KitView – A User Interface Tool for MetaKit

Steve Landers
Digital Smarties Pty Ltd
steve@digital-smarties.com

ABSTRACT

KitView is a user interface tool designed as a test bed for ideas in database design, data management, user interface generation and application deployment.

KitView is layered on MetaKit[1] – a flexible, efficient, portable and embeddable database library with bindings for Tcl and several other languages. KitView extends MetaKit, but requires no changes to MetaKit.

This paper gives an overview of the design and implementation of KitView, discusses some lessons learned to date, and considers possible future directions.

1 Introduction

KitView has its origins in a commercial software and hardware project.

The software component of the project includes a fairly typical distributed data management application, involving:

- a small number of centralized databases
- client software installed on a large number of machines (potentially thousands)
- a relatively low frequency of database updates
- semi-regular code updates

Key requirements were the usual wish list of:

- ease of deployment and upgrade
- good performance over slow networks
- ease of use
- portability across *nix and Windows

Typical solutions to meet these requirements seemed capable of addressing at most three of the above. Perhaps the most common solution would have been a database driven web back-end and a Java front-end. But however you look at it, this would still leave significant deployment, portability and performance issues.

The development organisation's existing commitment to MetaKit was re-confirmed after an evaluation of the available options. This decision was due to MetaKit's embedded nature (no need for database servers), Tcl bindings, small size and excellent performance.

The result was a solution that met all of the design objectives with a significantly lower development cost than more traditional implementation technologies.

The data management component - now named KitView - is being generalized in preparation for release as a general-purpose user interface tool for MetaKit.

Key features of KitView include:

- a layered data dictionary - implemented using MetaKit tables[1] - that can be incrementally added to existing MetaKit databases
- a data management user interface that is generated at run-time from the data dictionary
- centralized storage of application code, data and meta-data which is downloaded to clients at run-time
- distributed locking providing table, column or field locking with support for distributed waits
- data updates are distributed to active clients using a queued trigger mechanism
- small footprint
- easy client deployment using a single cross-platform Scripted Document[2]

The organization funding the development has actively encouraged the generalization and release of the data management component of the project as Open Source – in support of the Tcl/Tk community.

This paper gives an overview of the design of KitView and the implementation technologies used; assesses the success of the approach and offers some comments on the issues encountered along the way.

[1] Note that this paper uses the more traditional database terminology *table* rather than the MetaKit equivalent *view*

2 Architecture

KitView comprises four components

- A back-end server
- One or more front-end clients
- The MetaKit database file being managed
- A TclKit[3] interpreter for each supported platform

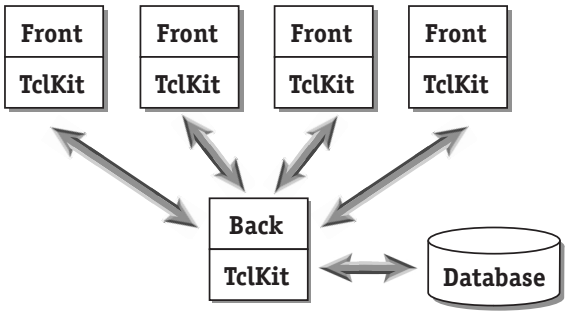

Figure 1 - KitView Architecture

The front-end client runs the application user interface. It contains any packages needed by the user interface (including copies of compiled packages for all supported platforms) and just enough logic to connect to the back-end server. Typically, all user interface code is downloaded from the back-end server to the client at run-time.

The back-end server performs several tasks on behalf of the front-end clients:

- data storage and access
- locking
- logging
- code storage

Both the front-end client and back-end server are deployed as single file Scripted Documents[2]. Typically, each application that uses KitView has a separate set of these files containing the required Tcl extensions and application specific code.

KitView stores only static data in the Scripted Documents - all other data is stored externally in the MetaKit database and managed via the back-end server. This allows the client to be made read-only and facilitates deployment via networked filesystems.

[2] Section 4 (Implementation Technology) gives more details about TclKit and Scripted Documents.

Communication between the front-end clients and back-end server is via a simple text based protocol that implements remote method invocations.

All communications are compressed at the protocol level. After initial testing it was found that latency was more of an issue than throughput, so the protocol was amended to combine (or "coagulate") several MetaKit library calls behind a single server method to minimise client/server communications.

