`editTable`A Generic Display and Edit Widget for Database Applications

Clif Flynt
Noumena Corporation,
8888 Black Pine Ln,
Whitmore Lake, MI 48189,
`http://www.noucorp.com`
`clif at noucorp dot com`

October 24, 2012

Abstract

An application that includes a database always requires a set of pages to edit the contents of the database.

The bulk of the edit pages are simple and easy to write. Even with Tk's ease in constructing simple data-entry pagespages, writing a dozen procedures takes time that could be spent on the more interesting parts of a project.

The goal of the `EditTables` package is to provide a set of good enough pages with no effort on the part of the programmer, better pages with a bit of effort, and a framework for building the fully featured pages an end user will demand.

The implementation uses TclOO for a base class with mixins to provide customized behavior for particular database engines. The test engines are sqlite3 and TDBC.

1 Introduction

One application of the 80/20 rule is that 80 percent of an application is the code that's boring to write, and 20 percent will be fun. In a database application with many data entry screens this ratio could push 95-5.

It would be nice if the 95 percent part of a database application could be generated without requiring 95 percent of the programmer's time.

It turns out that this can be done.

There is enough information in an SQL schema to create a simple entry screen. The following schema can be used to generate an adequate GUI:

```
table create phoneList {
  id integer unique,
  name text,  -- person
  phone text, -- phone number
  type text -- home, work, mobile
}
```

The GUI would resemble:

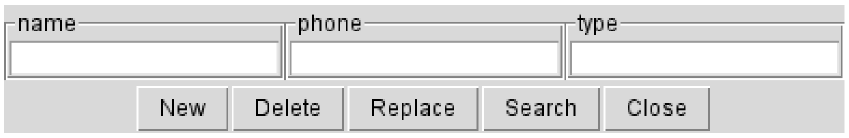


Figure 1: Simple, uninstrumented GUI

By adding some extra information about the fields, the GUI can be improved to resemble the following image:

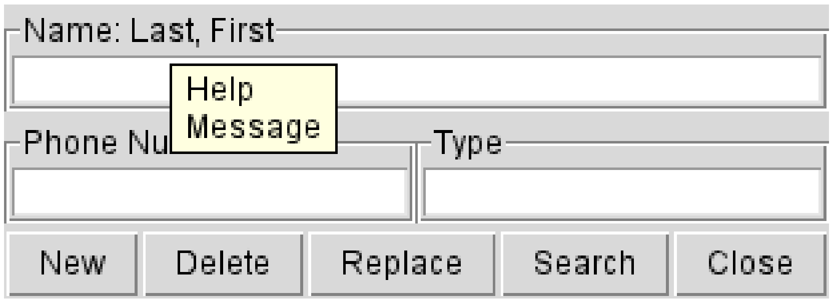


Figure 2: Simple, instrumented GUI

To paraphrase many lolcats, *Simple tables are Simple*. In order to be useful, the `EditTables` package needed handle one-to-one mapping schemas, such those containing references to other tables.

```
table create phoneList {
  id integer unique,
  name text,
  num  text,
  typeid integer references types
}
```

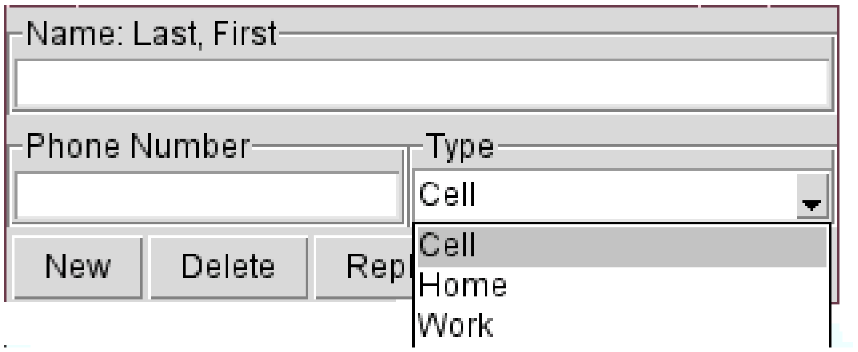


Figure 3: Reference, instrumented GUI

The package also needs to handle many-to-one mapping, perhaps a fixed number of fields as shown below:

```
table create person {
      id integer unique,
      name text,
      address text,
    }
  }
table create phone {
      id integer unique,
      num  text,
      personid integer references person,
      typeid integer references types
    }
```

