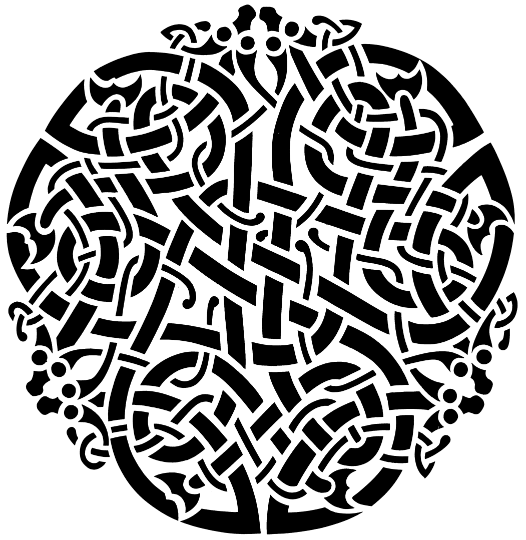


TAO/TK

A TclOO Based Toolkit for GUI Design

Presented at the 20th Annual Tcl Developer's Conference (Tcl'2013)
New Orleans, LA
September 23-27, 2013

Sean Deely Woods
Senior Developer
Test and Evaluations Solutions, LLC
400 Holiday Court
Suite 204
Warrenton, VA 20185

Table of Contents

Introduction

In this paper, I will describe TAO/TK, a comprehensive architecture for implementing user interfaces, control systems, and machine learning. TAO is a dialect of TclOO. TAO builds on TclOO and adds its own notations and policies in a way that allows it to interact with other TclOO code.

TAO adds features that are required to make complex systems of related classes easier. TAO's main feature is passing data through inheritance as well as code. TAO/TK extends TAO into a more familiar set of widget and megawidgets.

TAO/TK requires Sqlite and Tcl 8.6. Some of you may remember a past paper of mine, also on an object system called "TAO". This project is TAO 1.0's successor, having been rewritten from the ground up to use TclOO.

TAO Parser

Some of you may remember my paper from 2006 on my concept of TAO. TAO then was my answer to problems I had encountered with [Incr Tcl], and was essentially Pure-Tcl code, with a syntactic sugar parser, and held together with Sqlite.

And aside from the name and the fact that, yet again I seem to have invented a syntactic sugar parser held together with Sqlite, there aren't many similarities between the two projects.

The code that can be implemented is not the real code, after all. ;-)

TAO has been reinvented using TclOO at it's core. To introduce new keywords and design patterns all TAO code passes through the TAO Parser. The TAO Parser borrows heavily from the TclOO parser. Several TclOO keywords are intercepted and their contents logged into the in-memory sqlite database. Wholly new keywords string together a series of TclOO calls to dynamically produce TclOO code. A TAO produced object or class is 100% interchangeable with a TclOO object or class.

The TAO parser operates in it's own namespace: ::tao. The ::tao::class command works like a combination of ::oo::class create and ::oo::define. It will either create a new class, or modify an existing one.

```
::tao::class foo {
  # Works just like TclOO class
  # definitions
  method noop args {
    return {}
  }
}
::tao::class bar {
  superclass foo
  method someop args {
    return some
  }
}
```

Properties

In a complex system where a rule has to be written to handle a range of different classes of objects, it is often useful to be able to refer to some meta-information within the object. It is also helpful to have that meta-information inherited along with the methods.

```
::tao::class animal {
  property tkingdom Animalia
  property has_spine 0
}
::tao::class vertebrate {
  superclass animal
  property torder Chordata
  property has_spine 1
}
::tao::class mammal {
  superclass vertebrate
  property tclass Mammalia
  property has_fur 1
}
::tao::class carnivore {
  superclass mammal
  property torder Canivora
}
::tao::class feline {
  superclass carnivore
  property tfamily Felidae
}
::tao::class felis {
  superclass feline
  property tgenus Felis
}
```

In this above example, we are creating the taxonomic classification of the common housecat. In that classification, we are seeding some useful traits that will be passed along to descendents of the class above.