3 Features

3.1 Data Dictionary

The data dictionary is central to KitView, and contains the information needed to generate a user interface at run time, including such meta-data as:

- data description (table, column, format)
- validation, access and transformation rules
- an index of application specific scripts (that are themselves stored in the back-end server)

The data dictionary can be incrementally added to any MetaKit data file. KitView guides the user through adding the dictionary and, where possible, makes intelligent suggestions (e.g. it suggests a column width based on the length of the longest value in the column).

The KitView data dictionary is layered on a MetaKit database using MetaKit tables. Several normalized tables are used to store the dictionary:

- the list of tables within the application
- the columns within tables
- rules to be applied to each column
- the definition of the rules
- an index of scripts/logic/code

Rather than describe specific user interface components or display characteristics, the contents of these tables are abstracted and generalised. For example, there are brief, short and detailed descriptions of each table, information about master/detail relationships between tables and hints about how to express plurals (e.g. row vs rows).

In addition, KitView maintains several tables:

- log of application usage
- a log of data changes
- active clients
- active locks

Type	Rule	Command	Message
field	expr	expr $arg	
field	length	expr {$value == {} || [string length $value] == $arg}	$title must be $arg characters long
field	maxlength	expr [string length $value] <= $arg	$title must be $arg or less characters long
field	notnull	expr {$value != {}}	$title cannot be empty
field	phone	regexp {^$\|^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$} $value	Not a valid long distance phone number
field	regexp	regexp {$arg} $value	
field	string	string is $arg $value	$title must only contain characters of type
reformat	initcap	set value [string totitle [string tolower $value]]	
reformat	phone	regsub {^(.+)(....)$} $value {\1-\2} value	
reformat	strip	regsub –all {[$arg]} $value {} value	
reformat	when	set value [clock format [clock seconds]]	

Figure 2 –Rule Definitions

The entry validation and transformation rules are worth mentioning in more detail. The rules are one of four types:

- entry - trigger on each keystroke
- field - trigger when leaving the field
- row - trigger when leaving the row
- reformat - trigger when leaving the field and replace existing field value

Rules are written in Tcl, and can make use of the wealth of string and regular expression handling available.

Multiple rules can be defined for each field or row, and are evaluated until one fails or all succeed. A reformat rule causes any field rules to be re-evaluated.

Rules are defined by adding rows to a rule definition table. Each row contains the following information:

- the rule type (entry/field/row/reformat)
- the rule name (used when invoking the rule)
- a Tcl command that implements the rule (returns 1 for success, 0 for failure)
- default message to display if the rule fails

There are a number of variables that may be used within the Tcl command that implements the rule. The two main ones are **value** (the value of the just entered character, field or row) and **arg** (an arbitrary value passed from the rule invocation). In addition, KitView sets variables to indicate the table and column being validated, and a variable for each field in the row.

Figure 2 shows some generic rule definitions. Note that rules usually map straight on to Tcl commands, although they could refer to application specific scripts.

When specifying the rules to be applied to each row or field, the following information is provided:

- the table to apply the rule to
- the type of rule (entry/field/row/reformat)
- the column to apply the rule to (if not a row rule)
- the name of the rule to use
- an arbitrary argument to pass to the rule
- an indication of how severe the error is – used to highlight error messages by the generated user interface (e.g. red for an error)
- the error message to display if the rule fails (this will override the default rule message)

Figure 3 shows an example specification of rules to be applied when entering data to a phone number field. The rules apply as follows

1. make sure the field isn't empty
2. strip out space, parentheses and minus characters
3. check that only digits have been entered
4. check the entered data is 10 characters long
5. reformat the data to look like a long-distance phone number – e.g. "(619) 692 2265"

Table	Column	Type	Rule	Arg	Severity	Message
sample	phone	field	notnull		error	You must supply a phone number
sample	phone	reformat	strip	()-		
sample	phone	field	string	digit	error	
sample	phone	field	length	10	error	
sample	phone	reformat	phone			

Figure 3 –Rule Usage

The data access permissions have been designed but not yet implemented. These are specified using an hierarchical role model that allows particular roles to be assigned to each user or groups of users. Organizational roles and permissions can be abstracted nicely via this mechanism - allowing the developer to specify access to a table, a column or even a row. Like validation rules, access rules are expressed in Tcl and maintained using KitView.