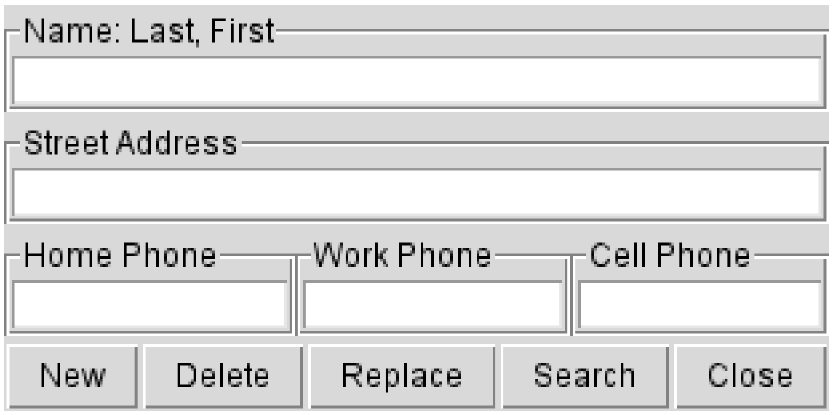


Figure 4: Reference to table, instrumented GUI

The trick to generating these sorts of GUI's instead of just a simple GUI from the pure schema requires adding some layout instructions.

The next requirement is to validate inputs. This may need to be done on a per-field basis, or when the user submits a page. Both of these options are supported in this package.

Finally, a useful package needs to work with multiple database back ends. It would be nice to only support TDBC, but in the real world, TDBC doesn't have enough penetration (yet).

There are features in Tcl 8.6 that allow all of the requirements to be met.

The package is written using TclOO. The base package understands Tk and how an SQL schema is laid out. It provides some primitives that the other classes can use to find a primary field, find references, etc.

If the project starts using the `EditTable` package, the layout instructions can be embedded into the SQL schema and extracted as needed. If `EditTable` is being shoe-horned onto an existing database, or if you find the idea of mixing layout commands and data definitions distasteful, the instrumentation can be provided by modifying the method that acquires the layout commands in the Tcl object.

Validation is accomplished by adding flags to fields with some common validation routines available in the main class, and the capability of adding custom validation if an application requires it.

Finally, support for multiple database backends is accomplished using TclOO's `mixin` feature.

The current version of `editTable` is the fourth or fifth iteration of ideas. It's being used in a commercial product. It's also likely to see a number of modifications before it gets used in another product.

2 Using `EditTable` package

2.1 Creation

The `EditTable` package is constructed using the TclOO megawidget design pattern described by Donal Fellows (Tcl/Tk Proceedings 2009) and in my book (`Tcl/Tk: A Developer's Guide`, Chapter 11). This pattern emulates a standard Tcl widget, allowing the new `editTable` object to be created like any other Tk widget.

Syntax: `editTable` *`widgetName mixin dbEngine args`*

`widgetName`	Name of the widget, normal Tk style
`mixin`	Name of the db engine mixin
`dbEngine`	Name of the db engine to use
`args`	Optional arguments for opening the db engine

Creating an `editTable` connected to an SQLite database named `test.db` would resemble:

```
set obj [editTable .t1 SQLITE3_support sqlite3 -dbArgs test.db]
```

Doing the same with a TDBC engine that's connected to the SQLite database would resemble:

```
set obj [editTable .t1 TDBC_support sqlite3 test.db]
```

Once an `editTable` object has been created, it can be used to query various parameters using the `config` or `configure` commands (both map to the same underlying code.)

Like Tk, the `configure` command will accept no arguments to return all configuration options and values, a single argument to report the value of an option, or a pair of arguments to set a value.

One of the options that can be extracted is the underlying database pointer. Extracting this allows an application to interact directely with the database object opened by `editTable`. This end-run functionality allows applications to easily extend the behavior of the `editTable` class.

