



Collaborating applications: Tequila takes Tcl further

Jean-Claude Wippler
Equi4 Software
jcw@equi4.com

ABSTRACT

Tequila is a "middleware" kind of infrastructure written in pure Tcl, which makes it easy to write collaborating applications. A first version has been extremely effective and succesful in two very different projects: a distributed telephone system testing system which has processed a quarter million daily events non-stop for over a year, and a highly interactive multi-session chat and discussion board used in high-school collaboration research. Several other projects have demonstrated the convenience of having such a generic and robust infrastructure. The next generation of Tequila will be presented, aimed at providing an even richer foundation to build on. It is heavily influenced by the first version as well as by GroupKit, a generic collaboration environment developed at the University of Calgary. Examples will be presented of how the new design simplifies building collaborative applications, such as a basic chat system in just a few dozen lines of Tcl. Tequila supports structured data and automatic persistence, which has a major impact on how clients and servers are designed. The basic components (pools, rpc endpoints, notifiers) will be presented to show how Tequila enhances Tcl's strengths, i.e. seamless integration of the Tk GUI with networking and persistence.

Introduction

It can take a considerable amount of code to implement applications that combine a user interface, network communication, and data storage. Even though this is an area that is extremely well suited to Tcl, there simply is a lot of ground to cover.

The Tequila package described here describes an approach which has been used successfully in the past, and which is now being re-implemented to apply the lessons learned so far as well as to better take advantage of the tools available today.

We will first look at the specific requirements of collaborative applications, how the first version of Tequila ("T1") dealt with these and what experience was gained. Then a new design ("T2") will be presented which cleans up the basic design, in order to make the code more modular and extensible, and extends it with "pools" as the basic concept for persistent state and data exchange and "notifiers" to handle change propagation within and between multiple collaborating application instances.

The last section will cover the details of Tequila, and can act as demonstration of the new implementation. It includes examples to illustrate the mechanisms and idioms in practical use.

Collaborating software

There are a number of aspects that set collaborating software apart from other types of software systems:

- Multiple processes, machines, platforms
- Everything happens all over the place
- Event-driven, asynchronous socket I/O

- Keeping running applications consistent
- Dealing with network failure at any time

First of all, collaborating software consists by its very nature of multiple applications, running independently, used by different people on workstations which are connected via a network. In the current design, Tequila is aimed at a configuration with one or more long-running central servers, and any number of client processes logging in and out over time.

Collaborating applications must be prepared to deal not only with changes coming via the GUI, i.e. due to local interaction, but also with state changes received via the network. Collaborating applications must be event-driven for both user interaction and network activity, to maintain a responsive user experience. They must also be very careful about consistency – so that inevitable latencies between the different clients do not cause confusion, or worse: inconsistencies across clients whereby different people end up working with different versions of the data.

Lastly, collaborating software has to deal with less-than-perfect communication channels, even if the hardware it runs on is reliable. Networks occasionally stop working, so loss of connectivity must be dealt with. In the general case, this can be a very complex requirement.

Tcl is a good fit

The Tcl scripting language, and the Tk GUI toolkit are an extremely good fit for collaborating software for a number of reasons:

- Event-driven I/O (fileevent)
- Everything is a string, easy to send around

- Traces for managing changes
- Platform independence
- Deployment, via Telkit

Together, these features combine to create a software development environment which is extraordinarily well-suited for networked visually rich collaborating applications, in fact. It takes perhaps a dozen lines of code to implement a basic server, and about the same amount of code again to let clients communicate with it.

The total code needed to implement Tequila is under a thousand lines of pure Tcl. No C-coded extensions are needed (beyond what Telkit includes) to create a system which handles the user interface, networking, and storage of data. The performance has turned out to be excellent, no doubt due to the fact that the level at which network activity takes place is very high.

It should also be mentioned that Starkits, the deployment technology which is now so common with Tcl/Tk 8.4 and particularly the Telkit self-contained application runtime for Tcl/Tk, have made deployment of Tcl applications trivial. In the case of collaborating applications, this is an essential ingredient for success.

Early case study

The original idea for Tequila came from a commercial project implemented in the course of 2000 and 2001. The system is used to continuously test the properties of a telephone network by making large numbers of test calls and collecting the results. Some details:

- Long-lived “dumb” central server
- Central process driving modem banks
- A scheduler drives the main workload
- Custom requests are piggy-backed on top
- All data stored centrally
- Interactive clients on Windows and Solaris
- Simple login/password authentication
- Client “loads itself” over the net on startup
- Watchdog process restarts server if it fails

It was decided early on that the networking and storage design would be developed separately, and tested with an elaborate simulation harness. This became Tequila version 1, although the core was in fact re-coded twice from scratch, as more experience was gained.

This system was extremely successful:

- Dimensioned to handle 250 dialers on an Ultra-SPARC, the system was in fact able to handle 500 dialers in continuous simulation mode (which placed a far greater load on the system) from a measly laptop running Linux.
- As part of a very extensive pilot test, the server has

stayed “up” for 18 months without a single failure, making roughly 250,000 test calls daily.

- Software changes do not require a server restart, the scheduler and modem-bank processes can both be stopped and restarted in mid-flight.
- Client deployment consisted of a single Telkit executable plus a Starkit (the Telkit + Starkit deployment model were used in a production setting for the first time, although it was still called a “scripted document” at that time).
- The core system became smaller and simpler as development progressed, a clear sign that the design concepts used were working out properly.
- Development and testing, and eventually also delivery of the final product took place over three continents. ISDN-based client access turned out to be workable, even though the system was only designed for LAN use.

One interesting observation is that all of the above was achieved with a beta version of Tcl/Tk (8.2b2, I think). So much for the notion that “beta” must be buggy!

Tequila T1

The first implementation will be called “T1” from now on, to distinguish from the newer T2 design presented later on in this paper.

T1 is based on a really simple API: arrays.

The idea is that client applications “attach” one or more global arrays to Tequila, and then automatically any changes made to them anywhere will propagate to all other clients via background network communication.

This is in fact exactly what T1 does. It relies very heavily on Tcl traces to detect changes to any attached array, and then takes care of sending and receiving such updates. Traces are then also used by the application to trigger processing and user interface updates when any item in an array changes due to incoming network messages. These changes can include addition and deletion of array elements, not just modification of existing ones.

Persistent storage comes for free. This rather surprising benefit comes from the fact that all data resides on the server, so as long as a server stays up, nothing needs to be done on the client side to store information. Every time a client starts up, it receives the relevant state from the server. In a way, the server defines a permanent workspace, and clients simply get to see more or less of that state as they attach their arrays.

However, given that a server could fail, such an “all in-memory” approach as described approach is evidently a bit too risky to build serious applications on. For that reason, T1 includes server-side logic to store array contents on file: either as views in a Metakit database or as individual files in a directory. The

latter can be convenient to see what the state is while a server is running, since all state of such arrays can easily be inspected from the command line.

There is a special array called “tequila” to which clients can attach. It contains one entry for each currently connected client. This allows clients to discover which other clients are running, and to detect their demise by setting an unset trace on the tequila array.

A simple lock mechanism was added later on by Steve Landers, to allow clients to synchronize their actions. This allows a client to claim a data item so that others will not alter it while the item is displayed and being edited, for example. All locking is advisory: all parties must follow the proper convention, and refrain from altering locked items.

Experience with T1

One of the most surprising outcomes of T1, was to see just how well it fits into Tcl. Applications need to be written in a certain style, but apart from this nothing changes between stand-alone applications that do not use Tequila and applications that do. The only thing that changes is on startup: decide which arrays are shared, and attach them as part of the initialization step.

The fact that arrays “simply persist” is a benefit that is hard to over-estimate. Coding can now deal with the structure of data and the effect of changes on that data, with no attention at all to loading or saving things. This works, because all previous state changes sort of get applied automatically on startup, so all an application has to do is set up the user interface, prepare the traces that act on changes, and then attach the array. As soon as the connection is established, the network will send all state, adjusting the user interface through the traces.

Performance is surprisingly good. It turns out that Tcl, even though scripted, has absolutely no trouble keeping up with network activity and traces over a LAN. Due to the asynchronous design, many delays drop beneath the horizon. This is even the case for locally generated changes, which normally also need to make a round trip to the server to be applied consistently. The trick to achieve good performance is to design in such a way that latencies have limited impact. Bandwidth can be more or less ignored (within reason), since the level of communication is very high, i.e. the rate of messages and their size often remains quite limited.

A design decision which has paid itself back in gold, is the choice of using a generic Tequila server, and keeping all application-specific business logic isolated in clients and secondary “worker processes” – clients which run on the server to perform tasks, but with no user interface. The generic server has allowed creating a system whereby the centerpiece is rock-solid from the very start, and requires almost no debugging as the rest of the application is imple-

mented and extended.

Not all is peachy with T1, however. One unforeseen issue has been that distributed code and especially traces are a nightmare to debug, even with ample logging in the server. The timing variations make it next to impossible to reproduce problems related to order-of-events, i.e. race conditions.

Another problem with T1 is that it does not scale that far when arrays contain a lot of data. The reason for this is that all data must be sent across to each client as the array is attached, leading to potentially long startup delays. Furthermore, being arrays, all data in the client ends up being memory resident. These issues can be alleviated up to a point by carefully splitting data across multiple arrays, and only attaching a subset. In T1, an extra escape hatch was built into Tequila, allowing a procedural get/set access mode for data, i.e. foregoing attached arrays for some cases, such as activity logs.

And lastly, Tequila grew out of a single project, so its features were tuned somewhat to that. One limitation is that the transport layer was built-in as being plain sockets only, using dedicated ports. Other uses such as an HTTP wrapper via port 80 were considered but never implemented. Neither was a more secure session layer, such as SSL.

Subsequent uses in other projects have shown that although Tequila provides a very effective model for collaborating software, it does have limited uses. Sometimes, a more traditional RPC-like mechanism can make things simpler still, rather than having to shoehorn the code into changes to attached arrays.

Beyond T1

In January 2005, a new project called “Stargus” presented a chance to revisit the whole design and structure of Tequila, with the opportunity to take things a bit further.

The following aspects were investigated:

- Improved data-model support
- Far better scalability
- Support for authentication & encryption
- Client-side caching and re-connects

Many of these capabilities are found in the GroupKit package, which was found too elaborate and complex for T1 in 2000/2001, but which does offer a very rich set of concepts and implementation solutions for most of the above aspects. The one thing lacking in GroupKit is persistence, and as T1 has shown that really is a very useful and effective part of Tequila.

As a result, and drawing on the experience of Mark Roseman, GroupKit’s principal architect, it was decided to re-implement Tequila, in a design much closer to GroupKit’s, while adopting as much as possible from T1. The result is Tequila T2, described here.

Tequila T2

T2 is a complete rewrite. It was written in a very modular fashion, so that individual parts can be extended or even replaced later on, as needed.

This is one of the situations where an object system really shines. It is trivial to extend a system when its main components are object oriented, regardless of whether the OO mindset is used in the rest of the application. For practical reasons, IncrTcl was selected for T2 – with the idea that any OO system could be substituted later, if the Tcl community ever gets its act together in designating one OO system as “the one”.

IncrTcl is part of every Telkit binary, the deployment system of choice for applications such as these. It is stable, documented, and fast.

For persistence, the Metakit database library is used. There are a number of reasons why this is a good fit:

- On-the-fly restructuring (adding properties)
- Compact both on file, and in-memory
- Easy to send serialized snapshot over a socket
- Included in Telkit, stable

Neither IncrTcl nor Metakit are essential ingredients for Tequila, they could be replaced when other packages are available which are more suitable in some sense.

A side-effect of using components which are present in every build of Telkit, is that Tequila can be used as is on every platform for which there is a standalone Telkit executable, which is several dozen by now. Since ActiveTcl also contains all the necessary packages, that too is an option.

Anatomy of T2

Tequila T2 consists of the following key components:

- Pools, as the logical building blocks of data
- RPC endpoints, used for all communication
- Notifiers, a generalized version of traces

Each of these will be covered in detail in the next sections. Together these form a package called, unsurprisingly, “tequila” which is a single Tcl-only script that needs to be included in each collaborative application, both clients and servers. There is a simple generic server which can be used out of the box for simple cases, or can be used as template and starting point for a more sophisticated system. And lastly, there are a number of examples to illustrate various aspects of T2.

At this stage, T2 is fully functional, but it has not been used “in anger”: there are no large-scale production systems based on T2 so far (April 2005).

Pools

Pools are the central concept on which applications get built. A pool contains one or more named collections of data, which automatically persist and get shared with all clients connecting to the same pool (via the server – T2 only addresses centralized client/server topologies at this stage).

A “collection” in turn, is a tabular data structure: it has rows, identified by a key, and named attributes. You can think of a collection as a table. Simple tables might contain just a key and a value, in which case they are very much like Tcl arrays, or they might contain more attributes. All values in a collection are strings and can – as usual in Tcl – contain anything.

Pools represent global state. In MVC (model, view, controller) parlance, pools are the model. Pools will normally persist. The structuring of pools into collections is a way to bring different data structures together – each collection can have a different set of attributes, whereas all rows within a collection have the same set of attributes.

In the simplest case, you only need a single pool, so the basic design effort is all about deciding what collections you need, and what the key and attributes of each should be.

Here’s how to create a new standalone pool:

```
tequila::pool mypool
```

This produces a new command, named “mypool”. From there, it is easy to create a collection and add a row to it:

```
mypool set mycoll.x value y
```

This creates the collection “mycoll”, and adds a row with key “x” and an attribute named “value” set to “y”. Fetching the row again uses the “get” sub-command:

```
puts [mypool get mycoll.x]
```

You can also find out the keys of all rows in the collection:

```
puts [mypool keys mycoll]
```

Or any other property for that matter:

```
puts [mypool values value]
```

One useful subcommand for debugging is “dump”, it prints the first few rows of every collection in a pool.

There are several more subcommands. The idea is that collections are like an “array with named columns”, with “rows” as entries, and that these arrays, rows, and attributes are created on-the-fly when a value is set.

As described so far, pools are merely local. To make them shared we need to tie them to “endpoints”.

RPC endpoints

An “RPC endpoint” is perhaps best described as “this half of a connection”. It is the entrance of the tunnel leading to a remote counterpart, i.e. for a client it is the link to a server.

The following discussion is geared towards sockets, although other types of endpoints could be created. The startup sequence is bound to be different, but - once set up - end points are simply objects which must respond to a certain API.

Endpoints are asymmetric (just as sockets are). To establish a connection, a server somewhere must create an endpoint and specify what port number to use:

```
set s [tequila::endpoint -server 8291]
```

The above creates a server socket, listening on port 8291. Once that is available, clients can connect to that endpoint using something like:

```
set c [tequila::endpoint \
    server.domain.net 8291]
```

Once a session has been established, we can tie a pool to them. This is done when the pool is created, so that its previous state can be restored right away. Here’s an example where “mypool” is associated with a pool of the same name on the server:

```
tequila::pool mypool $c
```

Keep in mind that all network events are asynchronous and processed in the background, i.e. at idle time. The above does not instantly fill the pool, it just ties things together so that everything happens later on, when the application setup has been completed and it starts waiting for events.

Notifiers

As it stands, the above is sufficient to make pools stay in sync across multiple clients. The one missing link is that an application wouldn’t find out about any of these changes, so it’d be rather boring.

That is where notifiers come in. A notifier is an object where apps can subscribe to be notified of events. The prime user of this mechanism is the pool – every pool includes a notifier, so that changes to the pool can be associated with application scripts to execute.

The notification mechanism is based on named “events”. Applications can “bind” to an event and cause it to execute a certain script, or they can bind to a range of events by using “*” and “[...]” and “?” wildcards.

Pools define various events, such as:

- connected – the pool has been initialized
- disconnected – connection has been lost
- collection.add / .change / .delete – a row has been added/changed/deleted

- collection.attribute – a change to specified attribute

The scripts that get evaluated when an event fires can obtain additional information about the event through a mechanism that is identical to Tk: event specifiers, i.e. codes in the script starting with “%” – these get replaced before the script is actually evaluated. Event specifiers are essentially a general way of passing arguments from an event producer to all event consumers.

In the case of pools, the following event specifiers are available:

%C – name of the collection

%K – key value

%E – full event name

%R – row number

Example of use:

```
mypool bind *.add \
    { puts “added key %K to coll %C” }
```

Not all specifiers are meaningful in all types of events (%C, %K, and %R are not set in connect/disconnect events, for example). Some events may define more specifiers – see the documentation for details.

Keep in mind that notifiers are not limited to being used by pools. The basic rule is just that event producers and event consumers must agree on consistent conventions.

Putting the pieces together

In the basic scenario, the following steps must be taken:

- A server must be started, on a well-known host and port. The server usually stays running for a long time, and runs unattended.
- Each client starts by creating a client-side endpoint connecting to that server.
- Next, clients set up one or more pools, and tie them to this end point. The convention is for a server to have a “system” pool, and in it a collection called “tequila”.
- For simple uses, clients can set up their end of the “system” pool and create the collections they need inside that pool.
- If multiple pools are required, then the server must either set up all pools, or be prepared to create such pools as needed when clients ask for them.
- Once pools exist, clients proceed by creating a set of bindings that determine what happens on changes to collections in the pool(s).
- Lastly, each client enters the idle loop, starting the phase where connections will become active and messages get sent around between the Tequila clients and the Tequila server.

From here on, the client application becomes an event-driven mechanism, dealing with user interface events as usual (with Tk) and dealing with network events to manage the sharing of state and the propagation of changes to that shared state.

An RPC example: chat

One of the simplest networked examples is a chat – people type lines into an input area, and everyone gets to see the lines when RETURN is pressed.

The code for this example is in Appendix A. It illustrates the absolute basics of Tequila:

- Setting up the Tequila package
- Connecting to a common server
- Sending lines to the server
- Responding to incoming lines

Although the code is very simple to read and may help to get started, it also is seriously flawed: the design used in this chat application is in fact precisely the wrong approach most collaborative applications...

The reason is that the chat works in terms of actions (i.e. send this line to everyone). This breaks down when you start thinking about clients connecting and disconnecting at various times. When that happens, you often want to be brought up to date to see the same state as what others see. This brings a sense of “being” in one place, and collaborating on a common project.

With purely event-driven exchanges, you’d have to think in terms of replaying changes to clients who join later. But as we shall see, there is a much easier way to accomplish the same thing.

Model-View-Controller

From Smalltalk comes the concept of splitting the work in an application into three distinct tasks:

- Model = the state, think “documents”, “projects”, and so on.
- View = what you see on the screen (or the browser, or even reports on file or paper). There can be more than one view on the model. There can also be a model without a view – i.e. when you are not displaying it.
- Controller = the way to deal with input actions, from point the mouse, to scrolling, to entering data and pushing buttons.

With collaborating applications in general and Tequila in particular, these aspects of a design come back in full force. One reason being that there will be potentially many views, and many controllers. But in general, what you want is that everyone works with a consistent copy of the same common model.

First the bad news: there are limits to that consistency. There is no way one can automatically resolve

the situation when people perform conflict actions at exactly the same moment in time. It’s a bit like Heisenberg’s uncertainty principle: either two people are next to each other and acting in sync, or they are remote and have to accept imprecise knowledge of what the other is doing at that very same moment in time.

The good news is that this need not be a problem. The simplest solution is to serialize actions via one common server. The server will, by the fact that it processes work sequentially, serialize incoming requests in order of arrival.

This does not prevent unintended clashes. It is as with long-distance phone calls: when delays are large (as in satellite-based sessions), you occasionally end up speaking at the same time as the other party. Right thereafter, you both realize that this won’t work, stop, and retry after a short while. Normal courtesy and occasional extra retries take care of the rest.

A similar mechanism is used in Tequila. Actions are transmitted to the server, which then broadcasts it to all clients – including the originating one. The result is that all clients receive all actions in the same order, and can maintain a 100% consistent state. In the case of accidental conflicting actions, one of them will end up preceding the other, all parties will see the result in the same way, and once they realize it they can react by undoing or repeating their action. That too will be sent to the server, and all clients see the corrective actions.

By sending all requests to a central server, a single consistent model is maintained. Clients can then deal with views by themselves, and never worry about synchronization. The worst that can happen are network delays, which is unnoticeable in the case of geographically remote clients (even a few seconds would usually go undetected!).

The controller side of all clients is synchronized by the fact that they cause no local effect but merely get sent to the server to wait in line and be accepted.

There are many issues related to synchronization that are beyond the scope of this paper. Suffice to say that when the user interaction is designed with potential latency in mind, the whole server-based approach works remarkably well. No nasty race conditions, no tricky recovery mechanisms are needed.

It’s all about the model

The key to designing collaborating applications with Tequila is to focus almost exclusively on shared state, and to avoid “telling everyone what to do”.

This is why the above chat example was in fact not such a good idea. If coded naïvely, the chat would not even present a consistent log to every client. This would be the case if clients decided to add their own entries to their log and only broadcast the entries to all the other clients.

In the code in appendix A, the chat avoids this problem. It uses the server to serialize, by sending a line to the server, which then broadcasts it to all clients, including the originating one. Once that request comes back in, the originating client adds it to the log window. Just as all other clients do, and in precisely the same order.

What the chat does is use the central server to serialize all events. Each client then maintains its own copy of the model. Note that the server does not do anything with the requests, other than sending them around.

In other words, the chat relies on all clients to diligently follow all events and each manages a replica of the data model. This breaks down for clients who join later (or drop off and have to reconnect for any reason).

That's where Tequila's "pools" come in. They are the model and reside on the server. When a client connects to a pool, it first gets a copy of the exact state on the server. After that, it receives all changes to the pool, using the same broadcast mechanism as before.

Pools solve the problem of connecting at arbitrary points in time. Due to their shared design, pools also manage all further changes – using RPC, but taking care of everything transparently. With pools, you are encouraged to think in terms of state, not actions!

In a way, pools are Tequila's way of using a model. Distance and connection re-establishment are solved.

A better example: rooms

Appendix B contains the code for a more elaborate example, written by Mark Roseman. It represents a groupware system, whereby clients connect to a server with "rooms". Each room has some state, each client can view that state as well as modify it. Any client can also create rooms themselves – the system starts out with no rooms at all.

The model here is very simple and clear: each room is a pool. There is a special pool called "system", which is where clients can find a list with the names of all rooms. A simple explorer-style GUI lets you add to that list, or select an existing room from it.

When you select a room by double-clicking, the right-hand pane shows the room. Rooms have a random color assigned to them at creation time (just so it's easy to recognize different rooms). Clicking on a room adds a timestamp in the position that was clicked.

Needless to say, every client can do this, and every client sees changes in real time.

This example is illustrative of how designs with Tequila work:

- Each room is a pool – clients only deal with one room at the time. Changes to others don't concern them (and are not sent).

- The design is completely in terms of state – there are no explicit RPC calls in the rooms demo. The code works the same regardless of which and how many other clients are around.
- Connects and re-connects are easy – each time a room is selected, the client gets the current state (i.e. a copy of its pool on the server). After that, it automatically sees all changes.

The transparency of the network should become clear when you realize that an application can be developed standalone, by using local pools. All it takes to turn this into a networked system later on is to alter the way pools are initialized (i.e. associated with a server).

Current status

Tequila is currently called T2 (it's at version 2.02 at the time of this writing). That reflects the fact that it is a second-generation implementation, taking the best from the original T1 as well as from GroupKit.

T2 has not been used in a production setting. It has passed a certain amount of testing, but there are still some design changes underway – such as being able to use multiple pools on a server via single RPC channel.

The basic concepts of pools, RPC endpoints, and notifiers are stable, however. This design is expected to stay essentially as is from now on.

Performance appears to be good, judging by a few timing experiments. All the code is in pure Tcl, so there will be considerable room for improvement if bottlenecks do show up at some point. In GroupKit, "environments" (a more elaborate concept on which Tequila's pools are based) were coded in C, and so were notifiers, but so far Tcl has worked out well.

The design of collections (i.e. the tabular data structures which can be stored in pools) is relatively complete, but a few loose ends remain with respect to positional access and inserting/moving/deleting rows.

Next steps – T3

T2 is really an intermediate phase to transition away from T1's array-centric design and prepare for a number of more advanced features:

- Scalability – by using collections, the need to keep all data in memory (as with arrays) is gone. T2 introduces pools and collections, and maps them to Metakit sub-views. This is a start to maintaining data in memory-mapped files, i.e. letting the operating system do what it does so well: decide on when/how to page data in and out of RAM, transparently.
- Client-side caching – a major limitation of T1 and T2 is that all data in an array/pool is sent to the client when it connects. This does not scale well. T1 had some hooks to avoid the copying, the price

being increased application complexity. T2 was designed to eventually support a cache, so that the client side only need fetch changes – often a fraction of the complete dataset.

- Transport independence – T2’s RPC objects make it possible to insert extra layers or even replace them with something entirely different. This can be used to add compression and/or encryption to the communication channel between clients and servers
- Multiple servers – T2 makes a start with supporting multiple servers. These can be used for redundant/fail-safe scenarios, as well as for maintaining pools in separate locations (for security, or intranet vs. public).
- Database gateways – the collections in T2 support a rich table-like structure, whereas T1 only supports key-value associative arrays. This can be used to create rich gateways to relational databases via a Tequila server.
- Tools – with a generic framework such as Tequila, it is worthwhile to create generic tools that can be used by different applications, both during development and in released code. Think of auditing, debugging/tracing, web interfaces, and report-generation.

The plans for T3 have only just started. A number of the issues mentioned above will be addressed, but since Tequila is 100% open source, there is nothing to prevent anyone from taking what they like and extending it in ways they need. Preferably in such a way that others can benefit as well, of course.

Looking further

Tequila is an exciting mix of ideas and trade-offs. On the one hand, it completely bypasses conventional approaches such as the use of N-tier client/server databases and the use of HTTP and XML solutions. Instead, Tcl’s strengths are exploited in several ways:

- Using Tcl as protocol over-the-wire
- Using asynchronous file events for all I/O
- Using notifiers, as an equivalent for traces
- Using Metakit for data storage
- Using Telkit and Starkits for deployment

And of course: having Tk around the corner for building user interfaces on top of all this.

For people who know Tcl, the above are probably not very exciting, but it’s hard to overestimate the importance of having all these capabilities come together in the way Tequila uses them. Not to mention that T2 is well under 1000 lines of code so far – offering proof that all the components really are working together to form a powerful way of developing collaborating applications.

But the promise of T2 is in fact a different one: if

you ignore all the details of storage and networking and GUIs (as you can with Tequila + Tk), then what remains is the way pools and collections become the (structured!) “data backplane” of an application, regardless whether it is a single standalone script or a large multi-process / multi-site distributed system. What remains, is the application’s core itself, and the business logic that really defines what is being done.

With T3, the hope is that this potential to simplify software development with Tcl will be exposed and exploited further, much further eventually. The change in mindset needed to accomplish this, is that everything is about state and consequences of state change. As Tequila proves, location, persistence, sharing, and application launches & exits can become details that will need much less attention than before.

Conclusion

When started in 2000, Tequila immediately proved itself by dramatically simplifying the application it was initially designed for. By dealing with the general issues in a generic way, a surprisingly effective separation of (complex) application logic as well as (complex) collaboration logic was achieved. As a result, Tequila (T1) has been used in a number of projects since, with considerable productivity gains – it helped get software projects out the door faster, and it came with a solid set of features, tested and ready to build on, and with.

The current T2 re-implementation has been completed. It introduces pools, RPC endpoints, and notifiers. It also introduces collections as a more general way of managing structured data. The API is starting to look good, in that it remains relatively simple yet offers all the benefits that T1 had – as well as opening up the path to more advanced features such as client-side caching.

T3, the next phase of this project has only just started. It is likely to leave much of what T2 does alone, and simply extend on what there is to implement additional functionality.

Acknowledgments

The design of T2 was shaped profoundly by discussions with Mark Roseman, GroupKit’s principal author. As a result, it appears to be simpler (and leaner) as well as more flexible. The holy grail of collaborating software is probably the idea of tying networking, persistence, and the graphical user interface together in as automatic a framework as possible. If T2 succeeds, it will be mostly due to Mark’s involvement and experience (and if it fails, yours truly did 95% of the coding, so you’ll know who to blame...).

I would also like to thank Steve Landers, for participating in many of T2’s discussions, and for always coming up with good ideas and questions.

The original work on Tequila would not have been

possible without Steve Landers and Cameron Laird, who worked on the project where Tequila was originally born. Between us, we came up with the whole idea and together we carried it through to make it a pretty effective product. I'd like to thank Larry Blasingame for making some of the toughest go-ahead decisions ever by placing so much trust in a bunch of excited software engineers, halfway across the globe.

And last but not least, I'd like to thank Mike Doyle and Eolas Technologies Inc, for funding my work done on T2 and for providing an exciting opportunity to take Tequila further for everyone, by supporting its public release as open source software.

References

Tequila home – <http://www.equi4.com/tequila.html>

The Tcl'ers Wiki - a collaborative web site for the Tcl community, <http://wiki.tcl.tk/>

GroupKit by Mark Roseman, this was part of the GroupLab project at the University of Calgary – now released independently, <http://www.groupkit.org/>

IncrTcl – Object oriented extension for Tcl, <http://incrtcl.sourceforge.net/>

Metakit – embedded database extension for Tcl (Mk4tcl), <http://www.equi4.com/metakit.html>

Tclkit – a standalone runtime for Tcl/Tk, includes IncrTcl and Metakit, <http://www.equi4.com/tclkit.html>

Appendix A – chat demo

```
# Tiny chat system: assumes generic tequila server is running (port 18396)
# Clients broadcast their msgs to everyone currently connected to the server.
```

```
package require Tk
package require tequila 2.02
```

```
set rpc [tequila::rpc localhost 18396 -command chatrecv]
```

```
grid [text .t -width 40 -height 10 -yscrollcommand ".s set"] \
      -column 0 -row 0 -sticky news
grid [scrollbar .s -command ".t yview"] -column 1 -row 0 -sticky ns
grid [entry .e] -column 0 -row 1 -columnspan 2 -sticky we
bind .e <Return> sendChat
```

```
proc sendChat {} {
    $::rpc send to all chatmsg [.e get]
    .e delete 0 end
}
```

```
proc chatrecv {conn data} {
    if {[lindex $data 0] eq "chatmsg"} {
        eval $data
    }
}
```

```
proc chatmsg {msg} {
    .t insert end $msg\n
    .t see end
}
```

Appendix B – rooms demo

client.tcl:

```
package require Tk
package require tequila

set rpc [tequila::rpc localhost 18396]
tequila::pool system $rpc

frame .rooms
button .rooms.new -text New -command newRoom
listbox .rooms.l -yscrollcommand ".rooms.sb set"
scrollbar .rooms.sb -command ".rooms.l yview"
frame .room
canvas .room.c -width 400 -height 400 -background #ccc
grid .rooms -column 0 -row 0 -sticky ns
grid .rooms.new -column 0 -row 0
grid .rooms.l -column 0 -row 1 -sticky news
grid .rooms.sb -column 1 -row 1 -sticky ns
bind .rooms.l <Double-1> enterRoom
grid .room -column 1 -row 0 -sticky news
grid .room.c -column 0 -row 0 -sticky news
grid columnconfigure . 1 -weight 1
grid rowconfigure . 0 -weight 1
grid rowconfigure .rooms 1 -weight 1
grid columnconfigure .room 0 -weight 1
grid rowconfigure .room 0 -weight 1
wm title . "<No Room>"

proc nextid {} {
    if ![info exists ::nextid] { set ::nextid 0 }
    expr {[system cget -clientid]+1}*1000000 + [incr ::nextid]}
}

system bind rooms.* roomsChanged
system bind connected roomsChanged

proc newRoom {} {
    set roomid [nextid]
    $::rpc send poolmanager create room$roomid
    system set rooms.$roomid name "Room $roomid" pool ::room$roomid \
        backgroundcolor [format "%02x%02x%02x" [expr int(rand()*256.0)] \
            [expr int(rand()*256.0)] [expr int(rand()*256.0)]]
}

proc roomsChanged {} {
    .rooms.l delete 0 end
    foreach i [system keys rooms] {
        .rooms.l insert end [system get rooms.$i name]
    }
}

proc enterRoom {} {
    set idx [.rooms.l curselection]
    set roomid [system get rooms!$idx key]
    if {[info exists ::currentroomid] && $roomid == $::currentroomid} return
    if {[info exists ::room]} {
        .room.c delete all
        .room.c configure -background #ccc
        bind .room.c <Double-1> {}
        $::room destroy
    }
    set ::currentroomid $roomid
    set ::room [tequila::pool room$roomid $::rpc \
        -servername [system get rooms.$roomid pool]]
}
```

```

    $::room bind connected roomEntered
}

proc roomEntered {} {
    wm title . [system get rooms.$::currentroomid name]
    set color [system get rooms.$::currentroomid backgroundcolor]
    .room.c configure -background $color
    bind .room.c <Double-1> { clickWorkArea %x %y }
    $::room bind notes.* { noteChanged %K }
    foreach i [$::room keys notes] { noteChanged $i }
}

proc clickWorkArea {x y} {
    $::room set notes.[nextid] x $x y $y text [clock format [clock seconds]]
}

proc noteChanged {key} {
    if {$key==""} return
    foreach {x y text} [$::room get notes.$key x y text] break
    if {[.room.c find withtag note$key] eq ""} {
        .room.c create text $x $y -text $text -tags note$key
    } else {
        .room.c coords note$key $x $y
        .room.c itemconfigure note$key -text $text
    }
}

```

server.tcl:

```

package require tequila

class poolMgr {
    variable rpc
    method constructor {rpc_} {
        set rpc $rpc_
    }
    method create {name} {
        uplevel #0 ::tequila::pool -server $name $rpc
    }
    method connect {name clientname} {
        # later use this instead of connectPool
    }
}

# mainline
set rpc [tequila::rpc -server 18396]
poolMgr poolmanager $rpc
[$rpc cget -receiver] allow poolmanager
tequila::pool -server ::tequila::system $rpc -file try.db
[$rpc cget -receiver] allow tequila::connectpool
catch { vwait forever }

```

Tequila

Jean-Claude Wippler



The Netherlands

Let's build a...

CHAT SYSTEM

network stats collector

PBX

wlan admin tool

backdoor?

groupware app

multi-player game

data backup manager

Wiki

home entertainment center

MP3 JUKEBOX

custom intranet utility

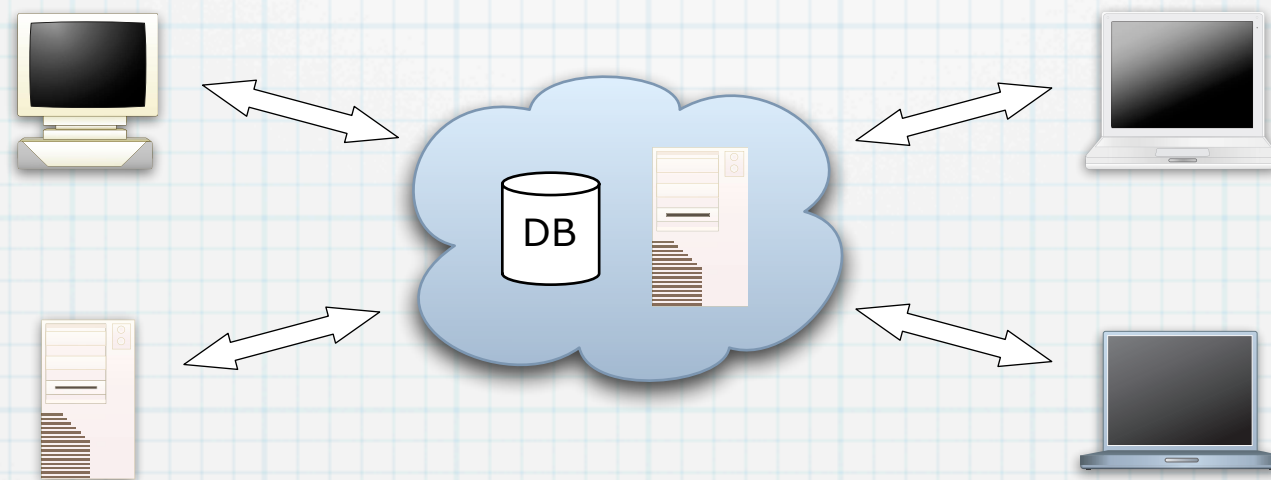
whiteboard

Collaboration

- * people - any time, any place
- * events everywhere
- * consistency
- * network failures

Overview

- * clients connect to permanent server



- * focus on star-shaped topology for now
- * must be multi-platform & responsive

Tcl/Tk

- * **an absolutely perfect match!**
- * event-driven I/O
- * everything is a string - EIAS!
- * platform independence
- * Starkit deployment

Tequila

- * general-purpose server
- * simple protocol: count + arg list
- * data stored on server:
 - * array = dir of files, or Metakit
- * client + server < 600 lines of Tcl

Version 1

- * shared global arrays:

- * connect

- * attach

- * traces

- * events

- * vwait

```
package require tequila
```

```
tequila::connect here.com 20000
```

```
tequila::attach mydata
```

```
set mydata($key) $value
```

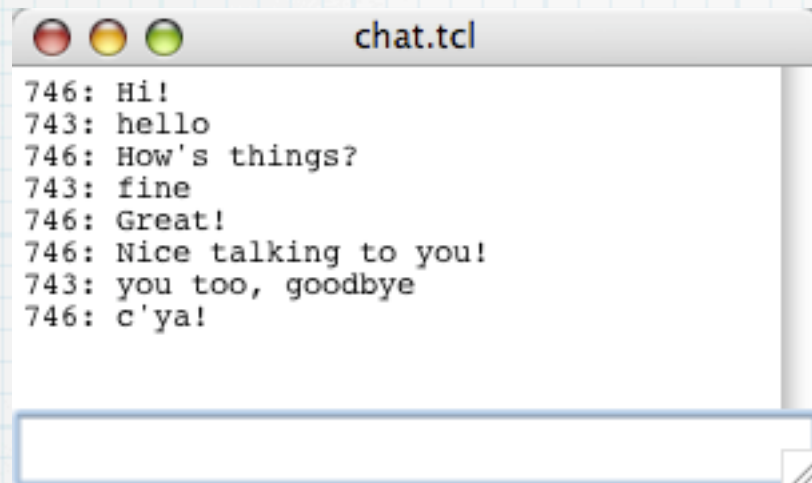
```
puts $mydata($key)
```

```
foreach x [array names mydata] {
```

```
    ...
```

```
}
```

Let's chat

A window titled 'chat.tcl' with a white background and a scroll bar on the right. It contains a list of chat messages.

```
chat.tcl
746: Hi!
743: hello
746: How's things?
743: fine
746: Great!
746: Nice talking to you!
743: you too, goodbye
746: c'ya!
```

A window titled 'chat.tcl' with a grey background and a scroll bar on the right. It contains the same list of chat messages as the first window.

```
chat.tcl
746: Hi!
743: hello
746: How's things?
743: fine
746: Great!
746: Nice talking to you!
743: you too, goodbye
746: c'ya!
```

A terminal window titled '%1' with a yellow background and a scroll bar on the right. It shows the execution of a Tcl script named 'tequilas.tcl'.

```
%1
biggie t!chat $ tequilas.tcl
Tequila server on port 20000 started.
Define data 0 S
GetAll data 1
Notification set up for 'data': sock6
Define data 0 S
GetAll data 1
Notification set up for 'data': sock8
Set data 746 Hi!
Set data 743 hello
Set data 746 {How's things?}
Set data 743 fine
Commit done (2710)
Set data 746 Great!
Set data 746 {Nice talking to you!}
Set data 743 {you too, goodbye}
Commit done (2455)
Set data 746 c'ya!
Closing sock8
Forget notify for data
Closing sock6
Forget notify for data
No more notifications for data
^C
biggie t!chat $
```

Look ma, no hands

```
package require Tk
package require tequila

tequila::connect localhost 20000
tequila::attach data

text .t -width 40 -height 10 -yscrollcommand ".s set"
grid .t -column 0 -row 0 -sticky nwes
grid [scrollbar .s -command ".t yview"] -column 1 -row 0 -sticky ns
grid [entry .e] -column 0 -row 1 -columnspan 2 -sticky we

bind .e <Return> {
    set data([pid]) [.e get]
    .e delete 0 end
}

trace add variable data write chatrecv

proc chatrecv {a e op} {
    .t insert end "$e: $::data($e)\n"
    .t see end
}
```


POTS network test

- * POTS = Plain Old Telephone System
- * central site: 250,000 modem calls/day
- * driven by periodic scheduler
- * engineers add (big!) ad-hoc tests
- * Windows & Solaris clients
- * built 100% with Tcl/Tk (Tclkit) & Tequila

It works!