3.2 User Interface Generation

KitView separates the internal and visual representations of a user interface.

The internal representation comprises the meta-data stored in the data dictionary. The visual representation is inferred from the contents of the data dictionary and is generated at run time.

There are four user interface modes:

- database design
- user data management
- logging/status and audit trails
- application specific user interface

Only the first three have been implemented.

The Database Design mode allows the developer to modify the database structure, add data dictionary information, validation rules and rule definitions. The developer uses this mode when building the application or adding a data dictionary to an existing database. This is an administration mode that is run stand-alone (i.e. when no clients are active).

The User Data Management mode is typically invoked via an application specific window. It can be invoked multiple times on different tables (e.g. from several options on a pull-down menu).

Figure 4 shows the User Data Management interface generated when maintaining the Data Dictionary itself (not the usual procedure – but it serves as a good example).

The style of interface is simple but effective. The tables to be managed are presented using a tabbed notebook, with one tab per table.

The various items in Figure 4 are:

1. there is a tab for each table being managed – the tab name is inferred from the brief table description (Item 7)
2. the long description of the table
3. the table heading uses the short table description, and is enhanced by the "Count" field (Item 6) if specified.
4. if the table access is "read" then the table contents cannot be modified
5. the sorting order of data in the table (or don't sort if empty)
6. whether to display a row count in the table heading (Item 3)
7. a brief description of the table – used in the tab name (Item 1)

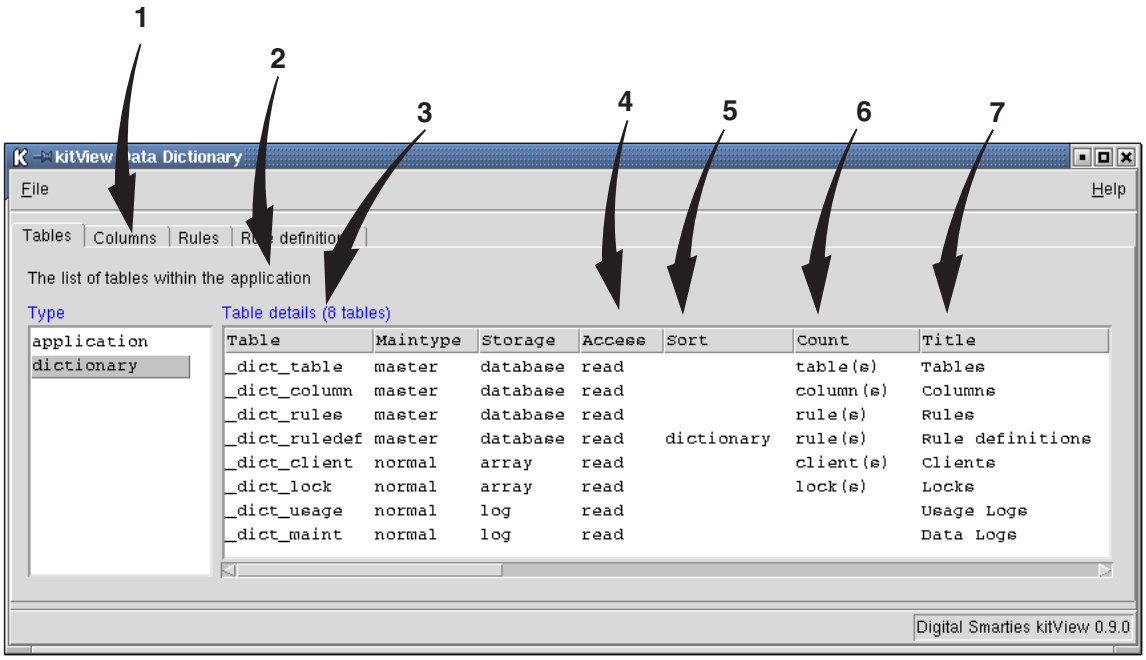

Figure 4 - User Data Management Interface

Note also that KitView supports a master/detail abstraction – i.e. a simple two level hierarchy within a table – specified by the "Maintype" field in the data dictionary.

