval [linsert $g 1 $var]]
    return [MakeQuotient \
                [MakeDifference \
                     [MakeProd $df $g] \
                     [MakeProd $f $dg]] \
                [MakeSum \
                     [MakeProd $f $f] \
                     [MakeProd $g $g]]]
}
```

Here we see the recursive nature of the differentiator at work: it begins by differentiating the two arguments to atan2 with respect to the given variable, and then applies the rule:

$$\frac{d}{dt}\tan^{-1}\frac{f}{g} = \left(g\frac{df}{dt} - f\frac{dg}{dt}\right) \Big/ \left(f^2 + g^2\right).$$

A handful of functions, no more than a couple of hundred lines of code in all, implement this part of the differentiator.

The [MakeSum], [MakeDifference], [MakeProd], … functions could have been implemented simply as invocations to [list]. In the actual implementation, though, they are done with a modicum of "peephole optimization." [MakeSum], for instance, has special cases:

- If either operand begins with a unary minus, the sum is rewritten as a difference.
- If either operand is a constant 0, the sum is rewritten as the other operand.
- If both operands are constants, the sum is folded to a constant representing their sum.

Similarly, [MakeProduct] has special cases to lift unary minus out of products; to simplify multiplications by zero, one and –1; and to reduce the product of two constants to a constant. The other expression constructors have similar peephole optimizations.

A simple (twenty-line or so) Tcl script then converts the list representation back to Tcl's notation so that [eval] can deal with it.

The resulting derivatives are hardly a minimal representation, but they are good enough for numeric evaluations. The differentiator has been integrated successfully with:

- A multidimensional root-finder using modified Newton-Raphson iteration.
- A function minimizer using the conjugate-gradient method and explicit derivatives.
- A non-linear least-squares curve fitter using the Levenberg-Marquardt algorithm.
- A solver for stiff ODE's based on the Fortran code LSODAR (and described in a companion paper).

The differentiator as it stands is serviceable, but there are ample opportunities for future work. Among the most obvious challenges are:

- Better algebraic simplification for the output. This sort of technique also moves Tcl toward the realm of a true symbolic-algebra system that could be used for more than just differentiation.
- Cleaning up the horrible API of the TclPro parsing extension and integrating it more tightly into the Tcl core (failing to expose the parser at script level is an egregious oversight).
- Reworking the Tcl-command representation to use the "math functions as commands" and "math operators as commands" syntax of Tcl 8.5. This change would (in 8.5) make it possible to evaluate the generated derivatives directly without reconverting to infix notation.
- At the same time that such a change is made to use tcl::mathfunc::F notation for the built-in mathematical functions, the ability for users to extend the set of supported functions dynamically should be included. Such an extension would allow special functions such as exponential integrals, elliptic integrals and Jacobian elliptic functions, Bessel functions, and so on to be differentiated symbolically.

Tcl 8.5 is not far from being an extremely capable system for ad-hoc mathematical calculations; extensions such as this one point the way.