- * plenty of performance
- * ran for 18 months, no restart
- * Tk-based custom client app suite
- * organically grown by 3 programmers
- * Tequila was a spin-off, re-used later

The good

- * **forget about distribution details**
- * **persistence is automatic**
- * **the API is trivial**
- * **split & merge sub-tasks as needed**

The bad

- * **arrays are hard to virtualize:**
 - * **array names**
 - * **all data must be present**
- * **single server**
- * **weak data structures**
- * **lack of RPC**

The ugly

- * sends all data on attach:
 - * does not scale
- * debugging traces is messy
- * reentrant traces are lost
- * no reconnect / recovery

The Fix

- * richer data structures: pools
- * transport layering: rpc endpoints
- * alternative to traces: notifiers
- * client-side caching: Metakit DB
- * implemented in phases: T2, then T3

Tequila T2

- * learn from T1 and from GroupKit:
 - * keep it simple
 - * keep it simple
 - * keep it simple
- * only requires Tclkit or ActiveTcl
- * code is still 100% Tcl, so far

The big picture

- * connect to your server via an endpoint
- * keep your data in one or more pools
- * shared pools are kept in sync by Tequila
- * changes to pools generate notifications
- * use bindings to act on incoming changes
- * your own changes also come that way

MVC

- * **separation of responsibilities:**
 - * **Model** = the “state” of your app
 - * **View** = ways of presenting it
 - * **controller** = effects of user actions
- * **very effective for distributed apps**

Pools

- * a pool is a logical group of collections
- * a “collection” is a table with data:

key	value
Mary	London
Bill	Paris
John	Berlin

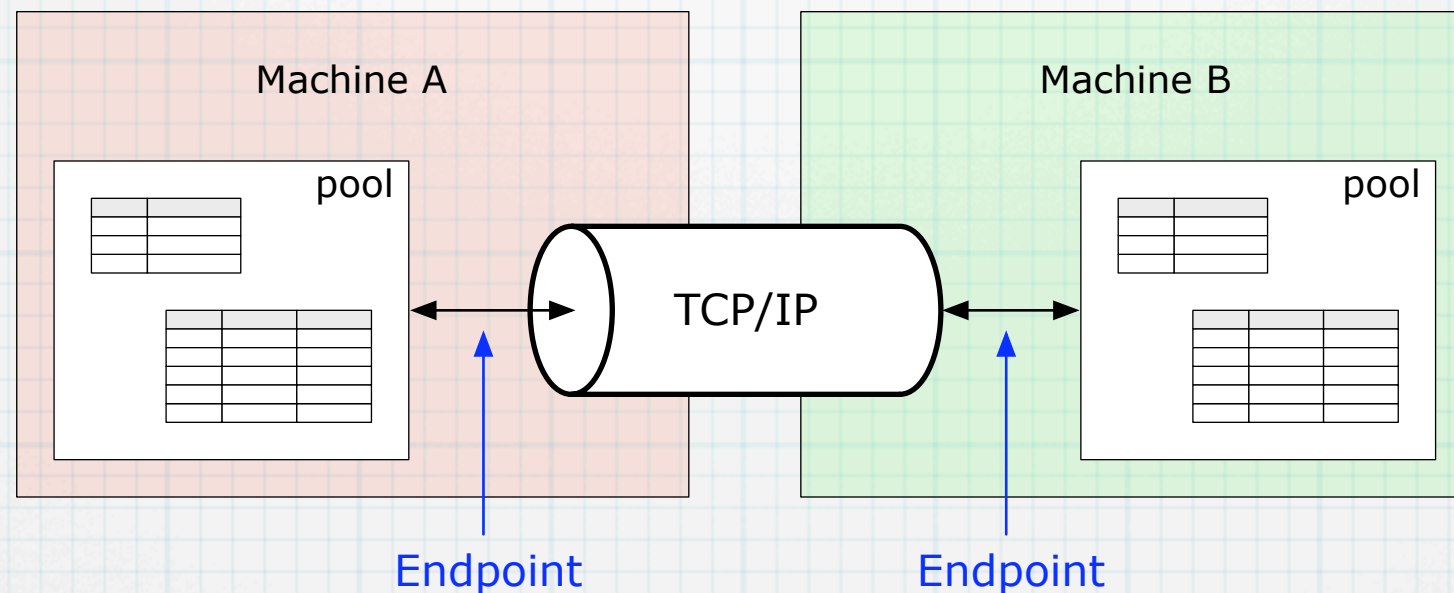
- * by key or by row#: multiple attributes:

	key	location	since
#0	Mary	London	1956
#1	Bill	Paris	2001
#2	John	Berlin	1989

- * represents a “sharable unit”

RPC endpoints

- * the interface “in & out of the tunnel”:



- * client usually has 1, server has many
- * new: can be shared for multiple pools

Notifiers

- * **generic event mechanism:**

```
$notifier bind <event> <script>  
$notifier notify <event> <info>
```

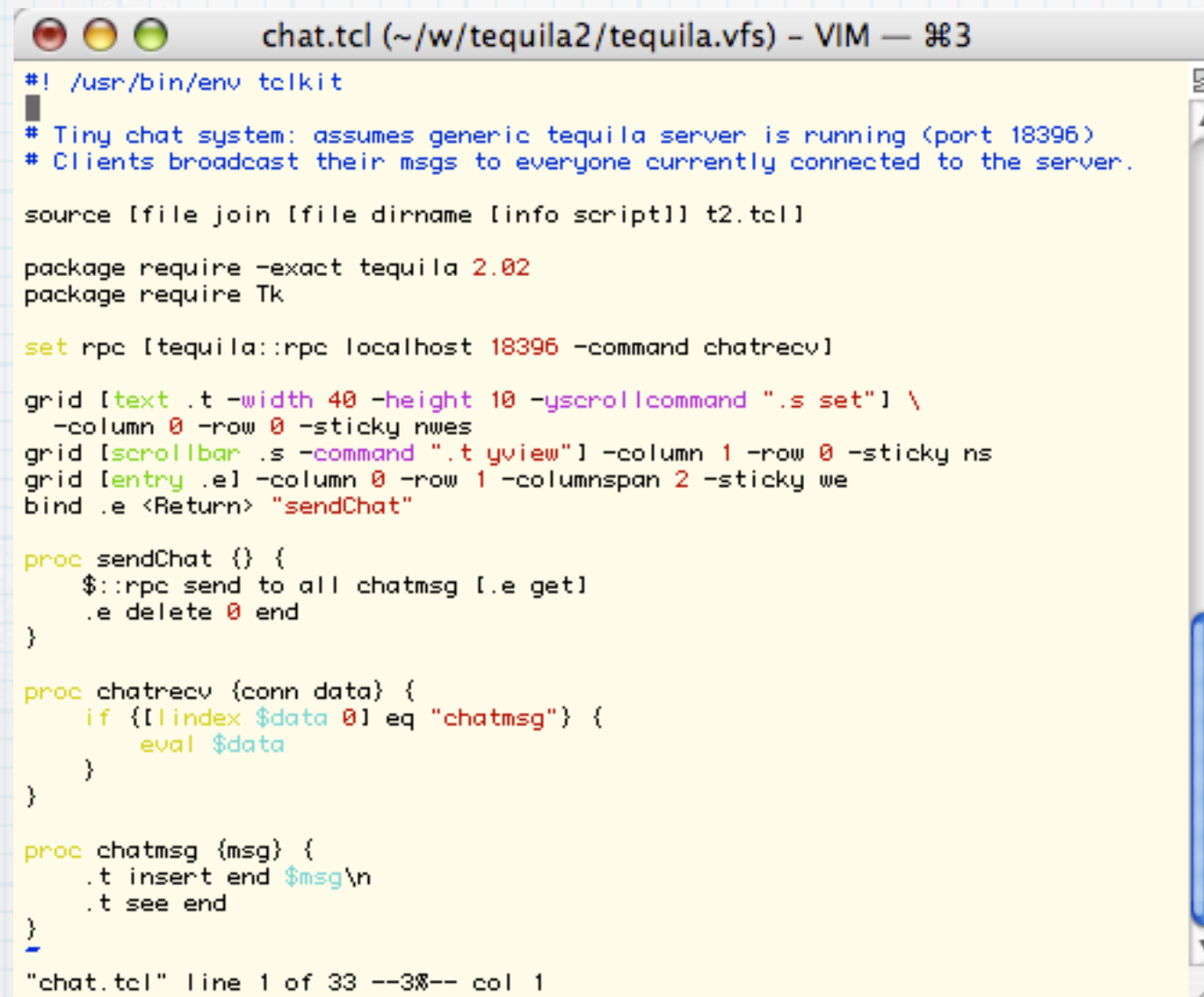
- * **like Tk's bind, but for general use**

- * **arbitrary "%x" expansions**

- * **bind to wildcard events, using ? and ***

- * **each pool has its own notifier**

T2 chat



```
chat.tcl (~/w/tequila2/tequila.vfs) - VIM — 3
#!/usr/bin/env tclkit
# Tiny chat system: assumes generic tequila server is running (port 18396)
# Clients broadcast their msgs to everyone currently connected to the server.

source [file join [file dirname [info script]] t2.tcl]

package require -exact tequila 2.02
package require Tk

set rpc [tequila::rpc localhost 18396 -command chatrecv]

grid [text .t -width 40 -height 10 -yscrollcommand ".s set"] \
     -column 0 -row 0 -sticky nwes
grid [scrollbar .s -command ".t yview"] -column 1 -row 0 -sticky ns
grid [entry .e] -column 0 -row 1 -columnspan 2 -sticky we
bind .e <Return> "sendChat"

proc sendChat {} {
    $::rpc send to all chatmsg [.e get]
    .e delete 0 end
}

proc chatrecv {conn data} {
    if {[lindex $data 0] eq "chatmsg"} {
        eval $data
    }
}

proc chatmsg {msg} {
    .t insert end $msg\n
    .t see end
}

"chat.tcl" line 1 of 33 --3%-- col 1
```

Current status

- * **Tequila 2.02**

`http://www.equi4.com/pub/sk/tequila.kit`

- * **several demos:**

chat, checkers, chess, lines, calendar

- * **manual page**

- * **working, but rough - mostly API demo**

T3

- * **planned later in 2005**
- * robust, test suite, docs, examples
- * client-side caching - instant startup
- * thin clients & software self-updates
- * connect over HTTP - detached use?
- * security - Steve Landers' CryptKit

Beyond...

- * the “killer” net-enabled app platform
- * some ideas:
 - * web interface on server
 - * collaborative app development
 - * server as gateway to other DB's
- * anything - it's all 100% open source

Switching to Tile

Rolf Ade

April 2005

Abstract

The Tile package is probably the most thrilling and ambitious effort to revitalize Tk so far. Tile adds new abilities to control and change the look and feel of Tcl/Tk applications with so-called 'Themes'. Especially, tile provides native looking widgets at Windows and Mac OSX. Plus the package provides some additional widgets. This paper gives an overview over tile at version 0.6.2 and tries to guide Tcl/Tk developers thru the first steps of using it¹.

1 Overview

The tile package had an astonish career. Less then two years ago, Joe English made the first sources public available. Today, although the version number suggest, that tile is still in its early days, a few Tcl Core Team Members are semi-official committed to push the inclusion of tile as a bundled package of the upcoming 8.5 Tcl/Tk release². If that should in fact happen, that wouldn't be a radical change. The tile package is in some sense orthogonal to the current Tk core - just don't use it, and you get the familiar Tk.

At the first look, tile is in short about 'eye candy'. The debate about the look and feel of Tk has a long history. In the early days of Windows and Mac support (version 7.5), Tk looked the same on every supported platform. Later on (starting with 8.0), Tk used some platform specific controls under Windows and Mac. But the world moved on. Windows XP, Mac OSX and under Linux KDE and Gnome brought theming support on the desktop - and that renewed the look and feel problem for Tcl/Tk application programmers. For example, while it is possible with some effort, to write a Tcl/Tk application, which looks nearly native on Windows 2000, this isn't really possible on Windows XP - the Tcl/Tk application will look 'foreign' on this platform.

Tile aims to solve this problem by theme-able reimplementations of the core

¹The author has followed the development of and discussion about tile from early states on but he isn't involved so far into its development. This is a report from an outside viewpoint and like an article about using a new technology. At the Eleventh Annual Tcl/Tk Conference 2004, Joe English presented a paper more from a tile developers view point[4].

²Though there is up to now no TIP, which proposes that really officially.

widgets³. The basic idea of this reimplementaion is, to separate the code responsible for the appearance of a widget from the other parts of the implementation. Selecting a theme means then just switching from one widget drawing code to another behind the scene.

But from where does this code come? Well, sure, it has to be written. Much better: that code is already written and is part of tile. For the first steps, Joe Tcl/Tk Programmer need not to care about creating complete new themes. For the ambitious, this is of course possible. New themes can be implemented as add-on packages written in Tcl or in C, depending on the level of customization required. But on windows, there is already a theme **xpnative**, which uses the Windows "Visual Styles" API to make tile widgets indistinguishable from native controls. And on Mac OSX is the **aqua** theme available, which uses the Carbon Appearance Manager. The default theme has a new, streamlined look, compared with Tks current Motif-like appearance on X11 (which is also available as theme **classic**). And there are more themes included⁴.

Beside all that 'eye candy' it should not be forgotten, that tile provides a handful additional (of course theme-aware) widgets. One additional gift, that tile *may* bring is a new kind of meta-widget framework, but that word isn't spoken yet.

2 Getting started

The tile sources are hosted under the umbrella of the TkTable project on SourceForge[5]. Given, that a recent Tcl/Tk version is installed (the current Tcl/Tk 8.4.9 will do well) building under linux and probably any other sane unix implementation is just a matter of **configure; make all; make install**. For compiling on Windows, a VC++ makefile is included. The tile download page at SourceForge provides a multi-platform stargit. Additionally, tile binaries are provided by (among others) ActiveStates ActiveTcl distribution[1] and Daniel Steffens TclTk Aqua Batteries-Included distribution[8]. Since version 0.6 every released version has an even last number. Version numbers with an odd patch number indicate CVS snapshots. So, the next release will have at least the version 0.6.4.

Tile is a well-behaving Tcl package. To use it, just put

```
package require tile
```

near the top of your main script. The scripted library code of the package selects at package loading time depending on the platform the 'right' theme (that is **xpnative** on Windows XP, **winnativ** on other versions of Windows, **aqua** on Mac OSX and the **default** theme on X11⁵). There is always exactly one theme

³To do this, tile uses the Tk theme engine. Without much notice outside the tcl-core mailing list Frédéric Bonnet laid the ground of Tk theming support with TIP 48[3]. Tile is the first known package, which uses this theme engine. The tile developers revised and enhanced the theme engine on the way.

⁴In fact, early adaptors have already written 3-party themes. Most of them are scripted themes, but Georgios Petasis has provided the start of a new C level theme, which instruments the Qt styling engine, to draw the widgets[7]

⁵If not otherwise set in the X resource database.

in effect at any one time. The proc `tile::availableThemes` returns a list of all available themes. The name of the current theme is stored in the variable `tile::currentTheme`. To switch the current theme, use the helper proc⁶

```
tile::setTheme <theme name>
```

When you switch themes, the tile widgets are redrawn automatically with their new look - unlike Tk, where changes to the option database with `option add` let existing widgets alone and change only the appearance of newly created widgets.

The tile reimplementation of the Tk core widgets are implemented in the `ttk` namespace and have the same name as their Tk counterparts. It's a nice experiment to locally override the Tk core widgets in your application's namespace(s) with their tile reimplementations with

```
namespace import ttk::*
```

Normally, your application should at least start in a new look and even mostly work as normal. But in most cases it will also be obvious, that this was not completely all work to do to switch to tile. Before we discuss the most common problems with the migration to tile, we take a closer look at how tile works.

3 How styling works

Programming GUIs with Tk is like writing a text in a word processor as OpenOffice or Microsoft Word. As the writer may change the font, foreground, background etc. of every text item, so has the Tk programmer with the help of tons of options the control over the appearance of the Tk widgets. Tile is more like a markup language as \LaTeX . Lots of appearance details like border, font, foreground, background etc. are handled by the chosen theme. Even more: the tile widgets doesn't allow to change a lot of this appearance aspects with widget options any more. Though, for compatibility reasons (to make migration easier), the tile widgets 'know' all the options of their corresponding Tk widgets, but they ignore some of them. Figure 1 shows this in detail for the Tk and tile button widgets.

As familiar from the Tk widgets, the default behavior of the tile widgets is controlled by the widget class bindings. The tile reimplementations of the Tk core widgets have the same class name as their counterparts, prefixed with a 'T' (so, the tile button class is `TButton`, the tile entry class is `TEntry` and so on). The class bindings are independent from the theme - switching themes doesn't affect class bindings. In Tk, only the 'container widgets' `toplevel`, `frame` and `labelframe` have a `-class` option, to set the widget's class at creation time. In tile, every widget has a `-class` option. That makes it easier, to create

⁶It is not recommended to use `style theme use <theme name>`, because `tile::setTheme` loads the theme, if necessary and keeps the variable `tile::currentTheme` up to date. The latter is necessary, because the natural `[style theme use]` unfortunately doesn't return the current theme so far.

Options common to Tk 8.4.9 button and tile ttk::button		
-command	-compound	-cursor
-default	-image	-takefocus
-text	-textvariable	-underline
-width		
Tile ttk:button options present for compatibility, but ignored		
-activebackground	-activeforeground	-anchor
-background (and -bg)	-bitmap	-borderwidth (and -bd)
-disabledforeground	-font	-foreground (and -fg)
-height	-highlightbackground	-highlightcolor
-highlightthickness	-justify	-overrelief
-padx	-pady	-relief
-repeatdelay	-repeatinterval	-state
-wraplength		
New tile ttk::button options		
-class	-padding	-style

Figure 1: Tk 8.4 button options versus Tile button options

different behaving controls based on the same widget. The tile distribution has the custom widget class **Repeater** (to be used for tile buttons) as an example.

The **-style** option of the tile widgets may be used to specify a custom widget style.

3.1 Widget Elements

With Tk, the widgets are the 'atoms' of the GUI. The tile widgets are not monolithic blocks, but itself build from smaller, simpler parts, the 'quarks' of a widget or, as they are officially called, the widget elements. For example, the Windows-style **button** has a border, a focus ring and a label, each of which are distinct elements.

Widget elements have options very much like widgets. For example, the default border element has **-borderwidth** and **-relief** options (in fact, most of the options, which have disappeared from the widgets will be found as element options).

Widget elements are usually implemented in C. The tile core provides a default set of elements. Every theme inherits this core elements, but it may overwrite the drawing code of a part or all elements with its own implementations (to get a different visual representation), with more or less or other element options. A theme may even add new elements. Elements can also be defined from Tk images to create pixmap themes.

While

style element names

returns the list of elements defined in the current theme, there isn't as of version 0.6.2 any other way than source code diving to know the valid options of that elements⁷. To give an overview, Appendix A has a table of all core elements with their valid options.

3.2 Widget layouts

Since tile widgets are composed of a collection of elements, that elements has to be layed out somehow to build the widget — this is done by style layouts⁸. Every theme may build a tile widget out of more or less elements and/or may arrange the elements in a different way. For example, the `classic` theme layout of the `scrollbar` widget places an arrow button on each side of the scrollbar, while this sample code by Joe English places one arrow button on the left and two arrow buttons on the right side:

```
style layout Horizontal.TScrollbar {
    Scrollbar.trough -children {
        Scrollbar.leftarrow -side left
        Scrollbar.rightarrow -side right
        Scrollbar.leftarrow -side right
        Horizontal.Scrollbar.thumb -side left -sticky ew
    }
}
```

The arrangement of elements work like a simplified version of Tk's `pack` geometry manager. It's even possible, to adjust the layout of a tile widget after creation.

3.3 Styles and States

Under the umbrella of a style are all the settings collected, which affects the appearance of one group of widgets, (normally) all of the same class. Styles are named, in a hierarchical way. For example, the programmer could use the style name `Toolbar.TCheckbutton` to collect all special settings needed for checkbuttons used in a toolbar. All settings, not explicetely set by the `Toolbar.TCheckbutton` style are looked up in the `TCheckbutton` style. If there are still not explicetely set options, then the settings for the 'root' style `.` will be used, and that style has always a default value for every option, per implementation.

For example, to set the default relief for the example customized style `Toolbar.TCheckbutton` simply use

```
style default Toolbar.TCheckbutton -relief flat
```

⁷The next tile release will allow to query the name of the options of a given element with `style element options <element name>`.

⁸The layout system is one of the enhancements of the TIP 48 style engine made by the tile developers.

But the value of a style option isn't just a static value. The value of a style option for a certain widget depends on the state of that widget. Every tile widget has a map of several flags, currently:

- `active`
- `disabled`
- `focus`
- `pressed`
- `selected`
- `background`
- `alternate`
- `invalid`
- `readonly`

Every state flag is independent from each other. Every tile widget has the widget commands `state` and `instate` to modify and query any combination of that flags. Now, so called 'state maps' could set specific values for every option of a style for any possible combination of states. While this is a powerful concept, the simple cases are easy to understand:

```
style map Toolbar.TCheckbutton -relief {
    disabled flat
    selected sunken
    pressed sunken
    active raised
}
```

If a widget state is changed, then the state map of its style is searched for the first combination of states, that matches. If there is a match, that value is used for the option. If there isn't a match, the default value will be used. Widget state changes usually happen in widget class bindings like:

```
bind TCheckbutton <Enter> { %w state active }
```

3.4 Themes

A theme is a named umbrella for widget elements, layouts and styles. The layouts arranges the elements to widgets and styles control the visual appearance of that widgets. At scripting level, a 3-party tile theme is an ordinary Tcl package, with the package name `::tile::theme::<theme name>`.

4 Migration Problems

It's the details, that matters. While it is often quite easy to start to migrate an application to tile (as shown above), there are also typically some issues to solve. One really obvious problem is, that tile widgets and Tk widgets often doesn't look good side by side in one dialog.

Currently, tile provides reimplementations of the following Tk widgets:

- `button`
- `checkbutton`
- `entry`
- `frame`
- `label`
- `labelframe`
- `menubutton`
- `radiobutton`
- `scrollbar`

Also already in the code (and at least basically working) is the start of a themed `scale` widget, but that isn't currently documented. Not as a replacement (which mimics the interface) of the Tk `panedwindow`, but as a similar widget there is the `ttk:paned` widget.

That means, a few Tk widgets currently haven't a theme-able tile counterpart. There isn't much problem with the `canvas` widget — it simply doesn't need themability. Mostly the same is valid for the `text` widget⁹. But even if we rule this out that means, that theme-able counterparts of the `menu` and the `spinbox` (and eventually `listbox`) widgets are currently missing pieces. If you have, for example, a `spinbox` in your otherwise tile-ified dialog, which looks foreign like a blain, there isn't currently much you can do other than adjusting the `spinbox` options as possible or even rewriting the dialog without the `spinbox`.

Unfortunately, it is as yet not easily possible, to query the styles of the current or other themes, to get, for example, the default font for `entry` widgets. This makes is harder, to adjust the option settings of Tk core widgets as close to the current theme as it may be possible.

Probably even more important also most scripted meta-widgets (like, for example, the popular BWidget package) and additional C coded widgets (like the BLT toolkit[6]) will look foreign side by side with tile widgets. For the most common non Tk core widgets like `combobox` and `notebook` widgets, tile provides its own theme-able versions. But even if they fit feature-wise, using them means rewriting parts of the GUI code.

⁹Though it may asked, why for example the font of `entry` widgets is controlled by the theme, but the font of a nearby `text` widget is not.

Tile has merged the `-padx` and `-pady` options into a single `"-padding"` option, which may be a list of up to four values specifying padding on the left, top, right, and bottom.

Another minor interface difference, which may require a bit code editing, is that the tile `label` widget has `-background` and `-foreground` switches (which overwrite the theme defaults), but the Tk shortcuts `-bg` and `-fg` are only present as unused backward compatibility switches.

For all tile widgets with a `-compound` option the the `-width` option always specifies the width in characters to allocate for the text string. In Tk, it's either the width in characters, or in screen units, depending on the value of `'-compound'`, for the widget.

Even the widget commands of the tile widgets are only *mostly* compatible with the corresponding Tk widgets. A few widget commands are not implemented yet. Figure 2 has the complete list. Really important are probably only the missing `checkbutton` and `radiobutton` widget commands. The obvious work-around is, to use the `-variable` option and then to change the associated variable value.

Widget	Missing widget commands
button	flash
checkbutton	deselect flash select toggle
radiobutton	deselect flash select
scrollbar	activate

Figure 2: Tk widget subcommands not yet implemented by tile

We've already discussed a few times, that the tile widget's appearance isn't specified on a per-widget base with options but is controlled by the settings of the current theme. That means, if you had, for example, an important red button somewhere in your application with the help of the `-background` option, this button looks like any other button in your application, if you use the tile `button` widget. This isn't a bad thing. It's a good basic rule for standard applications, to use standard controls and a standard appearance - and red buttons are not really common. Tile enforces this principle by design¹⁰. But what, if this is all good and well for you, but you need for whatever reason just that red button? A solution is, to sub-class the style of your widget:

```
style default Red.TButton -background red
::ttk::button .redbutton -text "The Red Button" -style Red.TButton
```

¹⁰Although, tile allows you also to give your application a special visual 'branding', which emphasis how outstanding your work is, by creating your own theme.

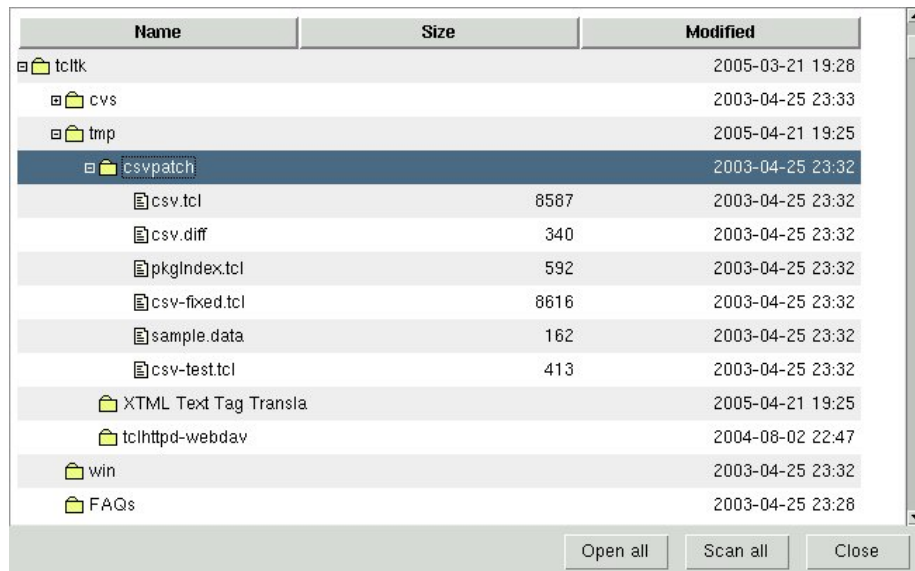
Depending on the style you sub-class you may also need to adjust the style map of your new style.

5 Additional widgets

Beside the styled counterparts of some of the core widgets, the tile package provides a few additional widgets. None of them is really novel. In fact, most of them are desired by Tcl/Tk developers since years and therefor there are (often several) alternative Tcl scripted meta-widget or even C coded implementations available.

For the early adaptors, which are already switching an existing code base to tile or writing a new application with it, especially the tile **combobox**, **notebook** and **progressbar** widgets are very helpful, because the current available scripted counterparts doesn't fit well into an otherwise tile-ified GUI. With the **classic** or **default** theme, the additional widgets are well usable within an otherwise 'pure classic' Tk application.

The tile **combobox** is, well, an entry field with an associated pop-down single-selection listbox. It seems to be thought-out comparatively mature. The tile demo directory even has a simple version of an inline auto-completion code. The tile **notebook** widget is a simple, single-tier notebook widget, similar to BWidget notebook. The **progressbar** widget supports two modes. The **determinate** mode shows the amount completed relative to the total amount of work to be done, and the **indeterminate** mode provides an animated display to let the user know that something is happening.



Name	Size	Modified
tcltk		2005-03-21 19:28
cvs		2003-04-25 23:33
tmp		2005-04-21 19:25
csvpatch		2003-04-25 23:32
csv.tcl	8587	2003-04-25 23:32
csv.diff	340	2003-04-25 23:32
pkgIndex.tcl	592	2003-04-25 23:32
csv-fixed.tcl	8616	2003-04-25 23:32
sample.data	162	2003-04-25 23:32
csv-test.tcl	413	2003-04-25 23:32
XHTML Text Tag Transla		2005-04-21 19:25
tclhttpd-webdav		2004-06-02 22:47
win		2003-04-25 23:32
FAQs		2003-04-25 23:28

Figure 3: A screen-shot of the treeview demo script demobrowser.tcl

The tile **separator** widget is a simple Widget, very much like the BWidget

separator (and probably any other separator) widget. By default, it has no behavior in response to the user and just displays a horizontal or vertical separator bar. It is typically used for visual grouping of toolbars, but the tile demo has also an example for using it to visually structuring a dialog. The tile `treeview` widget (see Figure 3) is at the moment a much simple widget, than the also C coded `TkTreeCtrl` widget[2] or, for the tree part, the scripted `BWidget` tree widget. It can be used like a pure tree widget (without headings and columns). If it fits feature wise, its simple interface is an advantage. On a recent computer, this widget is able to handle up to a few hundred thousand tree nodes in fairly low time and with moderate memory needs. That means, it is able to handle a lot bigger trees than `BWidget` tree. The `treeview` widget currently support only `-yscrollcommand`, there is no `-xscrollcommand`. Multi-line entries in a `treeview` column cell doesn't really work as yet.

A Tile Core Elements

Tile Core Element	Options
Checkbox.indicator	-background -borderwidth -indicatorcolor -indicatordiameter -indicatormargin -indicatorrelief
Labelframe.text	-background -embossed -font -foreground -justify -text -underline -width -wraplength
Menubutton.indicator	-background -borderwidth -indicatorheight -indicatormargin -indicatorrelief -indicatorwidth
Progress.bar	-background -borderwidth -orient -sliderlength -sliderrelief -width
Radiobutton.indicator	-background

	-borderwidth -indicatorcolor -indicatordiameter -indicatormargin -indicatorrelief
Treeheading.cell	-background -rownumber
Treeitem.indicator	-diameter -foreground -indicatormargins
Treeitem.row	-background -rownumber
arrow	-arrowcolor -arrowsize -background -borderwidth -relief
background	-background
border	-background -borderwidth -relief
client	-background -borderwidth
downarrow	-arrowcolor -arrowsize -background -borderwidth -relief
field	-borderwidth -fieldbackground
focus	-focuscolor -focusthickness
hsash	-sashthickness
hseparator	-background -orient
image	-background -image -stipple
label	-anchor -background -background -compound -embossed -font

	-foreground -image -justify -space -stipple -text -underline -width -wraplength
leftarrow	-arrowcolor -arrowsize -background -borderwidth -relief
padding	-padding -relief -shiftrelief
pbar	-background -bar-size -borderwidth -orient -pbarrelief -thickness
rightarrow	-arrowcolor -arrowsize -background -borderwidth -relief
separator	-background -orient
slider	-background -borderwidth -orient -sliderlength -sliderrelief -width
tab	-background -borderwidth
text	-background -embossed -font -foreground -justify -text -underline -width

	-wraplength
textarea	-font -width
thumb	-background -borderwidth -orient -relief -width
trough	-borderwidth -troughcolor -troughrelief
uparrow	-arrowcolor -arrowsize -background -borderwidth -relief
vsash	-sashthickness
vseparator	-background -orient

References

- [1] ActiveState. Activetcl.
<http://www.activestate.com/Products/ActiveTcl>.
- [2] Tim Baker. Tktreectrl.
<http://sourceforge.net/projects/tktreectrl>.
- [3] Frédéric Bonnet. Tip 48: Tk widget styling support.
<http://www.tcl.tk/cgi-bin/tct/tip/48>.
- [4] Joe English. The tile widget set.
<http://tktable.sourceforge.net/tile/tile-tcl2004.pdf>, 2004.
The so far only published paper about tile from one of the main tile makers.
- [5] Joe English Pat Thoyts et al. The tile package.
<http://tktable.sourceforge.net/tile/>.
- [6] George A. Howlett. Blt toolkit.
<http://blt.sourceforge.net>.
- [7] Georgios Petasis. The tile-qt theme.
<http://cvs.sourceforge.net/viewcvs.py/tktable/tile-themes/tile-qt/>.
- [8] Daniel Steffen. Tcltk aqua batteries-included.
<http://tcltkaqua.sourceforge.net>.



Making Tcl "legacy" code object oriented

Artur Trzewik
mail@xdobry.de

<http://www.xdobry.de/xotclIDE>

5. European Tcl Workshop 2003 - Bergisch Gladbach



Motivation

- OO in Tcl still hot discussed
 - Posting „Do you want OOP?” in comp.lang.tcl 31 items
 - Wiki „Poll: do you want OOP?” - 131 times edited
- Growing popularity of XOTcl (part of Active State Tcl Distribution)
- Still many questions and unclear answers?



Agenda

- Examples: OO-like code in pure Tcl
- Porting an Wiki example to XOTcl by using XOTclIDE
- XOTclIDE – what is new.
- Pro & Contr for OO with Tcl



Tk is object oriented

We can use window as object

```
button .tl.button
.tl.button configre -text „new text“
destroy .tl.button
```

How it works internal

```
proc ::tk::dialog::file::chooseDir::DblClick {w} {
    upvar ::tk::dialog::file::[wininfo name $w] data
    set selection [tk::IconList_Curselection $data
        (icons)]
    ...
}
```



Variant – global arrays

```
proc callOnObject {structureRef} {  
    uplevel #0 $structureRef myData  
    set attr $myData(attr)  
  
}
```

Most solution use arrays as primary structure.
Also global lists are possible.

Helper – Lset, keyed lists (tclx), dict (Tcl5)



Variant – arrays in caller context

```
proc addNode {_g node args} {  
    upvar 1 $_g g  
    set id [llength $g(nodes)]  
    set g($id) [concat $node $args]  
    lappend g(nodes) $id  
    set id  
}
```



Bigger Example

- Package for tree processing.
- Source. Wiki (<http://mini.net/tcl/1664>)
Simple tree layout by Richard Suchenwirth
- Clear pattern

Migration to XOTcl

Original code

```
proc terminals _g {
    upvar 1 $_g g
    set res {}
    foreach i [nodes g] {
        if {[sons g $i]==""} {lappend res $i}
    }
    set res
}
```

migrated to XOTcl

Class OTree

```
OTree instproc terminals {} {
    my instvar g
    set res {}
    foreach i [my nodes] {
        if {[my sons $i]==""} {lappend res $i}
    }
    set res
}
```



Tcl/XOTcl – client view

Original code

```
graphInit g
treelist2graph {A {B C D} {E F G}} g
sons g B
```

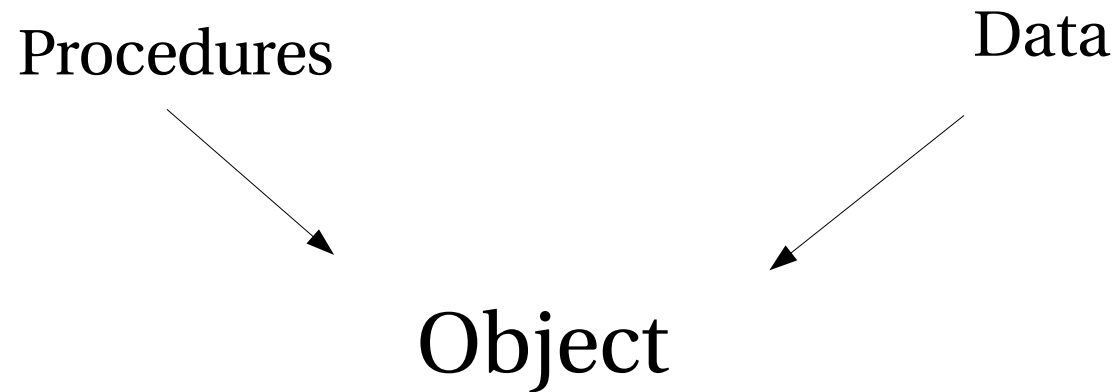
migrated to XOTcl

```
OTree g
g treelist2graph {A {B C D} {E F G}}
g sons B
```

Also with pure Tcl is easy to use oo-like calls. Just define structure reference as procedure and forward commands . In this case reference management should be hidden from client.

```
proc $ref {name args} {
    eval $name $ref args
}
```

Idea of OO



Data structures are hidden for Object user.



OO Programming vs. OO Programming Language

OO Programming with not OO Language

Tk, Tix, GTK (even with heritage)

C++ was initially compiled to C

Not OO Programming with OO Language

Anti pattern – Blob, Ghost, many C++ and Java projects.

DataSet in .NET

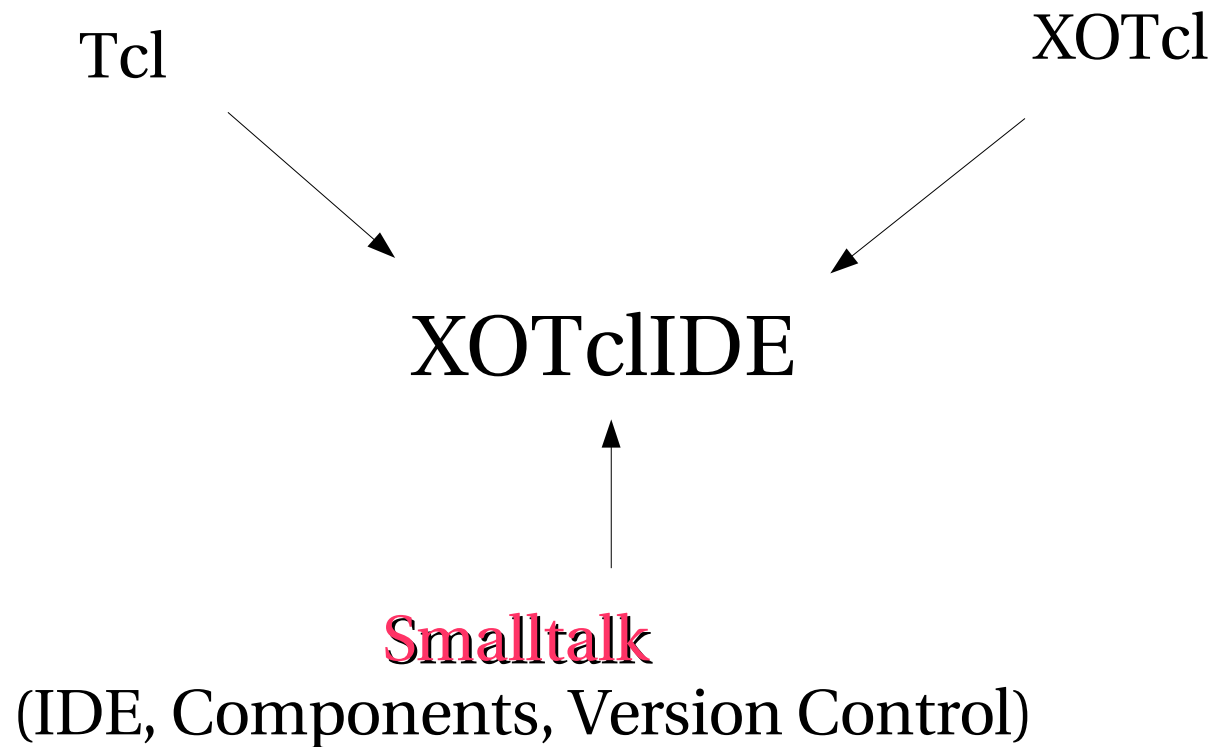
Object orientation is the way to think about programming.



OO in Tcl vs. mainstream OO

- not static typed
- more dynamic (introspection=reflection)
- more consistent (class is object – XOTcl)
- better for Aspect Oriented Programming
- better for Pattern (with filters and mixins)
- more similar to Ruby, Smalltalk, Self, Lisp CLOS as to Java, C++, C#

XOTclIDE





XOTclIDE - news

- Starkit support
- New databases (Oracle, Access, sqlite) for version control
- New Plugins
- Usability improvements (by Michael Heka and Fabrice Pardo)
- Supports new XOTcl features.



Migrating Tcl – live

- Importing code
- Creating components
- Migrating to XOTcl
- Adaptations
- Tests



Advantages of OO

- Better and clear structure
- Lifetime support (no memory leaks)
- Good for complex structures (no need for references)
- ? More Productivity and Reuse
- Better for complex projects
- Additional Features (mixins, filters, delegation, assertions, ...)
- No reinventing the wheel



Disadvantages of OO

- Additional Extension needed
- Less portability (jacl, jim, tclsharp)
- ? Performance
- Many extensions (XOTcl, ITcl, Snitt, Stoop, Classytcl)



That's all!

Questions ?
Fragen, Anregungen?
Demandoj?

<http://www.xdobry.de/xotclIDE>

Relational algebra for Tcl: introducing Ratcl and Rasql

Jean-Claude Wippler

Equi4 Software

jcw@equi4.com

ABSTRACT

There are a number of ways to manage data in Tcl, from native lists and arrays to various database bindings. The choice involves trade-offs regarding persistence, robustness, performance, memory use, query capabilities, scalability, portability, standards conformance, and convenience. This is an issue even for simple scenarios, given Tcl's limited support for data structures. A new approach will be presented which is based on relational algebra, supplied through two packages: Ratcl, which provides a relational algebra extension that fits naturally with Tcl. And Rasql, which provides a layered SQL interface for those who prefer it. Both are based on over a decade of experience gained with Metakit's vector-oriented internal data model, and use a new very compact and efficient C-coded engine called Thrive. Ratcl introduces a terminology and set of conventions which minimize the impedance mismatch caused by having databases added on instead of native persistence and query support, while Rasql takes this one step further to map standard SQL queries onto Ratcl. Working examples will be presented, along with performance results so far. This covers the first phase, which focuses on access and querying. The second phase is work in progress and will be briefly described - it deals with modifications, transactions, and multi-user scenarios.

Introduction

Tcl has a range of mechanisms to deal with data, both in-memory and on-disk. One of the more unusual and very powerful aspects of Tcl is that (almost) “everything is a string” (EIAS) as far as the programmer is concerned.

This is both a blessing and a curse. The total lack of inherent type with EIAS that makes it very easy to quickly write code, can also be a cause for trouble:

- Type-less data can lead to code where bugs are caught later in the development process.
- Fewer opportunities to work with highly optimized data representation formats.
- No simple solution for missing values, i.e. null as being distinct from the empty string.

The lack of explicit type translates directly to the lack of explicit structure, i.e. compound types (records). While Tcl offers some very convenient mapping such as the new “dict” convention in 8.5, this type is not as easily enforced or stored truly efficiently on file.

When large volumes of data are involved (more than can conveniently be held in memory) or when high performance data manipulation is required, the EIAS approach by itself tends to lead to a lot of Tcl coding to deal with the unwanted consequences. This is usually just about the time when people start looking at databases as a way to address these issues.

There is something odd going on: as a language, Tcl offers amazing productivity gains when it comes to developing large-scale production software, but when a substantial amount of data is involved, much of the benefits are left behind as the coding switches to a

very different database style, such as SQL.

This paper presents a conceptual model based on Relational Algebra (RA) and shows how it can be embedded in Tcl in such a way that the benefits of scripting and quick ad-hoc coding remain, while the data gets managed in a completely new way, with high performance and persistence thrown in for free.

The Ratcl and Rasql extensions described here are part of a larger research project called “Vlerq”, which will also be presented briefly later on.

So what’s the problem, really?

The main reason why data storage has so many implications for programming is *copying*.

Programs are strange beasts: when launched, they start with a completely empty slate – all data processed by a program needs to be brought in, either directly from file or via a database layer. Worse, all new data, and all results produced by the program need to be saved back to “persist”. Do nothing, or crash, and the data will vanish. Imagine us working that way, knowing nothing when we wake up, and forgetting everything when we go to sleep!

So what most programs do, and have been doing for decades, is to create mechanism to facilitate this task of “fetching” and “storing” data, or “loading” and “committing” in database parlance.

The mindset that goes along with this is very deeply entrenched in most programming lan-

guages. One possible exception is Smalltalk / Squeak, where the system itself loads all data on startup and saves it again on quit – treating code and data uniformly.

For a database to be usable in Tcl we expect it to be good at fetching the data we need, and good at saving it back robustly and efficiently when changed.

Let's examine the different existing approaches to data storage, before attempting to offer an alternative.

Flat files

The simplest form of data storage by far is to dump everything to file, and restore it all in full later. The term “flat” is used, because in most programming languages this tends to destroy all structure, i.e. inter-data relationships.

In Tcl, structure can be saved on file for free. If you store a list, it'll come back as a list. This is one of the immense benefits of the EIAS approach of Tcl. So in a way, Tcl is actually much better equipped to work with (flat) files than most other languages.

There are drawbacks to flat files, though. For one, you have to dump and restore all data at once. There is no easy way to work with subsets (especially in terms of saving only the changes back). This makes the dump/restore slow as more data is involved, and means a copy of all data has to be held in memory.

Another problem is robustness. When saving changes, you have to be very careful not to lose all data altogether if the system were to crash or be switched off at just the wrong moment in time.

Yet another problem is evolution. How do you deal with old files when a new version of the application requires the data format to be extended in some way?

All of a sudden, flat files turn out to be not so trivial anymore. We're getting bitten by the fact that having data on-file and in-memory as totally disjoint forms of the same is not that convenient after all.

Relational databases

The next solution is to adopt a database of some kind. By now, this is almost always a relational database, based on decades of work leading to a very sound theoretical foundation for both the way to structure data and the way to manipulate it.

There are many relational database implementations, of varying complexity and sophistication. The most common ones in Tcl are probably Oracle, PostgreSQL, MySQL, and SQLite. The latter is quite interesting because it is embeddable, i.e. part of the application, whereas the others mentioned here are client/server solutions using a separate process or even machine.

With a relational database, all the problems mentioned for flat files are solved. Access and modification in subsets of the data are easy and quick. Data no longer needs to be loaded on startup or saved on exit. Changes are saved as transactions, so that fail-

ure is completely controlled: either a set of changes makes it into the database or it does not – there is no intermediate or inconsistent state.

The robustness of relational databases is summarized with the “ACID” acronym: changes are Atomic, Consistent, Isolated, and Durable. It's good to be able to truly rely on a database – after all, a crashed program can usually be restarted, but damaged and inaccessible data is essentially unrecoverable.