The idea being that by the time we get down to putting together the final leaf classes, the code is simply:

```
::tao::class housecat {
  superclass felis
  property tspecies domesticus
}
```

And should a question arise, all objects of that class can answer based in information inherited by ancestral classes.

```
::tao::class housecat {
  superclass felis
  property tspecies domesticus
}

housecat create Thomas
Thomas property torder
> Chordata
Thomas property has_fur
> 1
Thomas property has_backbone
> 1
```

Different kinds of Properties

Constant properties, being the most common, have the simplest notation. However, constant properties are just one type that TAO supports. To use one of the other property types, specify an additional argument. When the property keyword sees 4 arguments, the third is interpreted as the type of property.

```
::tao::class housecat {
 property dob option {
  storage date
 }
 property age eval {my Age}

 method Age {} {
   set dob [date_to_julian [my cget dob]]
   set now [today_julian]
   return [expr {($now-$dob)/365}]
 }
}
Thomas configure -dob 2010/01/01
# Assuming we ask on 2013/09/01
Thomas property age
> 3
```

Type	Description
`const`	A property that always returns a constant
`eval`	A property whose value is generated by evaluating a command run within the object's namespace.
`subst`	A property whose value is generated by evaluating an expression run through subst.
`variable`	A property whose value is retrieved from an internal variable of the same name.
`option`	A property which is treated as an option. See ***Option Handling***.
`signal`	A property which is treated as a signal. See Locks, Signals, and Notifications

Option Handling

All TAO objects (not just he graphical ones) can take on of Options that can be given to the constructor and/or modified during runtime via the configure command. Much like Tk Widgets. Options are tracked using the same mechanisms as properties. Thus they are inherited like properties. Options are specified in Dict format, because we need to track a lot more than a constant value. For convenience, options have their own keyword.

```
::tao::class housecat {
  superclass felis
  property gender option {
    widget select
    storage string
    values {{} male female}
    default {}
  }
  option weight {
    widget scale
    units kg
    range {0 10}
  }
}
```

With the modification above, we can specify the animal's weight and gender at creation time:

```
housecat create Thomas \
  -gender male -weight 10
housecat create Thomasina \
 -gender female -weight 8
# After a year of snacking
Thomas configure -weight 11
```

Because in large systems one may need to perform a dump of information from a database or some other source, TAO will also accept configure options as a key/value list with or without the dashes (See: Constructor Option Syntax).

```
housecat create Thomas {
  gender male
  weight 10
}
housecat create Thomasina \
 gender female \
  weight 8
# After a year of snacking
Thomas configure weight 11
# And just to show off, how about we
# create objects from db records
db eval {select * from animals where
species='cat'} {
 housecat create ::cat::$uuid [db eval {
    select key,value from attributes where
    uuid=:uuid
 }]
}
```

Option Properties

The TAO parser specifies the following rules for elements given within an option-dict:

Option	Description
`class`	Reference to another option or option class to clone
`default`	The default value for the option
`default-script`	A script to use to generate the default value. If both default and default-script are present, default-script is used. See: `Option Substitution`.
`description`	A human readable comment.
`get-command`	Script to retrieve the value in leu of storing the value internally: See: `Option Substitution`.
`set-command`	Script to set the value externally in leu of storing the value internally. See: `Option Substitution`.
`storage`	Storage type for C, sql, etc.
`validate-command`	Script used to validate incoming values before they are incorporated into the state of the object.
`values`	Specifies a finite list of possible values for the option.
`values-command`	Specified a command to generate the list of finite values. If both values and values-command are specified, values-command is used. See: `Option Substitution`.
`widget`	Widget to use when generating an automated GUI. See `Dynamic Widgets`

Option Substitution

In Tk what information is sent along with a –command option is often widget specific. Many handlers for options need to work over a range of options. Some need to specify an elaborate path that includes the name of the field, the object, and/or the new value. Rather than force the developer to follow a template, TAO allows the developer to specify how information is sent to scripts in the option dict. It works in a mechanism similar to the substitution used by the Tk bind command:

Field	Substitution
`%self%`	The object's name
`%field%`	The field that triggered the script
`%value%`	The value being input (when appropriate)

Option Event Processing

To keep the outcome of events tied to an option consistent, TAO enforces the following order of operations:

1. **validate-command** is run for all incoming values with the property specified. If an error is thrown, the process is aborted without modifying the object. If the problem is encountered in the constructor, the object is destroyed and an error thrown.
2. The **set-command** is run for all incoming values with the property specified. No local value for the option is kept.
3. For values for which the **set-command** property is null, the new value is saved as a dict element in a local variable ***config***.
4. For calls to the configure command, but not during the constructor, the **Option_set** ensemble is called for all incoming values. (More on Method Ensembles in a moment.)

