Reporting Tools With Tcl

About 1995 some colleagues and myself started in payed work with Tcl when beginning to implement a reporting tool for a sales department. We started with a client/server approach using a Tcl socket interface. That included also a second application for entering data and verifying these data as a front end for the MM module of SAP.

The client was built with a Tcl/Tk UI and was built as an installation package for Windows using the WISE tool for building an installation package. The Server was running on a Sinix (= Siemens version of Unix at that time) machine.

For the socket interface we used a private protocol, we invented for that purpose. It was in general a key value or key values protocol.

The client was divided in 2 parts, a COMMON part with the commonly used procs (there were some other application built on that base later on) and an application specific part which contained the unique information for the UI parts of that application, like report masks, input masks fields in these masks etc.

This version already had an automatic update of the Tcl scripts of the client included, which was done by comparing the time stamps of the client version of the file and the version on the server provided for that file. For that purpose we had a copy of the client directory tree in a dedicated directory path on the server side. So for deploying a new version, it was enough to copy the modified sources (Tcl scripts) into that tree on the server and every client – when starting again – did get always the newest version of the client UI. Because of that approach we did not have a fixed release cycle, as we installed bug fixes – when tested – in the server tree.

In parallel to that PRODUCTION tree there were 2 other trees called TEST and DEVELOPMENT, these were for the developers to implement and to test new parts or bug fixes. At that time the team had up to 8 persons, so synchronization of work was still relatively easy.

When starting the client out-of-date files were fetched from the server and stored in the file hierarchy on the client, so the next time the client was started these files (when no longer out-of-date) could directly be loaded from the client. The direct loading was also done for the files not out-of-date on the client. A Tcl script therefor was either read from the client hierarchy or fetched from the server and then executed using eval. At that time there was no use of the source command by us.

The goal of the 2 tools was to present financial reports about sales activities as tables like Excel sheets and to provide something similar to a forms input mask to be able to enter data concerning material data.

For the reports there were some comboboxes for selecting the desired report and contents of the report. The contents of the report was also driven by the function of the person using the reports, as the sales people for example were not allowed to see the costs of production.

In a second step there was a replacement of the private socket protocol by use of apache server with rivet on the server side. The socket protocol was hereby replaced by http protocol as provided be apache server. The client now was used like a browser serving http requests instead of the private socket requests. On the server side there was the need for several changes by building .ttml files for all requests which the client did send. The modifications on the client side very small as the socket protocol was already handled at one place, the change was mostly to change that place to send http requests and interpret the answers sent back instead of the socket requests.

In a third step the client was changed to be a starkit instead of a WISE installation package. The advantage was that providing the virtual file system for build the starkit was much easier. Another advantage was, that there was no longer the installation on the user's PC instead of that there was an URL which pointed to the starkit .exe file which as transferred and executed, if out-of-date, otherwise execute from the browser cache. To make the transferred data as small as possible, this .exe file only contained the executables for Tcl, Itcl, Bwidgets and Iwidgets etc. and a small script, which did check, if the .kit file which contained the client scripts was up-to-date or not and if necessary did copy the .kit file to the PC. In the .kit file there was the client file hierarchy as before, so from the execution point of view it was nearly the same as before.

The .kit file did contain a generic client, which did get the exact information on how the reports and masks were built from the server using http requests. With that technique also the change rate for the .kit file was relatively low.

In a fourth step the layout of the client was made still more generic in putting the meta info for the layout in a DB (mysql or oracle) and using a common meta interface for accessing the databases. This meta interface was partly generic and partly specific for these kind of requests. The client itself was changed to be more generic, which means it had generic parts like a “TableReport” a report which did present data like a spreadsheet table, a “TreeReport” which did present data in a tree structure, a “combobox” part, a “label” part, an “EntryField” part etc. TableReport and TreeReport were both based on a Report class for the common interfaces of both parts. Also the other parts were based on a class which did contain elements common for all of these parts like a label a layout etc.

The specific layout of the report parts was driven by attributes, which could be set for all of these parts and did drive the appearance of these parts including relief, justifying, anchoring etc.

Based on these report parts a report could also be an “InputReport” which means that type of report was built from elements like labeled widgets, entryfields, comboboxes, datefields, calendar widgets etc.

A Report further on could be built from a TableReport part and an InputReport part so it did have a spreadsheet area and a forms for input area.

To allow several reports in one application there was use of the tabnotebook widget. The report when selecting a tab was built dynamically, if it did not yet exist to avoid long startup times.

That was the general evolution of this kind of application during about 15 years. I have omitted a lot of details, which I will partly present later on.

Based on that experience I decided in December 2008 to start an open source project named “Reporting Tools With Tcl”, which should be mostly a reimplementation of the ideas of these applications with lessons learned influencing the design of the tool.

This is step five which uses also newer features found in itcl-ng and newer features found in Tcl/Tk. Another change uses sqlite as the database for the attributes and meta information and splitting up of common meta information for all applications in one sqlite database scheme and one sqlite scheme for the meta information for every application.

The existing tool for entering and changing the meta layout/attribute information in the database, which is also based on the same technique will be replaced by a new version a still more generic interface. Doing that also the technique for storing the layout and attribute infos will be reworked, to simplify the information on a lot of places and make it more easy to work on/enter that information in the administration databases.