This comes at a price, however.

Data in a database can be orders of magnitude slower to manipulate than in-memory data. Try comparing a relational “join” with a Tcl array lookup, which are more or less the same operation, in abstract terms.

Apart from speed, databases tend to highlight the huge difference between in-memory data and on-file data. Something as simple as “\$a(\$b)” in Tcl ceases to be available. Instead, you're faced with, say:

```
[$db {select * from a where key = '$b'}]
```

... with not just a performance loss but also the issue of accurate quoting when \$b is an arbitrary string.

In theory, relational databases are wonderful. In practice, they can be a pretty lousy fit for programming languages.

Other ways to store & manage data

There are a number of other solutions to dealing with large amount of persistent data.

OODB – To overcome the impedance mismatch between databases and programming languages, a number of object-oriented solutions have been built. The idea is to treat everything as an object, and to then add a mechanism whereby objects transparently move between their in-memory form and a “backing store”, using techniques such as “pointer swizzling”.

A hybrid is “object-relational” mapping (OR), where objects are mapped to records in relational databases.

The OODB approach will not be explored further – one reason being that Tcl uses EIAS as basic model, not OO. But more importantly, OODB suffers from a major flaw when compared to relational databases: they tie the navigational access model to the data structure. In other words: when using an OODB, you have to make choices on how the data will be accessed, whereas the relational model separates the data structure from the way it is used. This is a very fundamental issue, at the heart of many OODB vs. RDB debates.

XML – Another approach is to fully abandon the relational model and treat everything as a hierar-

chy. XML was designed as general-purpose interchange format, and is now occasionally touted as solution as the model to use for storage and manipulation of that data as well.

There is little benefit to doing so, actually. Apart from the fact that it does not address the main issue of avoiding the gap between on-disk and in-memory formats, the main drawback is that by ignoring inherent repetitive structure in large data-sets, it prevents a number of optimizations and notational conveniences from being used.

Lastly, XML data can in fact very efficiently be represented via the relational model.

Berkeley DB – This represents a range of different database implementations actually (such as gdbm). The model used is the key/value association. With the *DBM packages, all data is stored by key, and looked up by key, plus the ability to traverse all keys.

This can be summarized as the persistent equivalent of Tcl arrays.

The speed of keyed access can be quite high, due to the use of hashing, although that tends to break down when large numbers of accesses are performed, where hashing leads to excessive disk seeking.

This approach is not used much, despite the fact that it has been around for ages. One reason is no doubt that richer data structures are often needed, and that as with OODB and XML solutions it often is useful to be able to navigate through the data in other ways than by key. A request such as “find all keys X for which the value is 123” ends up traversing all data.

Metakit – Metakit is a mix between the flat-file, relational, and hierarchical database approaches. It uses an inverted column-based format for efficient brute-force searching across all data, and uses the “stable storage” algorithm for transacted changes.

The Metakit database is a bit of everything and a bit of nothing. It has been used as basis for a relational SQL layer, although the Tcl binding does not really expose all functionality of the core.

The basic goal was to try and combine the powerful relational database concepts while using a column-wise internal structure for performance. To put it another way: Metakit presents a row-wise interface to what is essentially an “inverted” format. It favors fast access/searching, at the cost of slower updating.

Searches in Metakit can use hashing or binary search, but they are usually done by brute force. The reason this works so well is that copying is avoided to an extreme degree. Iterating over one field in all rows often outperforms other databases, even when they use indexes (up to a point, of course).

Brute force searching also works well with imprecise searches, i.e. “globs” and regular expressions, where a full scan is usually needed anyway. In Metakit, text searches are cheap.

One consequence of the inverted design is that data

structures can instantly be extended or modified. Adding a field to all rows is a matter of adding a single column internally. This encourages gradual development – extend the data as your code grows, instead of designing it all up front. This is a great match for the dynamics of scripting.

But Metakit is not perfect. Its Tcl binding does not expose some of the more advance capabilities of the underlying engine. And although quite snappy, the design is far from optimal in terms of performance.

Lastly, Metakit’s documentation is lacking. It takes some work to get the best mileage out of the system.

Home grown – There will always be data storage solutions that are custom-designed for a specific task. The challenge of any new solution is of course to try and offer sufficient performance and flexibility to cover an increasing number of these cases. The trend towards using “standard” solutions appears to be increasing, no doubt because home grown code is much more work to maintain, and because more and more open source alternatives present themselves.

Client/server – Lastly, one could say that the easiest way to use a database is to not use one at all. Instead of incorporating code for storing and manipulating data inside the application, the alternative is to simply connect to a database on a remote server. This relies on permanent network connectivity – an obvious trend, as the rise of websites with databases behind them shows.

Looking for alternatives

Wouldn’t it be great if we could *somehow* combine SQL’s relational foundation with Metakit’s column-wise performance and embed it all really cleanly in Tcl?

This is precisely the aim of Ratcl and Rasql.

The strength of SQL is that it has a strong relational foundation that is extremely effective (even though some will argue that SQL is severely flawed). It is a great benefit to be able to specify data processing tasks in a non-procedural way, i.e. in terms of what needs to be done, not how it is done.

Not only is it easier to say “find all the names of the part numbers I have on this list” than “go through each item on this list and lookup the name associated with in the parts catalog”, it also leaves more room for the underlying code to choose between different implementations. In cases where performance is not at a premium, the benefit of not having to spell out the details surely does simplify programming.

Then again, the SQL world is rife with examples where changing the order of a request makes a huge difference in performance, or where one is

expected to add an index briefly for use in a specific task, and drop that index again to avoid hampering other tasks. The last thing we need is a system where we have to fight and apply counter-intuitive tricks to get good performance.

If you think this is a minor issue, think again: people abandon SQL all the time due to the unacceptable performance they get (for whatever reasons).

Metakit proves that an inverted column structure has the ability to outperform traditional databases, sometimes by an order of magnitude. Examples are known for each and every database mentioned so far, where Metakit was able to perform the same task an order of magnitude faster. The very high-end “Kx” commercial database using a similar design shows that the limits of scalability and performance have not yet been reached, not by a long shot.

The challenge ahead, is to embed these techniques into Tcl in such a way that one stops thinking in terms of getting data “out” of a database and storing changes back “in”. Better still, we should try to create a system whereby the whole concept of a “database” separate from the language fades away.

This is similar to the way Tk has pushed “graphics contexts”, “ports”, “screen coordinates”, “refresh”, and “updates” out of the mind of the application programmer. We don’t think of Tk as a place to copy Tcl data to. We create a view hierarchy in terms of widgets, and then events do the rest.

There is a tremendous opportunity here. A lot of effort in programming deals with moving data around, altering its shape and structure a bit, and transforming it – often in very simple ways. At every point, we have to think where to copy data from, what variables to put it in, and how to deal with the end results – on-screen and on-disk.

Already, Tcl has many types of data collections. Internal data, such as channels, widgets, commands, as well as external data, such as returned from glob, stat, events, I/O.

Already, we lack a consistent way of combining this data. An example of this is: give me a list of a read-only files in a directory. In Tcl, we have to get a list (glob), iterate over them (foreach), check the file’s attributes (file stat), and generate a list with results (lappend). Why can’t we join the glob to the file stat and apply a condition?

Relational algebra provides a simple formalism, which is every bit as powerful as SQL (more so, some will say), and which lets us specify (as opposed to spell out) what needs to be done.

To get there, we need to “let go of the data”, i.e. stop thinking in terms of storing it in variables. Instead, we need to set up our processing in terms of operators (and use variables to manage those structures).

We need to let Tcl do what it does so well: glue.

Introducing Ratcl

The Ratcl extension for Tcl takes a first step towards a non-procedural approach to programming.

To use Ratcl, you have to be prepared to place all data under its control. Doing so will give you low memory consumption, persistence, and performance in return. Data in Ratcl can be manipulated through relational operators (join, groupby, and so on), set operators, expressions to produce calculated results, conditions to define subsets, and sorting.

The central concept in Ratcl is the “view” – think of it as the widget of the data world. A view is a tabular structure with the following properties:

- Views consist of rows, indexed by position.
- Views consist of columns that can be referred to either by name or by position.
- At every (row,column) position is a data item, which is either a basic value such as an integer or string, or a nested “sub-view”.
- All items in a column are of the same type.

The above terminology will be used in the rest of this paper, but usually very similar designs underlie most database systems. Here is a comparison with some familiar concepts:

- SQL’s “tables” are similar to views – they do not support positional access, usually, nor nested sub-views. In SQL, rows are called records and columns are called attributes. Views are indexable, they can also represent result “rowsets”, there is no need for cursors.
- The “relations” of pure relational database theory differ from views in that neither positional access nor order is supported, for rows as well as columns.
- Tcl arrays (and Python dictionaries) are very similar to a view with a “key” and a “value” column. However, views treat keys and values on equal terms, and allow either of them to consist of multiple columns.

It might be tempting to see views as matrices of rows and columns, but this is in fact not such a good idea. For one, matrices are uniformly typed, whereas each column in a view can hold different types of data. The other reason is that views will be extended later to support dimensions independent of row structure (so you could have a 3-dimensional space of rows of arbitrary complexity, not just single values).

Views are the central interface between Ratcl and Tcl. In Tcl, a view is a command object. You create a view explicitly and fill it with data in one command:

```
% set V [view A B C \
           { a1 b1 c1 a2 b2 c2 }]
```

To dump the view in Tcl, simply execute the command with no arguments:

```
% $V
  A   B   C
  --- ---
a1  b1  c1
a2  b2  c2
%
```

As you can see, V was a view with two rows and three columns, named A, B, and C.

Yes, V *was* a view, not *is*, as you can see here:

```
% $V
invalid command name "::vlerq::o::1"
%
```

Views are command objects in Tcl, but they require a slightly modified style to be usable transparently in Tcl. The details of this will be explained later, for now it is sufficient to note that with view objects, you should use “vset” instead of “set” when storing their name in a Tcl variable (or array element). To repeat:

With views, use “vset” instead of “set” !

This idiosyncrasy is only needed in Tcl, btw. Other languages can handle views like any other object.

With these preliminaries out of the way, let’s see what Ratcl has to offer.

A little tour

Ratcl includes a wide range of view operators. A few basic examples are given here. See the Ratcl pages on the web for more complete examples and some preliminary reference documentation.

Let’s assume the following views have been defined:

```
% $R
  A   B   C
  --- ---
a   b   c
d   a   f
c   b   d

% $S
  D   E   F
  --- ---
b   g   a
d   a   f

% $T
  A   B   C   D
  --- --- ---
a   b   c   d
a   b   e   f
b   c   e   f
e   d   c   d
e   d   e   f
a   b   d   e

% $U
  C   D   E
  --- ---
c   d   e
c   d   f
d   e   f
%
```

Then we can do things like:

```
% [$R product $S]
  A   B   C   D   E   F
  --- --- --- ---
a   b   c   b   g   a
a   b   c   d   a   f
d   a   f   b   g   a
d   a   f   d   a   f
c   b   d   b   g   a
c   b   d   d   a   f

% [$T project {A B}]
  A   B
  ---
a   b
b   c
e   d

% [$T if "B > 'b'"]
  A   B   C   D
  --- --- ---
b   c   e   f
e   d   c   d
e   d   e   f

% [$T join1 $U]
  A   B   C   D   E
  --- --- --- ---
a   b   c   d   e
a   b   c   d   f
e   d   c   d   e
e   d   c   d   f
a   b   d   e   f

% [$T join0 $U]
  A   B   C   D
  --- --- ---
a   b   e   f
b   c   e   f
e   d   e   f
%
```

Note how we used “[\$R product \$S]”, instead of “\$R product \$S”. The reason is that “\$R product \$S” returns the name of a view command object, not its contents. By adding an extra pair of [], we cause it to dump its contents, just like “\$R” does. We could also have used the following equivalent sequence:

```
% vset x [$R product $S]
% $x
  A   B   C   D   E   F
  --- --- --- ---
a   b   c   b   g   a
a   b   c   d   a   f
d   a   f   b   g   a
d   a   f   d   a   f
c   b   d   b   g   a
c   b   d   d   a   f

% unset x
%
```

View operations can be nested at will:

```
% [[ $T project {C D} ] minus \
    [ $U project {C D} ] ]
  C   D
  ---
e   f
%
```

And lastly, views can be tied to a Metakit data-file:

```
% vset M [mkopen mydata.db]
% $M names
dirs
% [$M sub 0 dirs] names
name parent files
% [[ $M sub 0 dirs] sub 0 files] names
name size date contents
%
```

Here's an example combining much of the above:

```
% vset D [[mkopen mydata.db] sub 0 dirs]
% [[ $D project {parent name}] sort]
parent  name
-----
-1  <root>
0   doc
0   lib
2   Class1.0
2   ClassyTk1.0
2   Extra12.0
2   Mpexpr10
2   Tktable2.7
(etc...)
```

Here is the set of view operators currently available:

*add addcol all as at blocked cmp col cols
concat counts decref delete divide expr first
flatten get groupby if ifmap incref insert
intersect join join0 join1 last mapcol maprow
meta minus names norows nspread omitcol
omitrow pair pick print product project
rename repeat reverse row rowid rows set
single slice sort sortmap spread sub subcat
types union uniqmap unique vid*

The list of operators is still evolving, but as you can see all key relational- and set-operators are included.

Advanced aspects of Ratcl

There is a lot more to say about Ratcl than will fit in this paper. A few highlights:

Calculated fields – data can be generated as a result of calculations based on other fields:

```
% [$T pair [$T expr F:I {B > 'b'}]]
A  B  C  D  F
-  -  -  -  -
a  b  c  d  0
a  b  e  f  0
b  c  e  f  1
e  d  c  d  1
e  d  e  f  1
a  b  d  e  0
%
```

The current parser is not yet able to handle callbacks, but once this is implemented, arbitrary Tcl-based computations will also be usable inside views.

Derived views are cheap – views are “lazy”, i.e. the information extracted from views is produced on-demand, at the latest possible moment in time. For example, setting up a sorted view is instant, only when rows in it are accessed does the sorting take place. For the same reason, access to views stored on

file can be extremely quick, since only a minimal amount of information is actually read in.

This has profound implications for situations where only a subset of the results is used. One example is the presentation of views on-screen: large views need not be fully accessed when only a small part of the view is showing on the screen.

Sub-views – in contrast to traditional relational database systems, views can be nested. The result of the standard “join” and “groupby” operators is in fact just that: a view with nested sub-views. This greatly simplifies processing, and is dramatically more efficient than producing a result where all data is expanded to fully “flat” tabular form.

The “flatten” operator can be used to force a flat operation when needed, though.

As all other operators, “join” and “groupby” are lazy performers, with everything happening behind the scenes in a totally virtualized manner. This means, for example, that joining two huge views takes little more than two integer vectors of memory, which are set up the moment access to the result is requested.

Cleanup – the view command objects of Ratcl use an elaborate reference counting mechanism to make sure they are kept around as long as needed, but no longer.

The consequence has already been seen in the use of “vset” instead of “set”. The reason for this is that an “unset trace” is needed in Tcl to make sure views are cleaned up when its variable goes away (implicitly on return, in the case of local vars in a procedure).

A somewhat unusual aspect of view command objects is that by themselves they will self-destruct after a single call. This allows the combination of multiple view operations into a single statement, without creating uncollected “debris”. The flip side is the need to use “vset”. This restriction could be lifted if a future version of Tcl were to make the standard “set” just a little smarter, by the way.

Related packages

For reference, here is a brief list of Tcl packages which offer some of the same functionality as Ratcl:

- NAP (“Numeric Array Processor”) by Harvey Davies offers vectorized processing of data. It is geared towards numeric processing whereas Ratcl works equally well with strings.
- TclRAL by Andrew Mangogna is Relational Algebra system that stays very close to the pure relational model, using the “relvar” and “relation” terminology. It is entirely value-based, and as such a good fit for Tcl, but it has no persistence, other than dump/restore.

As has become clear with Metakit over the years, there are very few systems around with relational algebra as basis, and offering the persistence of data-bases without adopting the SQL language.

The case for Ratcl

Ratcl aims to bridge that gap between databases and Tcl, offering the benefits of both as much as possible.

By “claiming” control over all data, it provides very efficient view “operators” as well as persistence.

The current set of operators is already reasonably complete, but a number of planned improvements will take this even further, such as allowing arbitrary bits of Tcl code inside view expressions – very similar to the way Tcl’s “expr” commands adds an algebraic notation to Tcl while still allowing “[...]” inside any expression to escape back to Tcl.

The central concept is the view, which maps to a Tcl command object – much like widgets map low-level GUI concepts via Tk. Views can be passed around and combined at will. Unlike most commands, views represent lazy evaluation, where the actual processing takes place behind the scenes at various points in time. Setting up complex nested calls to view operators is about preparing for processing, rather than having data handling actually being done.

As a consequence, Ratcl can do a lot of internal optimization, delaying file access and computations until the time they are actually needed. Combined with the column-wise structure of data, this often leads to a substantial reduction of processing time.

The efficiency of views in Ratcl will be presented in the next section.

Size and performance

The Ratcl extension consists of a tiny “core engine” coded in C, a bit of Tcl glue code, and some auxiliary data. A complete system, including all the relational and set operators, a Metakit data file reader, and an expression parser is about 75 Kb. With compression, a standalone exe containing all of the above as well as a Zlib de-compressor ends up being 22 Kb.

The source code of all the pieces of Ratcl amounts to some 3000 lines of code, half of which is C.

Small is beautiful, not just as an academic challenge, but because less code means fewer places for bugs to hide, and fewer cases to deal with and test. The layering used in Ratcl means that the system consists of a small set of carefully chosen components, each highly dedicated and aimed at only performing a few tasks, but doing those real well.

The performance of Ratcl has not been optimized at all so far. Key operations such as join and groupby use algorithms which are far from optimal right now, the reason for this is that this implementation focuses on functionality and took many shortcuts to get the basics working, regardless of overhead.

Nevertheless, Ratcl can open and access Starkits, which are Metakit data files, faster than the Mk4tcl extension itself. In plain integer column iteration, Ratcl can outperform Mk4tcl by a factor 4, in string iteration it is about on par.

In another test, using an Apache log file with about a million entries, it takes 1.66 sec to locate 3 copies of a specific IP address in today’s basic Ratcl (Metakit: 2.18, SQLite: 3.85). All timings are done on a relatively slow PIII/650 notebook to get a decent timer resolution.

The comparison with SQLite is a bit unfair, since one should use an index, in which case the time drops to 0.32 mSec. Then again, note that adding the index took 37 sec, and dropping it again took another 3 sec, so the choice of what to index is an important one to make up front.

To construct a comparable case in Ratcl requires creating a view which projects the key and then sorts it. With sorted data, binary search can then be used to locate a key. In Ratcl, project + sort take about 0.4 sec, and searching takes 28 microseconds).

The conclusion at this point should be that although Ratcl’s brute force is surprisingly efficient, it is no match for indexed access when the number of records involved is large (we’re comparing $O(N)$ brute force with $O(\log N)$ binary search). At this point, similar tricks must be used to gain optimized access, after which a Ratcl-based solution again outperforms other databases by an order of magnitude. Similar results and ratios can be expected with hashing, by the way.

Now, as everyone doing benchmarks knows, it’s fairly easy to “construct” examples that support any type of conclusion. Therefore, in the following discussion all further comparisons have been omitted.

Instead, let’s simply examine how long it takes to perform certain tasks using the high-performance primitives built into Ratcl (but not yet used very much!).

Opening the above data file takes 720 mSec. Using a primitive call, locating 3 ints in a million on file takes 20 mSec (80x as fast as Ratcl’s current dumb code).

One point to make is that most database timings are severely skewed towards single accesses, a metric which is usually irrelevant. What matters, is the performance figures when large amounts of data are processed as a whole. This is where databases can get dogged down to hours of processing time and I/O-bound disk thrashing. This is also where Ratcl’s column-wise model tends to make a dramatic difference.

The above example of finding 3 matching ints in a million takes exactly as much time regardless of the number of results – i.e. 20 mSec to find all values larger than K, for any K.

At the time of writing, not many more performance results are available. As mentioned before, Ratcl does not yet hook into the optimized vector-oriented code that is part of the system – most of the effort so far has simply gone into getting the data structures ready for vectorized use, and implementing basic functionality.

One more result which ought to give an impression of what lies ahead for joins and groupby is available: a hash-based algorithm which identifies all identical values in a set of the same million integers as above, takes 0.15 sec. For comparison, Tcl's "lsort –unique –integer" takes 3.7 sec to produce the same results (about 20,000 groups). Note also that these integers consume 4 Mb memory in Ratcl and 28 Mb in Tcl.

The explanation for these results, which show orders of magnitude higher performance figures than current database systems, is that the combination of an inverted column-wise design with a very efficient data format which is identical on-file and in-memory, work together to take maximum advantage of today's CPUs. Not only is a column-wise structure optimal for file access, it also lets CPU caches work at their best. All it takes is a highly vectorized internal design of the underlying code engine.

Reasons to use SQL

Despite these nice results in Ratcl, there are still a number of reasons to use SQL in an application:

- It's a standard – there is a lot of code based on SQL and a lot of experience with it.
- It's convenient to write tasks in a non-procedural way. The ability to think in terms of *what* instead of *how* is a huge time-saver, even if performance might suffer a bit.
- And lastly: you may not have a choice, if your boss dictates it. The same holds for Tcl itself, of course!

SQL is a complete language of its own (several in fact, sometimes frustratingly so). By adding SQL to an application, you are bound to get more or less of an impedance mismatch – quoting rules change, variable naming and expansion changes, even simple operators change ("<" versus "!=" for example). There is also some duplication of functionality, such as SQL's "like" versus Tcl's "string match". And lastly, you may find that SQL does not offer regular expressions, and that Tcl's "regexp" cannot be used for string searches in data managed by the database.

SQL is a language (from the 60's, in fact) - and its use in Tcl unavoidably implies working with two sometimes very different ways of looking at data.

Even though SQL is quite well standardized, the availability and lack of features differ widely across different database implementations and their bindings to Tcl. There are database independent wrappers and there is ODBC – but be prepared for quite a bit of tinkering. SQL is nice, but definitely no panacea.

Introducing Rasql

Rasql aims to bridge the world of databases and Tcl, but in a very different way than Ratcl.

Rasql is an implementation of SQL, and as such offers the standard SQL notation for those who choose to work this way.

The crucial point to make is that Rasql is based on Ratcl – it is in fact a thin layer over Ratcl, parsing and translating SQL statements to relational algebra operations in Ratcl.

This has a several implications:

- Rasql simply presents itself as an extra set of view operators, the most important one being called "select".
- You can combine views constructed with Ratcl with Rasql's standard SQL syntax.
- Views use the same inverted-column design, and are very efficient in space and time.
- The result of a Rasql "select" is a view.
- There is some usefulness in having sub-views, but there are also some limitations on their use inside SQL, which was not designed for them.

That last note means that Rasql can also be used as basis for further Ratcl operations. So now you get the best of both worlds: use SQL's non-procedural notation when it is convenient, yet switch to view operators as needed.

Rasql is not a gimmick. It handles nested sub-queries and quite advanced cases of SQL. Its design differs fundamentally from most SQL implementations, in that it translates non-procedural requests to set-wise manipulation of data, just as Ratcl does – this takes full advantage of the internal column-wise design.

At least four different implementations of more or less complete SQL engines on top of Metakit have provided the insights needed to accomplish this. Rasql combines this experience and brings it to Ratcl.

Rasql's limitations

One pretty severe limitation of Rasql is that it is work in progress. Its last implementation is from 2004, and was based on a predecessor of Ratcl. This code is not ready for serious use, and needs to be rewritten to use the latest Ratcl code base.

Another limitation of both Ratcl and Rasql right now, is that there is no built-in support for storing NULL. This can be emulated quite efficiently by adding an extra flag to every NULL-able column, but computations with such an approach can become a bit tricky. The reason NULL has not been added yet is that it requires a change to the Metakit file format to allow persisting views where some data items can be NULL.

The use of NULL is extremely controversial in the formal relational database world. Still, to provide sufficiently compatible support for SQL it will need to be supported in Ratcl and Rasql. Sub-views also offer a way to avoid NULLs in join and groupby.

Rasql does not aim to support SQL 100% (if that were even possible). The goal of Rasql is to support enough of the language to perform all common tasks, and to offer as few surprises to people who are used to SQL as possible. Rasql is a gesture towards what has become a de-facto standard, not an endorsement, and certainly not “Yet Another SQL Database”.

Lastly, Ratcl and Rasql are single-process in their current design. A number of high-performance concepts for contention-free parallelism in Metakit will be ported to Ratcl (and hence Rasql), eventually.

Note that this does not mean that Ratcl and Rasql are single-user. Multi-user scenarios will be fully supported as client/server option, once transactions are added back in, with all the aspects of ACID (atomicity, consistency, isolation, and durability) covered.

Current status

Right now (early May 2005), the Ratcl package is about to enter its second public release. This release supports general-purpose views, a wide range of view operators, read-only access to Metakit-compatible data-files, and simple serialization of views to file.

The current performance level of Ratcl is “decent”, meaning it’ll compare just fine with other solutions, but also that it is still far from the intended levels. The reason for this is that a lot of the internal vector-oriented processing has not yet been activated.

This Ratcl release will not be suitable for production use, it’s really a technology preview – to allow others to get more experience with the design and comment on it, and to act as a baseline for optimization.

The stability of Ratcl is already very good, i.e. it does what it should do. Robustness is not quite there yet, i.e. if used incorrectly, Ratcl still falls over far too often to be usable in general.

There is a nice introduction to Ratcl on the web, but it refers to an earlier implementation – some details of the syntax have changed by now. The semantics of it all is largely unchanged, though.

Rasql will not be released in public for some time to come, although the code will be made available as soon as the port to the latest code base is completed.

The Vlerq research project

Ratcl and Rasql are part of a research project called “Vlerq”. Vlerq is an acronym for:

Take Vectors
Add a Language
Make it Embeddable
Use the Relational model
Include a Query mechanism

Ratcl and Rasql are the result of using several tools being developed in / for Vlerq. In particular, a high-performance vectorized virtual machine called Thrive (Threaded Interpreter Vector Engine), and a systems-level language called Thrill (Thrive Language Layer).

The Thrive VM is a very tightly coded stack machine in C with an emphasis on handling vector operations and persistent data with maximum efficiency. Thrive includes automatic garbage collection. The Thrill language is relatively low-level, and is loosely based on Forth and other “concatenative” languages. Most of the Ratcl logic is coded in Thrill.

Much of the expected performance of Ratcl and Rasql are due to the fact that Thrive and Thrill have been designed and implemented from the ground up to provide the necessary functionality. The results so far and the extreme compactness of the code show that by segmenting a project into different conceptual layers (combining C, Thrill, and Tcl), far more can be accomplished than with a single-language design.

In a way, the Vlerq project is really a tribute to John Ousterhout’s vision on scripting as a glue language.

Longer-term goals

The use of views as central mechanism for data exchange is only the beginning of a considerably more ambitious goal: to create a data-flow driven framework whereby processing becomes completely automatic.

The promise of data-flow is that it allows you to move away from “thinking about all the consequences all the time”. Instead of applying changes to data and hard-coding the consequences at each point where such changes are made in an application, data-flow computing provides the same capability as what spreadsheets have been offering for decades.

With data-flow as driving mechanism, there could be a revolution similar to event-driven programming in user interface development, but permeating all the aspects of application development this time around.

To achieve this, the distinction between data on-file and in-memory has to be removed, which is precisely what Ratcl’s “views” are for. This can only be done by “taking the data out of Tcl”, i.e. adopting a coding style whereby Tcl manage dependency structures, but not directly the data itself. This is nothing new: the same holds for GUI components in Tk.

Getting data-flow working “all the way to the GUI” will one day require some new “data aware” widgets. Discussion on this is beyond the scope of this paper.

Conclusions

This paper has presented some early results of Ratcl and Rasql, two packages for Tcl that aim to simplify data manipulation.

As several preliminary tests with Ratcl show, the performance that can be achieved is at least an order of magnitude higher than traditional databases.

The reason for this is that an “inverted” column-wise data structure offers significant benefits for vector-oriented data processing algorithms.

The consequence is that even when not using any auxiliary “indexes”, many tasks will be surprisingly efficient. This means that we can have your cake and eat it too: the flexibility of not having to design rigid data models up front, combined with performance which exceeds most databases, and sometimes even Tcl’s performance with its own data structures.

With Ratcl and Rasql, it becomes feasible to “just start coding”, which is one reason why scripting languages can be so effective. This should of course not be taken as an excuse to design scripted applications badly, or worse, to skip the design phase entirely!

The column-wise format of persistent data makes adding columns trivial and instant, and the very high performance of joins, groupby, and sort means that the usual agony of choosing just the right set of indices and entering SQL statements in just the right order becomes a thing of the past.

What this means is that with data in Ratcl, you can get the best of everything:

- Data structures which are easy to define *and* to later extend or alter.
- Efficient operations on large amounts of data.
- Compact representations in memory and on file.
- Tcl-like performance as well as robust persistence.

Much of this is not new. People programming with APL, J, and K have known for decades that a wide range of processing tasks can be done far more efficiently than is commonly known – and that a vectorized language can be extremely concise yet flexible.

What Ratcl and Rasql bring to the table is the ability to get the best of both worlds. By introducing view command objects as the one generic data structure for everything, and by embedding this very tightly in Tcl, the result is a system in which data manipulation becomes very convenient, avoiding the usual looping idioms and dealing with entire data sets in one step.

Ratcl, and especially Rasql, are still in their infancy. Although all results presented so far are based on working code, that code still is being revised daily.

It is hoped that the main benefits (and trade-offs) of the approach presented here will help others see how the impedance mismatch between traditional database systems and a programming language such as Tcl can be reduced, by using “views” as general-purpose data

structure, combined with relational algebra, set operators, and array operators.

The Vlerq project which has become the foundation of Ratcl and Rasql has its own home page on the web at <http://www.vlerq.org> - a wiki-based area for all discussion and news related to this project.

All software described in this paper is available under the MIT open source software license.

Acknowledgments

I would like to thank Mark Roseman and Steve Landers for the many discussions which led to the design of Ratcl, Rasql, and Vlerq over the years. I would also like to thank them for their help and review of this paper.

Much of the Vlerq architecture stems from the experience gained with the Metakit database library in over a decade. I would like to thank everyone who directly or indirectly helped me refine and improve that system, often simply by pushing for more performance or identifying subtle bugs and design limits.

A very big thank you also to Mike Doyle and Eolas Technologies Inc, for funding the Vlerq project since early 2005, which has allowed me to make very substantial and rapid progress with the Ratcl and Rasql software.

References

Ratcl home – <http://www.equi4.com/ratcl.html>

Rasql technology preview and online demo – <http://www.equi4.com/preview/>

Metakit – embedded database extension for Tcl (Mk4tcl), <http://www.equi4.com/metakit.html>

The Tcl’ers Wiki - a collaborative web site for the Tcl community, <http://wiki.tcl.tk/>

NAP – Numeric Array Processor by Harvey Davies, <http://wiki.tcl.tk/4015/> / <http://tcl-nap.sourceforge.net/>

TclRAL – by Andrew Mangogna, <http://wiki.tcl.tk/12348/> / <http://tclral.sourceforge.net/>

Ratcl & Rasql

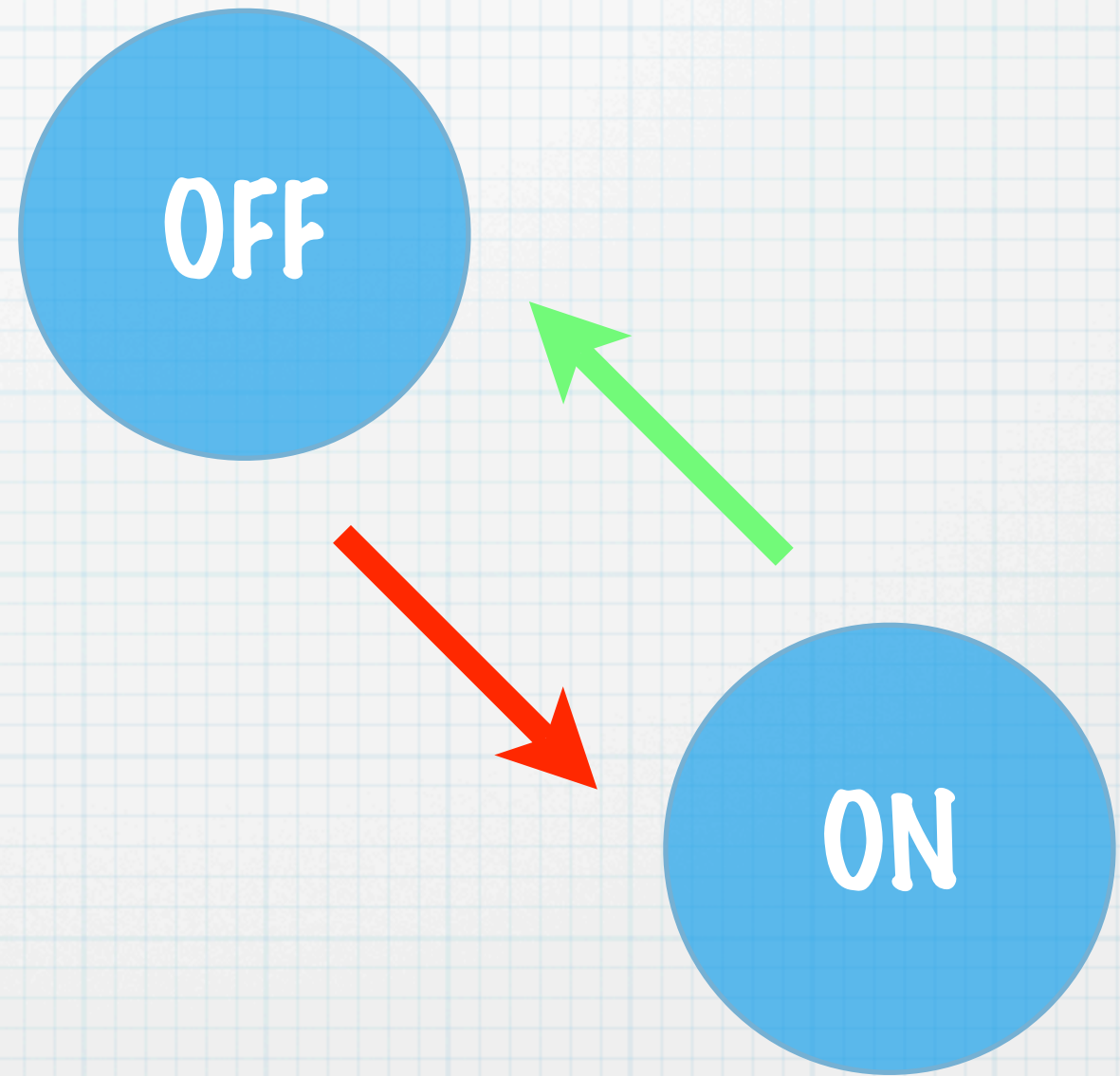
Jean-Claude Wippler

EQUI4
software

The Netherlands

The Data Dilemma

- * data on disk
 - * permanent
 - * passive
- * data in memory
 - * running apps
 - * modifiable



Everything in Tcl

- * application startup and exit:

```
array set mydata [read $fd]  
puts $fd [array get mydata]
```

- * sluggish - doesn't scale
- * risky - lose all if write fails
- * structure? search? share? speed?

Everything in a DB

- * learn a new language & mindset
- * which DB, pick ONE and stick with it
- * startup - can take a lot more code
- * copy, copy, copy data from DB to Tcl
- * copy, copy, copy data from Tcl to DB
- * design structure first - or be sorry later

Relational Algebra

- * best intro I've seen:

http://en.wikipedia.org/wiki/Relational_algebra

- * can do everything with 6 primitives:

select, project, product,
union, difference, rename

- * could RA be what SQL should have been?

Ratcl

- * stay in Tcl, think in Tcl, code in Tcl
- * manipulate data, still on disk
 - * access managed by Ratcl
- * lots of data manipulation operators
 - * relational, set-wise, vectors
- * transactions - commit/rollback, ACID

Implications

- * you control the data, but don't own it
 - * learn to work with "views"
- * stop writing loops to find & process
 - * think relational, set-wise, "wham!"
- * large speed & memory-use benefits
- * no hostages - can always import/export

Views

*** A view:**

name	age	shoesize
Mary	15	35
Bill	18	42
John	12	32
Eva	13	32
Julia	16	34

*** rectangular**

*** named columns**

*** rows 0 .. N-1**

“array”, “table”

uniform type

vertical: efficient

The “wham” mindset

* Relational product of views A and B:

A:

name	age
Mary	15
Bill	18
John	12



name	age
Mary	15
Mary	15
Bill	18
Bill	18
John	12
John	12

B:

shoe	size
left	32
right	38



+

shoe	size
left	32
right	38
left	32
right	38
left	32
right	38

=

#A = 3

#B = 2

A x B:

name	age	shoe	size
Mary	15	left	32
Mary	15	right	38
Bill	18	left	32
Bill	18	right	38
John	12	left	32
John	12	right	38

Views are virtual

- * the “product” example uses NO memory
- * it doesn’t read any data
 - * data is read when accessed
 - * memory-mapped files, no copying
 - * cached by the O/S, same as “paging”
- * combined operations are also virtual

Data exchange

* Tcl to view - “real data”

```
set r [vdef name age shoesize {Paul 15 32}]
```

```
set v [vdef name age [array get mydata]]
```

* view to Tcl - “real processing”

```
puts [view $v sort | get]
```

```
view $v each c { puts $c(name) }
```

```
array set a [view $v where {age >= 16} | get]
```

Meta-views

- * Every view has a meta-view ...
- * ... which describes its structure

* View:

name	age	shoesize
Mary	15	35
Bill	18	42
John	12	32
Eva	13	32
Julia	16	34



Meta-view:

type	name	subv
S	name	-
I	age	-
I	shoesize	-

- * #columns in view = #rows in meta-view

Repeating data

*** let's add a field to list their friends:**

name	age	shoesize	friends
Mary	15	35	Eva, Bill
Bill	18	42	Mary
John	12	32	Mary, Eva, Julia
Eva	13	32	John
Julia	16	34	Mary

*** how do you represent this?**

Repeat the rows?

* store each friend in a row copy:

name	age	shoesize	friend
Mary	15	35	Eva
Mary	15	35	Bill
Bill	18	42	Mary
John	12	32	Mary
John	12	32	Eva
John	12	32	Julia
Eva	13	32	John
Julia	16	34	Mary

* can (will!) become inconsistent - BAD

Relational: normalize

- * use two relations, link by common key:

- * Master:

name	age	shoesize
Mary	15	35
Bill	18	42
John	12	32
Eva	13	32
Julia	16	34

Detail:

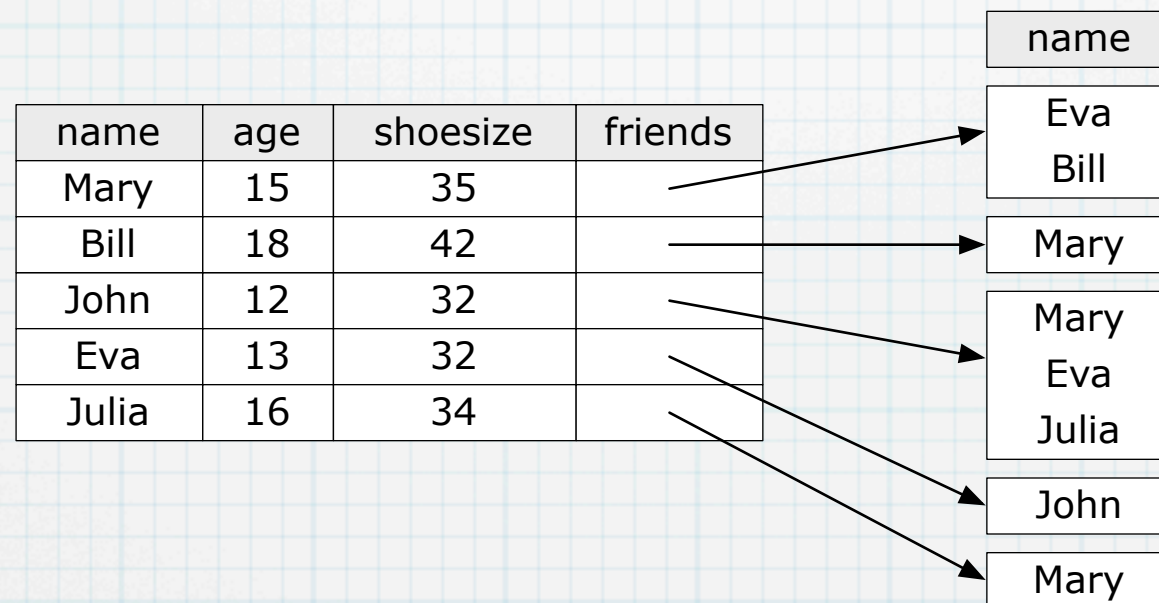
name	friend
Mary	Eva
Mary	Bill
Bill	Mary
John	Mary
John	Eva
John	Julia
Eva	John
Julia	Mary

- * simple & consistent

- * keys may require a lot of space

Sub-views

* embed 1:N in a hierarchical way:

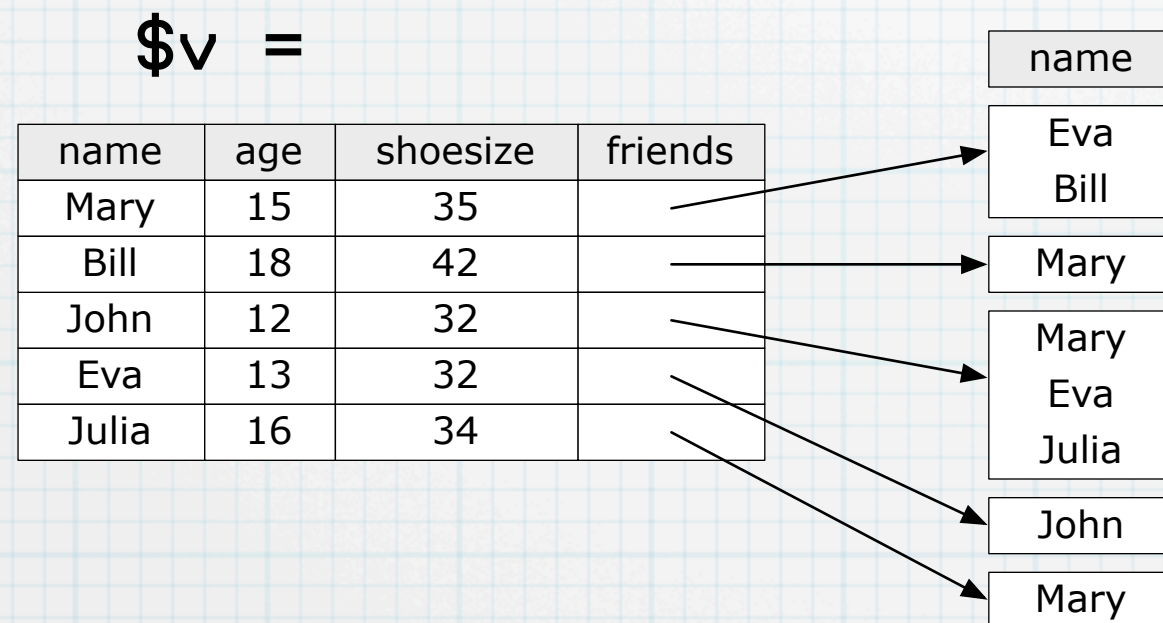


* still clean & tidy

* more efficient in time & space

Ratcl can "flatten" ...