Method Ensembles

For large projects, it is often useful to be able to clump similar functions together. At the same time, it's nice to be able to pop and swap chunks of that ensemble to handle the intricacies of your class system.

TAO has a rudimentary method ensemble system. If the parser detects a method with a "::" in the name, it assumes the portion before the “::” is the ensemble, and after the “::” is the submethod. Method ensembles are passed along to descendents, and descendents can override or extend the ensemble with their own submethods.

```
::tao::class vertebrate {
 superclass animal
  method has::spine {} {
    return 1
  }
}
::tao::class human {
  superclass vertebrate
}
::tao::class politician {
  superclass human
   # ^ Though that may be debatable
  method has::spine {} {
    error {Define ”spine”}
  }
}
```

Method Ensemble Implementation

Ensemble submethods are tracked and catalogued by the TAO parser as a special kind of property. After the class is parsed, TAO builds a series of dynamically generated methods. On is the **property** method, which you have seen earlier. The others include all of the ensemble methods.

Structurally, method ensembles are really a switch statement. Each body of the switch statement is the version of the submethod from the most recent ancestor.

The default for an ensemble is to throw an error when given an unknown submethod. This can be overridden by providing a **default** submethod. The **default** submethod is guaranteed to be the last evaluated. The method that was given on the command line is preserved as the **$method** variable.

```
# An example catch-all for the has method
::tao::class moac {
  method has::default {} {
    return [string is true -strict \
      [my property $method]]
  }
}
```

If we peer inside the **has** method, we can see how it works:

```
info class definition politician has
{method args} {
switch $method {
  <list> { return {spine} }
  spine {
::tao::dynamic_arguments {} {*}$args
   error {Define ”spine”}
  }
  default {
    return [string is true -strict \
      [my property $method]]
  }
}
```

As you can see, in addition to the submethods we have defined in the parser and **default**, our ensemble includes an additional submethod **<list>**. <list> provides a list of all of the valid submethods for the ensemble for this particular class.

Method Ensemble Argument Handling

From our code dump above, you well see a call to **tao::dynamic_arguments**.

tao::dynamic_arguments is a routine that converts the args given to the ensemble into the local variables the rest of the body is expecting and/or throw an error if all of the required values are not given. If the arglist ends with *args*, any number of arguments beyond the mandatory ones will be added to a list called args. If the arglist ends with *dictargs*, any arguments beyond the mandatory ones are placed into a key/value list called *dictargs*. If one argument is given, that argument is assumed to be the key/value list.

```
::tao::class thing {
  method event::random {who dictargs} {
   if {[dict exists $dictargs subject]} {
     set subject [dict get $dictargs subject]
   } else {
     set subject {}
   }
   puts [list subject is $subject]
  }
}
thing create it
it event random
> ERROR: Usage: who ?dictargs?

it event fester subject {Light Bulb}
> subject is {Light Bulb}

it event lurch {subject {Good Evening}}
> subject is {Good Evening}

# And it event does dashes!
it event mortisha -subject {Mon Cher}
> subject is {Mon Cher}
```

Beware: Method ensembles are dynamically generated. A method ensemble will trump a normal method. So, taking the example above, any attempt to implement "has" as a normal method in an descendent will simply be ignored.

```
::tao::class lawyer.honest {
  superclass lawyer
  method has {field value} {
    if { $value eq "spine" } { return 1 }
    …
  }
}
layer.honest create mrsmith
mrsmith has spine
> ERROR: Define "spine"
```

Class Methods

The **class_method** keyword creates a method which operates only on the class object itself. It's like calling **oo::objdefine**, and defining an instance method for the class object. But unlike **oo::objdefine**, **class_method** is passed on to descendents of the class.

One of the handiest provided is the *property* method. It provides the meta-data and constant value properties of the object version. It doesn't, however, supply any of the properties that require gazing into the state of the object.