When a user attempts to edit data that is locked by another user, a wait lock is requested from the back-end server, a waiting dialog is posted, and the foreground text color of the tab for the table is changed to indicate a warning (usually orange). The user can continue editing other tables by selecting their corresponding tab. When the data becomes available the waiting dialog is deleted, the table tab color is changed to blue (indicating an edit in progress) and the editing can proceed.

Data is edited in place, by placing an entry widget above the field being edited.

Any data entry rules are downloaded as required by front-end clients and cached for subsequent uses. The rules are evaluated using a safe interpreter.

If a rule fails, then the user is unable to leave the current entry field being edited.

The logging, status and audit trail interface is a read-only display of information maintained by the KitView back-end server. The list of connected clients and active locks are transient (i.e. non-persistent between server restarts). The application usage log records client connections and application events (configurable per application). The data logs record the changes to the database, with sufficient detail to allow both roll-forward and roll-back (although no tool exists to perform these yet).

3.3 Centralized Storage

As mentioned previously, there is a single KitView back-end server per database being managed. Data and meta-data are stored centrally in the database being managed. Scripts can be stored in the database or, more typically, in the server Scripted Document for the application.

Data, meta-data and scripts are retrieved by front-end clients as needed and cached for duration of the connection. Data is loaded the first time it is viewed and kept up to date via distributed updates.

To date there hasn't been a performance issue relating to cache size, but it would be possible to implement a simple "Least Recently Used" algorithm to keep cache size below a specified threshold.

Clients hold no state between sessions – they are therefore read-only and may be mounted from a network file system or shared volume.

3.4 Distributed Locking

Locking is performed via the back-end server, and provides centralized locking with a granularity down to individual data items.

The back-end server has an internal lock table and an array shared with front-end clients. The array is to allow front-end clients to view details for active locks, but the definitive state of locks is stored in the lock table. The lock table uses unnamed storage so that locks are not persistent across server restarts.

The locking key is a tuple of <table,master,row,column> where *row* is the row number (or range of row numbers) within the table.

If part of a table is already locked then it is necessary to test for overlapping lock requests. This is a little tricky since the row may contain ranges. On balance it was judged better to save bandwidth (and get more consistent performance) by sending ranges from the client rather than enumerating the rows to be locked.

If other clients hold locks on the table preventing a lock request from being granted, the server can grant a wait lock and arrange for the client to be notified when the lock becomes available. Wait locks are granted in the order they are requested. A client can hold several locks, or wait locks, concurrently on different tables.

From an implementation point of view the use of row indices works nicely, since the row number is also the index of the listbox displaying the data. However this becomes a problem if another front-end client is adding or removing rows since the row number could be invalidated. To get around this, add and remove operations always lock the whole table although it would be relatively easy to implement a method of updating the locking row numbers on existing wait locks.

Locks can be general (e.g. lock the whole table) or more specific (e.g. lock rows for a particular master value or a specific row). It is possible to lock an individual cell. If a lock is requested for a primary key (currently the first column in a master table) then the entire row is locked.

If a client holds a lock then this can always be made more specific (e.g. changing a table lock to a row lock or changing a row lock to a cell lock), but can only be made more general if this won't conflict with existing locks on the table. This scheme allows a user to highlight a range of rows and proceed to change them, releasing each row as it is updated, potentially allowing other users to proceed with their updates.

3.5 Distributed Updates

Front-end clients register their interest in displayed or cached data and are notified by the back-end server if the data changes. When the back-end server signals front-end clients that data has changed the front-end clients pull the data they need to update their displays and/or caches.

Change notifications are distributed to all connected clients using a queued trigger mechanism. The triggers are run "after idle" so that they are executed asynchronously with respect to the front-end call that changed the data.

Note that the central store is always the "reference" - if it is critical that an application has the latest copy of the data item it can request a lock on the individual data item before proceeding

This scheme (push notification followed by pull update) allows the front-end more control over when and how to retrieve data. Although the current implementation just pulls the data when notified, a number of alternative schemes would be possible:

- if the front-end is iconified then it could retrieve updates when de-iconified to avoid retrieving multiple updates of the same data item
- on high latency networks the front-end could limit the frequency of updates (e.g. only every few seconds) and aggregate several update requests into one "transaction"
- A random delay could be introduced between updates to avoid "data storms"

For relatively low data change rates (a few per second) the default scheme works well and none of these alternatives has been explored.