* subviews and expanded are equivalent:



name	age	shoesize	friend
Mary	15	35	Eva
Mary	15	35	Bill
Bill	18	42	Mary
John	12	32	Mary
John	12	32	Eva
John	12	32	Julia
Eva	13	32	John
Julia	16	34	Mary

* `view $v flatten friends =`

... and go back: "group"

* grouping is inverse of flattening:

`$v =`

name	age	shoesize	friend
Mary	15	35	Eva
Mary	15	35	Bill
Bill	18	42	Mary
John	12	32	Mary
John	12	32	Eva
John	12	32	Julia
Eva	13	32	John
Julia	16	34	Mary

name	age	shoesize	friends
Mary	15	35	
Bill	18	42	
John	12	32	
Eva	13	32	
Julia	16	34	

name
Eva Bill
Mary
Mary Eva Julia
John
Mary

* `view $v group name age shoesize =`

Relational Join

* the workhorse for normalized data:

$\$v =$

name	age	shoesize
Mary	15	35
Bill	18	42
John	12	32
Eva	13	32
Julia	16	34

$\$w =$

name	friend
Mary	Eva
Mary	Bill
Bill	Mary
John	Mary
John	Eva
John	Julia
Eva	John
Julia	Mary

name	age	shoesize	friend
Mary	15	35	Eva
Mary	15	35	Bill
Bill	18	42	Mary
John	12	32	Mary
John	12	32	Eva
John	12	32	Julia
Eva	13	32	John
Julia	16	34	Mary

* “classical” join result =

* physical = “v & w” versus logical = joined

Joins

- * a join is “N lookups in parallel”**
- * joins produce subviews in Ratel**
- * no NULLs, yet equivalent**
- * rely on flattening & grouping**
- * think in very high-level: data shapes!**

Ratcl's Join

* first, group repeated field to subviews ...

\$w =

name	friend
Mary	Eva
Mary	Bill
Bill	Mary
John	Mary
John	Eva
John	Julia
Eva	John
Julia	Mary

\$wg =

name	friends
Mary	
Bill	
John	
Eva	
Julia	

name

Eva
Bill

Mary

Mary
Eva
Julia

John

Mary

* view \$w group name =

Ratcl's Join - part 2

* ... then connect corresponding rows

$\$v =$

name	age	shoesize
Mary	15	35
Bill	18	42
John	12	32
Eva	13	32
Julia	16	34

$\$wg =$

name	friends
Mary	Eva Bill
Bill	Mary
John	Mary Eva Julia
Eva	John
Julia	Mary

* view $\$v$ join $\$wg =$ (same result)

name	age	shoesize	friends
Mary	15	35	Eva Bill
Bill	18	42	Mary
John	12	32	Mary Eva Julia
Eva	13	32	John
Julia	16	34	Mary

It's all Relational

* SQL

```
SELECT * FROM data  
WHERE name = 'John'  
ORDER BY age
```

* Ratcl

```
view $data where {name = 'John'} | sort age
```

* or maybe

```
view $data where {name == "John"} | sort age
```


SQL & Rasql

- * (S)tructured - tables & joins
- * (Q)uery - “what”, not “how”
- * (L)anguage - standard notation
- * Rasql translates SQL to Ratcl view ops
 - * thin layer to create an “access plan”

SQL?

- * 1 standard? - N dialects!**
- * optimization, trial and error**
- * NULL, 3-valued logic**
- * half a programming language**
- * Rasql doesn't try to be "big" SQL system**

Inside Ratcl

- * **guided by simplicity and performance**
 - * lessons from Forth, APL, and Metakit
- * **“obsessively vector-oriented” design**
 - * tiny special-purpose virtual machine
 - * portable implementation language

Minimalism

- * Forth & APL show that “less is more”
- * built on a very uniform data structure
 - * tiny and fast mark/sweep GC
- * VM is < 1000 lines of C code
 - * 1000 more for vector ops
- * it's all under the hood - Tcl is the API

How small?

- * VM is a 30 Kb C extension
- * Tcl wrapper is another 40 Kb
- * as starkit - which uses compression

<http://www.equi4.com/pub/vq/ratcl.kit>

- * 100 Kb for complete system
- * includes binaries for 5 platforms

Speed: think again

- * **SQL:** `SELECT * FROM data WHERE name = 'John'`
- * **“*”** often reads too much
- * **Ratcl:** `set v [view $data where {name = 'John'}]`
- * **USE** determines I/O: later & lazily
- * **column-wise “inverted” storage**
- * **like having indexes on everything**

How Rasql works

*** select name from students**

where age > 15

group by shoesize

having count(shoesize) > 1

*** 1. map to groups**

2. collect counts

3. omit some counts

4. flatten result

5. omit some ages

6. done!

Why it's fast

1. load column of shoe sizes: 1 read
2. locate duplicates via hash: $O(N)$
3. load column of ages: 1 read
4. select specific age range: $O(N)$
5. logical AND, bitmaps: fast
6. Done!

How fast?

*** Join 161,127 x 47,079 on 1 int:**

Ratcl: 0.08 s, Metakit: 3.16 s

*** Find unique IP's in 1,077,106 entries**

Ratcl 0.15 s, Tcl 3.7 s (lsort -unique)
(~ 4 Mb) (~ 28 Mb)

*** Find 3 matches in 1,077,106 values**

Ratcl 1.66 s, Metakit 2.18 s, SQLite 3.85 s
Ratcl 28 µs, SQLite 316 µs - indexed
(create 37s, drop 3.2s)

Current status

- * Ratcl 0.92**

- * it works, many operators

- * API has not been frozen yet

- * it's not very robust or fast right now

- * maps MK datafiles, and writes dumps

- * Rasql - only an older preview**

Progress

- * on the web as “Vlerq” research project**

<http://www.vlerq.org/>

- * good software is like good wine**

- * consumed quickly just gets you drunk**

- * take your time to enjoy its richness**

- * most of my 2005 time goes into Vlerq**

Biotcl – a framework for computing biology

Dr. Detlef Groth, MPIMG Berlin, Ihnestr. 73, 13149 Berlin, dgroth@molgen.mpg.de

ABSTRACT

Currently from the four scripting languages Perl, Python, Tcl and Ruby only for Tcl there is no structured framework of tools and applications for computational molecular biology. This is a surprising issue because of its clear syntax, high portability and extensibility Tcl is an ideal choice as a scripting language for biological problems. Furthermore with the Starkit [1] approach Tcl has an excellent system to deliver applications and libraries to the researcher without going into a nightmare of installation procedures and prerequisites. The foundations of the Biotcl project will be Snit, Itcl and possibly a Tcl-only emulation of Xotcl as object oriented extensions, Metakit databases as RDBMS, a small embedded webserver in order to present the documentation and graphical interfaces for the applications. The project will restrict furthermore mainly to Tcl only libraries in order to achieve maximum portability. However for speed and memory issues there will be some compiled extensions available separately as well. Applications, libraries and documentation (Wiki/HTML format) will be embedded in one single Starkit file in order to simplify installation and maintenance as much as possible.

INTRODUCTION

In contrast to Perl [2], Python [3], Ruby [4] and Java [5] Tcl has no actual framework for solving biological problems. In the past several bio-applications has been written like Biowish [6], GRS [7] or MASIA [8].

There are various problems with the current software approaches used by other scripting languages like:

- installation hell
- difficult to keep different version of the same package
- platform dependencies
- deep directory trees

Tcl could resolve some of those difficulties. However currently a consistent application programming interface is missing. Tcl has several advantages which makes it suitable for programmers and ordinary users (biologists) like:

- clear syntax
- great introspection capabilities
- dynamic code generation
- flexible and modern object oriented extensions
- high level network functions
- Starkit-technology for easy deployment

In Tcl it is possible to create or redefine code at runtime and to inspect the values of variables or the definitions of classes or procedures. This make it suitable for developing distributed applications where otherwise debugging is much more difficult to achieve. An other advantage is the possibility to create stand alone applications and libraries with the Starkit technology. Applications, documentation and libraries are wrapped into one file which might be portable to all platforms where the Tclkit runtime exists (currently about 20 platforms). The user has just to download two files which can be placed anywhere. No further installation step is necessary. The application is just run with a "/path/to/Tclkit /path/to/application.kit parameter ..." command. A developer might just use "source /path/to/application.kit" inside a Tcl-program in order to use

Biotcl – a framework for computing biology

the libraries. If the application is updated just this kit–file must be redownloaded. Because all necessary libraries are wrapped as well into the kit–file it is not possible that using a new kit–file might give problems for an older file due to library incompatibilities or interferences with other installed libraries. Such problems are common for scripting languages like Python and Perl where the libraries are stored inside a deep hierarchy and newer packages are overwriting older packages.

Furthermore the documentation will be included as well inside the Starkit avoiding the problem of grabbing and maintaining the documentation separately. In order to show the documentation a little webserver is used having the advantage that the user can use the most sophisticated tool – a web browser – to view and manage the documentation. The documentation is provided as ing the problem of grabbing and maintaining the documentation separately. In order to show the documentation a little webserver is used having the advantage that the user can use the most sophisticated tool – a web browser – to view and manage the documentation. The documentation is provided as a writable Wiki documentation based on the famous TcLers–Wiki [9].

RDBMS

Data storage is an important requirement for applications. Tclkit comes with an inbuilt relational database system called Metakit. The Tcl–API for accessing the contents of Metakit databases Mk4tcl [10] is quite capable of supporting storage of a large amount of data in a structured and reliable way. However until now an easy to use query language was missing for inserting and accessing the data inside the database. Jean–Claude Wippler has taken several ways to simplify the access to Metakit databases like Oomk [11] using Snit [12] and relational operations.

Although Oomk was a big step forward for programming Metakit databases most developers are more familiar with SQL as a query language for databases. I therefore developed tsq4mk "Tiny SQL for Metakit" [13] based on the relational operator provided by Oomk. Tsq4mk implements quite a subset of the SQL–language. The tsq4mk–API is modelled after the SQLite–API which makes it possible to use tsq4mk as a drop in replacement for SQLite where SQLite is not available. A typical session for tsq4mk would be:

```
source tsq4mk.kit
package require tsq4mk
tsq4mk tsq dogtest.mk
tsq eval "BEGIN TRANSACTION"
tsq eval "create table dogs (name text, breed text,
    age integer, weight float, owner text)"
tsq eval "insert into dogs (name,breed,age,weight,owner)
    values ('fido', 'spanish spaniel', 4, 9.5, 'Will')"
```

```
tsq eval "COMMIT TRANSACTION"
tsq eval "select * from dogs" v { puts [array get v] }
    weight 9.5 age 4 name fido * {name breed age weight owner {}}
    breed {spanish spaniel} owner Will # 0
```

Due to some memory leaks of the underlying Oomk–architecture, making it necessary to fix them manually, it was unfortunately not possible to use a sophisticated scanner/parser system like fickle/tackle [14]. Fickle and tackle are the Tcl equivalents of the tools for C/C++: flex and bison. Instead a handwritten scanner was used, limiting therefore extensibility and maintenance of the library. For the required functions inside the Biotcl–framework however the current implementation status of tsq4mk is sufficient. With the ongoing vlerq–project [15] a successor of tsq4mk can be easily dropped into the Biotcl–kit.

WEBSERVER

The webserver inside the Biotcl–Starkit will be used for two tasks. At first it will be used to serve the documentation for users – application documentation and for the developers – library documentations. As

Biotcl – a framework for computing biology

shown by the Wikipedia and the TcIers' Wiki, Wikis are very successful examples of cooperative communities to write documentation. Therefore a Wiki will be the delivery platform for the documentation. The Wikit Wiki implementation is a Starkit containing a little webserver and utilities to view the documentation as a cgi process or inside a Tcl/Tk-application. I choose therefore the Wikit webserver as an inbuilt webserver for Biotcl.

However the Wikit webserver had two drawbacks limiting it's use for the second task: building web applications via HTML and crossbrowser javascript. The webserver could not serve static html-pages, images, javascript files or stylesheets. For this reason a separate more sophisticated webserver had to be used till recently. I removed this pitfalls by adding static page access to the Wikit webserver via some adjusted code from the minihttpd webserver, part of the tclhttpd-webserver. Furthermore I was adding procedure to url mapping in order to build GUI's for the Biotcl-applications.

We can provide javascript code if the webserver is started in the following way:

```
WIKIT_JS=/javascript/jsPreColor.js ;  
export WIKIT_JS ;  
WIKIT_CSS=/css/sepia.css ;  
export WIKIT_CSS ;  
WIKIT_BASE=http://localhost:8015/wiki ;  
export WIKIT_BASE ;  
tclkitsh dgwikit.kit wikit.tkd -httpd 8015
```

GUI

There are two natural choices for programming a graphical user interface (GUI) for Biotcl-applications. The first is Tk, the second is using the webserver serving html pages as the application server and the webbrowser as a client. The advantage of using Tk is it's rich set of widgets which can be used to build sophisticated applications. The disadvantage is that it can be used only on one machine, if the users want to present its data to more people, it is necessary to write a web based application. Therefore it is feasible to select the webserver/webbrowser approach for building the GUI. It is possible later to tunnel certain parts of the webserver through an existing browser like apache and to present data or services to the public.

A possible disadvantage of HTML for building user interfaces are that the capabilities of HTML for this task are somehow limited. There are several ways to overcome this limitations like Java-applet, Active-X controls or the Extensible User Interface Language (XUL). All those approaches have their drawbacks like high memory requirements (JAVA) or being attached to a certain webbrowser. In contrast with the javascript implementation, the Document Object Model and the XMLHttpRequest-method of modern browsers (IE 5.5, Mozilla 1.4++ and Opera 8) currently exists a highly capable programming platform for building rich user interfaces in a very dynamic way.

After studying the excellent behaviour implementation of Dean Edwards for Mozilla I decided to implement a similar but less complicated framework for the three major browser platforms (IE,Mozilla/Firefox/Opera) which strictly separates the javascript code from the HTML code. The result of this work are jsComponents which can be easily used to build richer user interfaces in HTML like trees, sortable tables, tabboxes, tooltips and much more. These controls can be used inside Biotcl to program the GUI.

Biotcl – a framework for computing biology

Example of a collapse widget:

```
<div class="JSCollapse">
  <ul>
    <li>HTML Authoring
      <ul>
        <li><a href="#">Beginner's Guide</a></li>
        <li><a href="#">Authoring Tips</a></li>
        <li><a href="#">HTML Coding Tips</a></li>
      </ul>
    </li>
    <li class="opened">HTML References
      <ul>
        <li><a href="#">Elements</a></li>
        <li><a href="#">Character Sets</a></li>
      </ul>
    </li>
  </ul>
</div>
```

DOCUMENTATION

Documentation is always the most important part which determines the success or failure of a certain project. Using a Wikit Wiki inside the Biotcl-kit is a natural approach for delivering information to the user. The advantages are: more people can contribute to the documentation, the user can use the excellent full text or title search capabilities of the Wiki, the user can easily bookmark important informations and the user can use a very sophisticated application to view the documentation – the webbrowser. It is even possible to modify the existing documentation inside the downloaded Biotcl kit in order to store notes of the user directly where they are needed.

With the added functionality of accessing static html–pages, stylesheets, images an javascript code the Wiki can be even more tweaked and styled to increase pleasure and reuse. The possibility to style the Wiki should not be underestimated. It is not only helpful to convince the user that a certain product is worth to being investigated it also helps to divide visually different areas of the documentation for instance sections for users and developers, therefore simplifying mental switching between different tasks.

LIBRARIES

There is a distinction between applications and libraries. Some of the libraries have interfaces to the user, so there are both libraries and applications at the same time. Developers can directly call the libnsible to translate the various tokens submitted via the scanner and to build a Metakit database. This database contains a index table which is used to retrieve entries of the datafile for a certain id. In order to simplify the building of web applications for each id a md5–hash is put into the index table as well. Furthermore a metadata table is generated which contains data for the datafile at all. For a blast datafile this could be for instance the type of blast program used, the chosen cutoff e–value and so on. The parser is the interface presented to the user – it is also an application which can be invoked from the console. If the parser is called for a certain datafile for the first file the scanner/consumer classes are invoked and the database is build. Afterwards the requested informations are submitted to the user. Example for a FastaParser getting a sequence for a certain id:

```
FastaParser fp -fastafile /mypath/to/orf_trans.fasta.tmp \
  -indexfile /myhome/data/orf_trans.fasta.tmp.mk
puts [fp getSeq YAL001C]
>YAL001C TFC3 SGDID:S0000001, Chr I from 151168-151099,151008-147596, ...
MVLTIYPDELVQIVSDKIASNKGKITLNQLWDISGKYFDLSDDKKVKQFVLSCVILKKDIE
VYCDGAITTKNVTDIIGDANHSYSVGITEDSLWTLTLTGYYTKKESTIGNSAFELLLEVAKS
....
```

APPLICATIONS

There are currently the main types of applications. The runner applications are presenting an easy to use interface for the user via a command line or via the HTML–GUI. They can be easily configured via the Biotcl inifile as described inside **CONFIGURATION**. Furthermore additional runner sections can be added to the inifile so that the user can easily add its own favourite applications. The example for a user to access the a certain sequence of a fastafile would be:

```
/path/to/tclkit biotcl.kit FastaParser
    -fastafile /mypath/to/orf_trans.fasta.tmp \
    -indexfile /myhome/data/orf_trans.fasta.tmp.mk
    -command 'getSeq YAL001C'
>YAL001C TFC3 SGDID:S0000001, Chr I from 151168-151099,151008-147596, ...
MVLTIYPDELIVQIVSDKIASNKGKITLNLWDISGKYFDLSDKKVKQFVLSCVILKKDIE
VYCDGAITTKNVTDIIGDANHSYSVGITEDSLWTLTGTGTTKKESTIGNSAFELLLEVAKS
GEKGINTMDLAQVTGQDPRSVTGRIKKINHLLTSSQLIYKGHVVKQLKLLKFFSHDGVDSN
PYINIRDLATIVEVVKRSKNGIRQIIDLKRELKFDKEKRLSKAFIAAIAWLDEKEYLKK
....
```

Parser applications are extracting information from biological data or from the output of biological programs which can be stored for later processing or piped into other applications.

Browser applications can be used to interactively explore informations via the HTML–GUI (GeneOntolBrowser). They must be configured as well via the inifile. See **CONFIGURATION** for more details.

INSTALLATION

The user has to download a Tclkit appropriate for its platform from <http://www.equi4.com> . Afterwards the it should be checked that the Tclkit is working properly. Try on the console:

```
/path/to/tclkit
% puts Hello
% package require Itcl
```

If everything is fine it is time to download the biotcl.kit from <http://goblet.molgen.mpg.de/biotcl> and put it somewhere.

As the final step it is useful to ensure that the steps has been done successfully via executing:

```
/path/to/tclkit /path/to/biotcl.kit -httpd 8016
```

and to view the documentation at: <http://localhost:8016/>

If the port address is in use you should select an other one which is higher than 8016.

CONFIGURATION

The configuration is done via ini–files. Normally the inifile is located at the uses home directory. However in order to use a different inifile the filename can be given at the command line:

```
/path/to/tclkit /path/to/biotcl.kit -inifile /home/www/user/biotcl.ini -httpd 8018
```


Biotcl – a framework for computing biology

This can be used to present different interfaces for instance one version to publish results to the community and an other inifile for your current work and so on.

Ini-files have the advantage that they are human readable and can be easily edited. Here an example for the configuration of the blastrunner:

```
[BlastRunner]
PROGRAM=blastall
TYPE=blastn,blastp,blastx,tblastn,tblastx
DATABASE=uniprot,/home/user/data/myseqs.fasta,yeast
EVALUE=1e-10,1e-20,1e-30,1e-40,1e-50,1e-60,1e-70,1e-100
FASTAFILE=
OUTPUT=
RUN=$PROGRAM $TYPE -d $DATABASE -i $FASTAFILE -e $EVALUE -o $OUTPUT
```

If the program is running via the console the user will be asked for each parameter passed to the program. If the value is empty than the user will be asked to select the value manually. If the user is using the HTML-GUI a form is presented which let the user select the appropriate options via the GUI. For empty values a entry field is presented, if the key ends with FILE a file select entry is presented o the user. From those data a command line is constructed which can be easily pasted into the console. The RUN key is used to assemble the command line. This interface is build in that way that the user can easily drop in own sections for runners without the need to hard code the runners into the system. In fact any console program can be put in not only biological relevant ones.

Browser applications must know where there data to be display are stored. The entry for the GeneOntolBrowser is structured as follows:

```
[GeneOntolBrowser]
OBOFILE=/home/user/data/gene_ontology.obo.2005-04-01.gz,gene_ontology.obo.2005-01-01.gz
GOAFILE=/home/user/data/gene_association.goa_mouse.gz,...
```

Here the user can select obofile/goafile pairs an interactively explore the data.

LICENSE

This software and documentation is distributed under the MIT License, as documented in <http://www.opensource.org/licenses/mit-license.php>.

CONCLUSIONS

The presented framework has some unique advantages over existing similar toolkits. Those are:

- easy deployment and administration
- several versions of Biotcl tools and libraries can be used
- applications, libraries and documentation in one file
- crossplatform Starkit for more than 20 platforms
- console or GUI interface for the applications can be used
- embedded webserver for presentation of data

A user has just to download and can immediately start using the biotcl-kit. If the user/developer wants to update the kit he needs just to download the Biotcl file again and to store it separately from older versions. If it is desired to use the older version again just use the older file. All libraries are stored inside the kit file. So you can keep different versions without hassle. The same Biotcl-kit can be used on about 20 different

Biotcl – a framework for computing biology

platforms from AIX64 to Win32 providing support for different operating systems and different processor architectures.

The interface is either graphically from the command line. The embedded webserver allows to present the data to the community without further trouble or the need to download any other software.

The documentation is included inside the application and can be easily viewed and searched via the webbrowser. The user can drop in it's own documentation into the manual to store notes etc.

Taken together Biotcl will provide an excellent platform for performing reliable analysis of biological problems.

ACKNOWLEDGEMENTS

At first I would like to thank Jean–Claude Wippler for very helpful discussions and providing all those great technologies like Tclkit, Metakit and the Wikit Wiki.

Further I would like to thank William H. Duquette for Snit and answering all my questions about using it. D. Richard Hipp I would like to thank for his work on SQLite – a wonderful small database which inspired my to program tsq4mk as a poor man alternative for SQLite.

REFERENCES

- Tclkit <http://www.equi4.com/tclkit>
- Metakit <http://www.equi4.com/metakit.html>
- Mk4tcl <http://www.equi4.com/metakit/tcl.html>
- Oomk <http://www.equi4.com/oomk.html>
- tsq4mk <http://goblet.molgen.mpg.de/tsq4mk>
- SQLite <http://www.sqlite.org>
- Snit <http://www.wjduquette.com/snit/>
- fickle <http://wiki.tcl.tk/fickle>
- taccle <http://wiki.tcl.tk/taccle>
- Tcl software for biology <http://mini.net/tcl/biotcl>

Biotcl – Framework

Dr. Detlef Groth

Table of Contents

<u>Biotcl – Framework for Computing Biology</u>	1
<u>Dr. Detlef Groth</u>	1
<u>Max–Planck–Institut for Molecular Genetics Berlin, Germany</u>	1
<u>Outline</u>	2
<u>Motivation</u>	3
<u>Syntax</u>	4
<u>Prerequisites</u>	6

Table of Contents

Biotcl – Framework for Computing Biology

<u>Architecture</u>	7
<u>Metakit RDBMS</u>	8
<u>TSQL4MK</u>	9
<u>TSQL4MK console</u>	10
<u>TSQL4MK library</u>	12
<u>Parser Infrastructure</u>	13

Table of Contents

Biotcl – Framework for Computing Biology

<u>Tcl–Scanners</u>	14
<u>WC with ifickle: iwc–fickle.fcl</u>	15
<u>Scanner Usage</u>	17
<u>Source iwc–fickle.tcl</u>	18
<u>WC–Results</u>	21
<u>FastaScanner</u>	22

Table of Contents

Biotcl – Framework for Computing Biology

<u>Scanner–Results</u>	23
<u>Database Structure – dgMKViewer</u>	24
<u>Webserver</u>	25
<u>Webserver–Implementation</u>	26
<u>GUI – JSComponents</u>	29
<u>Downloading == Installation</u>	31

Table of Contents

Biotcl – Framework for Computing Biology

<u>Configuration</u>	32
<u>Status</u>	33
<u>Scanner–Parser Status</u>	34
<u>Outlook</u>	35
<u>Acknowledgement</u>	36
<u>About Me</u>	37

Table of Contents

Biotcl – Framework for Computing Biology

<u>Note</u>	38
-------------------	----

Biotcl – Framework for Computing Biology

Dr. Detlef Groth

**Max–Planck–Institut for Molecular Genetics
Berlin, Germany**

Outline

- Motivation
- Implementation
 - ◆ RDBMS
 - ◆ Parser
 - ◆ Webserver, GUI, Documentation
 - ◆ Packaging
- Status
- Outlook

Motivation

	Tcl	Perl	Python
Syntax	+++	+	++
OOP	+++	+	++
Delivery	+++	—	—
Community	++	+++	++
Maintainance	+++	+	+

Syntax

Main datatype creation.Pperl:

```
my $x = 3 ;  
my @y = ('Spot', 'Spike', 'Fufu', 'Waldi');  
my %z = (Mouse => 'Jerry', Cat => 'Tom', Dog => 'Spot');
```

Python:

```
x = 3  
y = ['Spot', 'Spike', 'Fufu', 'Waldi']  
z = {'Mouse' : 'Jerry', 'Cat' : 'Tom', 'Dog' : 'Spot'}
```

Tcl:

```
set x 3
```

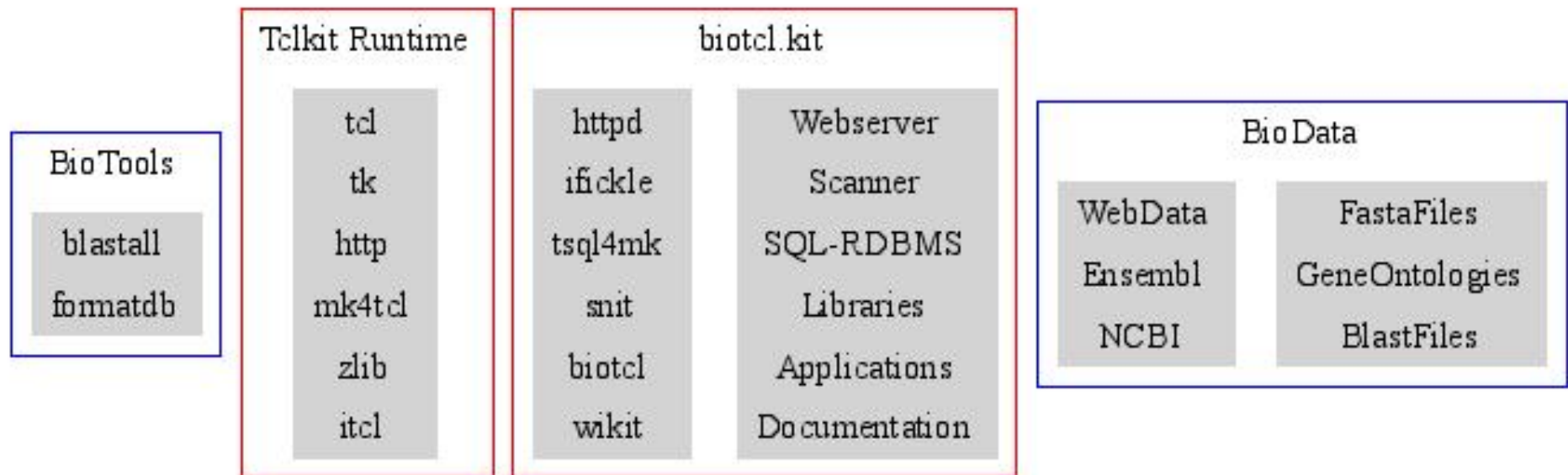
```
set y [list Spot Spike Fufu Waldi]
```

```
array set z [list Mouse Jerry Cat Tom Dog Spot]
```

Prerequisites

- Delivery Platform: Starkit
- SQL-RDBMS: oomk+snit → tsql4mk
- Scanner: itcl+fickle → ifickle
- Webserver: wikit-httpd + minihttpd → bhttpd
- GUI-Interface: Javascript+HTML → JSComponents
- Documentation: wikit based wiki

Architecture



Metakit RDBMS

- views alias tables
- rows
- properties alias columns
- cursors
- simple but flexible API
- but no query language
- Oomk provides relational algebra
- tsq4mk : Tiny SQL For Metakit

TSQL4MK

- `snit::type tsql4mk`
- on top of Oomk
- api modeled after sqlite api
- drop in replacement for sqlite
- tsql4mk starkit is:
 - ◆ console application
 - ◆ library
 - ◆ webserver

TSQL4MK console

```
-- file test.sql
BEGIN TRANSACTION;
create table dogs (name text, breed text, age integer,
                  weight float, owner text);
INSERT INTO dogs VALUES('fido','spanish spaniel',
                        4,9.5,'Will');
INSERT INTO dogs VALUES('fido II','spanish spaniel',
                        1,3.5,'Will');
...
COMMIT;
```

```
cat test.sql | tsq14mk.kit test.mk
```

```
$ tsq14mk.kit test.mk \
```

```
    "select * from dogs where owner == 'Will'"
```

```
name      breed                age  weight  owner
```

```
fido      spanish spaniel      4      9.5  Will
```

```
fido II   spanish spaniel      1      3.5  Will
```

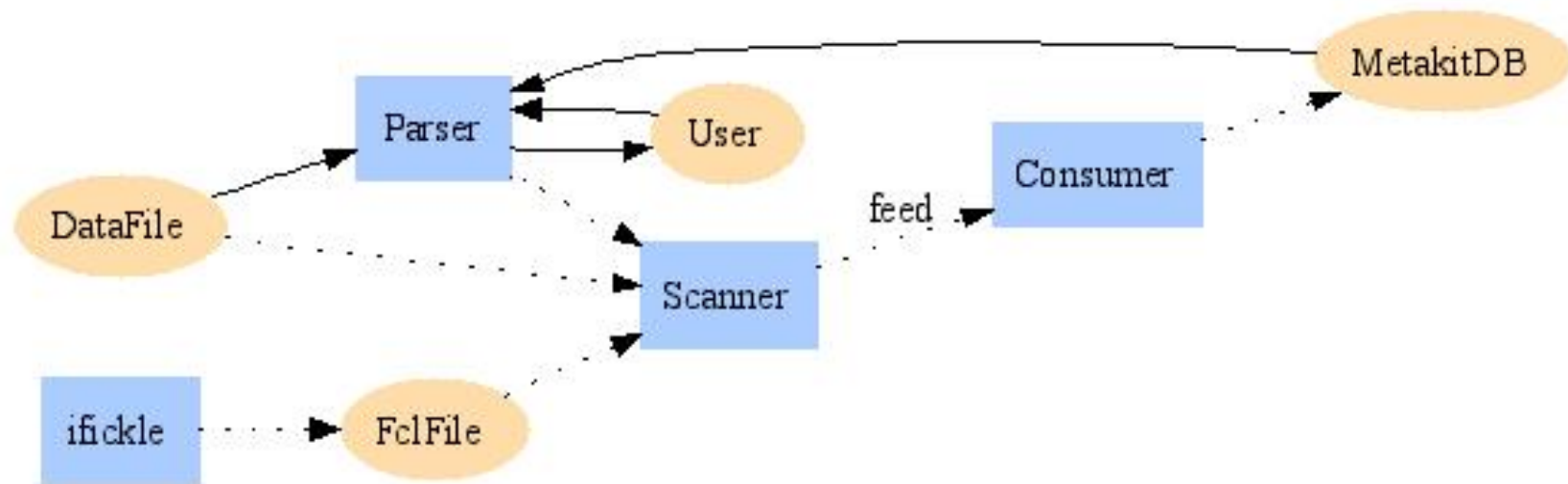
```
susi      retriever           6     12.0  Will
```

```
spot      dalmatian           12     18.0  Will
```

TSQL4MK library

```
(tsql4mk) 5 % source tsql4mk.kit
(tsql4mk) 6 % package require tsql4mk
0.1.4
(tsql4mk) 7 % tsql4mk tsql test.mk
::tsql
(tsql4mk) 8 % tsql eval \
    "select owner from dogs group by owner" v \
    { puts [array get v]}
* {owner group_owner} group_owner ::view7 owner Detlef # 0
* {owner group_owner} group_owner ::view8 owner Jean # 1
* {owner group_owner} group_owner ::view9 owner Jeff # 2
* {owner group_owner} group_owner ::view11 owner Will # 3
```

Parser Infrastructure



Tcl-Scanners

- fickle – by Jason Tang , GPL but Scanner free
 - ◆ Pros: no library dependencies, flex-like file input, faster
 - ◆ Cons: no encapsulation
- ylex: by Frank Pilhofer , License: Tclish
 - ◆ Pros: encapsulation via incrTcl-class
 - ◆ Cons: no flex-like input, slower, string input only
- ifickle: by Tang/Groth, License: see Fickle
 - ◆ Pros: see fickle, ylex;
 - ◆ Cons: none of fickle's, ylex's

WC with ifickle: iwc-fickle.fcl

```
%{  
#!/usr/bin/tclsh8.4  
public variable nline 0  
public variable nword 0  
public variable nchar 0  
%}  
%bufferize 1024  
%%  
\n      { incr nline; incr nchar 2 ; }  
[^\t\n]+ { incr nword; incr nchar $yylen ; }  
      { incr nchar ; }  
%%  
if {[llength $argv] == 0} {  
    puts stderr "usage wc-fickle inputfile"}
```

```
    exit 0
}
if {[catch {open [lindex $argv 0] r} yyin]} {
    puts stderr "Could not open [lindex $argv 0]"
    exit 0
}
set sc [iwcfickle \#auto -yyin $yyin]
$sc yylex
puts [format "%7d %7d %7d %s" \
    [$sc cget -nline] \
    [$sc cget -nword] \
    [$sc cget -nchar] [lindex $argv 0]]
close $yyin
```


Scanner Usage

```
$ tclkit ifickle.tcl iwc-fickle.fcl
$ tclkit iwc-fickle.tcl iwc-fickle.fcl
    26      102      601 iwc-fickle.fcl
$ wc  iwc-fickle.fcl
    26      102      627 iwc-fickle.fcl
```

Source iwc-fickle.tcl

```
package require Itcl
itcl::class iwc-fickle {
    public variable nline 0
    public variable nword 0
    public variable nchar 0

    variable yytext
    variable yyleng
    variable yyindex
    variable yylineno
    public variable yyin
    public variable yyout
    ...
    constructor ...
```

```
method yywrap {} {}  
method ECHO {{s ""}} {}  
  
...  
method yylex {} {}  
method input {} {}  
}
```

```
itcl::body iwcfickle::yywrap {} {  
    return 1  
}
```

...

```
itcl::body iwcfickle::yylex {} { ...
```

...

```
set sc [iwcfickle \#auto -yyin $yyin]
```

```
$sc yylex  
puts [format "%7d %7d %7d %s" \  
          [$sc cget -nline] \  
          [$sc cget -nword] \  
          [$sc cget -nchar] [lindex $argv 0]]  
close $yyin
```

WC-Results

Mode/Bytes	1	10	100	1000	10000	100000
tcl-fickle	0.361	0.362	0.396	0.635	1.948	14.990
tcl-yeti	0.824	0.826	0.861	1.161	4.230	36.344
tcl-ifickle	0.720	0.717	0.746	0.955	1.987	12.861

FastaScanner

```
>ENSMUST00000070533 cdna:novel chromosome:NCBIM33:1:3220 ...  
GGGCTCACCTCTTCTTCGTGGTGCTGGGCTCCCTTTCTGTGCAAGTGTTTCAGCTTCCGC  
TGGTTTGTGCATGATTTTCAGCACCGAGGACAGCTCCACGACCACCACCTCCAGCTGCCAG  
CAGCCTGGAGCAGATTGCAAGACGGTGGTCAGCAGTGGGTCTGCAGCCGGGGAAGGCGAG  
GTTCGTCCTTCCACGCCGCAGAGGCAAGCATCCAACGCCAGCAAGAGCAACATCGCCGCC  
CCATCTCCTCCAAGGCTGCAGTACAAGGATGATGCCCTTATTCAGGAGAGGCTGGAATAT  
GAAACCACTTTATAA
```

```
>ENSMUST00000073465 cdna:novel chromosome:NCBIM33:1:3673820:3  
gccgtgtacttcgcggatgtgggaacggacatctggctcgcggtggactactacctgcgt  
ggccagcgcctggtggtttgggctcaccctcttcttcgtggtgctgggctccctttctgtg  
caagtgttcagcttccgctgcaacggggccacccggaccagcggcaaacacaggtctgcg
```

Scanner-Results

scanner/kb	100	500	1000	2000	10000
fickle	0.32	1.05	2.21	3.77	18.24
ifickle	0.29	0.88	1.62	3.03	14.66
ylex	0.57	1.95	3.57	6.94	34.62

```
biotcl.kit FastaParser -fastafile /path/to/100kb.fa  
    -command 'getSeq ENSMUST0000073678'  
source biotcl.kit ; package require biotcl ;  
set fp [FastaParser \#auto -fastafile /path/to/fastafile]
```


Database Structure – dgMKViewer

dgMKViewer 0.4b5 - 100kb.fa.mk

File Options Help

- index
 - id
 - md5
 - pos:l
- sqlite_master
 - tbl_name
 - type
 - sql
- info
 - id
 - title

view	size	columns
index	52	id md5 pos:l
sqlite_master	2	tbl_name type sql
info	52	id title size:l

Data View S

start:

select join

index -first 0 -glob id *

n	rec	id	md5	pos:l
1	0	ENSMUST00000070533	c27cf03edbd1d782a3a...	0
2	1	ENSMUST00000073465	9313c795954c70d9e6...	1636
3	2	ENSMUST00000059871	a26f08807228fec5676...	1984
4	3	ENSMUST00000027032	3d978d752c34174bbb...	2959
5	4	ENSMUST00000027035	2066f9600d36d03ddaf...	10120
6	5	ENSMUST00000068739	61b79a89c6bfa9b8a82...	13397
7	6	ENSMUST00000061002	...	16700

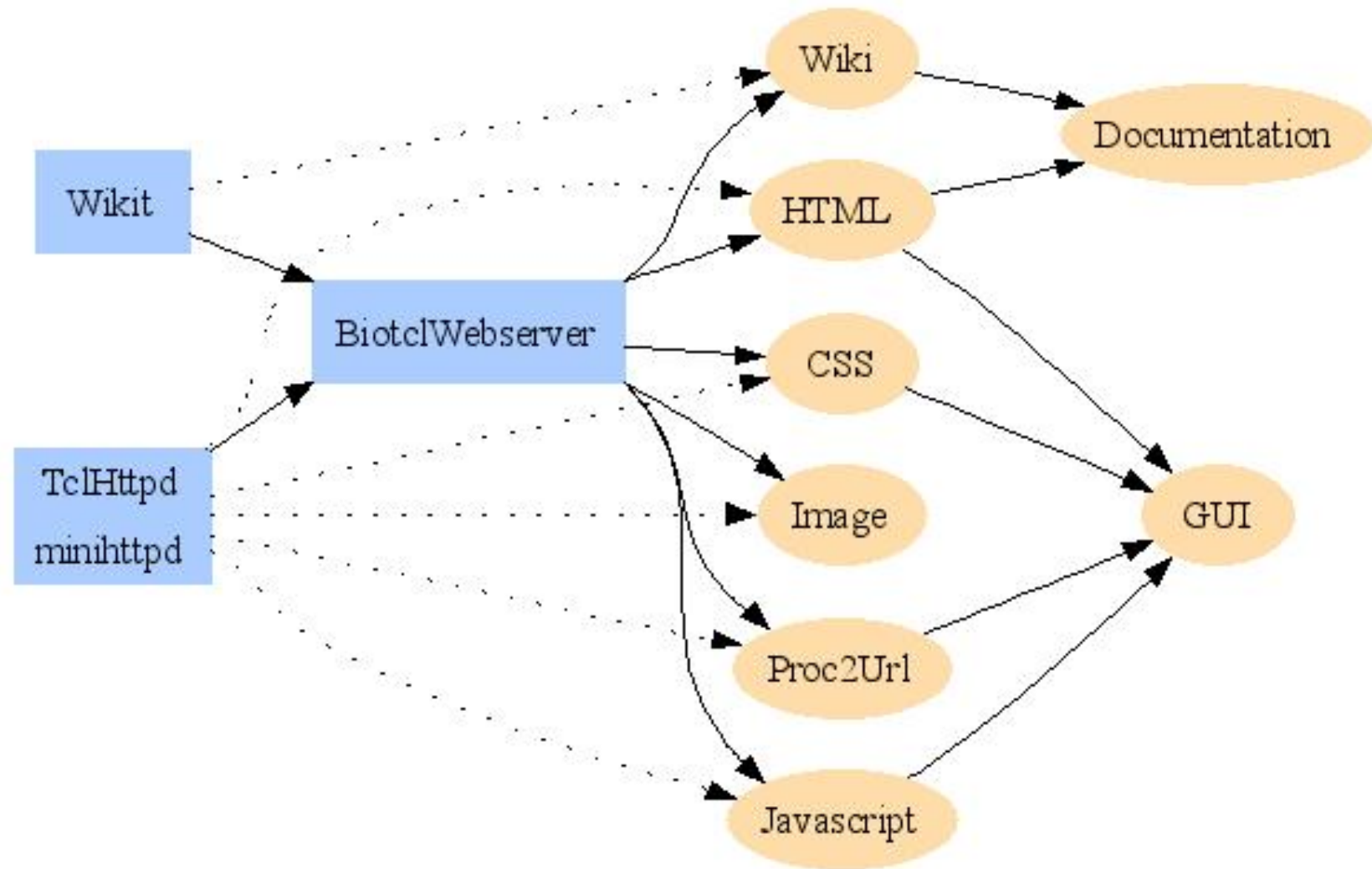
Webserver

- Wikit Documentation
- Static HTML Pages
- Cascading Stylesheets
- Javascript Support
- Image Support

```
biotcl.kit WebServer -httpd 8015
```

=> full fledged webapplications

Webserver-Implementation





GUI - FastaParser

orf_trans



getFasta



YAL002W

submit

```
>YAL002W VPS8 SGDID:S0000002, Chr I from 143709-147533, Verified ORF
MEQNGLDHDSRSSIDTTINDTQKTFLEFRSYTQLSEKLASSSSYTAPPLNEDGPKGVASA
VSQGSSESVSWTTTLTHVYSILGAYGGPTCLYPTATYFLMGTSKGCVLIFNYNEHLQITLV
PTLSEDPHSIRSPVKSIVICSDGTHVAASYETGNICIWNLNVGYRVKPTSEPTNGMTP
TPALPAVLHIDDHVNKEITGLDFFGARHTALIVSDRTGKVSLYNGYRRGFQVLVNSKKI
LDVNSSKEKLIRSKLSPLISREKISTNLLSVLTTHFALILLSPHVSLMFQETVEPSVQN
SLVNVSSISWTQNCSTVAYSNNKISVISISSDFNVQSASHSPEFAESILSIQWIDQLL
LGVLTISHQFLVLHPQHDFKILLRLDFLIHDLMIIPPKNKYFVISRRSFYLLTNYSFKIGKF
VSWSDITLRHILKGDYLGALFIESLLQPYCPLANLLKLDNNTTEERTKQLMEPFYNLSLA
ALRFLIKKDNADYNRVYQLLMVVVRVLQSSKKLDSIPSLDVFLQGLEFFELKDNAVYF
EVVANIVAQGSVTSISPVLFRSIIIDYYAKEENLKVIEDLIIMLNPTTLDVDLAVKLCQKY
NLFDLLIYIWNKIFDDYQTPVVDLIYRISNQSEKCVIFNGPQVPPETTIFDYVTYILTGR
QYPQNLSISPSDKCSKIQRELSAFIFSGFSIKWPSNSNHKLYICENPEEPAFPYFHLLI
KSNPSRFLAMLNEVFEASLFNDNDMVASVGEAELVSRQYVIDLLLDAMKDTGNSDNIRV
LVAIFIATSISKYPQFIKVSNOALDCVVNTICSSRVQGIYEISQIALESLLPYVHSRTTE
NFILELKEKNFNKVLFIYKSENKYASALSILETKDIEKEYNTDIVSITDYILKKCPPG
SLECGKVTEVIETNFDLLSRIGIEKCVTIFSDFDYNLHQEILEVKNEETQQKYLDKLF
TPNINNKVVDKRLRNLIHIELNCKYKSKREMILWLNGTVLSNAESLQILDLLNQDSNFEAAA
IIHERLESFNLAVRDLLSFIEQCLNEGKTNISTLLESRRAFDDCNSAGTEKKSCWILLI
TFLITLYGKYPSHDERKDLCKNKLQEAFLGLVRSKSSSQKDSGGEFWEIMSSVLEHQDVI
LMKVQDLKQLLLNVFNTYKLSLSLQKIIEDSSQDLVQQYRKFLSEGWSIHTDDCEI
CGKKIWGAGLDPLLFLAVENVQRHQDMISVDLKTPLVIFKCHHGFGHTCLENLAQKPDEY
SCLICQTESNPKIV*
```

GUI – JSComponents

- Tree
 - Collapse
 - Tabbox
 - TableSort
 - Barchart
 - Labelframe
 - u.a
-
- Supports: IE5+6, Mozilla/Firefox 1.4+, Opera7+
 - No need to code Javascript!!!
 - Separation of code and content

[Introduction](#)[Demo](#)[History](#)[Tree Control](#)[Collapse Control](#)[SearchList Control](#)[ToolTip Control](#)[Table Controls](#)[More](#)

first text

-   HTML Authoring
-   HTML References
 -  [Elements](#)
 -  [Character Sets](#)
-   HTML Applications (HTA)

HTML Source:

```
<div class="JSTree" id="idJSTree">
  <ul>
    <li>HTML Authoring
      <ul>
        <!-- space for IE5 in order to make the tree with an href op
        <li> <a href="#">Beginner's Guide</a>
          <ul>
            <li><a href="#">Beginner's Guide 1</a></li>
            <li><a href="#">Beginner's Guide 2</a></li>
```


Downloading == Installation

- Documentation
 - ◆ biotcl.kit WebServer –httpd ...
- Applications
 - ◆ biotcl.kit XYZParser –command ...
- Libraries
 - ◆ source biotcl.kit ; package require biotcl ...
- SQL–RDBMS
 - ◆ ln –s biotcl.kit tsql4mk.kit ; tsql4mk.kit test.mk "select * from .."
- Help
 - ◆ biotcl.kit

Configuration

```
$ head ../mytcl/biotcl.ini
```

```
[FastaParser]
```

```
01,FILE=/project/goblet/data/orf_trans.fasta.tmp(orf_trans)
```

```
01,INDEX=/project/goblet/data/orf_trans.fasta.tmp.mk
```

```
02,FILE=/project/goblet/data/orf_trans.fasta(orf_trans2)
```

```
02,INDEX=/project/goblet/data/orf_trans.fasta.mk
```

```
METHODS=getFasta,getTitles,getTitle
```

Status

- Webserver → ok
- RDBMS, tsq4mk → ok,, released
- Scanner construction → ok, release next week
- Applications → in progress
- Parsers → in progress
- official Release → 2–3 weeks, at least 6 scanners

Scanner–Parser Status

	Lib	Console	GUI
Fasta	+	+	+
Blast	+	–	–
GeneOntology	–	–	–
EntrezGene	–	–	–
1 per day...	–	–	–

Outlook

- critcl, re2c for speed
- vlerq
- soap or xmlrpc services
- jsComponents for improved GUI
- wiki collaboration
- goblet system
- insitu annotator

Acknowledgement

- Jean–Claude Wippler <http://www.equi4.com>
- William H. Duquette <http://www.wjduquette.com>
- D. Richard Hipp <http://www.sqlite.org>
- Csaba Nemethi
- Georgia Panopoulou MPIMG Berlin
- Albert Poustka MPIMG Berlin
- Steffen Hennig RZPD Berlin

About Me



1982 1984 1986 1988 1990 1992 1994 1996 1998 2000 2002 2004

Laborant -> Biochemistry :)

Army ---->

Biotech/GeneTherapy Bioinform

Programmer

Perl =====

Tcl--==

Note

This manual was written in Wikit–Wiki format and converted via an ifickle–generated scanner into HTML. HTML was afterwards converted into PDF usingHTMLdoc [<http://www.easysw.com/htmldoc/>] .