The next snippet shows creating an `editTable` object, extracting the TDBC database object and using that to create a table in the underlying sqlite3 database.

```
set obj [editTable .t1 TDBC_support sqlite3 test.db]
set db [$obj config db]
$db allrows {
  CREATE TABLE person (
    id integer unique primary key,
    loginid text, -- loginID
    fname text, -- First name
    lname text, -- Last name
    addrRef integer references addr
    );
}
```

There are several methods defined for the `editTable` object. The most commonly used are:

- *`editObj`* `config`
- *`editObj`* `getSchema`
- *`editObj`* `makeGUI`
- *`editObj`* `populateBySearch`

The config command will let the application query or set configuration options. If it is called with no key, it will return a list of all currently defined keys and values.

Syntax: `editObj` *`config ?key? ?value?`*

`editObj`	A widget created with `editTable` command
`config`	Query or set a config option
`key`	Name of the config option or blank
`?value?`	Optional value to define an option

The `getSchema` method will retrieve a schema for a table from the database, or optional schema retrieval process. The default behavior is to query the database. This is used internally to build GUIs for tables containing relations.

Syntax: `editObj` *`getSchema tableName`*

`editObj`	A widget created with `editTable` command
`getSchema`	Return the schema for a table
`tableName`	Name of the table to return Schema for

The `makeGUI` method is the workhorse that builds a GUI within the `editTable` frame. It can build a GUI for any table defined within the database. The `editTable` object can rebuild itself to display a different table as necessary. The default GUI includes buttons to perform simple searches, and add, delete or modify a record.

Syntax: `editObj` *`makeGUI schema`*

`editObj`	A widget created with `editTable` command
`makeGUI`	Construct a GUI within the editTable object frame.
`schema`	a Schema - may be return from `getSchema`

The `populateBySearch` method will load values into the fields of a GUI. The query can be any valid query suitable for an SQL `SELECT` on the currently active table.

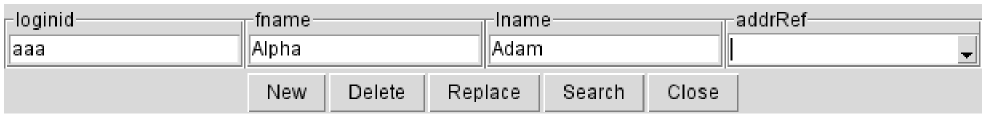
Syntax: `editObj` *`populateBySearch query`*

`editObj`	A widget created with `editTable` command
`populateBySearch`	Populates the values in a GUI
`query`	An SQL query

The next example shows initializing a sample database and creating a simple GUI for a table.

```
toplevel .tt
set obj [editTable .tt.t1 TDBC_support sqlite3 -dbArgs test2.db]
set db [$obj config db]

$db allrows {
  CREATE TABLE person (
    id integer unique primary key,
    loginid text, -- loginID
    fname text, -- First name
    lname text, -- Last name
    addrRef integer references addr
    );
    CREATE TABLE addr (
    id integer unique primary key,
    street text, -- Address
    city text, -- Address
    state text -- Address
  )
}

$db allrows {INSERT INTO addr VALUES (1, '123 St', 'Acity', 'AA')}
$db allrows {INSERT INTO addr VALUES (2, '234 St', 'Bcity', 'BB')}
$db allrows {INSERT INTO person VALUES (1, 'aaa', 'Alpha', 'Adam', 1)}
$db allrows {INSERT INTO person VALUES (2, 'bbb', 'Beta', 'Blocker', 2)}
$db allrows {INSERT INTO person VALUES (3, 'aa2', 'Abel', 'Adam', 2)}
$db allrows {INSERT INTO person VALUES (4, 'aa3', 'Aard', 'Adam', 2)}

$obj makeGUI [$obj getSchema person]
$obj config -table person
$obj populateBySearch "loginid = 'aaa'"

pack .tt.t1
```

The generated GUI resembles this:



loginid	fname	lname	addrRef
aaa	Alpha	Adam	

Figure 5: default GUI

2.2 Layout Information

Because we are polite and civilized, we will not call the previous example butt-ugly.

But it is.

An SQL schema is designed to convey the logical date relationships between fields and tables. It is not designed to convey any information about aesthetics.