The most immediate example I have is in taotk's user widgets. We trap the **unknown** handler, and detect if the first argument, instead of being create or new is a tkpath:

```
tao::class taotk::frame {
  class_method unknown args {
    set tkpath [lindex $args 0]
    if {[string index $tkpath 0] eq "."} {
      if {[winfo exists $tkpath]} {
        error "Bad path name $tkpath"
      }
      set obj [my new $tkpath {*}[lrange
$args 1 end]]
      if {![winfo exists $tkpath]} {
        catch {$obj destroy}
        return {}
      }
      $obj tkalias $tkpath
      return $tkpath
    }
  }
}
```

This could work if we did it such:

```
oo::class create Frame {
}
oo::define Frame method unknown args {
  # same as above
}
```

The ideas being that with either case, the class behaves like a tk command:

```
taotk::frame .foo
Frame .bar
```

The difference comes in when we pass this behavior onto descendents:

```
tao::class create taotk::customFrame {
   superclass ::taotk::frame
}
oo::class create CustomFrame {
   superclass Frame
}

taotk::customframe .baz
CustomFrame .bang
> ERROR. Unknown command .baz.
> Valid: create new
```

The DB Backend

TAO uses an in-memory database to index classes and track classes, methods and properties. The database uses sqlite, and can be accessed directly via the **::tao::db** command. A complete schema is available in the Appendix under TAO DB Schema.

If we combine class properties with database backend, we can do some useful searches throughout our library of classes.

```
# Example, find all animals that are
# carnivores
set result {}
tao::db eval {select name from class} {
  if {
    [$name property torder]
    eq “Canivora”
  } {
    lappend result $name
  }
}
```

You may be asking, “Why didn’t you just pull the property from the database directly?” Ok:

```
Select class from property where
property=’torder’ and dict=’Carnivora’;
> carnivore
-- We only get back the one class where the
-- property was defined
select property,dict from property where
class=’carnivore’;
> torder|Carnovora
-- We only defined the one property for
-- that class
```

Now, if we ask the property method:

```
carnivore property const dict
> tkingdom Animalia torder Chordata
> has_spine 1 tclass Mammalia has_fur 1
> torder Carnivora
```

We see that the property method’s picture includes the most recent ancestor’s copy of every property that has been inherited.

Class Regeneration

Every ancestor of every class is indexed in the *ancestry* table, along with which order. That index makes looking up all of the descendents of a class quite simple.

Included with the class table is a simple flag **regenerate**. When the flag is true, the class needs to be regenerated. At the conclusion of the tao::class command, every class that listed **carnivore** as an ancestors was marked **regenerate=1**. After the affected classes are marked, a search is run, the affected classes have their dynamic methods regenerated, and they are marked as **regenerate=0** once again.

The cool part is that if we add a new property to the class:

```
tao::class carnivore {
property has_teeth 1
}
carnivore property const dict
> tkingdom Animalia torder Chordata
> has_spine 1 tclass Mammalia has_fur 1
> torder Carnivora has_teeth 1
```

The new property shows up instantly. This happens because every call to **tao::class** sets off a sequence of events that will cause every class that is effected by the change to regenerate their dynamic methods. If we ask if Thomas (our housecat from Option Handling) has teeth:

```
Thomas property has_teeth
> 1
```

If we had asked that before, the property method would have returned a null.

The Mother of all Classes

All TAO objects descend from a single class: moac, the Mother of All Classes. The moac provides methods which enforce TAO policies and design patterns. This next section is intended as an overview of the key features. See Keyword: class_method

```
Syntax:
  class_method name arglist body
```

The **class_method** keyword defines a method for the class itself. If no **method** of the same name is defined of the class, the implementation will be copied to the objects of the class.

class_method is equivalent to declaring a local method for the class object via:

```
oo::objdefine class method arglist body
```

A method defined by **class_method** will be inherited by descendents of a class, whereas the above example would not.

The Mother of all Classes in the Appendix for a complete reference.