Sharpen : A Static Analysis Tool For Vignette's "TCL"

Brian Passingham
Passingham Software Ltd
327 Oxford Road
Macclesfield
Cheshire SK11 8JZ
England

E-mail: brian@passisoft.com

<http://www.passisoft.com/sharpen.html>

© Passingham Software Ltd, 2005

ABSTRACT

Early versions of the Vignette™ content management system (up to version 6) make use of an extension of the Tcl programming language, which we shall refer to as "TCL" to distinguish it from the mainstream Tcl language. We briefly discuss how various features of "TCL" can encourage a particular set of "Frequently Made Mistakes", leading to high development costs. Unfortunately, "TCL" is not understood by the best and most popular Tcl IDEs, and so these cannot be used to address these issues. We present a new tool, Sharpen, which can cope with the "TCL" extensions, and discuss how it can be used to reduce the maintenance costs of those Vignette legacy sites still using the "TCL"-based product. We also outline some possible future applications of the technology.

Introduction - Vignette and "TCL"

Early versions of the Vignette™ content management system (up to version 6) make use of an extension of the Tcl programming language, which we shall refer to as "TCL" to distinguish it from the mainstream Tcl language. The behaviour described here is that of the most commonly encountered configuration post V/5, in which interpreter re-use mode is enabled. We also assume that the original StoryServer behaviour with respect to EVAL's use of backslashes and SET's use of EVAL is present. V/6 provides options for more sensible functionality in these areas, but the costs and risks of updating a codebase to make use of these options seem to deter companies from making use of them, except, obviously, on new installations.

For a more complete description of the architecture of StoryServer, as the Vignette/TCL product was initially named, refer to the Vignette documentation. For our purposes here all we need to know is that a pool of page generator processes, each containing a Tcl interpreter, are used to deliver dynamic web pages by evaluating “TCL” templates. Template code is held in a database, and propagated to public-facing (“live”) and private (“development”) web servers. Typically, a content management application is run on the “development” web servers and used to publish data through to the live servers. Libraries of code may be included in templates through either the INCLUDE command (for TCL code necessitating the use of EVAL) or the SOURCE command (for normal Tcl code). In interpreter re-use mode, the SOURCE ... PERSIST command may be used to load code once for the duration of a page generator's existence.

Each “TCL” template consists of text with Tcl commands embedded using []. So far, so *subst*-like. However, Vignette wished to make scripted HTML generation as easy as possible, and so also introduced some control flow constructs of their own, aimed at the generation of text. A Vignette-specific mechanism for long comments of the form [# ...] is also supported. Thus a simple template generating a barely formatted table of some selected fiction might look like this:

```
[# Template: Simple Template
  Path : /simpleminded/library
  ... further identification details ...]

<table>
[SEARCH TABLE books INTO bookDetails SQL “select * from library”
FOREACH book IN [SHOW books] {
  [IF {[FIELD $book category] == “novel” && [BookSatisfiesUser $book]} {
    [# Generate a table row for each novel the user “likes”, according to their criteria]
    <tr><td>[FIELD $book author]</td>
      <td>[FIELD $book title]</td>
      <td>[FIELD $book date]</td></tr>
  ]
}]
</table>
```

Note that the bodies of the FOREACH and IF statements are treated in the same way as the template itself – they are passed to the Vignette EVAL command.

Some Frequently Made Mistakes

In some ways the most important type of mistake made with StoryServer is the common-or-garden Tcl coding error.

Web applications are all-too-commonly tested almost entirely by a black-box method. Path coverage tends, therefore, to be weaker than one would wish. In the context of a language which is parsed and compiled on demand this can lead to embarrassingly basic errors finding their way through to live sites. For example, I have seen all-too-many page generation failures which have causes as simple as this :

```
[IF {[diceThrow] == 6} {
  <p>[string strange [SHOW text] 0 20]</p>
}]
```

The frequent need to refer to characters needing a backslash is also often a cause of syntax errors, since code within N levels of EVAL requires 2 to the power N backslashes to be used! Sixteen backslashes look much like fifteen to the naked eye...

There are also subtler, Vignette-specific issues to contend with.

Consider the case where a developer omits a call to SOURCE a frequently called library. Statistically, it is very likely that any particular interpreter in the development environment will already have loaded the library. The code will only fail when the template is the first one called from a page generation process. Once the failure has occurred, of course, it is more likely to occur again, since a new page generator instance will be restarted...

There is also a similar, but thankfully rarer, problem in which code in a re-used library will only work on its first call, since its use invalidates its own pre-condition.

Mismatches in data scoping and control flow behaviour between the Vignette commands and “normal” Tcl can also cause problems – a full description of these would divert this paper unduly. Suffice to say that best practice at StoryServer sites tends to frown on the excessive usage of SET and SHOW.

If we're lucky (read disciplined!), the ease of making mistakes just leads to additional costs and delays when the problem is detected in regression testing. If we're unlucky, parts of the live web sites either fail completely, or misbehave in subtle ways. If we're really unlucky, there are enough problems to start destabilizing the system to a point where the delivery of any web pages at all becomes problematic.

For this reason alone, even a basic parser and usage checker for StoryServer can lead to important quality improvements.

Unfortunately, the main Tcl IDEs such as ActiveState's TclDev and T-IDE (apologies to any I've missed) provide no support for StoryServer. The issue of accessing code in a database can easily be resolved, of course. The issue is that of parsing TCL. The precise syntax of Vignette's [#...] comments is undocumented, and the excess backslashes of EVALed code defeat Tcl's own parser.

Implementation of Sharpen

For this reason we have implemented a parser package for TCL, **::sharpen**.

We had initially hoped to build on Tcl's internal parser, as exposed by a Tcl interface in TclPro. This is clearly the best way of guaranteeing a correct parse. However, experiments in this direction proved the strategy not to be viable. The extra backslashes present (due to EVAL's backslash escaping) in most TCL code cause real difficulties. Working with a “backslash eliminated” version of the text, along with a map back to the original text, we made some headway, but abandoned the approach after considering the memory overheads (StoryServer codebases can be *big*, since multiple sites are often maintained in a single environment).

As a result our approach is broadly similar to that of the incomplete 'parsetcl', though our representation of the parse tree differs considerably. We use an associative array to represent the parse tree, mainly since our code needs to run in an 8.2 context when tightly integrated with Vignette. We use a callback mechanism to reparse calls of procs which themselves have a significant syntactic component. We were tempted to drive this mechanism by a file of rules, as **Nagelfar** does, but decided that this might close the architecture - we intend to extend **::sharpen** to

perform parses of several sublanguages.

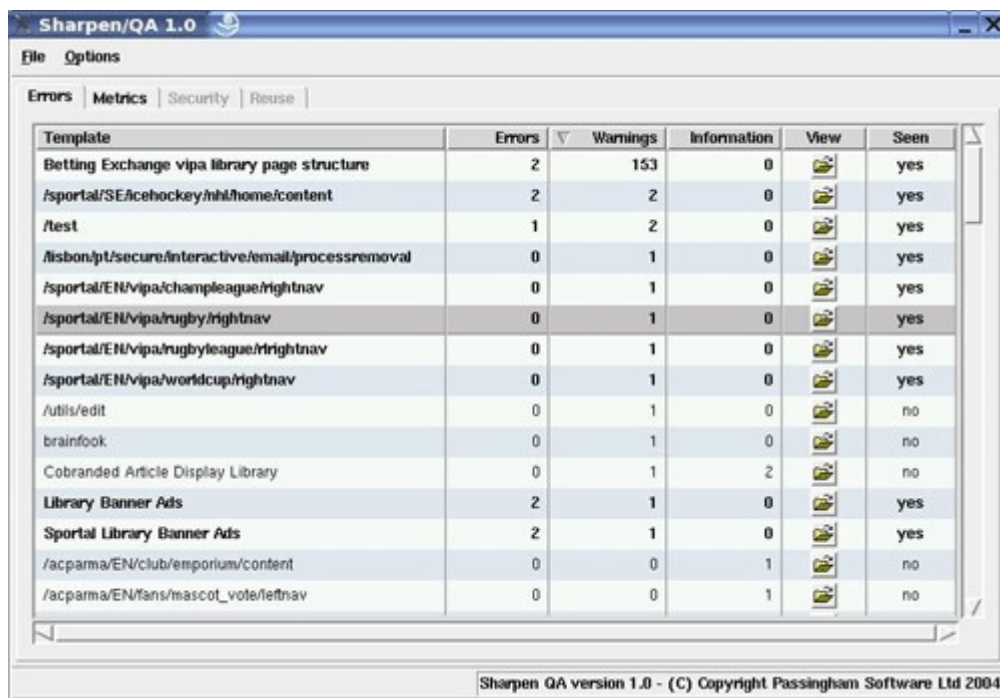
We also implemented a Vignette package parsing package, `::vpkg`, to cope with the serialized data files produced by the *transferproject* utility used to move code and its meta-data between environments. (It seems likely here that we have duplicated the efforts of the TrapEd product, which provides an editor for such packages.)

Current Capabilities of Sharpen

The current stable release of the Sharpen product consists of versions of the `::sharpen` and `::vpkg` packages, along with an example tool, **Sharpen/QA**, which uses these to provide basic QA facilities for Vignette TCL and Tcl templates, either individually, *en masse*, or from a package.

Sharpen/QA is a configurable tool for identifying quality assurance issues in Vignette TCL code, with support for reviewing individual templates, packages, or entire codebases. Reports from **Sharpen/QA** identify the *procs* defined and used by particular templates, allowing the cross-checking of software releases against particular environments. In addition to the basic capabilities of identifying syntax errors and usage problems, it is also possible to use Sharpen/QA to identify common problems such as attempts to release code that contains inappropriate calls (such as the use of `ERROR_TRACE` as a coarse-grained diagnostic). These static analysis features should reduce errors when undertaking releases. They are also of continual use during the development process. Integration with existing development tools may be possible, depending on the software used.

Sharpen/QA uses Csaba Nemethi's tablelist widget by default.



The screenshot shows the Sharpen/QA 1.0 application window. It has a menu bar with 'File' and 'Options'. Below the menu bar are tabs for 'Errors', 'Metrics', 'Security', and 'Reuse'. The 'Errors' tab is selected, displaying a table with the following columns: Template, Errors, Warnings, Information, View, and Seen. The table lists various templates and their associated error and warning counts. At the bottom of the window, a status bar reads 'Sharpen QA version 1.0 - (C) Copyright Passingham Software Ltd 2004'.

Template	Errors	Warnings	Information	View	Seen
Betting Exchange vipa library page structure	2	153	0		yes
/sportal/SE/icehockey/nhl/home/content	2	2	0		yes
/test	1	2	0		yes
/lisbon/pt/secure/interactive/email/processremoval	0	1	0		yes
/sportal/EN/vipa/chamleague/rightnav	0	1	0		yes
/sportal/EN/vipa/rugby/rightnav	0	1	0		yes
/sportal/EN/vipa/rugbyleague/rightnav	0	1	0		yes
/sportal/EN/vipa/worldcup/rightnav	0	1	0		yes
/utils/edit	0	1	0		no
brainfook	0	1	0		no
Cobranded Article Display Library	0	1	2		no
Library Banner Ads	2	1	0		yes
Sportal Library Banner Ads	2	1	0		yes
/acparma/EN/club/emporium/content	0	0	1		no
/acparma/EN/fans/mascot_vote/leftnav	0	0	1		no

The above screenshot shows the Error frame following analysis of a codebase, with templates

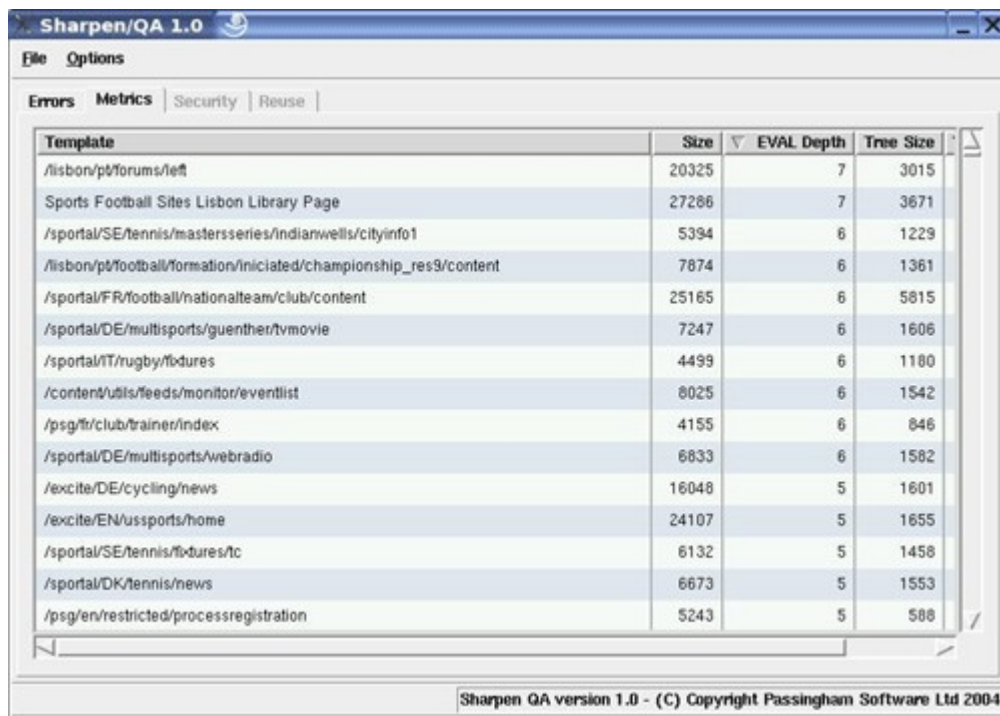
shown along with the number of categorised issues found within them. When applied to the now-defunct Sportal codebase (which supported the main Sportal portal as well as the websites of numerous top-ranked European football teams), Sharpen's QA tool uncovered numerous templates with serious problems that would have led to page generation failures, along with numerous instances of inefficient or risky coding practices. Experience to date suggests that this pattern is likely to be repeated at many Vignette TCL installations.

The following screenshot shows the Sharpen/QA template viewer, built using a Tk text widget tagged by the parse tree and its annotations. Due to the particular tag settings in use, it shows lightly syntax-highlighted TCL code (normal Tcl code in this example). Several errors have been identified and highlighted in red.



The next screenshot shows the statistics of the Metrics frame, which can be used to readily identify a site's most complex templates (since these are likely to require the most testing/development effort). The statistics shown here are:

- File size - the simplest measure of a template's complexity, which can be misleading. There is nothing inherently complex about a flat "terms and conditions" template, for example.
- Tree size - a more accurate measure of the template's complexity, recording the size of the parse tree (as well as some overhead caused by repeated reparsing). There is some evidence from Beizer that claims a significant correlation between this type of figure and the expected number of errors caused by the code.
- EVAL depth - a Vignette-specific metric, recording the extent to which commands like EVAL, IF, and FOREACH have been nested. Excessive nesting can lead to serious degradation in performance - we would recommend the use of normal Tcl control structures instead for all but the simplest of cases.



The screenshot shows the Sharpen/QA 1.0 application window. It has a menu bar with 'File' and 'Options'. Below the menu bar are tabs for 'Errors', 'Metrics', 'Security', and 'Reuse'. The 'Metrics' tab is selected, displaying a table with the following data:

Template	Size	EVAL Depth	Tree Size
/lisbon/pt/forums/left	20325	7	3015
Sports Football Sites Lisbon Library Page	27286	7	3671
/sportal/SE/tennis/mastersseries/indianwells/cityinfo1	5394	6	1229
/lisbon/pt/football/formation/initiated/championship_res9/content	7874	6	1361
/sportal/FR/football/nationalteam/club/content	25165	6	5815
/sportal/DE/multisports/guenther/tvmovie	7247	6	1606
/sportal/IT/rugby/fixtures	4499	6	1180
/content/Utils/feeds/monitor/eventlist	8025	6	1542
/psg/fr/club/trainer/index	4155	6	846
/sportal/DE/multisports/webradio	6833	6	1582
/excite/DE/cycling/news	16048	5	1601
/excite/EN/ussports/home	24107	5	1655
/sportal/SE/tennis/fixtures/tc	6132	5	1458
/sportal/DK/tennis/news	6673	5	1553
/psg/en/restricted/processregistration	5243	5	588

At the bottom of the window, it says 'Sharpen QA version 1.0 - (C) Copyright Passingham Software Ltd 2004'.

Future Work

The detailed parse trees obtained through **::sharpen** and **::vpkg** can be used as the basis of more sophisticated tools – **Sharpen/QA** is a fairly straightforward application designed to test the technology and provide immediate benefits to Vignette/TCL developers.

One key area for future work will be extending the tool's abilities in the area of data flow analysis. There are of course, theoretical obstacles here – it is always possible for a Tcl developer to produce an example of dynamic code that will defeat a particular analytical technique – but even a naïve data flow analysis will work on the majority of StoryServer code and be capable of detecting common errors, such as code branches in which particular variables are used before they are set.

We are considering supporting further metrics. In particular, interest has been shown in some form of McCabe's cyclomatic complexity metric. We believe that the conventional definition of this will

need to be supplemented by a further metric indicating the presence of non-imperative coding structures. A preliminary search of the literature has failed to find anything immediately suitable.

Operations to transform code whilst preserving its meaning are definitely possible, and offer great potential.

Particularly useful operations might be:

1. To eliminate EVAL-based control structures such as IF and FOREACH in favour of a byte-compilable procs which use normal Tcl control structures and *append* to build up a result string. This can yield significant performance improvements.
2. Excess backslash elimination to allow code to be migrated to a V/6 installation using the “sane backslashing” configuration. Such code is undoubtedly easier to read, and will be easier to develop, simply by eliminating the common confusion as to how many backslashes are required at a particular point.
3. Code instrumentation, allowing the construction of a StoryServer debugger and/or a code coverage tool.

More ambitiously, a full-blown TCL refactoring editor is a tempting prospect, though, given the limited time-frame within which Vignette will continue to support their TCL-based products, we are unlikely to embark on this (for this purpose at least).

We intend to demonstrate some proof-of-concept implementations in some of these areas at the meeting, and will publish these on <http://www.passisoft.com> after the meeting.

VCRI: A groupware application for CSCL research

Jos Jaspers (J.Jaspers@fss.uu.nl)
Marcel Broeken (m.h.broeken@fss.uu.nl)

Educational Sciences
Utrecht University
The Netherlands

Paper presented at the European Tcl/Tk User Meeting
Bergisch Gladbach (Germany), 27-28 May 2005

Abstract

One of the main research topics of the Educational Sciences group at Utrecht University is Computer Supported Collaborative Learning (CSCL). The CROCiCL^{*} project¹ is a research project, which focuses on CSCL and the effects of visualization of social aspects of collaboration processes in CSCL. The project started in September 2003 and will take about 4 years to complete. We have developed a groupware environment called VCRI, which enables users to collaborate and communicate. In this paper we will give an overview of the development history of the VCRI, its features, problems we encountered, our plans for the future and of course Tcl's part in all this.

Introduction

Secondary school students in The Netherlands – as a result of recent changes in the curriculum of the final years (the 'study house') – are doing increasingly independent research in preparation for college studies. The focus has shifted towards working actively, constructively and collaboratively, as this is believed to enhance learning. We have developed a groupware computer environment that supports these research activities that should fit well within this curriculum. The purpose of our research is to investigate the effect of the computer supported research environment and its tools on the final product through differences in the participants' collaboration processes.

VCRI overview

^{*} The CROCiCL project (Computerized Representation of Coordination in Collaborative Learning) is funded by N.W.O., the Dutch Organization for Scientific Research, under project number 411-02-121

The VCRI (Virtual Collaborative Research Institute) is a client-server based groupware environment providing a customizable tool-set. Currently there are about 14 tools, ranging from a collaborative text processor (Co-writer) to an instant messaging client (Chat). Any subset of tools can be used, to give users true flexibility. Adding new (third-party) tools is currently not supported but we are working on providing a clear framework for developers to make this possible.

One of the key ideas behind the VCRI is WYSIWIS (What You See Is What I See). Users share most tools. A shared tool is continuously synchronized and looks the same to every user. All users can edit the content of the tool, for some tools even simultaneously. The VCRI server is in charge of synching all tools. This mechanism of sharing gives the impression of real-life collaboration, even in cyberspace.

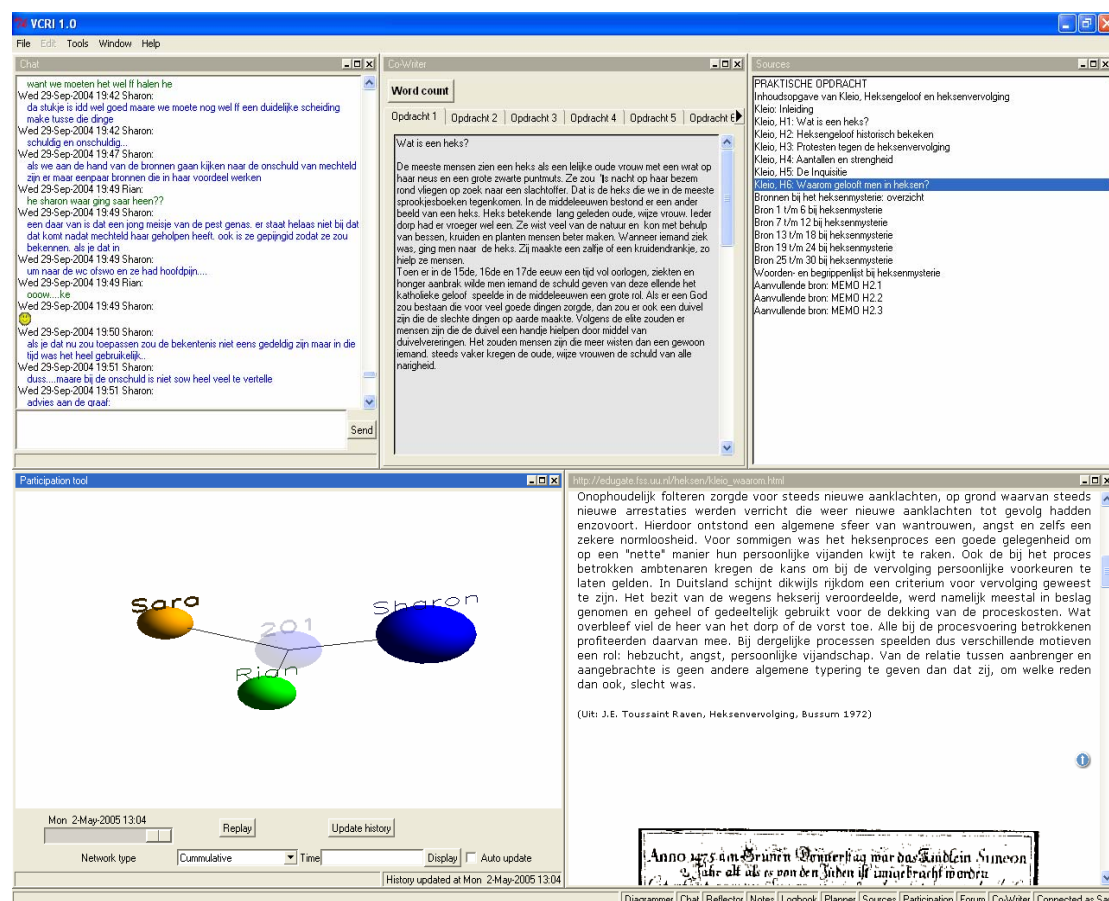


Figure 1 : The VCRI environment

Figure 1 shows a screen dump of the VCRI with some of tools. From left to right and from top to bottom:

- Chat: a synchronous communication tool
- Co-writer: enables the participants to write the texts for the different assignments
- Sources: Contains links to source materials for the assignments
- Participation tool: provides participants with a graphic display of their participation
- Source : A particular source was opened by the student

In this study, students have to collaborate in groups of 3 participants on a inquiry project about witches in medieval society based on historical sources. The research question focuses on the effects of participation awareness on the collaboration of the students. The main purpose of the VCRI is to enable students to collaborate on research projects, to help teachers to guide students while they are collaborating, and to enable researchers to collect data on the process of collaboration. Therefore all significant user events are logged by the server to enable our researchers to study the effects of our tools on the collaboration process. From the main log file, MEPA² files can be extracted. MEPA is an application for the annotation, coding and statistical analysis of verbal or nonverbal observational data or protocols, making analysis easier.

A bit of history

The VCRI is the result of years of developing different tools for CSCL research at the Utrecht University. The first version was developed around 1995. This version was written in Visual Basic. One of the more surprising results was the feedback we received from our subjects. They actually liked the collaborative writing. One of the problems was the handling of socket communication. This proved cumbersome and the program was rewritten in Delphi (Pascal). This provided some improvement.

As we started on a subsequent project we sought an additional programmer to speed up the development of the next version. We found the company Equi4 willing to assist us. One of the first decisions was the choice of the programming language. TCL was chosen for two main reasons:

1. Our application deals mostly with texts
2. Our application uses networking

This makes TCL a natural choice. Development on the VCRI is driven by the research demands. These demands tend to change frequently, making it very important to use a flexible and

interpreted programming language like Tcl. Its tight integration with Tk also makes it perfect for creating nice gui's with little effort. Another key feature was cross platform availability, giving us the ability to run in almost every school. The fact that Tcl is open source also was an important pro. Last but not least, is Tcl's clear and simple syntax making Tcl's learning curve steep.

Architecture

The VCRI is a client-server application. In principle, the client does all the heavy work while a customized TclHttpd³ server distributes updates and saves the tool content. Distributing the work over all clients reduces the server's workload and optimizes cpu usage. The client is kept as thin as possible, only providing a framework for communicating with the server and a login window. On user login the server queries a Metakit⁴ database for the toolset and lets the client remotely source the corresponding files. This kind of remote sourcing, or dynamic loading, makes rapid development and bug fixes possible. Users can keep using the same client while still getting all the updates and bug fixes from the server every time they log in.

Client and server communication is http based. The client issues http requests with Tcl commands to be executed by the server. The server returns scripts to be executed on the client side as result for the http request. The main reason for using this rpc like communication is firewalls. Until recently, most schools which participated in our experiments used KennisNet, a government funded intranet. This KennisNet blocked almost all ports (incoming and outgoing) and no exceptions were made. This prohibited a permanent socket connection. Since http requests and port 80 are always available, VCRI communication also became http based. The clients poll the server in regular intervals to exchange information.

Packages used

In developing the VCRI we've used some extensions and tools to provide the features pure Tcl/Tk was missing.

TclHttpd

This easily extensible and lightweight web server written in Tcl was perfect for our VCRI server. It provides a basic framework for client-server interaction giving us the room to focus on VCRI's specific features. VCRI's use of http(s) for server communication also made TclHttpd a logical choice. At this time, our customized TclHttpd server runs on (SUSE) Linux, Windows, and Mac OS X.

Starpack

Schools are more willing to join when the effort on their part can be made as small as possible. An easy and straightforward installation is an important way to reduce that effort. Packaging the VCRI client

as a Starpack makes installation equal to drag-and-drop and deinstallation to delete. A Starpack is a Starkit plus a Tclkit runtime. *Starpacks are standalone executables, which run out of the box, making them even easier to distribute and use than Starkits.*⁵

Metakit

Earlier versions of the VCRI used plain files for storing persistent data. Increasing complexity and scale of the VCRI have made maintaining, archiving and tweaking these files more painstaking and placed a need for more structured and consistent data storage. We have decided to use a lightweight database for data storage. We have chosen Metakit as our database because of its small size, efficiency, ease of use, and its easy to use bindings for Tcl (Mk4Tcl and Oomk⁶). Currently the VCRI server uses one Metakit database to store data, although in some cases plain files are still used.

TkHTML

The VCRI has a number of tools, which generate or display HTML. At first, we used OpTcl which is *a Tcl extension to provide connectivity to the resident hosts component model.*⁷ Unfortunately, it only works for COM on Windows, which clashes with our cross platform objective. Another con is the inability to interact with OpTcl at a low level, handling events and manipulating the HTML content.

Therefore, we decided to move away from OpTcl.

Finally, we have chosen TkHTML⁸, a Tcl extension written in C for rendering HTML-content. As a side note, the TkHTML project has recently been revitalized. CSS and XHTML support are among the project's main priorities.

TkOGL

Some of the tools offer 3D visualizations. Since Tcl/Tk lacks 3D capabilities we've used TkOGL⁹ for this purpose. TkOGL is a Tcl extension written in C providing an interface to the OpenGL framework. It currently works on Linux and Windows but a Mac OS X build is also planned. TkOGL's most recent version (3.2) makes it possible to use OpenGL commands without (almost) any modifications.

[Incr]Tcl

[Incr]Tcl¹⁰ is a language extension for Tcl enabling object oriented programming by introducing the notion of classes, objects and other OO related terminology. By adding this extra layer of abstraction, it's easier to write and maintain large programs.

Problems

During VCRI development, we have come across a number of problems of which a couple will be highlighted in this paper. In the

next paragraph, we will present some of our solutions to these problems.

Legacy code

As described in A bit of history the VCRI has come a long way since 1995. A lot of features and tools have been added but design wise things have stayed the same. This unchecked growth has lead to redundant and duplicate code, overlapping functionality and too many dependencies. At the beginning of the CRoCiCL project the decision was made to continue work on the current version of the VCRI instead of starting from scratch. The complexity of this legacy code has made it difficult to maintain and extend the code. Object oriented programming has turned out to be the solution to this problem.

User interface

The VCRI lacks eye candy. While this doesn't present a problem to those who are only interested in functionality, it's an important issue when designing software for teenagers, the target audience of the VCRI. In our experiments, it's crucial that the students enjoy working with the VCRI. An appealing user interface is key, especially for those users who have grown up with cool looking software. Another GUI related problem is the cross platform (and as a result non-platform) and inconsistent look of the VCRI, making it virtually impossible for users to make use of their knowledge of GUI conventions on their platform of choice. Unfortunately, lack of time and know-how makes it hard to fix this problem.

Platform dependencies

One of the main reasons for choosing Tcl is its cross platform availability. As development continued, shortcomings in Tcl/Tk's functionalities were patched using extensions. Not all of these extensions were 100% pure Tcl; as a result platform dependencies returned in the VCRI in the form of binary extensions such as TkHTML and TLS.

Different interests

Finally, there's been the problem of different interests among the project's stakeholders. Many times interests of users, researchers, and developers have conflicted. For example, adding a feature for research purposes can lead to unnecessary clutter (from the user's point of view). Furthermore, keeping everything as lean as possible makes logging all user events a bit tricky. Luckily, every one in the CRoCiCL team has at least a little experience in both research and programming, making it easier to resolve these conflicts by looking at the problem from different points of view.

Printing

One of our more recent problems is printing. For our application, we are looking for a printing solution to enable users to print the content of any VCRI tool window on any printer on any platform. Printing can be broken down into different sub problems. The first sub problem is to create a printable version of a tool's content. The other sub problem is letting the user (or the VCRI) select a printer and finally sending this printable version to the selected printer. Unfortunately, existing printing solutions only tackle one of these sub problems and are almost always platform dependent.

Solutions

In this paragraph we will present solutions to some of our problems mentioned in the Problems section. Please note that not all problems are discussed.

Object oriented programming

Some of the problems we mentioned earlier are closely related. The problems with complexity, redundancy and updating and using legacy code have to do with unraveling the relations between different chunks of code. Key ideas of object oriented programming are modularity and refactoring. By putting related code in one place (an object) interaction between different chunks of code is made simpler and more transparent. When all related functionality is provided by one object, the complexity of an application is greatly reduced. Instead of maintaining and adding code at different places, just one object needs to be changed when using an OO design. This also addresses the problem of redundancy because it's easier to recognize and fix redundancies.

Students and Tile

Another problem is the spartan look of the VCRI, which cripples the user experience a bit. Since lack of time is still a problem, we have been trying to involve students in information sciences and interaction design. At the end of June, our application will be used as the subject of an assignment on user testing as part of the course usability engineering at Utrecht University. Hopefully, this will provide usable feedback on usability and look of our GUI. In the future, we hope to be a part of similar projects.

Tile¹¹, *an improved themeing engine for Tk*, is another possible solution for our GUI related problems. Tile can use so-called themes to provide a consistent look and feel. Themes consolidate all GUI options in one place, which minimizes the effort to create customized themes for different platforms, experiments or users.

The Future

The VCRI will be used in at least two more experiments, one in September 2005 and one in September 2006. After the experiments and possible subsequent projects have finished, the VCRI will be released as free (and maybe open source) software for educational use. We're also planning to give third party developers the ability to extend the VCRI with new tools and functionality. Our aim is to release a product, which is easy to install (both client and server), easy to use, cross platform and fun.

Summary

Using Tcl to create the VCRI has proven to be a good choice. It's perfect for rapid prototyping, which has proved to be a pro in this research project with constantly evolving demands and requirements. The few shortcomings of Tcl we have encountered have been fixed by third party extensions. Being half way in the development of the VCRI in this project we can state that using Tcl has contributed a great deal to the result!

References

1. CRoCiCL: Computerized Representation of Coordination in Collaborative Learning, <http://edugate.fss.uu.nl/~crocicl/>
2. MEPA, <http://edugate.fss.uu.nl/mepa/>
3. Tcl Web Server, <http://www.tcl.tk/software/tclhttpd/>
4. Metakit by Equi4 Software, <http://www.equi4.com/metakit.html>
5. Beyond Tclkit - simplified deployment of scripted applications by Steve Landers, presented at the 2002 Tcl/Tk conference, Vancouver, <http://www.equi4.com/papers/skpaper1.html>
6. Object-oriented Metakit for Tcl, <http://www.equi4.com/oomk.html>
7. OpTcl, <http://www2.cmp.uea.ac.uk/~fuzz/optcl/default.html>
8. A HTML Widget For Tcl/Tk, <http://tkhtml.tcl.tk/>
9. TkOGL, <http://hct.ece.ubc.ca/research/tkoggl/tkoggl/index.html>
10. [incr Tcl] - Object-Oriented Programming in Tcl/Tk, <http://incrtcl.sourceforge.net/itcl/>
11. Tile: an improved themeing engine for Tk, <http://tktable.sourceforge.net/tile/>

Updated: April 26, 2005

Doing 3D with Tcl

Paul Obermeier
obermeier@poSoft.de

Abstract

This paper presents an approach called **tclogl**, which offers the 3D functionality of OpenGL at the Tcl scripting level. Tclogl is an improved and enhanced OpenGL binding based on the work done with Frustum by Roger E Critchlow. The paper starts with an overview of existing 3D libraries with a Tcl scripting interface. Different solution approaches are discussed and compared against the given requirements. The chosen implementation, which relies heavily on SWIG, is explained in detail in the main section of this paper. Common pitfalls when programming OpenGL in Tcl, as well as open issues of this approach are shown. Finally the results of a range of test programs and some demonstration applications are shown.

1 Overview

Hardware accelerated 3D capabilities are available nowadays on nearly every PC. There is also a broad range of programming libraries for doing 3D visualization, coming from different application domains, like simulation, gaming, visualization or animation.

These libraries differ in availability on computer architectures and operating systems, complexity and richness of supplied functionality, as well as the supported language bindings.

There are two low-level (light-weight) graphic APIs in common use today: OpenGL and DirectX. While DirectX from Microsoft is available only on machines running the Windows operating system, OpenGL is running on PC's as well as on workstations. OpenGL also has a software-only implementation called "Mesa", so you can run OpenGL based programs even in virtual machines or over a network. OpenGL libraries are part of all major operating systems distributions.

DirectX and OpenGL both offer a C based programming interface.

Based on one of these 2 low-level APIs lots of heavy-weight libraries exist, available as OpenSource implementations as well as commercial versions, adding features like scene-graphs, image handling, animation, advanced lighting models, etc.

Most of these libraries offer a C/C++ language binding, but only a few of them enable the user to "script" a 3D application.

Some examples of 3D libraries offering a Tcl language binding are listed in the following overview. The libraries are divided into the above mentioned categories heavy-weight and light-weight. Only non-commercial libraries are taken into account.

You may also take a look at the OpenGL related Tcl'ers Wiki page ([5]).

Name	Platforms	Source	Reference URL
Nebula	X11/Win/MacX	Yes	http://www.nebuladevice.org
Fltk	X11/Win/MacX	Yes	http://www.fltk.org
VRS	X11/Win	Yes	http://www.vrs3d.org
VTK	X11/Win/MacX	Yes	http://public.kitware.com/VTK
tk3d	X11/Win	Yes	http://www.gm.com/company/careers/career_paths/rnd/lab_manuf_sw.html

Table 1: List of heavy-weight 3D libraries with Tcl binding

Name	Platforms	Source	Reference URL
Glut/Tk	X11/Win	Yes	http://zing.ncsl.nist.gov/gluttk
Tkogl	X11/Win	Yes	http://hct.ece.ubc.ca/research/tkogl/tkogl
toogl	X11/Win/MacX	Yes	http://toogl.sourceforge.net
Frustum	X11/Win	Yes	http://www.elf.org/pub/frustum01.zip
XBit	Win	No	http://www.geocities.com/~chengye/opengl.html
tom	X11/Win	Yes	http://sourceforge.net/projects/om2t

Table 2: List of light-weight 3D libraries with Tcl binding

The following short excerpts from the libraries' home pages should act as a brief introduction and overview of their capabilities.

Nebula Device is an open source realtime 3D game/visualization engine, written in C++. Version 2 is a modern rendering engine making full use of shaders. It is scriptable through TCL/Tk and Lua, with support for Python, Java, and the full suite of .NET-capable languages pending. It currently supports DirectX 9, with support for OpenGL in the works. It runs on Windows, with ports being done to Linux and Mac OS X.

FLTK (pronounced "fulltick") is a cross-platform C++ GUI toolkit for UNIX®/Linux® (X11), Microsoft® Windows®, and MacOS® X. FLTK provides modern GUI functionality without the bloat and supports 3D graphics via OpenGL® and its built-in GLUT emulation.

The Virtual Rendering System is a computer graphics software library for constructing interactive 3D applications. It provides a large collection of 3D rendering components which facilitate implementing 3D graphics applications and experimenting with 3D graphics and imaging algorithms. VRS is implemented as a C++ library. Applications can incorporate VRS as C++ library based on the C++ API. In addition, we provide a complete Tcl/Tk binding of the C++ API, called iVRS.

The Visualization ToolKit (VTK) is an open source, freely available software system for 3D computer graphics, image processing, and visualization used by thousands of researchers and developers around the world. VTK consists of a C++ class library, and several interpreted interface layers including Tcl/Tk, Java, and Python.