Responsiveness as perceived by users has proven to be quite good - even when tunneling through the Internet via modem connections.

This same queued trigger mechanism is used when front-end clients are viewing transaction and usage logs.

4 Implementation Technology

KitView is implemented in Tcl/Tk (via TclKit) using several popular extensions:

- BWidgets[5]
- mclistbox[6]
- Tequila[7]
- the busy widget from BLT[8]
- the cmdline package from tcllib[9]

4.1 TclKit

KitView uses TclKit – a single file Tcl/Tk interpreter that includes the Tk library as a loadable extension. It also includes the Mk4Tcl package that provides bindings to the MetaKit database library.

Even without the use of MetaKit, TclKit makes the deployment of Tcl/Tk applications easy.

A separate TclKit interpreter is required for each supported platform.

4.2 Scripted Documents

Both the back-end server and front-end clients are deployed as cross-platform Scripted Documents.

A Scripted Document is a single file packaging of application scripts (in this case Tcl code), compiled extensions for supported platforms and application data.

With a little care, Scripted Documents are portable across any platform supported by TclKit. This is achieved by including in the Scripted Document any required compiled extensions/libraries for each platform to be supported. The Tcl package mechanism is used to load the appropriate shared library for the current platform.

KitView stores only static data in the Scripted Documents and all other data (including user options) is stored externally in the MetaKit database and managed via the back-end server.

Scripted Documents contain an internal Virtual File System (VFS). The TclKit VFS layer intercepts all Tcl filesystem-related operations, allowing them to operate on both external files and on the internal filesystem within the Scripted Document. In KitView, both scripts and platform specific packages are stored within the VFS. More information on the Virtual File System can be found on the MetaKit Wiki[4].

This approach allows scripts to be developed using traditional tools and packaged unchanged into a Scripted Document for deployment.

4.3 MetaKit

KitView is based on TclKit 8.4, so as to leverage the new Mk4tcl object API and features from MetaKit 2.3. Pre 2.3 database files are converted during use, but remain in the original format unless the data is modified and committed. This allows viewing of read-only data files (such as those on a CD-ROM).

MetaKit uses column-wise storage and memory-mapped files which, for many applications, provide significant flexibility and performance benefits. In particular, the column-wise storage means that brute force searching of medium sized databases (perhaps up to a million rows) is surprisingly fast. This certainly makes for easy deployment – there is no need for specialized database management systems with server processes and all the complexity they bring.

In addition, MetaKit supports dynamic reconfiguration of database schemas. This was particularly useful when prototyping the data dictionary.

4.4 Tequila

Tequila is a communications mechanism that implements persistent shared arrays between Tcl applications. It is part of the MetaKit distribution.

Tequila allows global arrays to be shared transparently between front-end clients and the back-end server. Any changes automatically propagate to clients attached to the array.

Within KitView, Tequila is used when sharing status arrays (such as the locking information and active clients). It is also used as a transport mechanism in the implementation of remote method invocations and queued triggers.

Tequila is built using Tcl traces and file events, and is a relatively lightweight client/server communications mechanism.

4.5 Widgets

The main widget set used in KitView is BWidgets.

Although there are a number of good alternatives, BWidgets were selected because they offered the most appropriate trade-off between:

- portability
- small size
- reasonable performance for the class of problem
- good look and feel on most platforms

In addition, extensive use is made of the mclistbox multi-column listbox widget when displaying tables.

The blt::busy widget was extracted from the BLT widget set. A true busy widget such as this is absolutely essential when implementing a quality client/server user interaction.

The winico[10] widget is used on Windows, both in the front-end clients and also to allow the back-end server to appear in the system tray.

5 Assessment

5.1 Deployment

Perhaps the most unusual feature of KitView is its deployment technology.

In fact, it could be said that deployment is a dream compared with more traditional approaches.

A client installation requires only two files (TclKit and the client Scripted Document). Both are read-only and can therefore be stored on a network file system. The TclKit interpreter is the platform specific part, the Scripted Document is cross-platform.

The only constraint on installation is that both these files must be stored in the same location. This is a small price to pay for not having to edit registries, set file associations, etc. Installation can be as simple as copying the two files, an un-install as simple as deleting the files.

Likewise, the back-end server requires only three files (TclKit, the back-end Scripted Document, and the MetaKit database file).