Default Constructor

The Default constructor for the moac is as follows:

```
constructor args {
  my InitializePublic
  my configurelist \
    [::tao::args_to_options {*}$args]
  my initialize
}
```

The **InitializePublic** method initializes all of the variable declared in the class definition as well as provide default values for all options. The next step is to run **configurelist** on the options feed in through args. Finally, the **initialize** method is called. **initialize** is intended to be a place for developers to be able to insert their code and know that all of the variables have been initialized and the object has been configured.

Constructor Option Syntax

The constructor uses the following rules to allow it to work with this range of inputs:

1. Arguments, beyond the mandatory arguments, are considered options.
2. All options must be given in the form of key/value pairs
3. If a single argument for *args* is given, that argument is assumed to be a key/value list of options.
4. Leading dashes (-) are stripped from keys

These rules are enforced by the procedure **::tao::args_to_options**, which the developer is encouraged to use if they should modify the constructor.

Locks, Signals, and Notifications

The moac for TAO defines several methods, and parser keywords to manage locks, signals, and notifications. The following is an overview of their interactions, with a complete reference available in the Appendix under Keyword: class_method

```
Syntax:
  class_method name arglist body
```

The **class_method** keyword defines a method for the class itself. If no **method** of the same name is defined of the class, the implementation will be copied to the objects of the class.

class_method is equivalent to declaring a local method for the class object via:

```
oo::objdefine class method arglist body
```

A method defined by **class_method** will be inherited by descendents of a class, whereas the above example would not.

The Mother of all Classes.

Locks

Locks are a handy way of getting an preventing an object from recursively calling the same routine.

```
::tao::class foo {
  method lock_demo {} {
    # Lock create only returns 1 if the
    # lock is already engaged
    if {[my lock create [self method]]} {
      # I must already be running
      return
    }
    … (Some elaborate action)
    my lock remove [self method]
  }
}
```

Because lock is a public method, other object and system calls can lock and unlock an object.

After the last lock is removed, the object looks to see if it was passed any signals.

Signals

Signals break a menagerie of tasks into discrete pieces that can be received piecemeal and assembled together into a single pipeline of operations.

The “signal” keyword in a class declares a signal. Like options, signals have a descriptor key / value list. Signals are also inherited just like properties and options. Signals have the following properties:

Option	Description
apply_action	Action to perform reflexively when the signal is passed to the object
action	Command to be performed when it is this signal’s stage in the pipeline is reached
aliases	List of names that this signal will respond to
description	Human readable comment.
excludes	List of signals that this signal prevents from running in the current pipeline.
preceeds	List of signals that this signal must preceed in the pipeline
follows	List of signals that this signal must come after in the pipeline
triggers	List of signals that this signal triggers

For a simple example, let’s have an object either fish or cut bait.

```
::tao::class fisherman {
  signal fish {
    follows cut_bait
    triggers cut_bait
    action {my action fish}
  }
  signal cut_bait {
    action {my action cut_bait}
  }
  variable has_bait 0
 }
```

As we can see, the fisherman can either be fishing, or he/she can be cutting bait. To fish, the cut_bait signal must be satisfied.

To see our cunning plan in action:

```
::tao::class fisherman {
  variable has_bait 0
  method action::fish {
    my variable has_bait
    if { $has_bait == 0 } {
      error “I have no bait”
    }
    puts “Fishing”
    # Do the fishing
    set has_bait 0
  }
  method action::cut_bait {} {
    my variable has_bait
    set has_bait 1
    puts “Cutting Bait”
  }
}
fisherman gordan
gordan action fish
> error: I have no bait
gordan signal fish
gordan lock remove_all
Cutting Bait
Fishing
```

Signal_Pipeline

When the last lock is removed from an object, it automatically schedules a call to a method called Signal_Pipeline. Signal_Pipeline figures out which signals have been called, which signals need to be triggered or suppressed as a result, and in which order they need to be executed. It then executes them.

Notifications

Notifications are a message passing system for objects. They are akin to Tk bindings. Object can emit notifications, and other object can receive those notifications.

In the case of emitting an event, as far as an object is concerned it simply belches the message to TAO. The TAO core then looks through subscriptions to find what objects would be interested in receiving a message of that type from the sender object. With list in hand, TAO goes about calling the **notify** method for each recipient with the sender, type, and content of the message.