Tk3D is a collection of extensions to Tcl/Tk that allow Tcl/Tk applications to manipulate large numerical arrays and generate 3D graphic displays. The Tk3D suite contains five packages, named Tns, Vtd, Fct, Fctr, and Tnsph. The "Tns" (tensor) package is a numerical array extension. It provides facilities for efficiently manipulating multidimensional arrays of numbers within Tcl. The Vtd package provides a Tk widget, called a view3d widget, in which to display 3D graphic images. This widget's functionality can be extended by adding "renderers," which are programs, written in C, for drawing objects in a view3d widget.

GLUT/Tk is a "light-weight" system that seeks to leverage GLUT and Tcl/Tk by tying them together in a stylistically consistent way with the addition of only a few commands to each. The basic implementation strategy is to enable a GLUT process to launch an independent Tk script. Thus, the built-in event loops of these two systems can operate as usual and the resulting programming style (registering callbacks for given events) is unchanged.

TkOGL is a package extension to the Tcl scripting language that enables a user to utilize OpenGL, a multi-platform API for interactive 2D and 3D graphics applications. TkOGL makes it possible for the user to display OpenGL graphics on the Tk canvas along with other Tk widgets.

Togl is a Tk widget for OpenGL rendering. Togl allows one to create and manage a special Tk/OpenGL widget with Tcl and render into it with a C program. That is, a typical Togl program will have Tcl code for managing the user interface and a C program for computations and OpenGL rendering.

Frustum implements a specialization of the Togl widget and a Swig generated Tcl binding for the opengl and glu libraries to allow 3d modelling to be done entirely from Tcl.

XBit has implemented a Tcl shell for OpenGL primitives at Windows platforms. The implementation focuses on scriptive programming in OpenGL rendering with an emphasis on code reusability and GUI. It provides an OpenGL rendering engine whose states can be changed with a greater flexibility during execution.

Tom is an OpenGL wrapper for Tcl/Tk. It provides Tcl procs very close to OpenGL C functions.

2 Wish and reality

2.1 Requirements

As has been shown in the previous chapter, a number of 3D libraries with Tcl bindings are currently available. But none of them fulfilled my personal wish list for a Tcl enabled 3D library: It should give me the ability to integrate small- to medium- sized 3D content into my Tcl/Tk based graphical user interfaces.

The preferred candidate should be an OpenGL based light-weight package, because OpenGL is available on nearly every platform. 3D functionality should be scriptable with Tcl commands and it should be possible to extend the functionality with C code. Graphical output should be displayed in a Tk widget.

The following table summarizes the requirements of my favourite Tcl-3D library.

#	2.1.1 Requirement	Comment
1	Light-weight	Small code size, Tcl package.
2	License	Source code availability under BSD license.
3	High automation	No need to write lots of wrapper/glue code. Easy upgrade to newer versions of the 3D library.
4	Portable	Availability on many platforms.
5	C and Tcl IF	Ability to program the library in both C and Tcl. Easy interchange between Tcl and C code.
6	Up to date	Buildable with actual tools and operating systems.

Table 3: Requirements for the Tcl 3D-library

2.2 Discussion of available solutions

Glut/Tk uses the GLUT library. Although GLUT is available on different platforms, it has not been actively supported for quite some time. GLUT contains lot of operating system dependent code covering features like event handling or simple menus, features that are already handled by Tk.

Tom has not been updated for a while and consists of a hand-crafted interface to OpenGL.

Togl allows programming OpenGL in C only.

X-Bit is not available as source code.

Out of the 6 possible solutions listed in Table 2 the following packages left over for a more detailed inspection:

Tkogl, currently maintained by the University of British Columbia in Vancouver and **Frustum** by Roger E Critchlow Jr, which is not maintained by the author anymore.

Both have a very similar approach: Wrap the OpenGL core libraries GL and GLU with SWIG ([6]), and display the contents in a Tk widget.

The next table lists the features which didn't fit my requirements:

	TkOgl	Frustum
Use of old SWIG version 1.1	Yes	Yes
OpenGL header files modified	Yes	Yes
Handcrafted tables for mapping GLenums	Ys	No

SWIG 1.1 is not supported any more and may be not available on newer versions of operating systems. The current SWIG version is 1.3.24 and this version offers lots of new features.

Edited OpenGL header files need manual changes when compiling on platforms with a newer OpenGL version, otherwise the additional commands are not available. Changes in the API have to be done by hand, too.

OpenGL declares a bunch of enumerations, as can be seen in the following table. These differ from platform to platform and keeping them up-to-date manually for the different platforms and versions would not be reasonable.

GL_VENDOR	SGI	Microsoft Corporation
GL_VERSION	1.1 Irix 6.5	1.1.0
GLU_VERSION	1.2 Irix 6.5	1.2.2.0 Microsoft Corporation
Number of gl commands	485	352
Number of glu commands	68	67
Number of gl enums	1041	588
Number of glu enums	138	116

So the final decision was to follow the Frustum approach, which only needed two parts to be cleaned up and extended.

3 Implementation

3.1 SWIG-based OpenGL wrapper

The first task was to create a language binding for the OpenGL core libraries GL and GLU with the help of SWIG ([6]). As stated earlier, it should work with an actual version of SWIG and the OpenGL header files should not be touched.

Due to the new version of SWIG and its extended typemap features it was possible to generate a consistent mapping between C functions and equivalent Tcl commands without changing the OpenGL header files `gl.h` and `glu.h`.

The following tables show, how parameters and return values of the C based OpenGL functions are mapped to Tcl command parameters and return values. Every type of parameter is explained with a typical example.

Note:

- The notation `TYPE` stands for any scalar value (`GLboolean`, `GLbyte`, `GLubyte`, `GLshort`, `GLushort`, `GLint`, `GLuint`, `GLfloat`, `GLdouble`). It is not used for type `void`.
- The notation `STRUCT` stands for any C struct.

Input parameter	GLenum
C declaration	<code>void glEnable (GLenum cap);</code>
C example	<code>glEnable (GL_BLEND);</code>
Tcl example	<code>glEnable GL_BLEND glEnable \$::GL_BLEND</code>

GLenum as an OpenGL function input parameter can be supplied as numerical value or as name.

Input parameter	GLbitfield
C declaration	<code>void glClear (GLbitfield mask);</code>
C example	<code>glClear (GL_COLOR_BUFFER_BIT);</code>
Tcl example	<code>glClear GL_COLOR_BUFFER_BIT glClear \$::GL_COLOR_BUFFER_BIT</code>

GLbitfield as an OpenGL function input parameter can be supplied as numerical value or as name.

Note:

- A combination of bit masks has to be specified as a numerical value like this:
`glClear [expr $::GL_COLOR_BUFFER_BIT | $::GL_DEPTH_BUFFER_BIT]`

Input parameter	GLboolean
C declaration	<code>void glEdgeFlag (GLboolean flag);</code>
C example	<code>glEdgeFlag (GL_TRUE);</code>
Tcl example	<code>glEdgeFlag GL_TRUE glEdgeFlag \$::GL_TRUE</code>

GLboolean as an OpenGL function input parameter can be supplied as numerical value or as name.

The mapping of the types GLenum, GLbitfield and GLboolean is handled in file *consthash.i*.

Input parameter	TYPE
C declaration	<code>void glTranslatef (GLfloat x, GLfloat y, GLfloat z);</code>
C example	<code>glTranslatef (1.0, 2.0, 3.0); glTranslatef (x, y, z);</code>
Tcl example	<code>glTranslatef 1.0 2.0 3.0 glTranslatef \$x \$y \$z</code>

Scalar types as an OpenGL function input parameter must be supplied as numerical value.

The mapping of scalar types is handled by the SWIG standard typemaps.

Input parameter	const TYPE[SIZE], const TYPE *
C declaration	<code>void glMaterialfv (GLenum face, GLenum pname, const GLfloat *params);</code>
C example	<code>GLfloat mat_diffuse = { 0.7, 0.7, 0.7, 1.0 }; glMaterialfv (GL_FRONT, GL_DIFFUSE, mat_diffuse) ;</code>
Tcl example	<code>set mat_diffuse { 0.7 0.7 0.7 1.0 } glMaterialfv GL_FRONT GL_DIFFUSE \$mat_diffuse</code>

Constant pointers as an OpenGL function input parameter must be supplied as a Tcl list.

The mapping of const TYPE pointers is handled in file ***autoarray.i***.

Note:

- This type of parameter is typically used to specify small vectors (2D, 3D and 4D) as well as control points for NURBS.
- Unlike in the C version, specifying data with the scalar version of a function (ex. `glVertex3f`) is faster than the vector version (ex. `glVertex3fv`) in Tcl.
- Note, that Tcl lists given as parameters to an OpenGL function have to be flat, i.e. they are not allowed to contain sublists. When working with lists of lists, you have to flatten the list, before supplying it as an input parameter to an OpenGL function. One way to do this is shown in the example below.

```
set ctrlpoints {
    {-4.0 -4.0 0.0} {-2.0 4.0 0.0}
    { 2.0 -4.0 0.0} { 4.0 4.0 0.0}
}
glMap1f GL_MAP1_VERTEX_3 0.0 1.0 3 4 [join $::ctrlpoints]
```

Input parameter	const GLvoid *
C declaration	<code>void glVertexPointer (GLint size, GLenum type, GLsizei stride, const GLvoid *ptr);</code>
C example	<code>static GLint vertices[] = { 25, 25, 100, 325, 175, 25, 175, 325, 250, 25, 325, 325}; glVertexPointer (2, GL_INT, 0, vertices);</code>
Tcl example	<code>set vertices [VectorFromArgs GLint \ 25 25 100 325 175 25 \ 175 325 250 25 325 325] glVertexPointer 2 GL_INT 0 \$::vertices</code>

Constant void pointers as an OpenGL function parameter must be given as a pointer to a contiguous piece of memory of appropriate size.

The mapping of const void pointers is handled by the SWIG standard typemaps.

Note:

- The allocation of useable memory can be accomplished with the use of the `vector` command, which is described later in this chapter.
- This type of parameter is typically used to supply image data or vertex arrays. See also the description of the Tk photo mapping later in this chapter.

Output parameter	TYPE *, GLvoid *
C declaration	<pre>void glGetFloatv (GLenum pname, GLfloat *params); void glReadPixels (GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid *pixels);</pre>
C example	<pre>GLfloat values[2]; glGetFloatv (GL_LINE_WIDTH_GRANULARITY, values); GLubyte *vec = malloc (w * h * 3); glReadPixels (0, 0, w, h, GL_RGB, GL_UNSIGNED_BYTE, vec);</pre>
Tcl example	<pre>set values [Vector GLfloat 2] glGetFloatv GL_LINE_WIDTH_GRANULARITY \$values set vec [Vector GLubyte [expr \$w * \$h * 3]] glReadPixels 0 0 \$w \$h GL_RGB GL_UNSIGNED_BYTE \$vec</pre>

Non-constant pointers as an OpenGL function parameter must be given as a pointer to a contiguous piece of memory of appropriate size.

The mapping of non-constant pointers is handled by the SWIG standard typemaps.

Function return	TYPE, STRUCT *
C declaration	<pre>GLuint glGenLists (GLsizei range); GLUnurbs* gluNewNurbsRenderer (void);</pre>
C example	<pre>GLuint sphereList = glGenLists(1); GLUnurbsObj *theNurb = gluNewNurbsRenderer(); gluNurbsProperty (theNurb, GLU_SAMPLING_TOLERANCE, 25.0);</pre>
Tcl example	<pre>set sphereList [glGenLists 1] set theNurb [gluNewNurbsRenderer] gluNurbsProperty \$theNurb GLU_SAMPLING_TOLERANCE 25.0</pre>

Scalar return values are returned as the numerical value.

Pointer to structs are returned with the standard SWIG mechanism of encoding the pointer in an ASCII string.

The mapping of return values is handled by the SWIG standard typemaps.

Note:

- The next lines show an example of SWIG's pointer encoding:

```
% set theNurb [gluNewNurbsRenderer]
% puts $theNurb
_10fa1500_p_GLUnurbs
```

The returned name can only be used in functions expecting a pointer to the appropriate struct.

Exceptions from the standard rules

The GLU library as specified in header file **glu.h** does not provide an API, that is as consistent as the GL core library. So one class of function parameters (**TYPE ***) is handled differently with GLU functions. Arguments of type **TYPE*** are used both as input and output parameters in the C version. In GLU 1.2, which is the current version, most functions use this type as input parameter. Only two functions use this type as an output parameter.

So for GLU functions there is the exception, that **TYPE*** is considered an input parameter and therefore is wrapped as a Tcl list.

Input parameter	TYPE * (GLU only)
C declaration	<pre>void gluNurbsCurve (GLUnurbs *nobj, GLint nknots, GLfloat *knot, GLint stride, GLfloat *ctlarray, GLint order, GLenum type);</pre>
C example	<pre>GLfloat curvePt[4][2] = {{0.25, 0.5}, {0.25, 0.75}, {0.75, 0.75}, {0.75, 0.5}}; GLfloat curveKnots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0}; gluNurbsCurve (theNurb, 8, curveKnots, 2, &curvePt[0][0], 4, GLU_MAP1_TRIM 2);</pre>
Tcl example	<pre>set curvePt {0.25 0.5 0.25 0.75 0.75 0.75 0.75 0.5} set curveKnots {0.0 0.0 0.0 0.0 1.0 1.0 1.0 1.0} gluNurbsCurve \$theNurb 8 \$curveKnots 2 \$curvePt 4 GLU_MAP1_TRIM 2</pre>

The two aforementioned functions, which provide output parameters with **TYPE*** are `gluProject` and `gluUnProject`. These are handled as a special case in the SWIG interface file `glu.i`. The 3 output parameters are given the keyword **OUTPUT**, so SWIG handles them in a special way: SWIG builds a list consisting of the normal function return value, and all parameters marked with that keyword. This list will be the return value of the corresponding Tcl command.

Definition in glu.h	Redefinition in SWIG interface file glu.i
<pre>extern GLint gluUnProject (GLdouble winX, GLdouble winY, GLdouble winZ, const GLdouble *model, const GLdouble *proj, const GLint *view, GLdouble* objX, GLdouble* objY, GLdouble* objZ);</pre>	<pre>GLint gluUnProject (GLdouble winX, GLdouble winY, GLdouble winZ, const GLdouble *model, const GLdouble *proj, const GLint *view, GLdouble* OUTPUT, GLdouble* OUTPUT, GLdouble* OUTPUT);</pre>

Example usage (see Redbook ([1]) example **unproject.tcl** for complete code):

```
glGetIntegerv GL_VIEWPORT $viewport
glGetDoublev  GL_MODELVIEW_MATRIX $mvmatrix
glGetDoublev  GL_PROJECTION_MATRIX $projmatrix
set viewList [VectorToList $viewport 4]
set mvList   [VectorToList $mvmatrix 16]
set projList [VectorToList $projmatrix 16]

set really [expr [$viewport get 3] - $y - 1]
set winList [gluUnProject $x $really 0.0 $mvList $projList $viewList]
puts "gluUnProject return value: [lindex $winList 0]"
puts [format "World coords at z=0.0 are (%f, %f, %f)" \
    [lindex $winList 1] [lindex $winList 2] [lindex $winList 3]]
```

3.2 Extension of the Togl widget

Now that we have a Tcl binding of the OpenGL functionality, we need to be able to display the 3D contents.

Togl is an actively maintained Tk widget with support to display OpenGL graphics, but the drawing commands have to be specified in C.

To be usable from the Tcl level, it has been extended to support 3 new configuration options for specifying Tcl callback commands:

-createproc	TclCommandName	Called when a new widget is created.
-reshapeproc	TclCommandName	Called when the widget's size is changed.
-displayproc	TclCommandName	Called when the widget's content needs to be redrawn.

These configuration options behave like standard Tcl options as shown in the example below:

```
% package require Togl
1.6
% togl .t
% .t configure -displayproc tclDisplayFunc
% .t configure -displayproc
-displayproc displayproc Displayproc {} tclDisplayFunc
```

So a minimal 3D application looks like the following “Hello, World” OpenGL program.

```
# hello.tcl

package require tclogl
package require Togl

proc tclDisplayFunc {} {
    glClear GL_COLOR_BUFFER_BIT

    # draw white polygon (rectangle) with corners at
    # (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0)
    glColor3f 1.0 1.0 1.0
    glBegin GL_POLYGON
        glVertex3f 0.25 0.25 0.0
        glVertex3f 0.75 0.25 0.0
        glVertex3f 0.75 0.75 0.0
        glVertex3f 0.25 0.75 0.0
    glEnd
    glFlush
}

proc tclCreateFunc {} {
    # select clearing color
    glClearColor 0.0 0.0 0.0 0.0

    # initialize viewing values
    glMatrixMode GL_PROJECTION
    glLoadIdentity
    glOrtho 0.0 1.0 0.0 1.0 -1.0 1.0
}

proc tclReshapeFunc { w h } {
    .fr.toglwin postredisplay
}
```

```

frame .fr
pack .fr -expand 1 -fill both
togl .fr.toglwin -width 250 -height 250 -double false \
               -createproc tclCreateFunc
.frb.toglwin configure -displayproc tclDisplayFunc \
                      -reshapeproc tclReshapeFunc
grid .fr.toglwin -row 0 -column 0 -sticky news

bind . <Key-Escape> "exit"

```

Note that `-createproc` is not effective, when specified in the `configure` subcommand. It has to be specified at widget creation time.

The changes in the widget code allow Togl to execute Tcl callbacks with the help of `Tcl_Eval`, while still maintaining 100% of it's original functionality. Only a few lines had to be added or changed in the Togl source code:

1. Add the 3 new configuration options to the `Tk_ConfigSpec` list.
2. Declaration and definition of the 3 new internal evaluation functions: `tcloglCreateProc`, `tcloglDisplayProc`, `tcloglReshapeProc`.
3. Change the default callbacks to point to the new internal evaluation functions.

These 3 changes are shown with the create callback as example:

1.


```

{TK_CONFIG_STRING|TK_CONFIG_NULL_OK, "-createproc", "createproc",
"Createproc", NULL, Tk_Offset(struct Togl, createCallback), 0, NULL},

```
2.


```

static int tcloglCreateProc (struct Togl *togl) {
    if (togl->createCallback) {
        if (Tcl_Eval (Togl_Interp(togl), togl->createCallback) != TCL_OK) {
            Tcl_BackgroundError (Togl_Interp(togl));
            free (togl->createCallback);
            togl->createCallback = NULL;
            return TCL_ERROR;
        }
    }
    return TCL_OK;
}

```
3.


```

static Togl_Callback *DefaultCreateProc = tcloglCreateProc;

```

3.3 Utility functions

All of the features listed in this chapter are not necessary for operation, but offer extended or easier functionality.

3.3.1 The Vector command

As stated in chapter 3.1, some of the OpenGL functions need a pointer to a contiguous block of allocated memory. SWIG already provides a feature to automatically generate wrapper functions for allocating and freeing memory of any type. This feature `%array_functions` also creates setter and getter functions for accessing the allocated memory.

The following definitions provided in file ***tclogl.i*** create the accessor functions for the OpenGL base types:

```
// Generate array functions (new, delete, getitem, setitem) for the
// following types.
```

```
%array_functions(unsigned char,GLboolean)
%array_functions(signed char,GLbyte)
%array_functions(unsigned char,GLubyte)
%array_functions(short,GLshort)
%array_functions(unsigned short,GLushort)
%array_functions(int,GLint)
%array_functions(unsigned int,GLuint)
%array_functions(float,GLfloat)
%array_functions(double,GLdouble)
```

The generated wrapper code looks like this (Example shown for GLdouble):

```
static double *new_GLdouble(int nelements) {
    return (double *) calloc(nelements,sizeof(double));
}

static void delete_GLdouble(double *ary) {
    free(ary);
}

static double GLdouble_getitem(double *ary, int index) {
    return ary[index];
}

static void GLdouble_setitem(double *ary, int index, double value) {
    ary[index] = value;
}
```

The file ***tclogl/Vector.tcl*** contains additional Tcl commands for encapsulation of these low-level accessor functions.

Tcl command	Explanation
Vector	Call the memory allocation routine <code>new_*</code> and create an OO like Tcl interface. (See example below)
VectorFromList	Create a new Vector from given Tcl list.
VectorFromArgs	Create a new Vector from given arguments.
VectorFromString	Create a new GLubyte Vector from given string.
VectorToString	Copy the contents of a GLubyte Vector into a string.
VectorToList	Copy the contents of a Vector into a Tcl list.

The following example shows the usage of the base `Vector` command.

```
set ind 23
set vec [Vector GLfloat 123] ; # Create a new Vector of size 123 GLfloats
set x [$vec get $ind]        ; # Get element at index 23
$vec set $ind 1017.0         ; # Set element at index 23 to 1017.0
$vec delete                  ; # Free the allocated memory
```

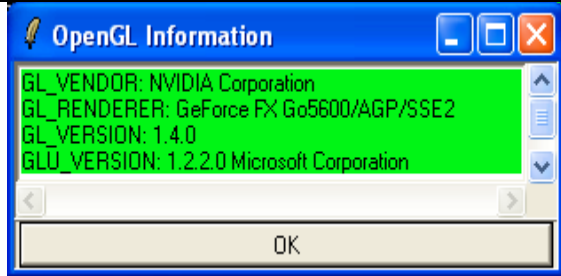
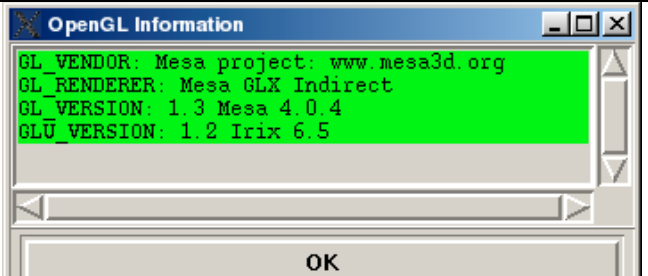
3.3.2 Information utilities

In file ***tcloglInfo.tcl*** three utility functions are currently implemented to get information about the OpenGL version, the installed extensions, as well as the current OpenGL state.

<code>tcloglGetVersions</code>	Query the OpenGL library with the keys <code>GL_VENDOR</code> , <code>GL_RENDERER</code> , <code>GL_VERSION</code> , <code>GLU_VERSION</code> and return the results as a list of key-value pairs.
--------------------------------	--

The following code snippet shows how to call `tcloglGetVersions` and place the result in a text widget.

```
foreach glInfo [tcloglGetVersions] {
    set msgStr "[lindex $glInfo 0]: [lindex $glInfo 1]\n"
    $textId insert end $msgStr
}
```

	
Example output of <code>glGetVersions</code>	Example output of <code>glGetVersions</code> on SGI/Linux

<code>tcloglGetExtensions</code>	Query the OpenGL library with the keys <code>GL_EXTENSIONS</code> and <code>GLU_EXTENSIONS</code> and return the results as a list of key-value pairs.
----------------------------------	--

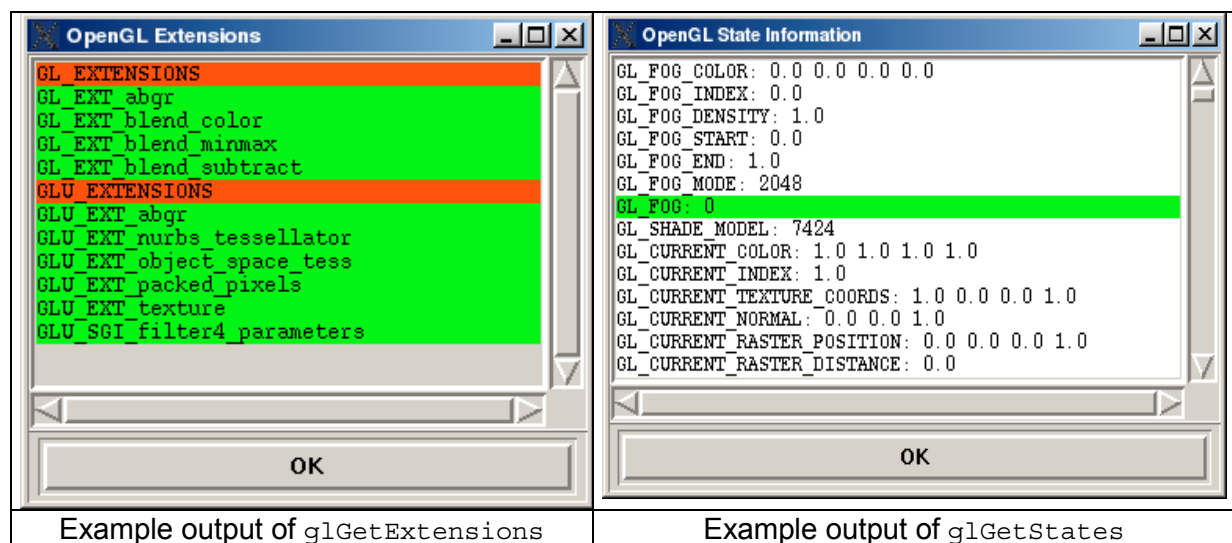
The following code snippet shows how to call `tcloglGetExtensions` and place the result in a text widget.

```
foreach glInfo [tcloglGetExtensions] {
    set msgStr "[lindex $glInfo 0]\n"
    $textId insert end $msgStr type
    foreach ext [lsort [split [string trim [lindex $glInfo 1]]]] {
        set msgStr "$ext\n"
        $textId insert end $msgStr name
    }
}
```


<code>tcloglGetStates</code>	Query all state variables of the OpenGL library and return the results as a list of sub-lists. Each sublist contains the querying command used, the key and the value(s).
------------------------------	---

The following code snippet shows how to call `tcloglGetStates` and place the result in a text widget.

```
foreach glState [tcloglGetStates] {
    set msgStr "[lindex $glState 1]: [lrange $glState 2 end]\n"
    if { [string compare [lindex $glState 0] "glIsEnabled"] == 0 } {
        set tag bool
    } else {
        set tag other
    }
    $textId insert end $msgStr $tag
}
```



Note:

The functions `glGetString` and `gluGetString` as well as the corresponding high-level functions `tcloglGetVersions` and `tcloglGetExtensions` only return correct values, if a Togl window has been opened, i.e. a rendering context has been established.

3.3.3 Tk photo mapping

In file *tkphoto.i* the following C functions are implemented to provide access to the Tk photo image functionality.

Tcl command	Usage
<code>PhotoChans</code>	Return the number of channels of a Tk photo.
<code>Photo2Vector</code>	Copy a Tk photo into a Vector in OpenGL raw image format. The Vector must have been allocated with the appropriate size and type.
<code>Vector2Photo</code>	Copy from OpenGL raw image format into a Tk photo. The photo image must have been initialized with the appropriate size and type.

These functions are best explained by looking at the following code excerpts from the simple image viewer *imgViewer.tcl*:

Example 1: Read an image into a Tk photo and use it as a texture map. Note: Texture map images must have width and height, that are multiples of 2.

```
proc ReadImg { imgName } {
    global gPo

    set retVal [catch {set phImg [image create photo -file $imgName]} err1]
    if { $retVal != 0 } {
        puts "Failure reading image $imgName"
    } else {
        set w [image width $phImg]
        set h [image height $phImg]
        set sqr [GetBestSquare $w $h]
        set gPo(texScaleS) [expr double ($w) / $sqr]
        set gPo(texScaleT) [expr double ($h) / $sqr]
        set sqrPhoto [image create photo -width $sqr -height $sqr]
        $sqrPhoto copy $phImg -from 0 0 $w $h -to 0 [expr $sqr - $h]
        update
        set vecImg [Vector GLubyte [expr $sqr * $sqr * 4]]
        Photo2Vector $sqrPhoto $vecImg
        image delete $phImg
        image delete $sqrPhoto
        glTexParameteri GL_TEXTURE_2D GL_TEXTURE_WRAP_S $::GL_CLAMP
        glTexParameteri GL_TEXTURE_2D GL_TEXTURE_WRAP_T $::GL_CLAMP
        glTexParameteri GL_TEXTURE_2D GL_TEXTURE_MAG_FILTER $::GL_NEAREST
        glTexParameteri GL_TEXTURE_2D GL_TEXTURE_MIN_FILTER $::GL_NEAREST
        glTexImage2D GL_TEXTURE_2D 0 4 \
            $sqr $sqr \
            0 GL_RGBA GL_UNSIGNED_BYTE $vecImg
        tclDisplayFunc
    }
}
```

Example 2: Read an image from the OpenGL framebuffer and save it with the Img library.

```
proc SaveImg { imgName } {
    global gPo

    set w $gPo(toglWidth)
    set h $gPo(toglHeight)
    set numChans 4
    set vec [Vector GLubyte [expr $w * $h * $numChans]]
    glReadPixels 0 0 $w $h GL_RGBA GL_UNSIGNED_BYTE $vec
    set ph [image create photo -width $w -height $h]
    Vector2Photo $vec $ph $w $h $numChans
    set fmt [string range [file extension $imgName] 1 end]
    $ph write $imgName -format $fmt
}
```

The actual size of the Togl window (`gPo(toglWidth)`, `gPo(toglHeight)`), which is needed in command `SaveImg`, can be saved in a global variable when the reshape callback is executed.

```
proc tclReshapeFunc { w h } {
    global gPo

    set gPo(toglWidth) $w
    set gPo(toglHeight) $h
    ...
}
```

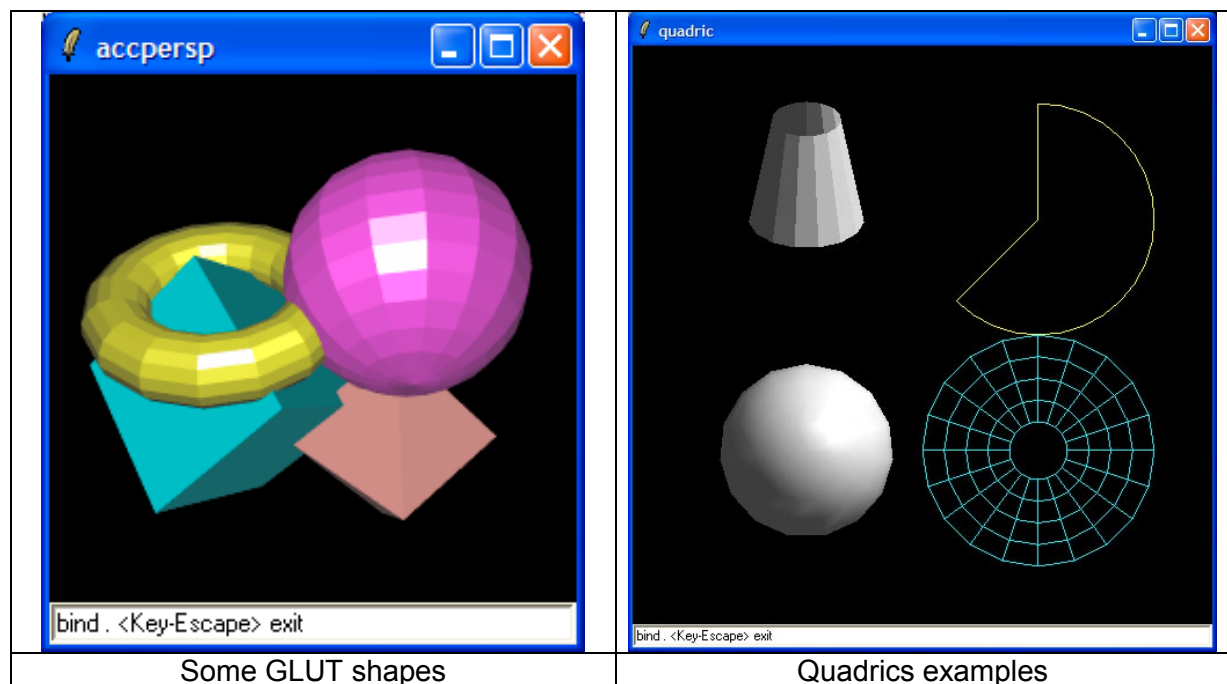
3.3.4 Additional tclogl utilities

The utilities in this chapter have been added for testing and demonstration purposes.

GLUT shapes library

The shape objects implemented in the GLUT library are available under the same names for running the test programs of the OpenGL redbook ([1]).

Solid shapes	Wire shapes
<code>glutSolidCube</code>	<code>glutWireCube</code>
<code>glutSolidCone</code>	<code>glutWireCone</code>
<code>glutSolidSphere</code>	<code>glutWireSphere</code>
<code>glutSolidTorus</code>	<code>glutWireTorus</code>
<code>glutSolidTetrahedron</code>	<code>glutWireTetrahedron</code>
<code>glutSolidOctahedron</code>	<code>glutWireOctahedron</code>
<code>glutSolidDodecahedron</code>	<code>glutWireDodecahedron</code>
<code>glutSolidIcosahedron</code>	<code>glutWireIcosahedron</code>
<code>glutSolidTeapot</code>	<code>glutWireTeapot</code>



Some GLUT shapes

Quadrics examples

The shapes library consists of the C files (*teapot.c* for the teapot, *shapes.c* for all other shapes and the common header file *shapes.h*) and the Tcl file *tcloglShapes.tcl*.

The shape library also acts as a demonstration, how to extend the tclogl package with C code.

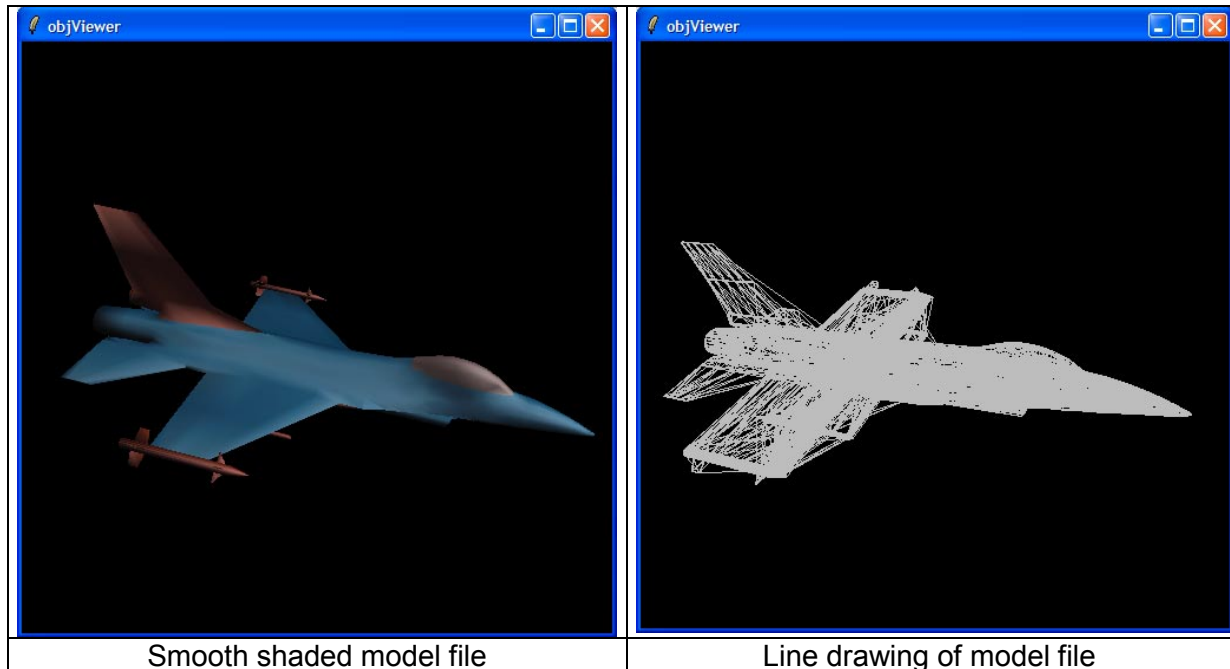
The steps necessary are:

1. Compile your C source files (*shapes.c*, *teapot.c*)
2. Put the name of the header file (*shapes.h*) into SWIG interface file *util.i*.
3. Call SWIG to create a new wrapper file.
4. Relink your dynamic library with the new object files (*shapes.o*, *teapot.o*).

Alias/Wavefront modelfile reader

A simple viewer for 3D models has been implemented in **objViewer.tcl**

It can read model files in Alias/Wavefront format. The code to read and draw the models is taken from Nate Robin's OpenGL tutorial ([4]). The corresponding files are **glm.c** and **glm.h**.



4 Caveats / Common pitfalls

Some OpenGL functions expect an integer or floating point value, which is often given in C code examples with an enumeration, as shown in the next example:

```
extern void glTexParameteri ( GLenum target, GLenum pname, GLint param );
```

It is called in C typically as follows:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

As the 3rd parameter is not of type `GLenum`, you have to specify the numerical value here:

```
glTexParameteri GL_TEXTURE_2D GL_TEXTURE_WRAP_S $::GL_REPEAT
glTexParameteri GL_TEXTURE_2D GL_TEXTURE_MAG_FILTER $::GL_NEAREST
```

If called with the enumeration name:

```
glTexParameteri GL_TEXTURE_2D GL_TEXTURE_WRAP_S GL_REPEAT
you will get an error message like this: expected integer but got "GL_REPEAT"
```

To correctly wrap the OpenGL libraries, a version of SWIG greater or equal to 1.3.19 is needed.

For performance reasons use OpenGL display list, where possible.

5 Open issues

- GLU callbacks are currently not supported. This implies, that tessellation does not work, because this functionality relies heavily on the usage of C callback functions.
- There is currently no possibility to specify a color map for OpenGL indexed mode. As color maps depend on the underlying windowing system, this feature should be handled by the Togl widget.
- Picking with depth values does not work correctly, as depth is returned as an unsigned int, mapping the internal floating-point depth values [0.0 .. 1.0] to the range [0 .. $2^{32} - 1$]. As Tcl only supports signed integers, some depth values are incorrectly transferred into the Tcl commands.
- The handling of Tcl errors inside of Togl callbacks could be improved.
- To evaluate the Tcl callbacks, Tcl_Eval is currently used, which does not compile the script into bytecode. Use the object-interface instead.

6 Results

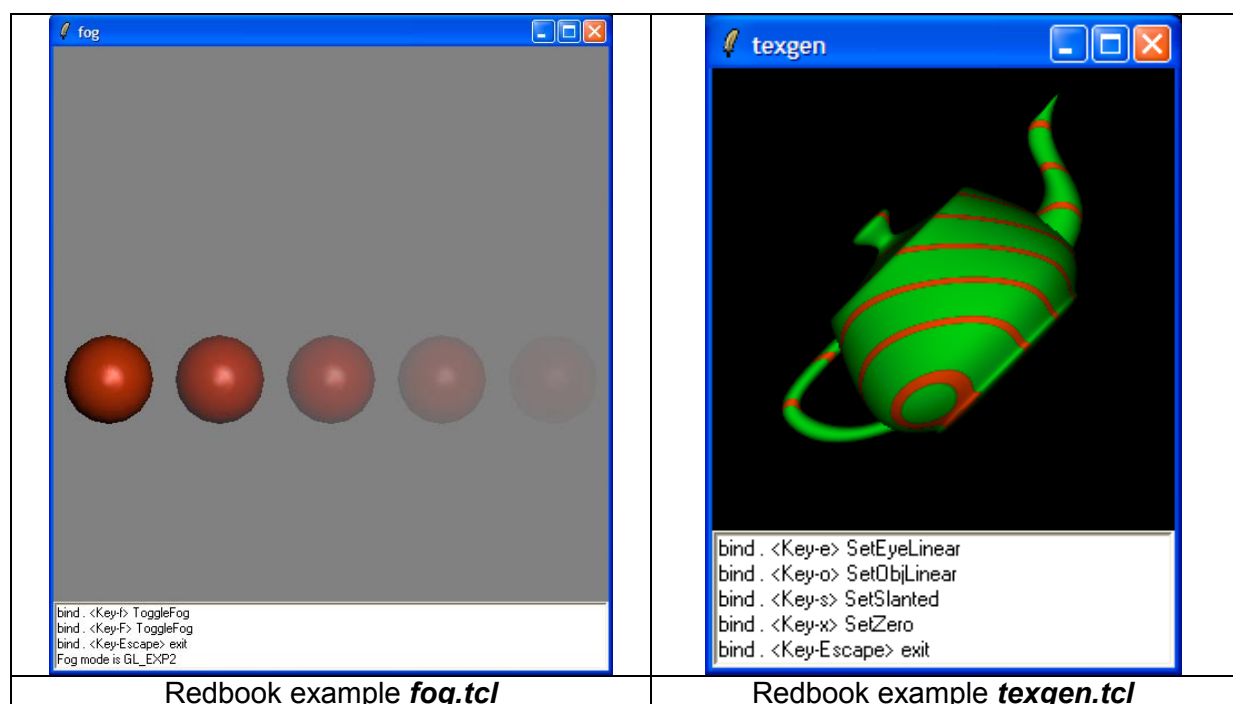
To test the correctness and completeness of the wrapped OpenGL library, the examples of the Redbook ([1]), which are available as C code ([2]), were ported into equivalent Tcl code.

The Redbook contains 56 examples, showing many aspects of OpenGL features.

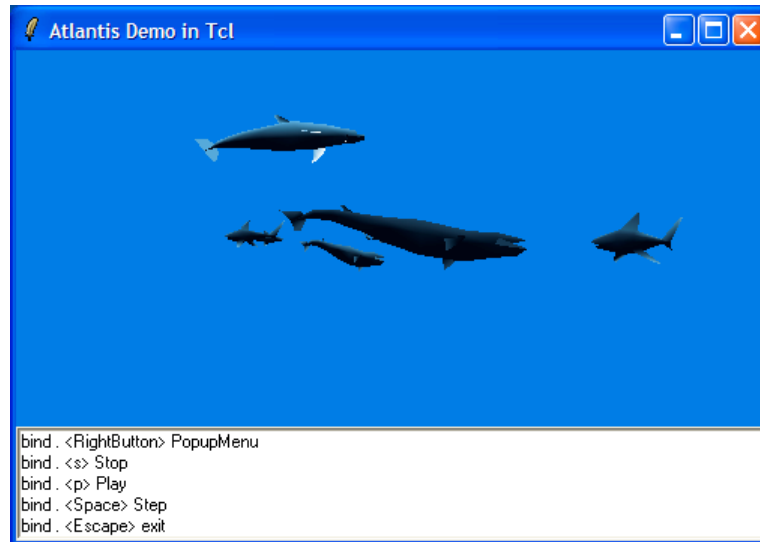
52 of them have been successfully converted into equivalent Tcl scripts and compared on different computers against the C version. All of them gave identical results, except depth-picking in some cases (see above).

Two of the missing four examples deal with tessellation, which is currently not supported, as stated in the previous chapter. The other two test programs not yet ported deal with color index mode, which is not yet implemented, too.

Tessellation and color index mode both are rarely used features, at least in my applications.



To demonstrate the easy transition of C to Tcl code, a more complex program, the "Atlantis demo" ([3]) has been ported. It behaves like it's C pendant, but performs a lot slower, as it has been not been optimized for running as a Tcl script.



Finally a simple image viewer has been implemented that allows realtime scaling of the image. The images can be read from files in all formats supported by the Img extension. The stretched image may also be written out to an image file.

The Togl and tclogl packages have been generated and tested on the following platforms:

Operating system	Compiler version	SWIG version
Windows XP	Visual C++ 6.0	1.3.19
SuSE Linux 9.0	gcc 3.3.1	1.3.19
IRIX 6.5	MIPSpro cc 7.30	1.3.24

The source code for the tclogl package, i.e. the modified Togl code and the SWIG interface files for the OpenGL wrapper, as well as the test and demo programs can be downloaded from my home page ([7]). A binary version of the actual SWIG version 1.3.24 for IRIX is available there, too.