There is a plan to enable use of different UI techniques like tile and to therefore keep the information in a still more generic way than it is done in the current version.

The general structure of the client will be very similar to the existing client. It will use paned windows for the toplevel structure. One contains the tree for selection of the report and one contains the report part which can be a report an input report or a combination of the two. The new client will use a tree instead of the tabnotebook for selection of a report.

Each report consists of a frame which contains as layout up to 9 frames, 3 rows of frames and 3 columns existing of a frame for each column for each row. The layout within such a frame can again consist of up to 9 frames and so on.

The frames are positioned using the grid manager. This allows a frame to span over more than one row or column. Each of the 9 parts of the layout frames can also be empty, which means can be not existing. This structure allows very different layouts as the experience of the last years with that layout scheme has shown. There was no need to have more than these frames in any application which has been built on that base.

A frame with all the selection elements can be placed on every of the 4 borders using the scheme as mentioned above. With the nesting of the frames it is for example also very easy to have the forms part (InputReport) and the table part (TableReport) in one of the outer frames by using the inner frames as containers for the parts. The layout where all the parts are placed are driven by information from the meta information database.

Many of the UI elements like entryfield and combobox are derived from the corresponding widgets in Iwidgets with some additional attributes like for example a text_id for being able to have language dependent texts or a row and a column to specify where the widget is placed within a frame or the name of the frame in the frame hierarchy. Other attributes are: adjust, label position and sticky. As all the UI elements have these additional attributes, the parts for putting the UI elements on the correct place can be rather generic. So it is possible to have a small and dumb client for building the UI which is get the information on what to do from the server using the UI elements and the attributes. In general a new UI element can be added without the need to modify the part for placing that element in the UI.

.To be able to provide rather generic TableReport and InputReport parts the information about database tables and columns within that table including attributes about data types of the fields was also held in the meta information database tables. There was additional information provided in other meta information db tables for example which function could see or even modify fields in the target database.

For that reason database fields were tied together with function information and there were tables containing the persons and the department infos for a person and the functions a person had.

A function was tied to an application and the target database fields also were tied to an application. So having a person assigned to a function and having the fields also assigned to a function it was possible to implement automatic rules on which fields within an application a person was allowed to read or to modify for a TableReport or an InputReport.

The information about the field name,table name, field attributes and the application was stored in meta information tables called apptabname, appfldname, appfldattr and application and the final information for all that and some additional attributes in table appfldinfo.

For a person the information was put into table appperson with additional attributes/fields like the phone number, the department reference, the cost center reference etc. The department info was in table appdepartment and the cost center info in table appcostcenter

Each person can have one or more functions assigned, which decide, what application and which fileds in an application a person can see or modify. The information is in table function and appfunction.

The appfldinfo entry and an appfunction entry are put into a table called fldfunctioninfo. This defines the dependency of an application field and a function of an application.

The combination of a person, an application and the function(s) for that person were stored in a table called apppersonfunction with references to appperson and appfunction.

The combination of these tables allows to build for example the layout of an InputReport by just adding information on which application specific fields should be shown in which row and column in the InputReport by putting the info in a table called fldgrids which add a row and a column to a fldfunctioninfo entry.

Information for a report is also stored in some tables. Appwidgettype is a table which has the information about the classes (Itcl classes) which build a widget. In there are for example all the 9 frames mentioned above or the TableReport and the InputReport.

The possible attributes of these widget(classes) are in table appwidgettypeoption together with the default values for the attributes. These include the optionname, the defaultvalue and the component name, if the option is for a specific component in more complex widget like for example the listbox component of a combobox. The information for a widget could be loaded with a small programm, which just created an instance of a widget and then asked for the options/attributes using the configure command and putting all that information into appwidgettypeoption table.

The information about a concrete instance of a report i.e. a TableReport or an InputReport is stored in table appreport with a reference to appwidgettype and information about the window path name part of that component and the information on the title of that report (again language dependent using also apptext for that).

As a report normally consists of frames and comboboxes for selection of the contents of that report there are tables appreportcombo for the comboboxes and appframes for the frames used for that report. An appreportcombo consists of a selection widget to be flexible, there can be different sorts selection widgets or special cases of comboboxes like a calendar combobox which are stored in appselectionwiget, which has a appwidgettype as the base.

The attributes/options for reports, comboboxes, frames etc. are in appwidget, which defines if that is a report, a combobox or a frame or a fldunctioninfo field and has the concrete defined values for that instance in appwidgetoption, which uses appwidgettypeoption and the default value from appwidgettypeoption if nothing is defined in appwidgetoption and the value in appwidgetoption, if that default value shall be changed for that instance.

When a report is constructed on the client a request is made to the server to collect all the information for the report instance, which is then sent to the client in a defined format and the client is using that information for constructing the report and configuring the report using the attributes from that info and doing a configure command for these options. Right now that is done using a format defined by the tool, but it would be better to use dicts at that place instead.

After the report widget has been constructed and configured on the client, the comboboxes are filled with information as desired. The Information on how to fill a combobox can be in a meta information table appcombofillselect including a select string with generic table names and with where clauses or it can be defined in a script on a the server side. Both kinds return the info in the same defined format.