If we extend our fisherman example, about, lets add a notification to the fisherman that a fish is on the line.

```
::tao::class fisherman {
  notify::fish_on {snd info} {
     set caught [dict get $info caught_by]
     if { $caught ne [self] } continue
     set fish [dict get $info fish]
     my action catch_fish $fish
  }
}
```

For the fisherman, we merely need to set up a handler for *fish_on* messages. Because messages a broadcast, we embed the *caught_by* field in the message. Thus, if we overhear another fisherman’s *fish_on* we don’t do something a rude and uncouth as to harvest it.

Let’s assume we have a pond object that is responsible for pairing fish with baited hooks.

```
::tao::class pond {
 method time_step {} {
   foreach fisherman [my list_fishermen] {
     if {![my random_criterial]} {
        # No fish caught
       continue
     }
     # Generate the fish
     set species [my random_species]
     set fish [$species -size random]
     # Hook it on the line
     my event_publish fish_on $fisherman
     set msg {}
     dict set msg fish $fish
     dict set msg species $species
     dict set msg size [$fich cget size]
     dict set msg caught_by $fisherman
     my event generate fish_on  $msg
   }
  }
}
```

The pond sets up a publication with an intended target of the fisherman. It then assembles the outbound message as a dict. And finally it broadcasts the message.

Setup, Cleanup and Renaming

When an object changes names, or is destroyed, all of the references it created in the notification sysyem.

To facilitate this, the TAO parser secretly adds a line to the top of every classes destructor which calls the **::tao::object_destroy** procedure. This procedure scrubs the notification system of all subscriptions, publications, and bindings.

When an object is renamed, developers are encouraged to use the **::tao::object_rename** procedure. This prodedure updates the references in the notification table to the new name.

[namespace code {}]

Because objects can change names, developers in TAO are also encouraged to publish calls to events using [namespace code {my thisorthat}] instead of [list [self] thisorthat]. While an object can change names, it never changes namespaces.

Here is a rigged demo of the phenomenon. We have a silly class that can delay gratification through the tcl event loop:

```
tao::class silly {
  method gratification {{how {}}} {
    return “[self] Ahhh $how”
  }
  method delayed_gratification {} {
   after idle [namespace code \
     [list my gratification [self method]]]
  }
  method interrupt_gratification {} {
    after idle [list [self] \
      gratification [self method]]]
  }
}

silly create whosit
whosit gratification
> ::whosit Ahhhh
whosit delayed_gratification ; update
> ::whosit Ahhhh delayed_gratification
whosit interrupt_gratification ; update
> ::whosit Ahhhhh interrupt_gratification
```

As we can see, so long as the object never changes names, everything is fine. Let’s throw the system for a loop though. We will trigger the events, but change the name before we give the event loop a chance to respond:

```
whosit delayed_gratification
whosit interrupt_gratification
rename whosit whatsit
update
> ::whatsit Ahhhh delayed_gratification
> BGERROR: Invalid command “whosit”
```

TAO/Tk

TAO/Tk is a GUI extension to TAO, geared toward the creation and operation of graphical user interfaces in Tk. TAO classes come in three distinct forms:

1. Meta-Classes, intended to be the building blocks for other classes.
2. User Widgets, classes intended to be called directly by the end user and behave like a Tk widget.
3. Dynamic Widgets, a special class of widgets designed for data entry screens. They obey a limited set of configuration critera and conform to a special template.

Meta Classes

In UI design, there is often the need/desire/lazy tendency to lump similar functions together. Very often though, this code re-use is only helpful on a high-level. Low-level widget design realities makes the approach cumbersome at best.

Let’s assume I want to make a widget that response to user commands, and then converts those commands to Tcl calls. I also want a widget to take in user commands, but make them SQL calls instead. We’ll assume we want them both to act as a freestanding toplevel window.

In TAO/Tk, I have that very example implemented. The result is an interaction of 3 classes to produce 2 more classes.

Class	Ancestor
taotk::meta::console	
taotk::meta::sql_console	**taotk::meta::console**
taotk::toplevel	**taotk::meta::widget**
taotk::console	**taotk::meta::console** **taotk::toplevel**
taotk::sqlconsole	**taotk::meta::sql_console** **taotk::toplevel**

So, why don’t I just take the differences between the Tk console and the Sql console and make a direct descendent of taotk::console?