The aesthetic information is conveyed as a six element list attached to each field that is to be displayed. The elements are shown in the following list. The first 2 are required, the others have default values that may be adequate.

`help`	A message to display in a popup help balloon
`label`	The label to display with this field
`reference`	If this field references another table, this element will contain the `table.field` name of the referenced table. If this field contains non-referential data, this element is blank.
`widget arguments`	arguments for the entry or combobox widget that will be created. These might include `-width` or `-background` arguments.
`row column`	A list of the row and column where this field is to be displayed.
`grid options`	Arguments to be attached to a grid command for this field. These might include `-columnspan` for example.

This technique for conveying layout info is sub-optimal. The requirements grew as the complexity of the application grew.

But it works.

The layout information is included in a Schema by adopting a modification of the standard SQL comment.

An instrumented schema has the layout string appended to a field definition with a triple-dash comment.

A simple GUI can be created like this:

```
set obj [editTable .t1 "" "" -table phoneList]

$obj makeGUI {
 table create phoneList (
 id integer unique,
 name text,  --- {First Last} {Name} {} {-width 40} {1 1} {-columnspan 2}
 num  text, --- {Number with area code} {Phone Number} {} {} {2 1}
 type text  --- {Type of phone} {Type} {} {} {2 2}
 );
}

grid .t1
```

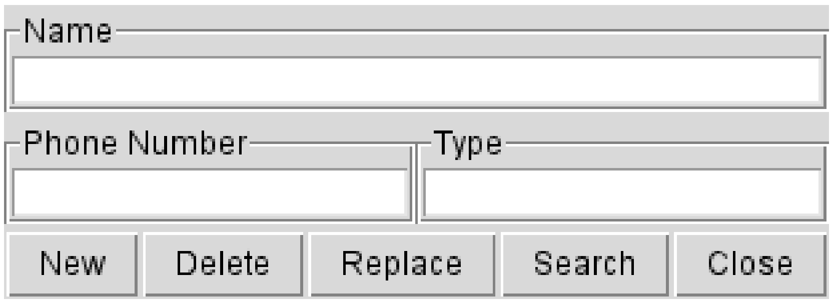


Figure 6: simple GUI with layout info

The `editTable` widget requires that the SQL schema follow strict rules and a field which references another table must include a *`references`* clause.

The presence or absense of the *`references`* clause determines how the third field - the `reference` element is to be treated.

If a *`references`* clause is present, then the `reference` element contains the database field name, or list of field names, to display in each element of a combobox. The fields to display must be named as `tableName.fieldName`.

One pattern used with references is for a table to reference one of several options. The next example is a database in which the book table has a string for title and a reference for author. The author table has separate fields for first and last name. The combination of first and last is displayed in the combobox for selecting an author.

```
set obj [editTable .book SQLITE3_support sqlite3 -dbArgs test4.db]
set db [$obj config db]

$db eval {
 CREATE TABLE book (
 id integer unique primary key,
 title text, --- {} {Title} {} {-width 40} {1 1} {-columnspan 2}
 authorid integer references author
     --- {} {Author} {author.first author.last} {} {2 1}
 );

 CREATE TABLE author (
 id integer unique primary key,
 first text, --- {First Name} {First} {}
 last text  --- {Last Name} {Last} {}
 );
}

$db eval "INSERT INTO author VALUES (1, 'Clif', 'Flynt')"
$db eval "INSERT INTO author VALUES (2, 'Mark', 'Twain')"
$db eval "INSERT INTO book VALUES (1,'Tcl/Tk: A Developer''s Guide', 1)"
```

```
$db eval "INSERT INTO book VALUES (2,'Tom Sawyer', 2)"
$db eval "INSERT INTO book VALUES (3,'Huckleberry Finn', 2)"

$obj makeGUI [$obj getSchema book]
$obj config -table book
$obj populateBySearch "title like '%Tcl%'"
grid .book
```

The GUI generated from this code looks like this:

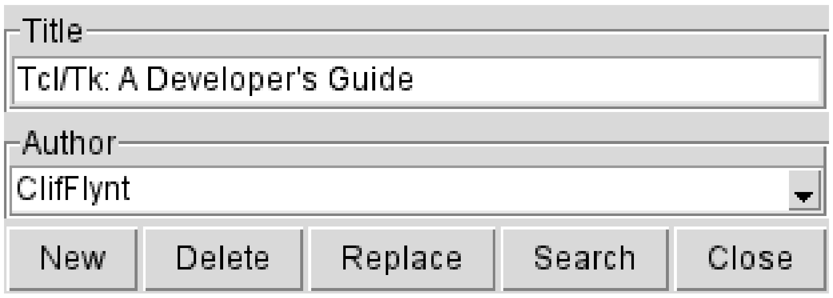


Figure 7: Schema with reference and layout info

Another common pattern is the many-to-one mapping implemented by having a field in one table point back to another table. In a book database, there are an undefined number of keywords that might be attached to a book.

The schema for this pattern would resemble:

```
 CREATE TABLE book (
   id integer unique primary key,
   title text,
 );
 CREATE TABLE keyword (
   id integer unique primary key,
   key text,
   bookid integer references book
 );
```

To generate a GUI for this database pattern, the `reference` element in the layout field in the `keyword` table is used to hold the name of a table. When a GUI is generated for a table listed in the `reference` element, an *`Associated values`* section is created in the GUI.

Expanding the previous example to include a reference table, the new table resembles this

```
 CREATE TABLE keyword (
   id integer unique primary key,
   key text, --- {Keyword} {Category} {book} {} {}
   bookid integer references book --- {book} {Title} {book.title} {} {}
 );
```

Which creates the following GUI:

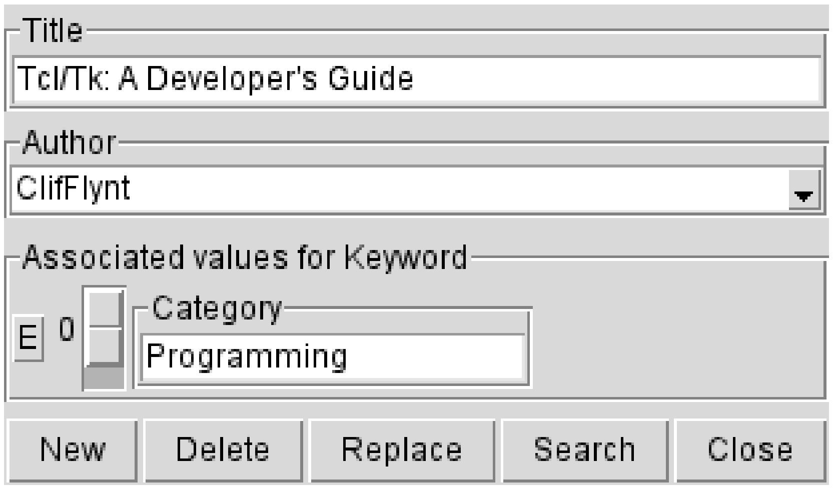


Figure 8: Schema with reference and layout info

The `E` button in the previous image opens a new toplevel to edit a keyword which will be associated with this table row.

2.3 Customization

The `editTable` widget follows the Tk dictum of being adequate with no tweaking, and open for customization if the application isn't suited to the base GUI.

Various degrees of customization are supported. These techniques include:

- using the layout elements
- configuring options for screen or field validation
- building a separate mix-in or inherited class
- tweaking the GUI before using it

Using layout elements to modify the appearance of the GUI works as described.

By default, a `editTable` GUI has no validation. Per-Field validation can be enabled by configuring the `validate,`*`fieldName`* attributes for a GUI. The validation attributes accept a script to evaluate when focus leaves that field. The widget name (entry or combobox) and field name are appended to the script when it's invoked.

The `editTable` class includes some trivial validation methods including `validatePhone` and `validateNumeric`.