7 References

[1] Woo, Neider, Davis: OpenGL Programming Guide, Addison-Wesley, "**The Redbook**"

[2] Redbook C examples: <http://www.opengl.org/resources/code/basics/redbook>

[3] Atlantis demo: http://www.opengl.org/resources/code/glut/glut_examples/demos/demos.html

[4] Nate Robins OpenGL tutorial: <http://www.xmission.com/~nate/tutors.html>

[5] OpenGL Wiki page: <http://wiki.tcl.tk/2237>

[6] SWIG (Simplified Wrapper and Interface Generator): <http://www.swig.org>

[7] Paul Obermeier's Portable Software: <http://www.poSoft.de>

Tcl/Tk Potpourri

Experiences by using Tcl/Tk in data and document management

Jörg Schmitz¹
arvato logistics services
D15E2
Fr.-Menzefricke-Str. 16-18
33775 Versmold
Germany

Abstract

In our department Tcl is used as it's natural character: a Tool command language.

To show the wide range of usage I'll give four examples: importing data from an XML document to a relational database schema, developing a parser for context free languages, e.g. compile EBNF into syntax diagrams, similarity search using the n-Gramm method to match different translation memories and a web front-end based on an idea of server pages but using CGI.

XML RDB Mapping

To import data from an XML document into a database you have to parse the XML file. Therefore you can take a DOM or a SAX implementation of an XML parser. If you don't know how big your XML document can get it will be impossible to use DOM because not enough memory to store the DOM tree of the complete document. Using SAX parser as an alternative will take you to another problem: no random access to the XML data (e.g. XPath expressions), only sequentially fired events for tags and character data. Therefore you can rebuild the interlaced data structure of the XML document by your own data structure or you can directly send attribute values and character data to a target system but you have to write procedures to handle SAX events and data.

Because of RDB as target system I decided to store minimal data structures and to write data into database when possible. Possible means a complete or incomplete set of data to store in a database table. Incomplete data sets needs to be filled up with type specific data.

```
<customer>
  <person>
    <name>Schmitz</name>
    <email>schmitz@bla.org</email>
  </person>
  <company>
    <name>arvato</name>
  </company>
</customer>
```

XML cut out

```
Schmitz
schmitz@bla.org

arvato
```

Data

Table: customer	
cid	integer primary key auto increment
person_name	varchar(42) not null
person_email	varchar(42)
company_name	varchar(42) not null

Table: customer

```
insert into customer values(
null,
'Schmitz',
'jschmitz@gmx.net',
'DUMMY');
```

SQL: incomplete set (DUMMY)

¹ E-Mail address: joerg.schmitz@bertelsmann.de | jschmitz@gmx.net

```
1
Schmitz
jschmitz@gmx.net
DUMMY
```

Data set after INSERT

```
update customer
set company_name = 'arvato'
where cid = 1
```

SQL: update set (arvato)

```
1
Schmitz
jschmitz@gmx.net
arvato
```

Data set after UPDATE

This approach forces a sequence of insert and update SQL statements to store data into database but it's simple and fast.

Because it's simple you can think in a standardized way of writing event handler and data handler to import XML data into a RDB schema. First step is to create an XML parser and to configure it:

```
01 set parser [eval ::xml::parser -validate 0]
02 $parser configure -elementstartcommand element_begin
03 $parser configure -characterdatacommand cdata
04 $parser configure -elementendcommand element_end
05 $parser parse $data
```

Then you can implement the event handler:

```
01 proc element_begin {name attlist} {
02     global PARSER
03     lappend PARSER(context) $name
04     foreach {item value} $attlist {
05         if {[catch {attribute "[join $PARSER(context) "."].$item" $value}
err]} {
06             error "attribute\ncontext: $PARSER(context)\nerror: $err"
07         }
08     }
09     if {[catch {begin_${name} [join $PARSER(context) "."]} err]} {
10         error "begin_${name}\ncontext: $PARSER(context)\nerror: $err"
11     }
12 }
```

```
01 proc cdata {data} {
02     global PARSER
03     set name [lindex $PARSER(context) end]
04     if {[catch {cdata_${name} "[join $PARSER(context) "."].data" $data}
err]} {
05         error "cdata_${name}\ncontext: $PARSER(context)\nerror: $err"
06     }
07 }
```

```
01 proc element_end {name} {
02     global PARSER
03     set PARSER(context) [lrange $PARSER(context) 0 end-1]
04     if {[catch {end_${name} [join $PARSER(context) "."]} err]} {
05         error "end_${name}\ncontext: $PARSER(context)\nerror: $err"
06     }
07 }
```

The event handler calls a data handler by XML element name (**_\${name}**):

- **element_begin** calls **begin_\${name}**
- **cdata** calls **cdata_\${name}**
- **element_end** calls **end_\${name}**

This data handler must not be written by hand. They can be generated during runtime. Therefore a very small mapping language was developed to describe what to do with data out of an XML document.

This mapping language is defined as:

INIT

Initialises a data structure to store data from the XML document. **INIT** creates a procedure

which is called by the event handler for a start tag to set the attribute values. **INIT** must be used in the **START** command of the XML document root element.

ERASE

Delete the data structure created by **INIT**. **ERASE** must be used in the **STOP** command of the XML document root element.

START name script

START must be defined for each element of an XML document. It's the container for the data handler of an XML start element. **name** is the element's name. **script** is a set of commands that should be executed by calling the data handler.

STOP name script

STOP must be defined for each element of an XML document. It's the container for the data handler of an XML end element. **name** is the element's name. **script** is a set of commands that should be executed by calling the data handler.

CONTEXT context script

If an XML element is used in different contexts you can define a set of commands (**script**) for each context. **context** is a complete path to the element in the interlaced structure. It's described as all element names to this context starting with the root element and joined by a dot, e.g.: `PODAccounting.Article.AcoountingArea`. **CONTEXT** is used in the **START** or **STOP** command of the XML element.

SET name value

Use **SET** to assign XML values to the target data structure. If target is a database use the complete path to a database field: `(<schema>.)?<table>.<column>` as a value for **name**. **value** is a complete path to the element in the interlaced structure. It's build up by all element names to this context starting with the root element and joined by a dot, e.g.: `PODAccounting.Article.Activity`. `.data` must be added to this path to assign the character data. `.<attribute_name>` must be added to assign data of an defined attribute.

VAR name value

With **VAR** you can define own variables to store temporary values. **name** is the name of the variable. **value** is the assigned value to this variable. It's typically used for result sets of a **DO** command.

OPTIONAL name script

Elements can be defined as 'optional' in the document type definition (DTD) of an XML document. If you have to run commands for a non existing (optional) element you can use **OPTIONAL** in the most recent **START** or **STOP** command to the optional element.

GET name

To get a value out of the target data structure use the command **GET**. If target is a database use the complete path to a database field: `(<schema>.)?<table>.<column>` as a value for **name**, e.g. `ArticleMaster.MasterNumber`. Usage of **GET** results in other commands:

```
SET Activity.ArticleID [GET Article.ArticleID].
```

UNSET name

Use UNSET command to remove values from the target data structure. If name is not a complete path to character data or attributes all structures below last element in name are deleted. UNSET is typically used in an elements STOP command.

DO ?tuple? script

DO typically executes SQL queries given by script. If tuple is not set and script is a select statement, DO returns a list of lists with name value pairs, e.g.

```
{ {ARTICLENUMBER 10.2034.64.00 ARTICLEARCHIVE VWoA} {...} ...}.
```

If tuple contains a name of a variable in the result set, only the first value assigned to this variable name (first row in result set) is returned:

```
[DO ArticleNumber {...}] → 10.2034.64.00
```

ID sequence

ID will return the current value of a given sequence of the used database schema.

FORALL items resultset script

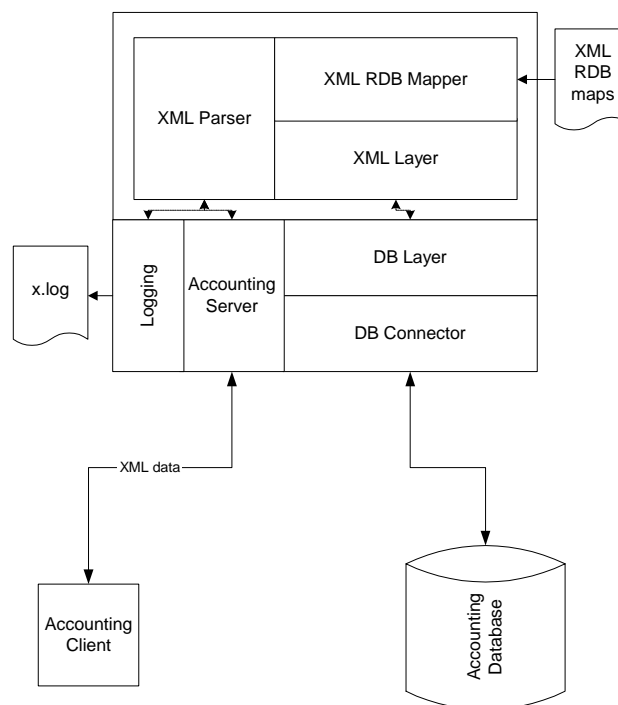
FORALL is an iterator to execute commands by using values of DO result sets. The DO resultset must be assigned to a VAR variable. items defines a list of variables which should be used during the iteration. Values can be used by GET command. name of GET is written as <resultset>.<item>, e.g.:

```
VAR customer [DO {select name, email from customer}]
```

```
FORALL {name email} customer { VAR x "[GET customer.name], [GET  
customer.email]" }
```

Developing an accounting application using XML RDB mapping

An accounting client sends XML data streams with settlement information to an accounting server. The accounting server parses the XML data and stores settlement information in a database system. Client, server and database are running on different computers in the local area network. Communication is done by TCP/IP and a small application specific protocol.



DTD, XML and XML RDB mapping examples

PODAccounting.xml

```
<!ELEMENT PODAccounting (Article+)>
<!ELEMENT Article (AccountingArea, SalesOrganization, CopyFactor, Description?, Activity?)>
<!ATTLIST Article
    number CDATA #IMPLIED
    archive CDATA #IMPLIED
>
<!ELEMENT AccountingArea (#PCDATA)>
<!ELEMENT SalesOrganization (#PCDATA)>
<!ELEMENT CopyFactor (#PCDATA)>
<!ELEMENT Description (#PCDATA)>
<!ELEMENT Activity (Item+)>
<!ELEMENT Item EMPTY>
<!ATTLIST Item
    type CDATA #REQUIRED
    quantity CDATA #REQUIRED
>
```

PODAccounting.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE PODAccounting SYSTEM "../dtd/PODAccounting.dtd">
<PODAccounting>
    <Article number="001.1245.44.20" archive="SAP">
        <AccountingArea>VW</AccountingArea>
        <SalesOrganization>VWoA</SalesOrganization>
        <CopyFactor>2</CopyFactor>
        <Description>Reparaturanleitung Golf</Description>
        <Activity>
            <Item type="BWPrintSingle" quantity="120"/>
            <Item type="CPrintSingle" quantity="2"/>
        </Activity>
    </Article>
    <Article number="030.2040.13.00" archive="VW">
        <AccountingArea>VW</AccountingArea>
        <SalesOrganization>Audi</SalesOrganization>
        <CopyFactor>10</CopyFactor>
    </Article>
</PODAccounting>
```

PODAccounting2SQL.tcl

```

START PODAccounting {INIT}; STOP PODAccounting {ERASE}
START Article {
    SET Article.ArticleNumber PODAccounting.Article.number
    SET Article.ArticleArchive PODAccounting.Article.archive
}
STOP Article {
    OPTIONAL PODAccounting.Article.Activity {
        SET Activity.ArticleID [ID ArticleSeq]
        VAR MasterProduct [DO {
            SELECT t1.productid, t1.masterquantity
            FROM masterproduct t1, articlemaster t2
            WHERE t1.masterid = t2.masterid
            AND t2.masternumber = [GET Article.ArticleNumber]
            AND t2.masterarchive = [GET Article.ArticleArchive]
        }]
        FORALL {ProductID MasterQuantity} MasterProduct {
            DO {INSERT INTO Activity (ArticleID, ProductID, ActivityQuantity)
                VALUES ([GET Activity.ArticleID],[GET MasterProduct.ProductID],
                    [GET MasterProduct.MasterQuantity])
            }
        }
    }
    OPTIONAL PODAccounting.Article.Description.data {SET Article.ArticleDesc "null"}
    DO {UPDATE article SET articledesc = [GET Article.ArticleDesc]
        WHERE articleid = [GET Activity.ArticleID]
    }
    UNSET PODAccounting.Article
}
START AccountingArea {}
STOP AccountingArea {
    SET Article.ArticleDomain PODAccounting.Article.AccountingArea.data
}
START SalesOrganization {}
STOP SalesOrganization {
    SET Article.ArticleSalesOrg PODAccounting.Article.SalesOrganization.data
}
START CopyFactor {}
STOP CopyFactor {
    SET Article.ArticleCopyFactor PODAccounting.Article.CopyFactor.data
    OPTIONAL PODAccounting.Article.number {SET Article.ArticleNumber "null"}
    OPTIONAL PODAccounting.Article.archive {SET Article.ArticleArchive "null"}
    DO { INSERT INTO Article (
        ArticleDomain, ArticleSalesOrg, ArticleNumber,
        ArticleArchive, ArticleCopyFactor, ArticleDesc)
        VALUES ([GET Article.ArticleDomain], [GET Article.ArticleSalesOrg],
            [GET Article.ArticleNumber], [GET Article.ArticleArchive],
            [GET Article.ArticleCopyFactor], null)
    }
}
START Description {}
STOP Description {SET Article.ArticleDesc PODAccounting.Article.Description.data}
START Activity {}; STOP Activity {}
START Item {
    SET Product.ProductName PODAccounting.Article.Activity.Item.type
    SET Activity.ActivityQuantity PODAccounting.Article.Activity.Item.quantity
    SET Activity.ArticleID [ID ArticleSeq]
    SET Product.ProductID [DO ProductID {
        SELECT ProductID FROM Product WHERE ProductName = [GET Product.ProductName]
    }]
}
STOP Item {
    DO {INSERT INTO Activity (ArticleID, ProductID, ActivityQuantity)
        VALUES ([GET Activity.ArticleID], [GET Product.ProductID],
            [GET Activity.ActivityQuantity])
    }
    UNSET PODAccounting.Article.Activity
}

```

Parser for context free languages

Sometimes regular expressions can't solve your problem to analyse an given expression because it's not a regular one but a context free expression. Recursively defined expressions are typically represented by context free languages, e.g.:

$A = "a" A "c" \mid "b"$.

Following an approach of Wirth² it is really simple to write a parser for context free expressions.

There are two steps to find out (parse) if an expression is one of the defined language:

1. Lexical analysis
2. Syntactical analysis

Lexical analysis is done by a scanner. The scanner splits the given expression into symbols. The parser uses an iterator to get a symbol from the scanner.

To create your own context free language, e.g. a business rule language to describe business relevant parameters in an application, you can use a well known notation, the Extended Backus Naur Form (EBNF).

Definition of EBNF (in EBNF):

```
syntax      = {statement}.
statement   = identifier "="
              expression ".".
expression  = term {"|" term}.
term        = factor {factor}.
factor      = identifier | string |
              "(" expression ")" |
              "[" expression "]" |
              "{" expression "}".
identifier  = letter {letter | digit}.
string      = {character}.
```

You have to decide between nonterminal symbols (NTS) and terminal symbols (TS). A NTS is a symbol that can be substituted by another symbol. A TS can not be substituted. A TS is written in quotation marks, e.g.: `"="`.

The basic constructs of EBNF are:

Construct	Semantics	Meta Code
<code>"x"</code>	Terminal symbol "x"	IF sym="x" THEN next ELSE error END
<code>(exp)</code>	Expression	Pr(exp)
<code>[exp]</code>	Optional expression	IF sym IN first(exp) THEN Pr(exp) END
<code>{exp}</code>	Repeated expression	WHILE sym IN first(exp) DO PR(exp) END
<code>fac₀fac₁...fac_n</code>	Series of factors	Pr(fac ₀);Pr(fac ₁);... PR(fac _n)
<code>term₀ term₁ ... term_m</code>	Alternation of terms	CASE sym OF first(term ₀): Pr(term ₀) first(term ₁): Pr(term ₁) first(term _m): Pr(term _m) END

The meta code can now be used to test if an given expression is one of the defined language. Remembering the example above a procedure to check the syntax of A is written as:

² Wirth, Niklaus: Grundlagen und Techniken des Compilerbaus, Addison-Wesley, 1996

Meta Code

```

PROCEDURE A;
BEGIN
  IF sym = "a" THEN
    next; A;
  IF sym = "c" THEN
    next
  ELSE
    Error
  END
  ELSIF sym = "b" THEN
    next
  ELSE
    error
  END
END A

```

Tcl Code

```

proc A {} {
  global sym
  if {$sym == "a"} {
    getSym; A
  } else if {$sym == "c"} {
    getSym
  } else {
    error "sym $sym must be 'c'"
  }
  } elseif {$sym == "b"} {
    getSym
  } else {
    error "sym $sym must be 'b'"
  }
}

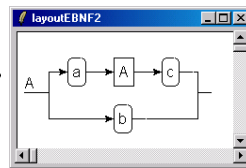
```

With procedure A you can check that aaabccc is syntactically correct.

Creating syntax diagrams out of EBNF descriptions

In this more complex example a scanner and parser for EBNF is combined with a stack to store some output created during scanning and parsing an expression. This generated output is Tcl code that is interpreted in a separated step as drawing instructions for a syntax diagram. This combination of a scanner, parser and code generator is called a compiler. It does not create machine executable code to run a program but drawing instructions to create a syntax diagram.

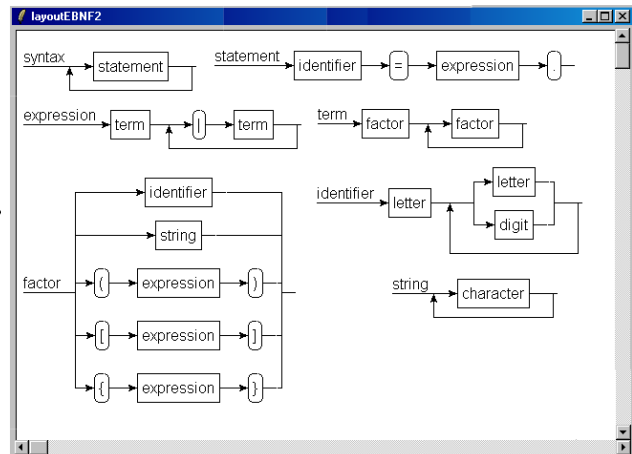
A = ("a" A "c") | "b".



```

syntax      = {statement}.
statement   = identifier "="
              expression ".".
expression  = term {"|" term}.
term        = factor {factor}.
factor      = identifier | string |
              "(" expression ")" |
              "[" expression "]" |
              "{" expression "}".
identifier  = letter {letter |
digit}.
string      = {character}.

```



Similarity search

To describe the similarity of two strings you can use well known similarity functions:

- Levenshtein distance (LD)
- Longest Common Subsequences (LCS)
- N-Gram
- ...

The N-Gram method with digram sets ($N = 2$) is exemplary used for similarity functions.

Similarity values were computed using the following string similarity scheme:

$$\text{SIM}(N_1, N_2) = |N_1 \cap N_2| / |N_1 \cup N_2|,$$

where N_1 and N_2 are digram sets of two strings. $|N_1 \cap N_2|$ denotes the number of intersecting (similar) digrams, and $|N_1 \cup N_2|$ the number of unique digrams in the union of N_1 and N_2 .

For example, the degree of similarity for the strings *rwanda* and *ruanda* is calculated as follows:

$$\text{SIM}(\{\text{rw,wa,an,nd,da}\}, \{\text{ru,ua,an,nd,da}\}) = |\{\text{an,nd,da}\}| / |\{\text{rw,wa,an,nd,da,ru,ua}\}| = 3/7 \text{ (0.428)}.$$

For the string compared to itself the similarity value is 1.0.

Example of use: Merge of translation memories

Our department for translation asked to create a new translation memory (TM) out of two given translation memories. This two TMs are created by a computer aided translation tool during translation of SGML documents from German (DE_DE) to English (EN_GB) and during translation of XML documents from English (EN_US) to simplified Chinese (ZH_TW). The content of both TMs describes repair manuals of different car types and brands of an international multi brand car manufacturer. The DE|EN file is around 15400 entries, the EN|ZH file is around 700 entries. It's as test case to find out how many entries can be identified by the English phrases in both TMs.

Because it's time wasting to do 700×15400 comparisons (10500000: if one test is about 1 millisecond it's around 180 minutes) between this two TMs the following normalisation and grouping steps are done:

- Normalisation
 - Lower characters and deleting all white space in the English terms
 - Deleting SGML, XML and RTF sequences as part of the English terms
 - Deleting some special characters: , . _ - + * # ' " ! \$ % & () { } [] | ? in the English terms
- Grouping
 - A group of strings is build up by the same quotient of a string length (integer) division by 4, that means one group of strings contains strings with $4n \dots 4n+3$ characters

I got 50 groups with an average of 15 terms for the EN|ZH TM and 300 terms for the DE|EN TM. This is around 225000 comparisons ($50 \times 15 \times 300$) between that two translation memories. This process needs around 6 minutes to read the two files, normalise and group the English terms, make the comparison and generate an XML output file. It theoretically needs 4 minutes (assumption: 1 millisecond per comparison).

The merged TM (DE|ZH) can be loaded by an special editor to check and correct terms that are not identical. The similarity is given as an distance between 0.0 and 0.1. This editor can store the translation memory as a TMX (translation memory exchange format) file to use this TM with different CAT tools.

TME editor			
File			
	Source Language: DE-DE	Distance	Target Language: ZH-TW
83	Zylinderkopfschraube	OK	汽缸頭螺絲
84	Zylinderkopfhaube	OK	汽缸頭蓋
85	Zylinderkopfdichtung	OK	汽缸頭墊圈
86	{\cs6V1\cf6\lang1024<unbruch><ext-rl><baugrup>}Elektrische Anlage	OK	電系
87	Motorhalter rechts	<input type="checkbox"/>	右引擎托架
88	für Einlass-Nockenwelle	OK	適用於進氣凸輪軸。
89	für Spannrolle	<input type="checkbox"/>	拉緊鏈性
90	weitere Information {\cs6V1\cf6\lang1024<ext-slp>}	OK	{\cs6V1\cf6\lang1024<?Pub Dtl?>}進一步資訊：
91	Lambda-Regelung	OK	含氧感知調節
92	am Zylinderkopfdeckel	<input type="checkbox"/>	汽缸頭蓋
93	Stoßfänger vorn ausbauen.	OK	拆卸前保險桿。
94	Keilrippenriemen abnehmen.	OK	拆卸聚合 V 形皮帶。
95	Keilrippenriemen ausbauen	OK	拆卸聚合 V 形皮帶
96	Ersetzen Sie bei Beschädigung die Dichtung.	OK	若有受損的話請換新油封。
97	Sekundär-Luftsystem	<input type="checkbox"/>	二次進氣系統
98	Sekundärluft-System	OK	二次進氣系統
99	Systembezeichnung	OK	系統配置
100	Spannrolle	OK	拉緊鏈性
101	Schraubverbindung	OK	螺紋連接
102	zum Zylinderkopfdeckel	<input type="checkbox"/>	汽缸頭蓋
103	Nockenwellenverstellung	OK	可變汽門正時
104	Kontrollieren, ob Paßhülsen zur Zentrierung Motor/Getriebe im Zylinderblock vorhanden sind.	<input checked="" type="checkbox"/>	檢查將引擎與變速箱體中的釘套位於缸體內。必要時安裝之。
105	20 Nm + {\cs6V1\cf6\lang1024<hoch>}1{\cs6V1\cf6\lang1024</hoch>}/{\cs6V1\cf6\lang	<input type="checkbox"/>	30 Nm + 步驟 {\cs6V37\cf6\lang1024<hoch>}1{\cs6V37\cf6\lang1024</hoch>}/{\cs6V37\cf6\lang
106	Klimaanlage:	OK	冷氣系統
107	Dichtung für Zylinderkopfdeckel	OK	汽缸頭蓋墊圈
108	Zylinderkopf, Ventiltrieb	OK	汽缸頭, 汽門機構
Distance: 0.059			
Source Key: tocyylinderheadcover			
Target Key: cylinderheadcover			
nGramm: 2 MaxDistance: 0.1 Encoding: utf-8			

Tcl CGI Pages

It's a hard job to generate HTML output in a CGI application if you use Tcl's `puts` statement:

```
puts stdout "<html>"
puts stdout "<head><title>hello world example</title></head>"
puts stdout "<body>[regive \"Hello, World!\"]<body>"
puts stdout "</html>"
```

It will be nice to write a HTML page (`hw.tcp`) with embedded Tcl code and run an interpreter to get the output:

```
<html>
<head><title>hello world example</title></head>
<body>[regive "Hello, World!"]<body>
</html>
```

And this is trivial with Tcl:

```
set rid [open hw.html r]; set html [read $rid]; close $rid
puts stdout [subst $html]
```

With the following result (`regive` returns the given string):

```
<html>
<head><title>hello world example</title></head>
<body>Hello, World!<body>
</html>
```


It's a little bit more complex to use if-then-else control structures in a Tcl CGI Page (TCP) to control which parts of the HTML code should be in the output but benefit is much higher than learning this syntax.

```
<table>
  [if {[is_data "session,id"]} {
    subst {
      <tr>
        <td>
          
        </td>
      </tr>
    }
  } else {
    subst {
      <tr>
        <td>
          <a href="application.tcl?show=main_site">
            [::msgcat::mc "Home Page"]
          </a>
        </td>
      </tr>
    }
    [tsp_menu "administration" "Administration"]
  }
}]
</table>
```

Taking this examples as a starting point I will present my "last web application".

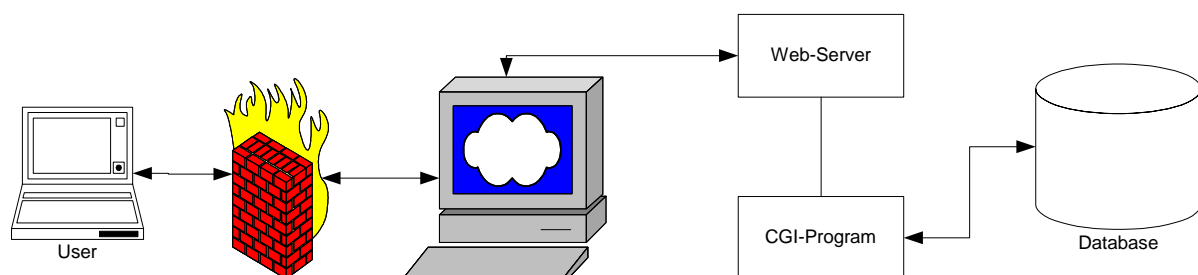
Translator Pool for Pangea: TP²

TP² is a database about translators and translation agencies. This application is used by our department of translation to find the right translator for a new translation project. After logon to the web based application you can use several functions depending on your user role and rights:

- Administrator
 - Adding, changing and deleting base information in the database (language / country combinations based on ISO lists of languages and countries, topics to specify a translation project, title and job title information etc)
 - Managing roles and rights to use the application
 - Adding, changing and deleting application user
- Project Manager
 - Adding, changing and deleting translators and translation agencies including their service offering (language pairs, prices, etc)
 - Search translation resources
 - Adding projects and rate projects
- Member
 - Search translation resources

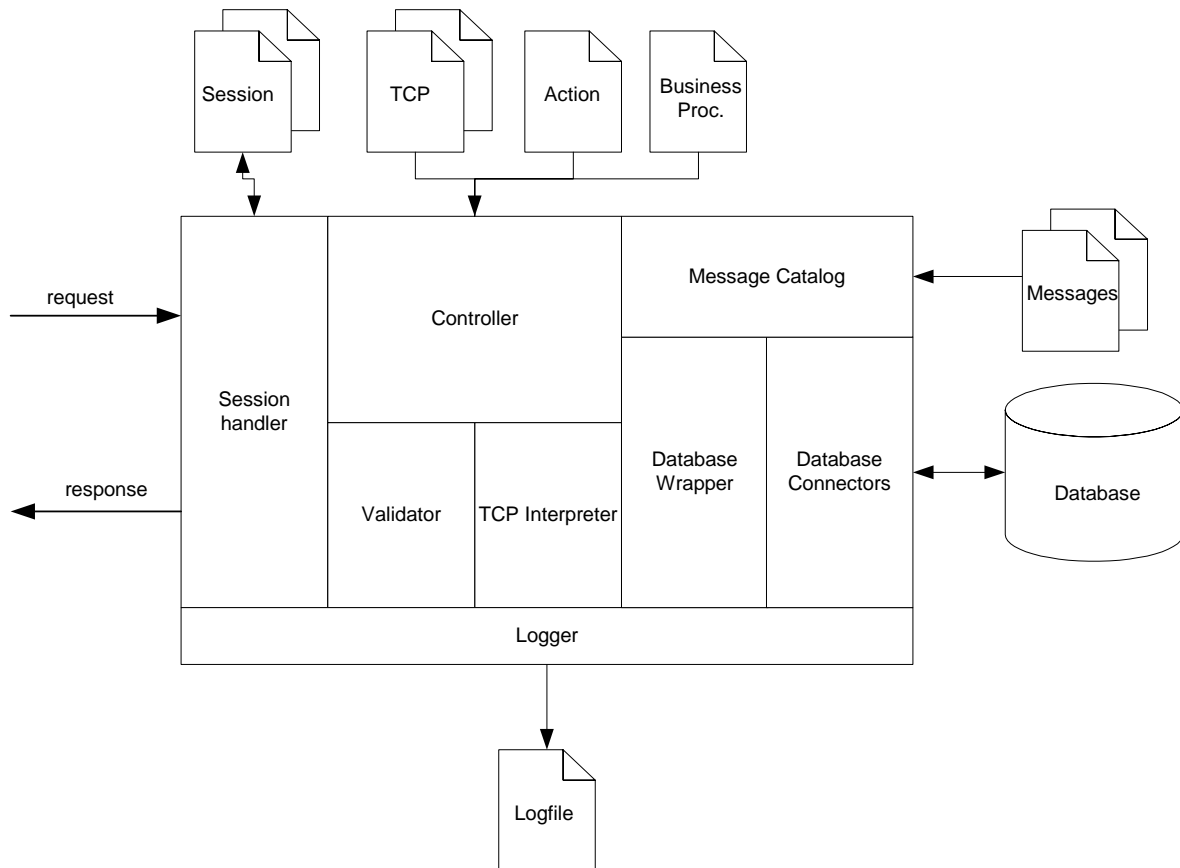
System structure

The system structure is the same as other CGI applications:



System architecture

The system architecture is more complex than a standard CGI program:



A request contains the name of the main application `application.tcl` and as a hidden value the name of the requested action, e.g.: `show=add_translator`. It also contains all form field variables and values from the requesting website. While a session is running the assigned cookie contains the session id. The session handler checks if there is a session file with the same id and sources this file. The controller sources the called action `action_add_translator.tcl` and if exists the corresponding business procedures `business_add_translator.tcl`. The action runs the validator and checks the form values. If there is an error, action sources an error Tcl CGI Page, if ok, action runs the business procedures and then sources the response TCP. The business procedures may query the model (database) and select, insert, update or delete data. The response site is evaluated by the TCP interpreter and send to the requesting client. Embedded Tcl code in a TCP can directly query the model. This has to be changed in the future. The session handler writes the session relevant data to the session file.

Code example: `action_reference_project.tcl`

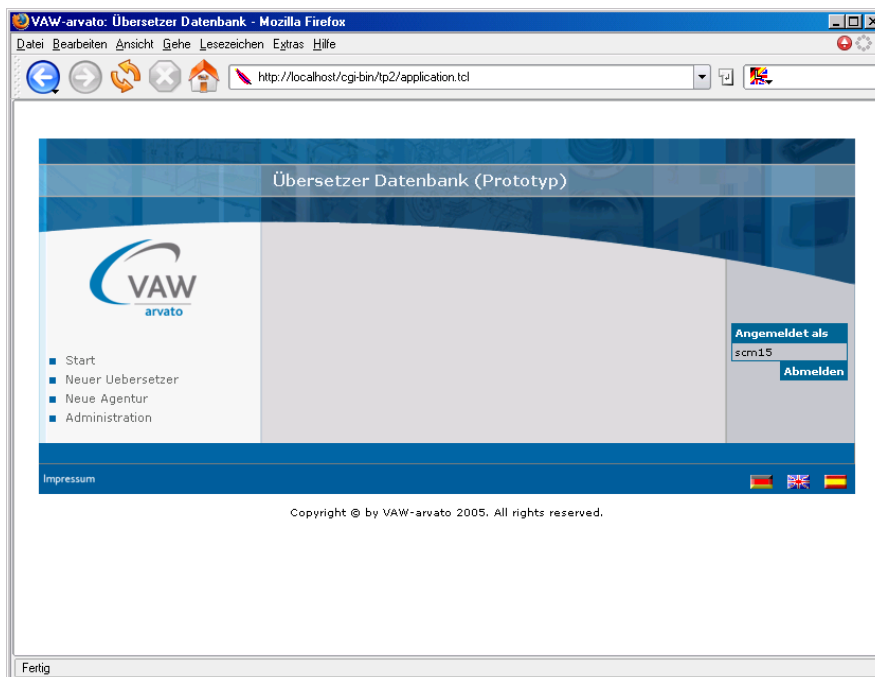
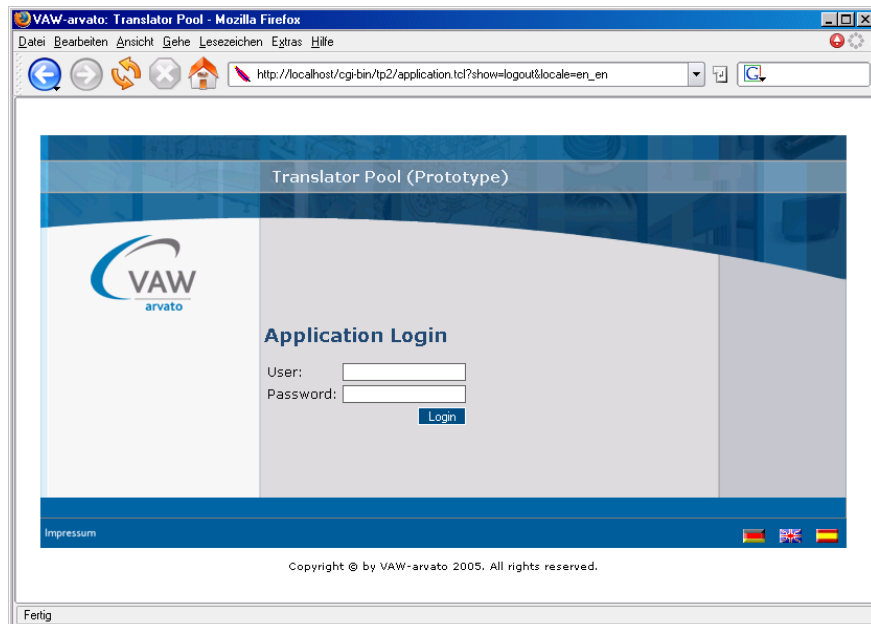
```

proc action_reference_project {} {
    puts stdout "Content-type: text/html\n\n"
    if {[running_session]} {
        puts stdout [include "index"]
    } else {
        set usecase [get_session "running_case"]
        set form_data [list [list "$usecase,languagecountry_source" {^([0-9]+_[0-9]+)?$}]\
            [list "$usecase,languagecountry_target" {^([0-9]+_[0-9]+)?$}]\
            [list "$usecase,capacity" {^.*$}]\
            [list "$usecase,discount" {^.*$}]\
            [list "$usecase,match_0_full" {^[0-9]?$}]\
            [list "$usecase,match_0_part" {^([0-9][0-9])?$}]\
            [list "$usecase,match_100_full" {^[0-9]?$}]\

```

```
[list "$usecase,match_100_part" {^([0-9][0-9])?$}\]\
[list "$usecase,match_fuzzy_full" {^[0-9]?$}\]\
[list "$usecase,match_fuzzy_part" {^([0-9][0-9])?$}\]]]
if {![valid_form "lappend" $form_data]} {
    puts stdout [include "reference_project"]
} else {
    if {[is_this performance_next]} {
        puts stdout [include "reference_performance"]
    } elseif {[is_this performance_submit]} {
        puts stdout [include "reference_project"]
    } else {
        puts stdout [include "index"]
    }
}
}
return
}
```

Screen shoots



VAW-arvato: Übersetzer Datenbank - Mozilla Firefox

Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe

http://localhost/cgi-bin/tp2/application.tcl?show=reference_company

Übersetzer Datenbank (Prototyp)



- Start
- Neuer Uebersetzer
- Neue Agentur
- Administration

Neue Agentur

Name:

Website:

Zuordnung VAW:

Referenz ID:

CAT Tools:

Trados, 6.5
Trados Teamworks, 1.x
across, 2.x
across, 3.x
Star Transit, ?

Fachgebiete:

law
literatur
media
religion
science

Kontakte:

Jörg Schmitz, Versmold
Manuela, Fersen, Bunde

Vertraulichkeitsvereinbarung:

Unterzeichnet am:

Vertrag:

Anfang:

Ende:

Letzter Kontakt:

Zusammenarbeit:

Währung:

Kontoname Inland:

Bank Inland:

Bankleitzahl Inland:

Kontonummer Inland:

Kontoname Ausland:

IBAN:

BIC SWIFT:

Angemeldet als

Fertig

Copyright © by VAW-arvato 2005. All rights reserved.

zogl Tcl/OpenGL integration

Alexios Zavras
zvr+tcl@zvr.gr

Motivation

3D

2D Tk canvas limitations

number of objects

tags

support

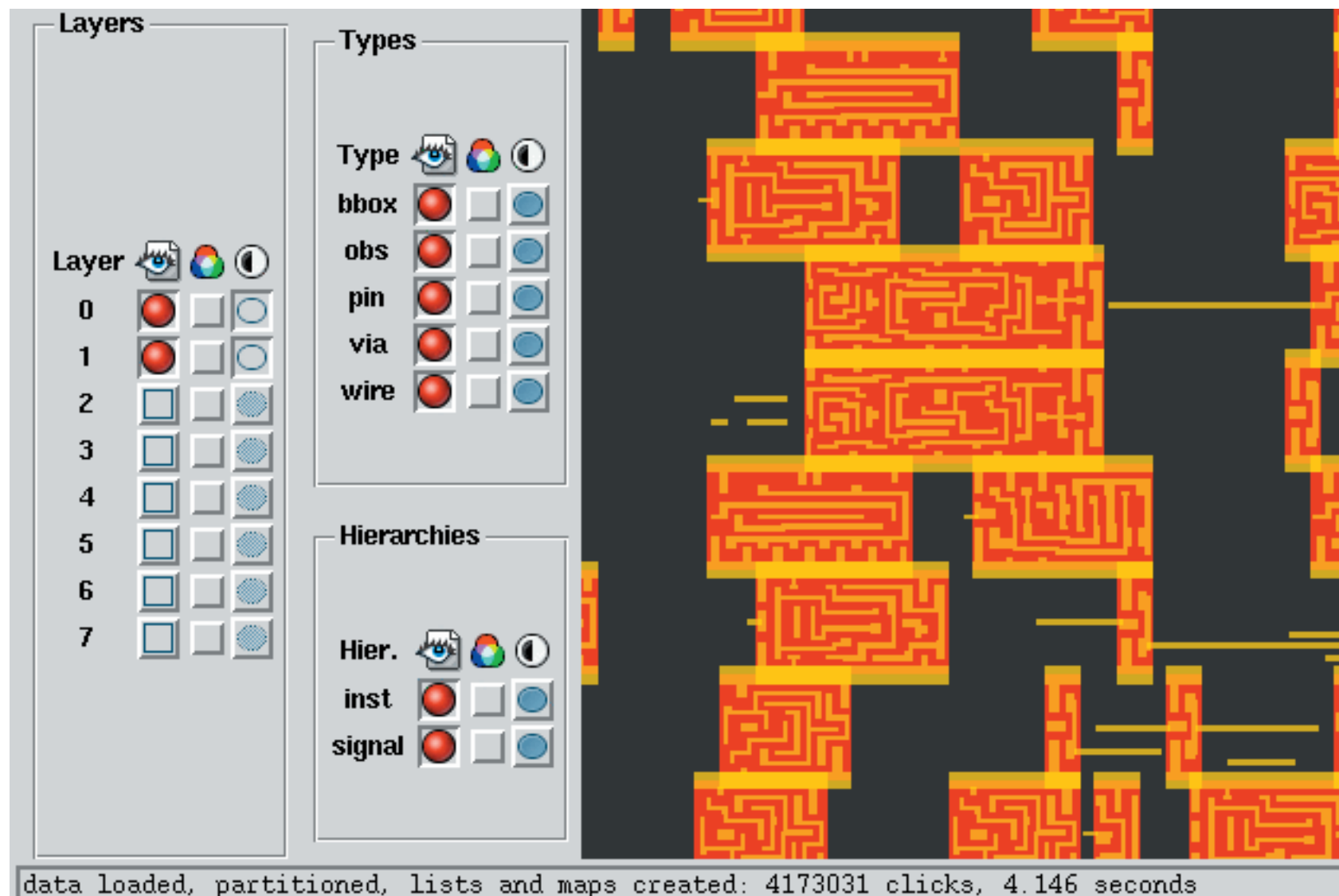
Usage Case Study

Athena Design Systems

CAD tools

view chip on screen

100K – 10M rectangles



Status

currently fully functional

Single command (`zogl`) for widget

`::GL` namespace

> 240 GL calls implemented

hand-written (not SWIGged)

only for X Window System

Code Example