Well a few reasons. The most important is that I often need chunks of the sqlconsole to operate inside of embedded frames. The embedded frame has a different constructor and connective tissue. A frame doesn’t have the concept of a window title, for instance.

Secondly, an interactive console is a generally useful thing. We may have to introduce a console targeting a different SQL backend in the future, which runs a slightly different dialect of SQL. Or perhaps we need a semi-verbal interface for a machine learning project. The possibilities are endless, and the later in the design process that we have to start drawing lines, the better.

And when I start introducing the dynamic widget set, you'll start to see that having a high level architecture apart from your low level architecture is a good thing™.

For coding consistency, taotk meta classes are located in the taotk::meta namespace.

User Widgets

User widget are designed to be readily useable as a Tk-Like entity. User widgets are located in the **::taotk** namespace, and are usable just like any other widget:

```
::taotk::browser .html -title {About:Blank}
```

To make TAO/Tk widgets behave like the Tk commands developers are familiar with, we use the unknown handler built into TclOO.

```
  class_method unknown args {
    set tkpath [lindex $args 0]
    if {[string index $tkpath 0] eq "."} {
      if {[winfo exists $tkpath]} {
        error "Bad path name $tkpath"
      }
      set obj [my new $tkpath \
         {*}[lrange $args 1 end]]
      if {![winfo exists $tkpath]} {
        catch {$obj destroy}
        return {}
      }
      $obj tkalias $tkpath
      return $tkpath
    }
    next {*}$args
  }
```

The **tkalias** method renames the tk object to something in the object's namespace, and then renames the object to take the tk object's place. When the object is destroyed, the native tk object will be destroyed along with it. Even though the tk object's command has been renamed, it still behaves within TK as if it had never been moved.

```
  method tkalias tkname {
    set oldname $tkname
    my variable tkalias
    set tkalias $tkname
    set self [self]
    set nativewidget [::info object \
      namespace $self]::tkwidget
    my graft nativewidget $nativewidget
    rename ::$tkalias $nativewidget
    ::tao::object_rename [self] ::$tkalias
    my bind_widget $tkalias
    return $nativewidget
  }
```

The **bind_widget** method ensures that when Tk destroys the widget, the object's destructor is called.

```
  method bind_widget window {
    my graft topframe $window
    my graft toplevel \
      [winfo toplevel $window]
    bind $window <Destroy> \
     [namespace code {my EventDestroy %W}]
  }
```

The graft statements allow your code to call out the tk objects as though the were methods:

```
  method change_title {} {
    set tl [my organ toplevel]
    wm title $tl {Changed the title}
    my <nativewidget> configure -bg green
  }
```

The **EventDestroy** method is a sanity check. When <Destroy> goes of, it is possible for a parent to see the <Destroy> event for it's children. Also, there are times where the only notification that goes out is for the toplevel window the object belongs to. It took me a bit of trial and error before I finally got this part right:

```
  method EventDestroy window {
    if { [string match "${window}*" $w] } {
      my destroy
    }
  }
```

Conversely, our destructor needs to destroy the tk object when called. But because developer may have their own destructor logic, the smarts for this process have been packed into a private method that destructor can call.

```
 destructor {
    my Widget_destructor
  }

  method unbind_widget window {
    my variable tkalias
    if {[winfo exists $window]} {
      bind $window <Destroy> {}
    }
    set tkalias {}
  }

  method Widget_destructor {} {
    my variable tkalias
    set alias $tkalias
    if {$alias ne {}} {
      my unbind_widget $alias
    }
    catch {my action destroy}
    # Destroy an alias we may have created
    if { $alias ne {} && \
      [winfo exists $alias] } {
       catch {
   rename [namespace current]::tkwidget {}
       }
    } else {
      catch {
      ::destroy [my organ nativewidget]}
      }
     }
  }
```

Dynamic Widgets

Dynamic Widgets are designed for producing automated data entry screens. They are designed to obey a limited set of commands, and fit into a relatively rigid template

1. Every element to be tracks is a field in a global array
2. For every element, a key/value list of metadata is provided.
3. The "widget" is a self contained frame.