Not only are there just a few files, but the file sizes are almost unbelievably small compared with alternative technologies.

The contents of Scripted Documents are transparently compressed when the Scripted Document is created. In addition, on Linux and Windows TclKit uses the UPX executable packer which results in very small size (Linux x86 is around 1Mb, Windows around 800k).

In a typical application using KitView, the file sizes are

- back-end - 65k
- front-end - 280k

Note that the front-end size includes bootstrap code, a 50k splash screen; the BWidget, cmdline, mclistbox and Tequila packages (all Tcl only); and shared libraries for the BLT busy widget (Linux, Windows and Solaris) and winico (Windows).

In addition to the small size, centralized code storage makes it relatively easy to upgrade clients. For example, an upgrade of application code involves building a new back-end server Scripted Document and restarting the server. When clients are restarted the new code will be downloaded.

5.2 Performance

Client/server performance has proven to be good – even when tunneling through the Internet over relatively slow modem links.

Remote users perceive locking, distributed updates, and log propagation as almost instantaneous.

In addition, MetaKit has performed flawlessly for the class of application that KitView has been applied to. It's small memory footprint and column-wise data storage more than makes up for the lack of indices and sophisticated architecture of more traditional database products.

The core MetaKit database is geared towards performance and has been used in a number of very high-performance contexts. If a performance bottleneck is encountered, it is expected that it will be due to the sequential operation of the single back-end server. There are a number of possible solutions if this does indeed become a problem - from buying a faster server to using multiple threads to service requests.

And, as with all scripted applications, there is always the option of moving time critical functionality into C or C++.

5.3 Technology

Tcl/Tk has been fundamental to being able to achieve the results. Within KitView it is used as an implementation language, extension language, and communications protocol (via Tequila).

In many ways building KitView involved standing on the shoulders of giants[3]. Without the foresight of those who developed Tcl/Tk to its current state, or those who built extensions such as BWidgets and mclistbox, it would have been necessary to start from a much "lower" platform. With these tools it was possible to concentrate on the architectural and design issues.

Also, KitView was an exercise in portability and cross platform development. It was built on Linux and deployed on Linux, Windows and Solaris. Windows binaries are cross-compiled on Linux using Mingw[11]. VMware[12] is used to test x86 environments and VNC[13] for remote testing.

5.4 Issues

There are a number of outstanding issues that still need to be addressed.

In particular, KitView contains no security or authentication mechanism. Whilst this isn't a problem for the original application (which runs on a secure, private network) it will need to be addressed.

Another implementation issue is whether to use an object system. Parts of KitView have been prototyped using [incr Tcl][14] but this is a little too heavyweight for the intended purpose. Of course, [incr Tcl] is a fine tool and heavyweight is a relative term – but adding 45k in size for each supported platform is still significant given the project objectives.

What KitView needs is a simple Tcl-only OO package supporting classes, methods, object variables and simple introspection. At some point the various Tcl-only options will be investigated, as will extracting parts of Tcl++[15]

Tracking the Tcl/Tk 8.4 alpha releases had a couple of nasty consequences. Tcl/Tk 8.4a2 broke backward compatibility with stubs-enabled packages. Also, changes to the entry widget broke BWidgets in some fairly gratuitous ways. Whilst not wishing to "point the finger", it was nevertheless quite a pain – but, that's what an alpha release is for.

Perhaps the most significant issue was coming up against the limitations of Tk when building general-purpose interface tools like KitView. Most can be worked around but not always satisfactorily.

For example, Tk lacks a scrollable container. This can be worked around by embedding a window in a canvas, but this isn't perfect – there are issues with resizing embedded windows and scrollbar positioning.

Another missing widget is a native multi-column listbox. The mclistbox widget does a good job but resizing and positioning can be a problem when imbedding in a canvas.

Tk is still the premier cross-platform GUI scripting language – but it could be much better. What is needed is a widget set that combines the best features of BWidgets, [incr Widgets] and Tix[16]; with the portability and flexibility of BWidgets; and the performance of Tix and [incr Widgets]. Before this can happen, Tk needs a standard mega-widget mechanism and a few additional fundamental widgets (such as those mentioned above).

6 Futures

KitView lacks security and authentication. Current thinking is to address this by adding security at the Tequila level.