```
zogl .gl -width 200 -height 200

glShadeModel $::GL::FLAT
glClearColor 0 0 0 0
glBegin $::GL::TRIANGLES
    glVertex3f 0.1 0.9 0.0
    glVertex3f 0.1 0.1 0.0
    glVertex3f 0.7 0.5 0.0
glEnd
```

Special Features

Tk image support

img2texture

tight X integration

glX calls

X window IDs

context sharing

Tasks for Release

licensing decision

documentation

manual page

tutorial

autoconfig tools

TEA compliance

Future Work

textures

pbuffers

extensions

extra functions

Word of Caution

OpenGL not one, but many
implementation details
nVidia \neq Mesa

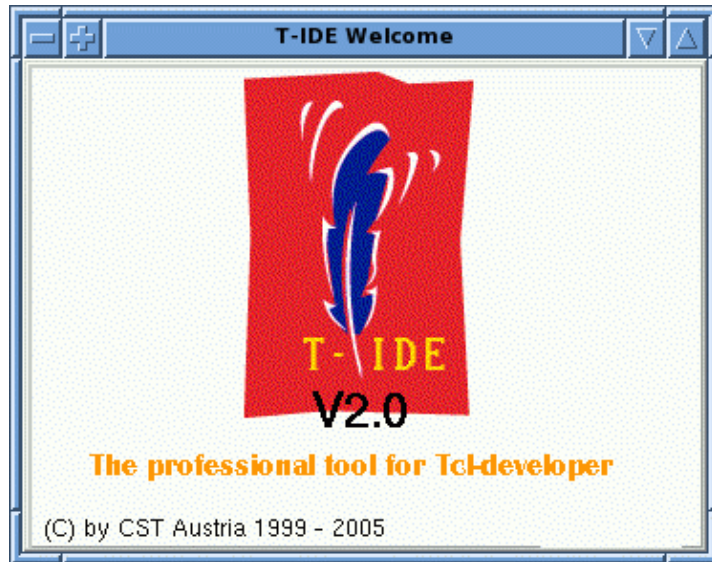
Demo time!

nk you!

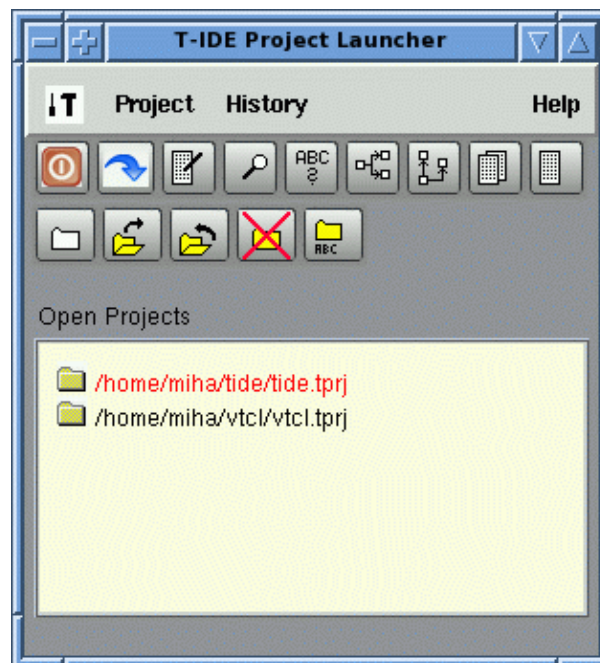
TIDE_Tour

Tour through T-IDE V2.0

T-IDE V2.0 is brand-new, includes themes for better look-and-feel on every platform (incl. WIN-XX). It includes several enhancements and speedups. After you have used it for a project you will know about the benefits you get through T-IDE and you will never use anything else (and it is not limited to Tcl only!).



After the "welcome-picture" has disappeared the T-IDE-starter pops up.



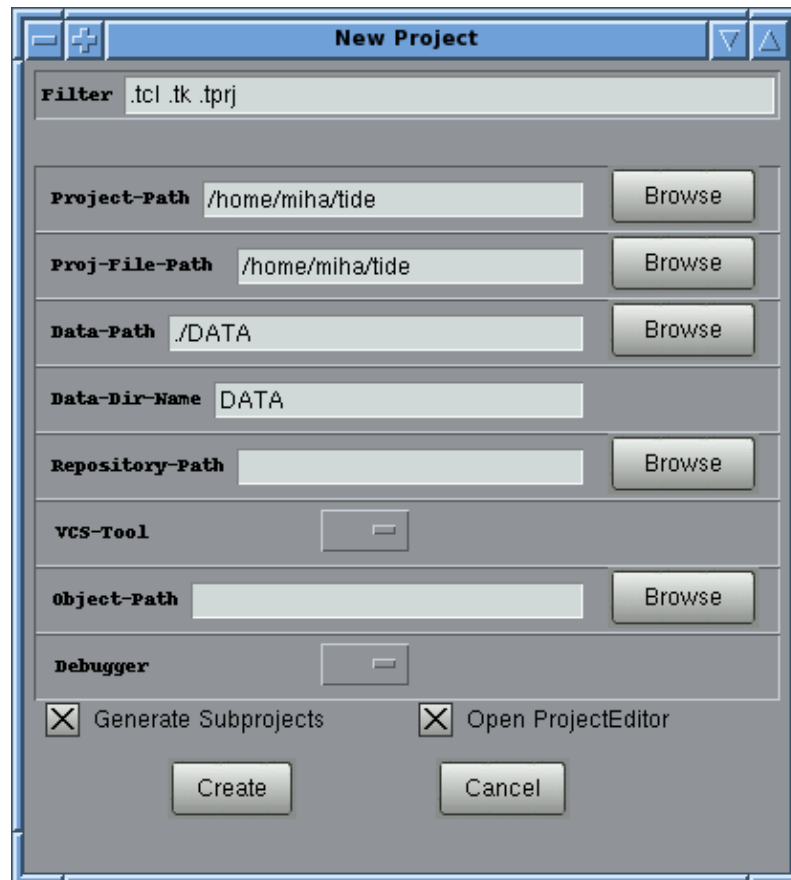
This is the place where you can create new projects, open existing ones (the last 10 can be easily accessed via the History-menu), close open ones, rename and delete projects. In the Tool-menu you will find an entry called "Workspaces". If you are programming in a team, this is the place where you define your workspaces (every programmer gets his own private workspace to work in and put then the result into the shared workspace to make it accessible to all others). But in our tour we will omit this step.

TIDE_Tour

First we have to create a project

By pressing the "New Project"-Button we get now a window called "**New Project**" to set all necessary values.

We will use a path with already existing files with the shown settings:



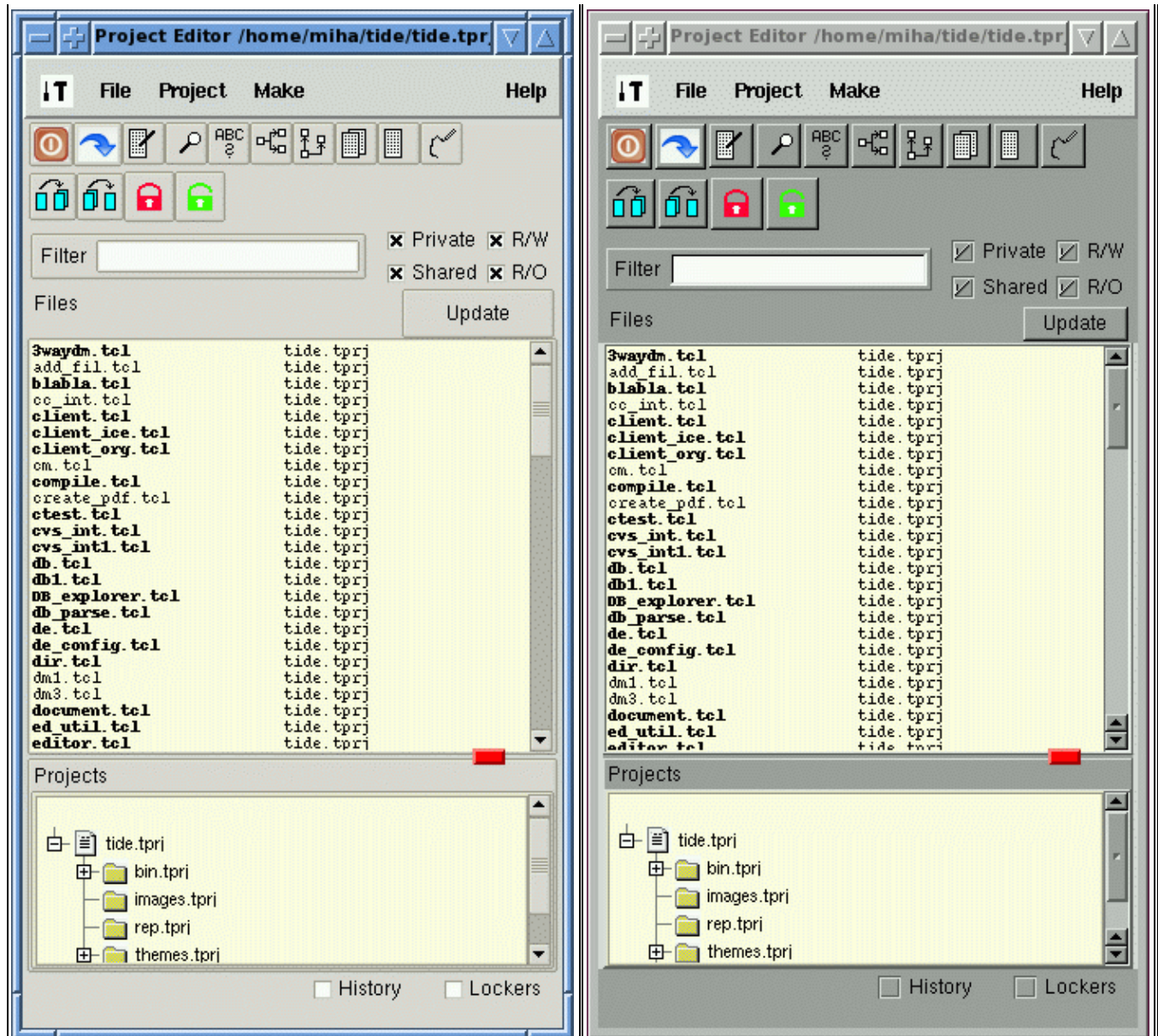
Lets press "Create".

The parser parses through all source-files and stores the result in a database.

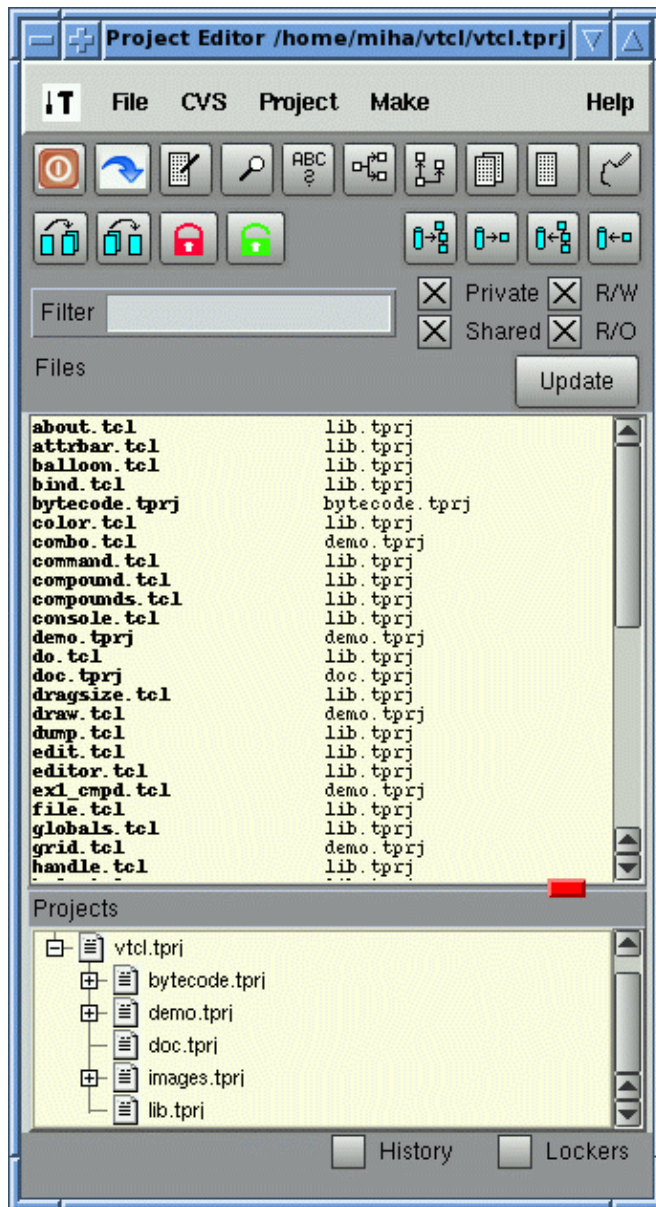
Then the **Project-editor** pops up.

Here showing some different styles:

Theme "Clam"	Theme "Next"	The



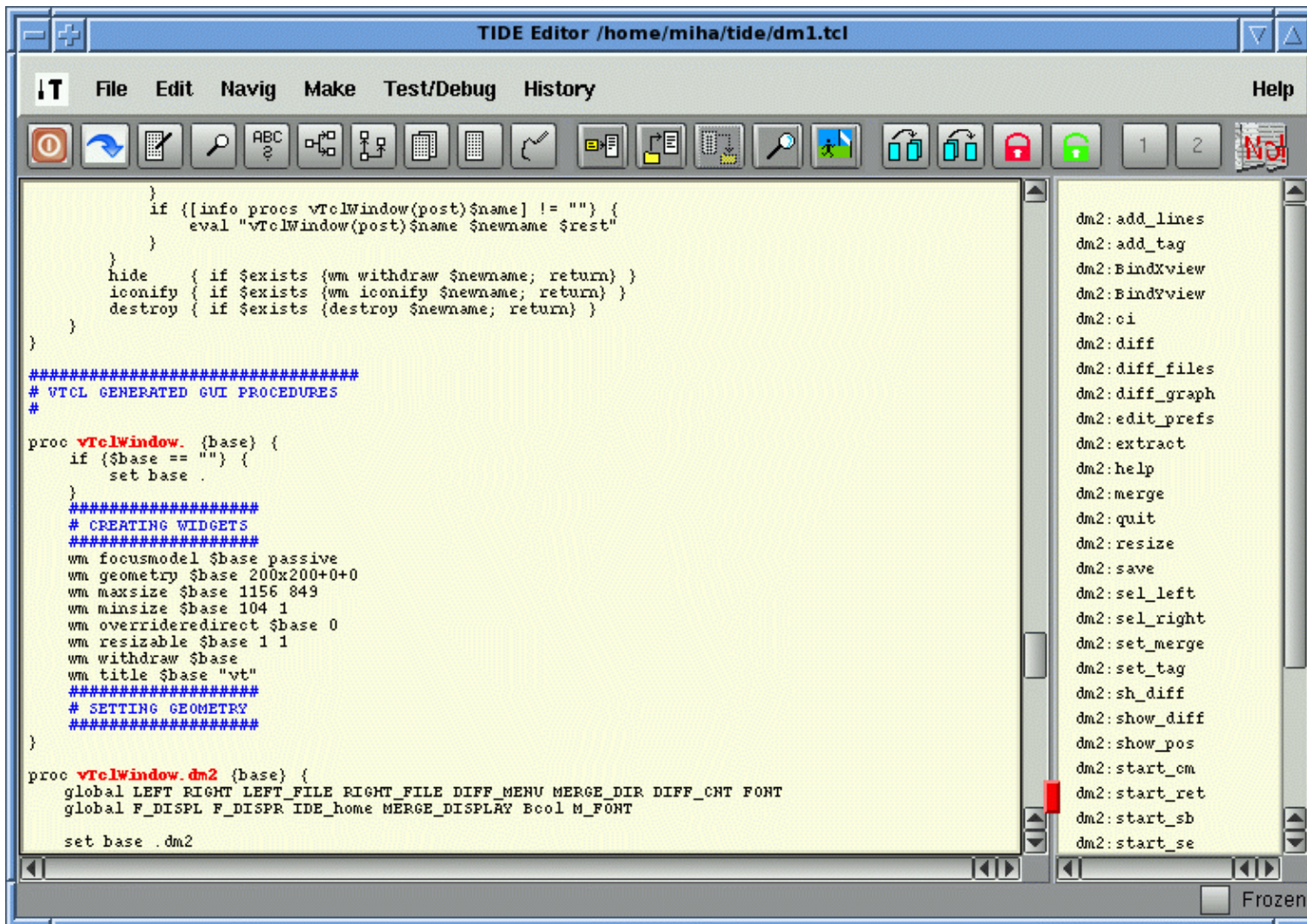
In case CVS is your Revision Control System the Project-Editor would look like this (with additional buttons to control CVS):



◁ The Project-editor lets you modify the structure of the project. You can not only add or remove files to every project, you can also add or remove subprojects (or complete trees), where the physical location of the single project is unimportant.

You can view the files of a specific project by clicking on the appropriate folder. The folder-icon itself is replaced by a sheet-icon. Files in bold are readable and writeable; files in normal script are read-only; this in italic are symbolic links. Symbolic links are used if the current project is opened in a private workspace (team-development). Enabling the switch "Lockers" shows the locker and the locked revision. Enabling the switch "History" shows the revision-tree for the selected file (only if a revision-tool has been specified).

Now we want to edit a file. A doubleclick on a source-file opens the editor :



On the right hand side of the editor you see a list of all procs defined in this file. Clicking on it relocates the editor to the location of the proc. Proc-names can be shown in colorized in a different font, comments also (can be modified in the Preferences).

The editor offers you all possibilities of navigation :

- retrieving a string from the current file, current project or all projects (see Retriever)
- looking up for a specific symbol (proc-name, global or variable) (see Symbol-Browser)
- call-tree-hierarchy (which procs are calling the actual proc, or where is global XX set or read,..) (see Referencer)
- widget-hierarchy-browser (see Hierarchy-Browser)

From the editor you can debug your current file (if its a part of an other one you can specify it). TUBA has been integrated as standard-debugger (thanks to John Stump). You can easily jump between editing and debugging (only for Tcl8.0.x!).

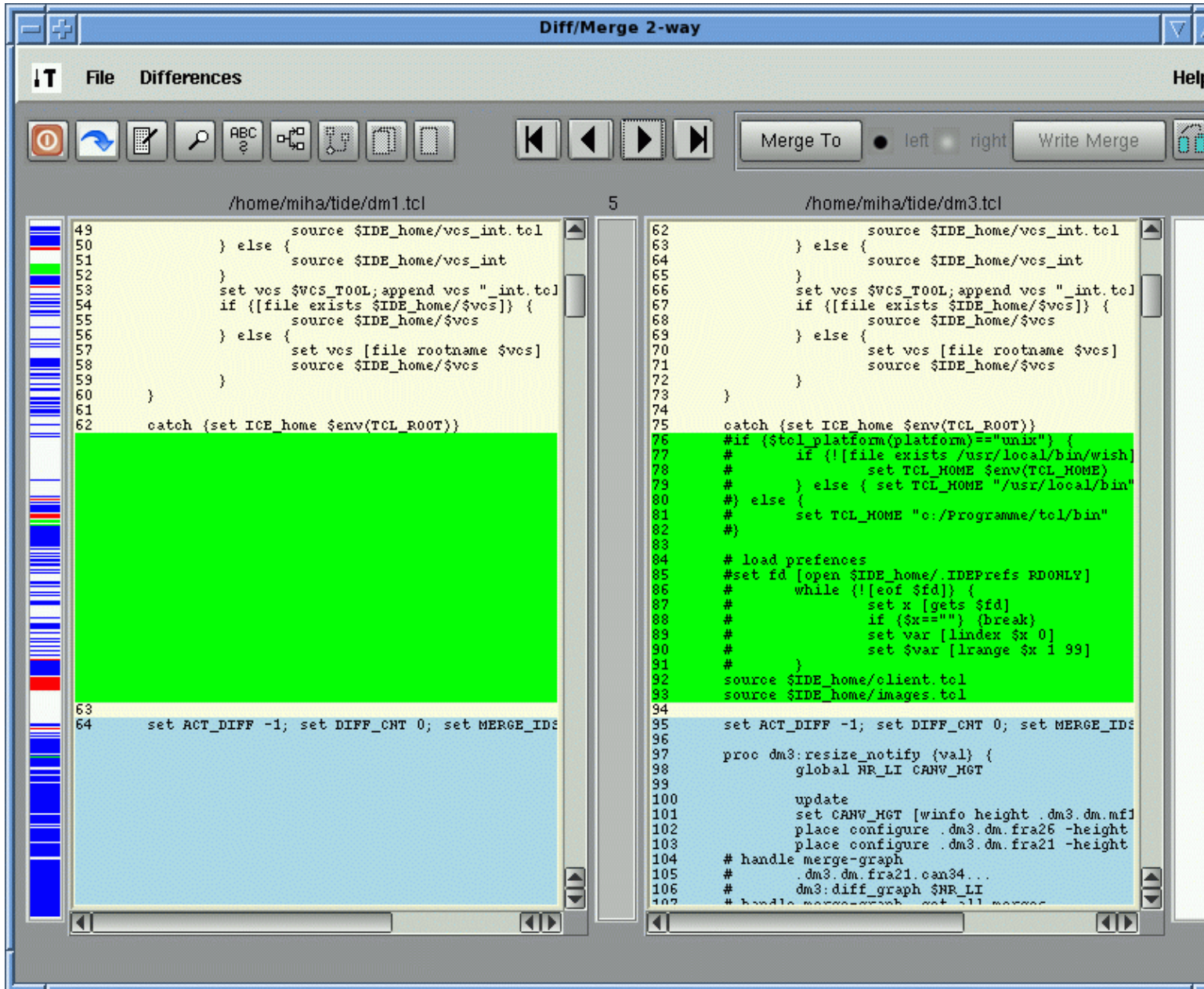
To test your application you have just to press the "RUN"-button....

If your application has been tested well you can compile it to byte-code by selecting "Make – make target".

TIDE_Tour

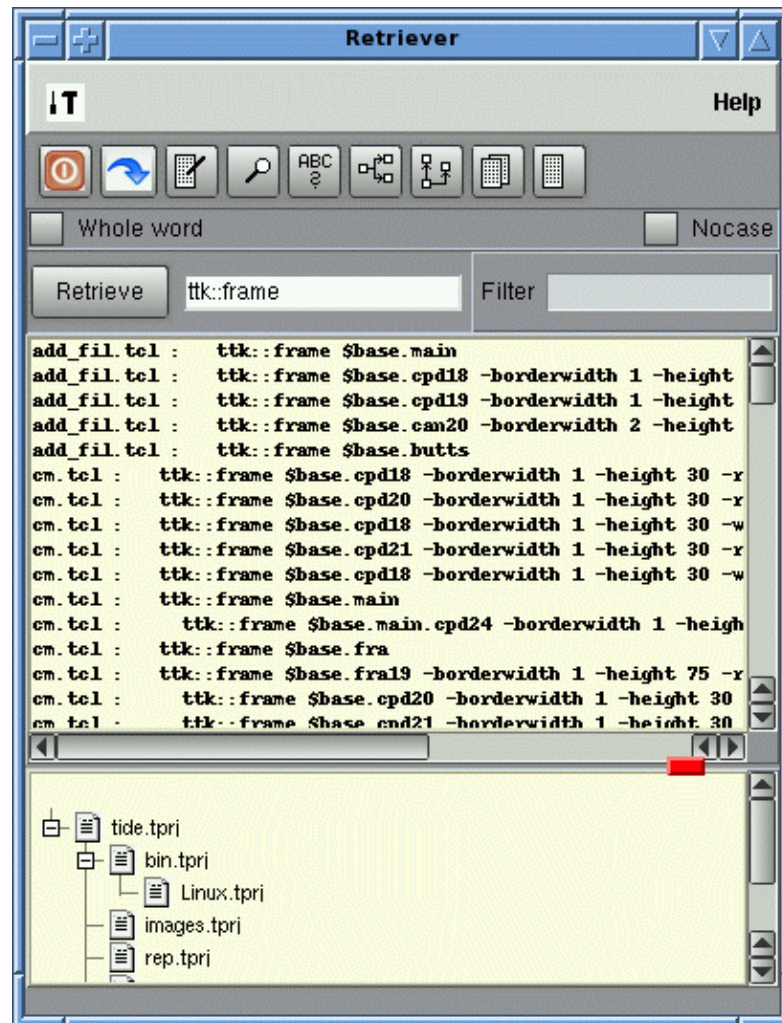
Other features of the editor : easy access to a VCS-tool (currently RCS and CVS, other will follow soon), to a GUI-builder, a user-menu to add own tools, mostly used commands as keyboard-shortcuts, use more than one editor at a time.

Lets continue with the **Diff/Merge-tool**



The Diff/Merge-tool is available as 2- and 3-way diff (e.g. to merge 2 branches into the main-branch).

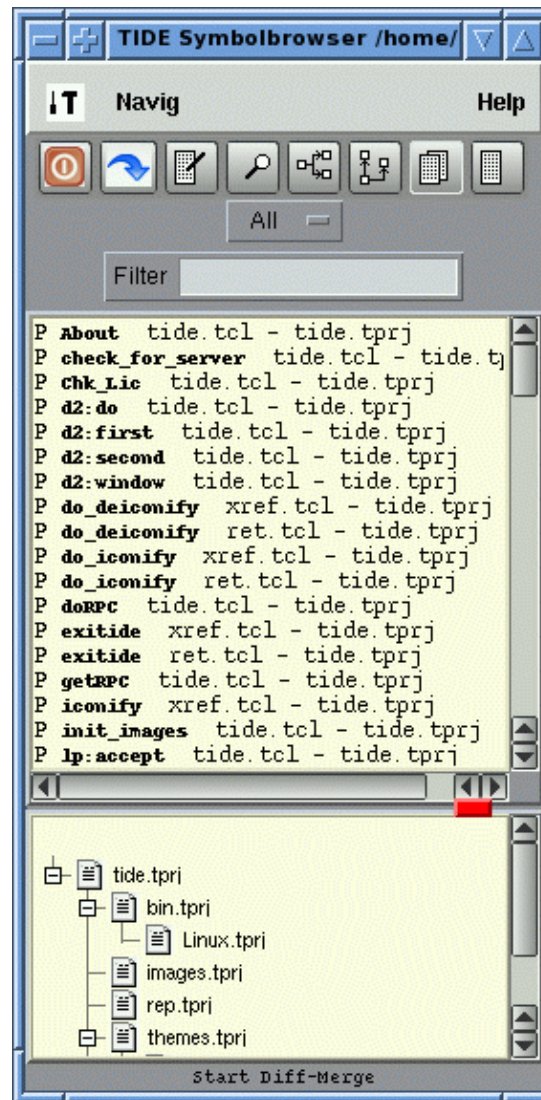
The Retriever :



A double-click on a listed item opens the source-code-editor just at the position of the retrieved item.

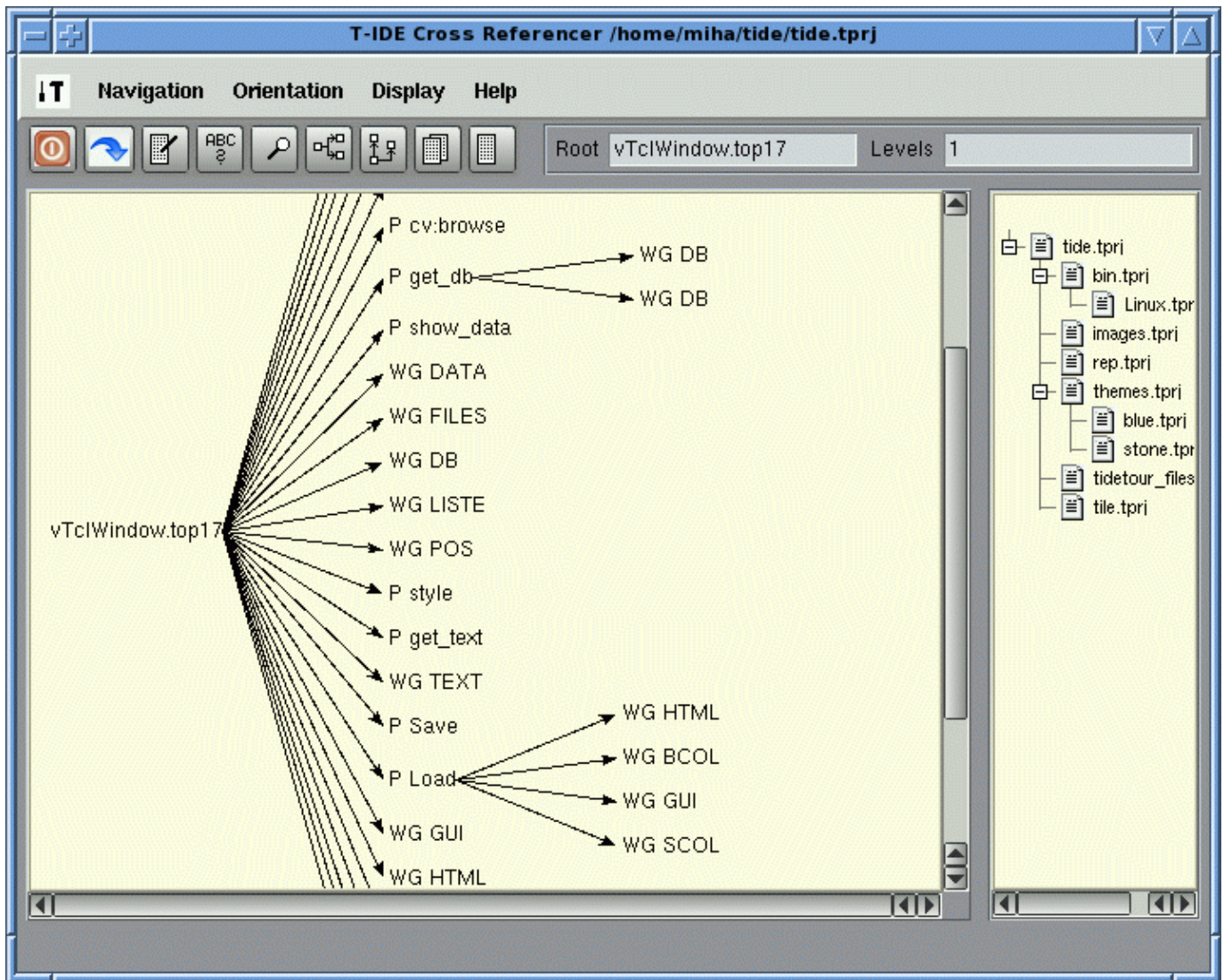
The Symbol-Browser :

TIDE_Tour



Shows procs, variables and globals used in the selected projects. The contents of the list may be filtered. This is a good way to find double-use of a symbol. Like above in the Retriever the location of the selected symbol can be edited by double-clicking on the symbol.

The Referencer :



The above screenshot shows a nested "ref-to" tree. To limit the displayed information Tcl/Tk-commands and globals/variables can be switched off.

You cannot only view the proc call-tree, but also find the procedures where global variables are referenced for write or read-access.

e.g. "global TREE referred by" gives you all procs reading or writing the global. Or : "proc Tree:build referred by" gives you all procs calling Tree:build.

By double-clicking on a node the source-code-editor opens at the position where the symbol is used. This is a very convenient way to understand the process-flow of an application.

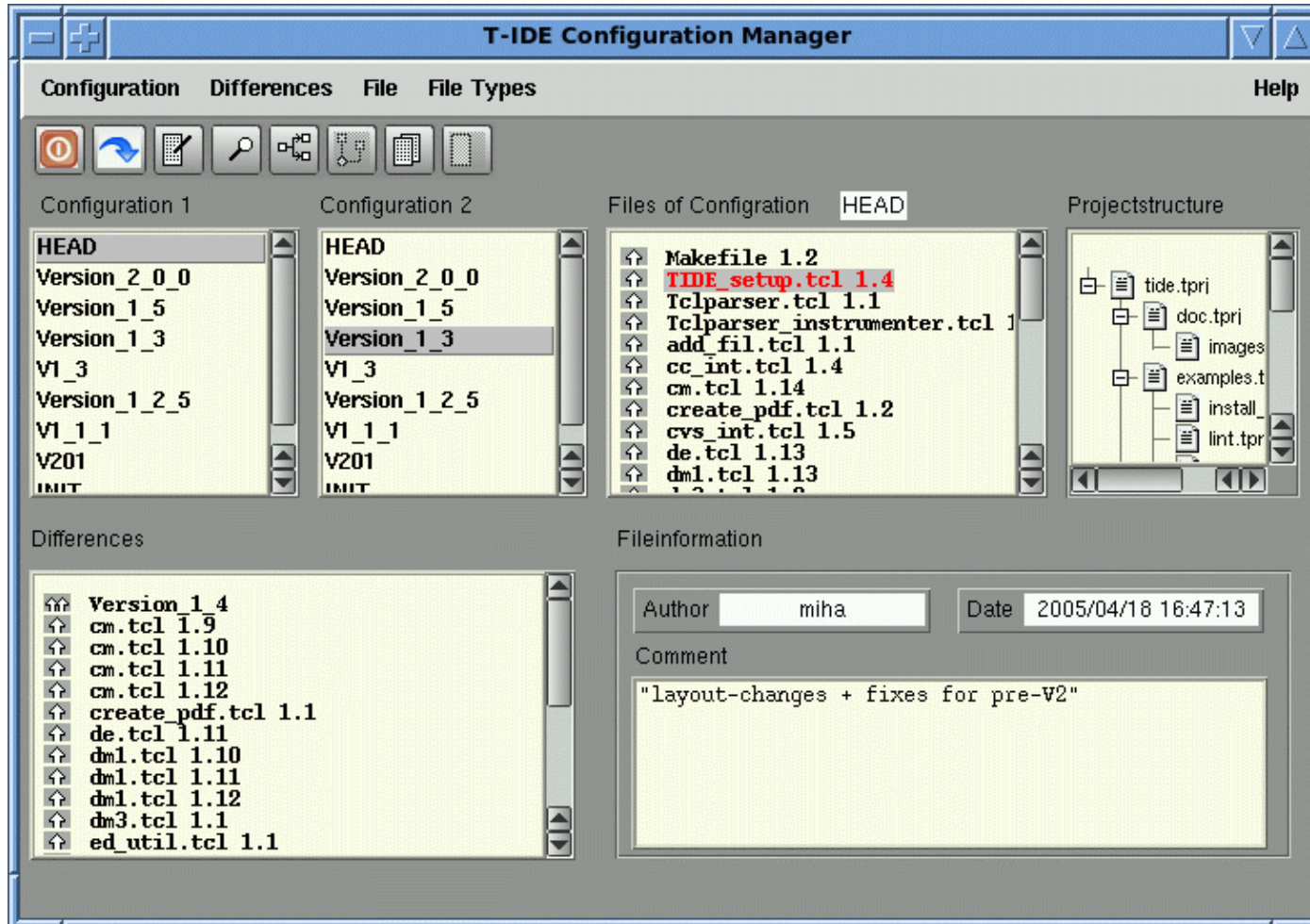
Even this view can be limited by selecting or deselecting projects.

The Hierarchy-Browser :

Shows the used widgets depending on the selected projects. Will be available with our GUI-Builder.

The Configuration-Manager :

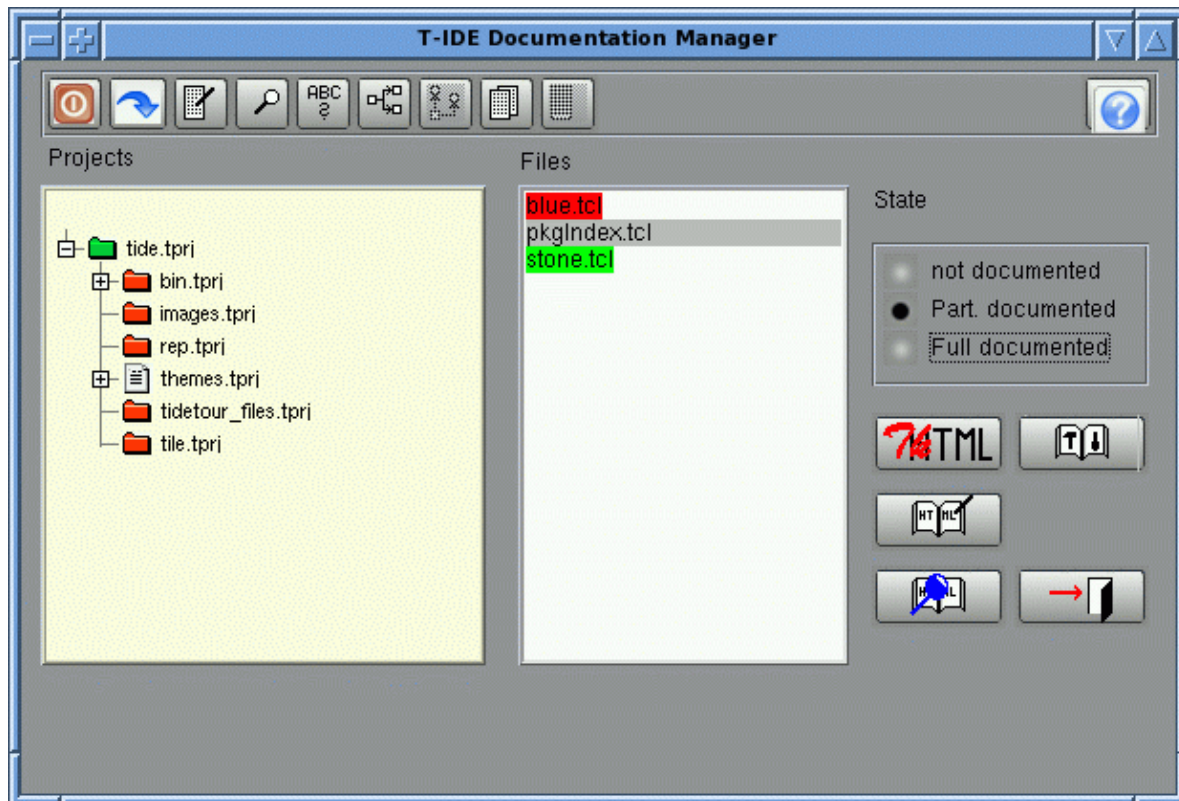
Helps to manage configurations within your preferred VCS-Tool.



Note: this tool is not available in TIDE light!

The Documentation-Editor :

TIDE_Tour



Produces HTML-documents of your source-files. The documentation can be done for the whole project or file-by-file. Colors indicate the documentation-state of the Project/File (red = not documented, yellow = partially and green = fully documented). A master-index can be generated by request. Each generated .html-file can be modified by the T-IDE HTML-editor.

This tool is available in the full version of T-IDE.

The Shell :

The purpose of this tool is to catch the results of a lint- or compiler-run. By selecting an error-message you can directly navigate back to the source-code-location to fix the problem.

Supported platforms (T-IDE 2.0) : Linux, Solaris, WIN-XX.

Other platforms will be available on request.

Note: This is just a preview to the new release, its not available yet!

NOTE: to bring this great tool to market, I am either looking for an investor, who helps doing the final things or an interested company willing to

purchase the source (including me).

If you have specific questions please contact [Michael Haschek, CST](#)

Logging by Example

A short introduction to the tcllib
logger package.

Michael Schlenker <mic42@users.sourceforge.net>

Why use logger?

- ✓ Part of tcllib
- ✓ Hierarchical logging
- ✓ Fully introspectable
- ✓ Cool debug features

Prelaunch Checklist

- ✓ Install Tcllib

<http://tcllib.sourceforge.net>

- ✓ Start your favorite Tcl console e.g. tkcon
- ✓ Load the logger package
package require logger

A basic logger

```
package require logger                ;# load the package
set log [logger::init global]         ;# Initialize a logger service
${log}::warn "A warning message"      ;# send a message to the log
${log}::setlevel error                 ;# ignore all messages below
${log}::warn "Second warning"         ;# this is ignored
```

Default format

[Thu May 26 03:08:20 +0200 2005] [global] [warn] 'A warning message'



A logger tree

```
package require logger                ;# load the package
set log [logger::init global]         ;# Initialize a logger service
set child [logger::init global::child] ;# Initialize a child service
${log}::warn "A warning message"      ;# send a message to the log
${log}::setlevel error                 ;# ignore all messages below
${log}::warn "Second warning"
${child}::warn "Second warning"       ;# this is ignored
```

Simple logprocs

```
proc myerror {txt} {puts "Error logged:\n$txt"}      ;# define a logproc
${log}::logproc error myerror                        ;# and assign it
puts [${log}::logproc error]                        ;# introspect it
proc mycritical {txt} {                             ;# a fancy logproc
  puts "Critical problem:\n$txt"                    ;# introspecting its caller
  puts "Caller: [uplevel 1 info level 0]"
}
${log}::logproc critical mycritical                  ;# assign it
proc someproc {} {${::log}::critical "A serious problem"}
someproc
```

Critical problem:

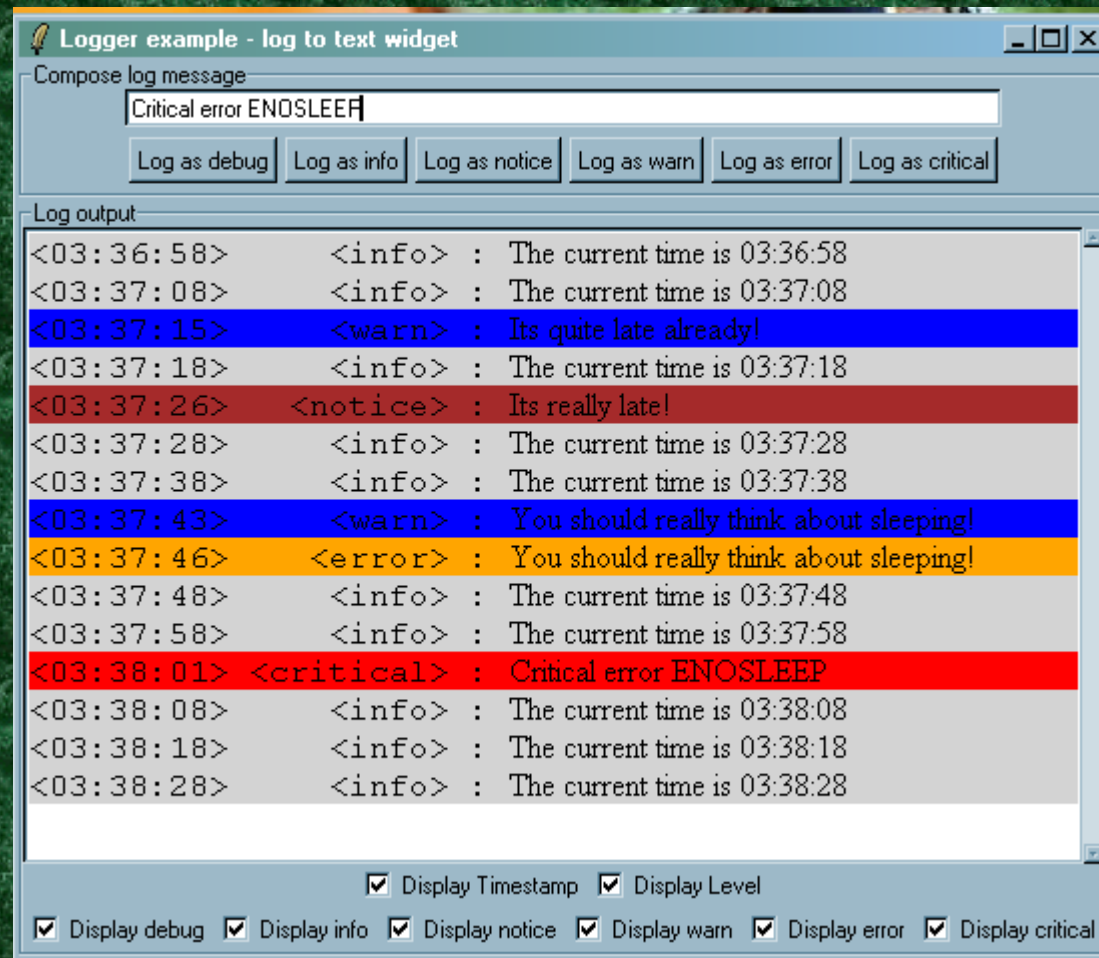
A serious problem

Caller: someproc

Complex logprocs

- Log to a file
- Log to syslog / Windows event log
- Log to a different thread
- Log to a different host using comm
- Log to a database
- Log to a Tk text widget

Log to a text widget



logtotext.tcl in examples/logger of tcllib CVS
(160 LOC including comments)

Using logger with XOTcl and SNIT

- **xotcl-logger.tcl (Proof of concept)**

```
logger::xotcl::Logger Log -servicename global -loglevel warn  
Log log error "A simple error message"  
Log setlevel warn
```

- **snit-logger.tcl (Experimental)**

```
logger::snit::Logger Log -servicename global -loglevel warn  
Log log error "A simple warn message"  
Log configure -loglevel error  
Log setlevel warn
```

logtotext.tcl in examples/logger of tcclib CVS
(160 LOC including comments)

Planned Development

- New logproc interface with format string support for timestamp, stack trace etc.
- Example modules for logging to file, syslog, distributed logging with comm
- `logger::util` package with useful but non-essential tools