2.4 Extending the class

The current implementation of `editTable` has mixins for Sqlite3 and TDBC. A programmer can add a new mixin to support another database engine by coding the engine-specific methods:

`init`	Initialize a connection to the database.
`getSchema`	Retrieve the schema for a given table. Setting `save` sets this to be the active table.)
`doSQL`	Execute an SQL command and return whatever the command returns. This provides a generic access to the underlying database.
`getTables`	Return a list of the tables defined by this database.
`getPrimary`	Retrieve the name of the primary key.
`doReplace`	Update the DB row based on the contents of the GUI.
`doNew`	Create a new row in the DB based on the contents of the GUI.
`populateBySearch`	Populate the GUI based on an SQL query.
`getValues`	Return a set of values based on a list of *`tableName.fieldName`* values
`getValueByRef`	Returns a value from table *`$tbl`* for field *`$fld`* where the primary is *`$value`*
`populateWithFwdRef`	Populate a GUI that includes forward references based on an SQL query
`closeDB`	Close the connection to the DB

2.4.1 GUI Modification

Since the GUI follows a fixed pattern, it can be modified post-`makeGUI` and before display.

For example, if users are only allowed to view and edit fields, the the delete button can be removed with a command like:

```
$obj makeGUI $specialSchema noClose
$obj configure -table book
```

Getting more aggressive, rather than showing the *`Associated values`* for a row, as they are displayed by default, an application can define a fixed quantity of associated values and a custom display procedure while taking advantage of the bulk of the `editTables` widget.

The next example demonstrates a book database that emulates a card catalog via a slider on the side to select books, and a simplified display of keywords.

It does this by with a special procedure to construct the GUI and a new procedure to extend the normal `editTable` population methods. The new procedure for creating the GUI (`bldBookGUI`) defines a modified schema with a couple extra fields to hold the keywords. The `showItem` procedure uses the `populateBySearch` method to populate the fields that are defined in the `book` schema, and has extra code to populate the extra fields.

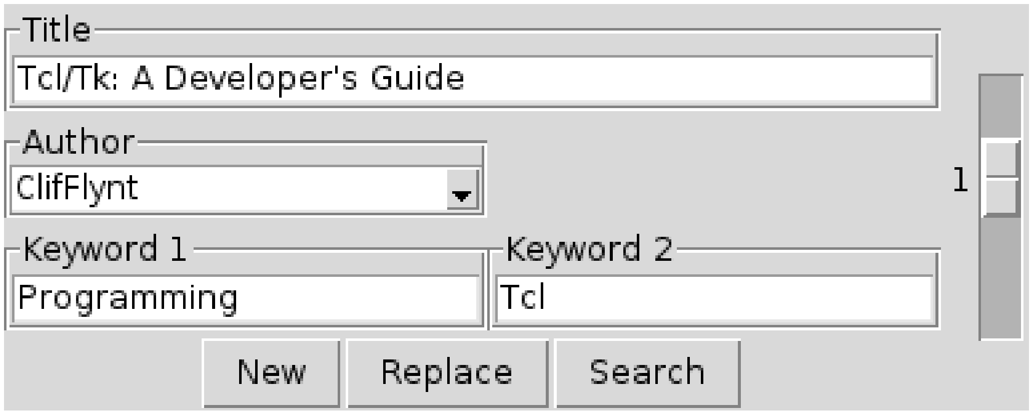

Title: Tcl/Tk: A Developer's Guide
Author: ClifFlynt
Keyword 1: Programming
Keyword 2: Tcl

Figure 9: Schema with reference and layout info

```
toplevel .t3

set obj [editTable .t3.t2 SQLITE3_support sqlite3 -dbArgs test3.db]

proc bldBookGUI {obj} {
  set specialSchema {
 CREATE TABLE book (
   id integer unique primary key,
   title text, --- {Title} {Title} {} {-width 40} {1 1} {-columnspan 2}
   authorid integer references author, --- {Author} {Author} {author.first autho
   keyword1 text --- {Keyword} {Keyword 1} {keyword.key} {} {3 1}
   keyword2 text --- {Keyword} {Keyword 2} {keyword.key} {} {3 2}
 );
  }

  $obj makeGUI $specialSchema noClose
  $obj configure -table book
  destroy $obj.buttons.b_Delete
  set prim [$obj getPrimary book]
  set count [$obj doSQL "SELECT count($prim) FROM book"]

  set scale [scale .t3.sc -from 0 -to $count \
      -command [list showItem $obj $specialSchema .t3.sc ]]
  grid $obj $scale
}

proc showItem {obj schema scale num} {
  $obj populateBySearch "id=$num"

  set pos 0
  foreach l [split $schema \n] {
    lassign [$obj splitSchemaLine $l] def layout
    lassign $layout help label ref args rowcol gridO
    if {($ref ne "") && ([string first "references" $l] < 0)} {
      set dfld [lindex $l 0]
      lassign [split $ref .] tbl fld
      set lst [$obj doSQL "SELECT $fld FROM $tbl WHERE bookid=$num"]
      set item [lindex $lst $pos]
      incr pos
      $obj configure -value,$dfld $item

    }

  }
}
bldBookGUI $obj
```