Whatever technology is used it must be both cross-platform and simple (thereby ruling out just about every piece of middleware known to humanity). KitView doesn't need much – just a text based API and remote method invocation. Some of the alternatives to be investigated include a home grown solution using TLS[17], a web based approach using an embedded TclHttpd[18] server in the back-end, or perhaps

[3] If I have seen further it is by standing on ye shoulders of Giants - Isaac Newton

something implemented using a Tcl SOAP binding. Time will tell.

A feature that is sure to be investigated is incremental booting. Rather than have an application specific client containing the necessary libraries, a generic client would contain only enough information to contact a back-end server. All the necessary parts would be downloaded and persistently cached locally, with mechanisms to update the cache if application components change. This would make deployment even easier than the present.

Although there is currently only one visual representation, it would be feasible to generate multiple interfaces depending on factors such as the type of display or level of experience of the operator.

The approach of storing a relatively high level UI description in the MetaKit database has merit - maybe even pre-parsed into XML. MetaKit would be particular suited to this application, supporting both hierarchical data and dynamic schemas.

This would not be a screen painter UI development tool but rather much higher level. Ideally, the definition would be sufficiently abstracted so that over time the UI could be improved by tuning the run-time tools without the need to change the UI definition.

Note that the UI need not be restricted to graphical/ bitmap displays - a design goal is that the interface should be mappable to character displays, PDA displays and web interfaces.

The other future is to track MetaKit development. This will include using the optional SQL layer for defining rules.

7 Conclusions

KitView is a very useful addition to the suite of Tcl packages and tools.

It is of particular value in typical data management situations, automating the generation of user interfaces and allowing the developer to concentrate on the application (or domain) specific functions.

The approach of separating the definition and visual representation will allow adoption of new UI techniques and different frameworks without the need to develop to a "least common denominator". Existing applications will adapt to new interfaces without the need to be redeveloped. This approach gives an application a degree of technological independence – and to some extent it becomes future proof.

The organisation funding the development of the commercial project views the use of Tcl/Tk, Tclkit and KitView as a strategic advantage because it is reliable, allows for true rapid prototyping and requires minimal installation.

The development model used for KitView is also noteworthy – developed as part of a proprietary product and then generalized for release as Open Source software. The Open Source community gets the benefit of software that might not otherwise be built. The company funding the original development benefits from broader use and support of the package. And the developers get to pay the bills. ☺

And finally, KitView has confirmed the validity and benefits of extending core compiled functionality by scripting. KitView enhances MetaKit with distributed access, locking, transactions (sort of) and logging – all without writing a line of C code - and still obtains good performance.

8 Acknowledgements

I would like to acknowledge the groundbreaking work of Jean-Claude Wippler in producing MetaKit, TclKit, Scripted Documents and Tequila. Also, thanks are due to Larry Blasingame for his support and encouragement, and Brad Entwistle for his assistance in preparing this paper.

References

[1] Wippler, Jean-Claude. *The MetaKit Database* http://www.equi4.com/metakit/

[2] Wippler, Jean-Claude. *Scripted Documents.* Proceedings of the Seventh Annual Tcl/Tk Workshop. Feb 2000. http://www.equi4.com/jcw/scripdoc.html

[3] Wippler, Jean-Claude. *TclKit* http://www.equi4.com/tclkit/

[4] *The MetaKit Wiki* http://www.equi4.com/metakit/wiki.cgi/114.html

Software mentioned in this paper

[5] The BWidget Toolkit http://sourceforge.net/projects/tcllib/

[6] mclistbox http://purl.oclc.org/net/oakley/tcl/mclistbox/

[7] Tequila http://www.equi4.com/tequila/

[8] The BLT Toolkit http://sourceforge.net/projects/blt/

[9] tcllib http://sourceforge.net/projects/tcllib/

[10] Winico http://ftp.bjg.de/pub/tcltk/winico31.README/

[11] Mingw http://www.mingw.org/

[12] VMware http://www.vmware.com/

[13] VNC http://www.uk.research.att.com/vnc/

[14] [incr Tcl] http://tcltk.com/itcl/

[15] Tcl++ http://www.sensus.org/tcl/

[16] Tix http://tix.sourceforge.net/

[17] TLS http://sourceforge.net/projects/tls/

[18] TclHttpd http://sourceforge.net/projects/tclhttpd/