3 Implementation

The `editTable` widget is implemented as a TclOO Megawidget with mixins to provide database engine customization.

Another standard pattern would have been to make each database engine a class that inherited the base functions from `editTable`. I considered and discarded this pattern in order to have a single widget class (`editTable`) rather than multiple classes (`editSQLite`, `editTDBC`, etc.).

The `editTable` class uses the technique for creating a megawidget described by Donal Fellows in his 2009 paper, and also described in my book (`Tcl/Tk: A Developer's Guide`, Chapter 11) (Shameless plug).

This technique uses a `classmethod` call to create an `unknown` method that checks to see if the first argument is a window name, and if so proceeds to create the new widget.

```
# Create a class method
# Avoid redefining classmethod if it already exists.
if {[info proc ::oo::define::classmethod] eq ""} {
proc ::oo::define::classmethod {name {args ""} {body ""}} {
   # Create the method on the class if the caller gave
   # arguments and body
   if {[llength [info level 0]] == 4} {
        uplevel 1 [list self method $name $args $body]
   }
   # Get the name of the class being defined
   set cls [lindex [info level -1] 1]
   # Make connection to private class "my" command by
   # forwarding
   uplevel forward $name [info object namespace $cls]::my $name
}
}

oo::class create editTable {
  classmethod unknown {w args} {
# puts "UNKL: $w -- $args"
    if {[string match .* $w]} {
      [self] new $w {*}$args
      return $w
    }
    next $w {*}$args
  }
...
}
```

When a new instance is created, one of the arguments describes the mixin to be added. Since mixins don't have constructors, the `editTable` constructor calls a `init` method defined within the mixin to perform special initializations.

The `editTable` class can create objects that have lives of their own. These includes GUI widgets, open database connections and may include slaved `editTable` GUIs. Because of these additional elements to the `editTable` class, the `editTable` class requires a destructor.

Each `editTable` object contains a list of cloned and referenced objects. The `desstructor` descends upon these like a wolf upon the fold destroying them in gay abandon.

While the `editTable` class is happy to have multiple channels to databases, many database engines are less happy with this. The `init` method in the mixins avoids opening multiple channels. The `editTable` class must also avoid closing a database channel until all users have been destroyed.

A class variable is used to keep track of the number of open database connections. Again, this code was stolen from Donal Fellows' 2009 talk and my book.

```
if {[info proc ::oo::Helpers::classvar] eq ""} {
proc ::oo::Helpers::classvar {args} {
    # Get reference to classs namespace
    set ns [info object namespace [uplevel 1 {self class}]]
    # Double up the list of varnames
    foreach v $args {
      uplevel 1 namespace upvar $ns $v $v
    }
}
}
```

4 Future

The method of conveying the layout information, and what information is required has been evolutionary. It is subject to a rework before the this package gets used in another application.

The schema parsing is a quick and dirty approach that assumes each field definition is contained on a single line. The schema parsing will be enhanced as a method of the base `editTable` class.

It may be possible to reduce the number of methods in the `mixin` classes by better use of the basic methods in the `editTables` class.

5 Summary

The current status of the `editTable` class is *`functional`*. It's in use in a commercial product and is scheduled for a several other in-house and out-house projects.

What started as a simple way to generate a *good enough* GUI that reflected the underlying database schema has expanded into a package that can generate a commercial-grade GUI with capability of hiding the schema and providing a simple interface to a user.

The underlying simple display has been retained with potential tweaks to extend the behavior beyond simple.

As the class is force-fed to other applications I expect to discover more things it *should* do, and streamline the current feature set.

The current escape is available at `http://www.noucorp.com`.