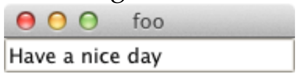
The syntax boils down to:

```
taotk::dynamic_widget tkpath fieldname \
   arrayname properties
```

Let's see an example:

```
toplevel .foo
array set ::record {
   message {Have nice day}
}
taotk::dynamic_widget .foo.bar \
 message ::record {}
grid .foo.bar
```

And we get back:

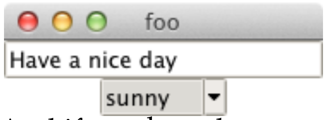

foo

Have a nice day

If we just alter the description, we get different behavior.

```
array set ::record {
   weather sunny
}
taotk::dynamic_widget .foo.baz weather \
   ::record {
widget select
values {cloudy sunny rainy foggy snowy}
}
grid .foo.baz
```

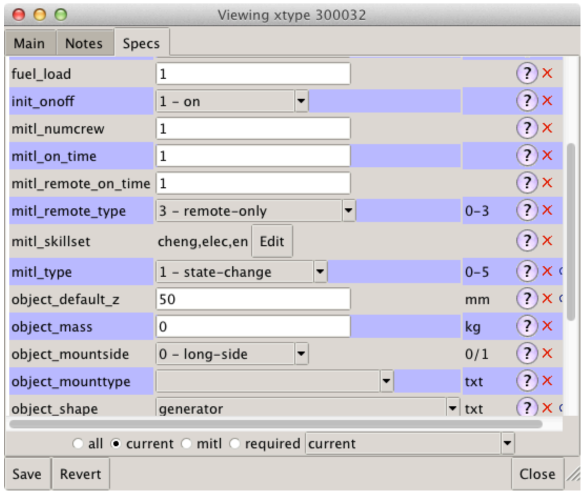
We now see:


foo

Have a nice day

sunny

And if you have been around me, soon enough you'll be generating screens that look like this:


Viewing xtype 300032

Main | Notes | Specs

fuel_load	1		
init_onoff	1 – on		
mitl_numcrew	1		
mitl_on_time	1		
mitl_remote_on_time	1		
mitl_remote_type	3 – remote-only	0–3	
mitl_skillset	cheng,elec,en	Edit	
mitl_type	1 – state-change	0–5	
object_default_z	50	mm	
object_mass	0	kg	
object_mountside	0 – long-side	0/1	
object_mounttype		txt	
object_shape	generator	txt	

all [✓ current] mitl required current

Save Revert Close

In our simulator, we have hundreds of "specs" that can be used to describe a piece of equipment, a room on a ship, a doorway, even crew members. We also have dozens of controls that we present to the user to drive the simulator. And even 50 or so visual preferences.

They are all described succinctly with a key/value list that is either stored in a parent object as an option, or in a database.

Property Inferences

The first step to making a Dynamic Widget is to read through the description. In the absence of any other information, the dynamic system will assume the field is a text string, represented by an entry box.

If the user has specified a "widget" property, that is which widget will be used.

Otherwise, the widget has to be inferred. With no widget property, it tries to guess by the presence of a “values” field. If “values” is present, the widget is then assumed to be a selection. With no “widget” or “values” property, the inferencer then looks for a field name “type” or “storage”.

Here are the basic dynamic widgets implemented by taotk:

boolean	A checkbox to handle the simple cases of 1 and 0
checkbutton	A checkbutton with stylized properties for controlling on/off values
color	Presents the user with a label previewing the current value and a button to activate the Tk color chooser
entry	An entry box. If the “read-only” property is set to true, reverts to a label.
filename	Presents the user with an entry field for a value as well as a button to launch the Tk file chooser
font	Presents the user with a label previewing the current value and a button to activate the Tk font chooser
label	A label
real	An entrybox, but intended for numerical values
scale	Presents the user with an entry box coupled with a scale slider.
script	A button that, when pressed, presents the user with a popup window with a text widget.
vector	Breaks the entry into pieces and presents an entry box for each component.

All of the dynamic widgets live in the ::taotk::dynamic namespace.

Appendix

TAO Parser Keywords

Keyword: option

```
Syntax:
 option fieldname keyvaluelist
```

The **option** keyword defines an option that will be conferred to all object of this class, and passed on to descendents of this class. It is equivilent to:

```
property fieldname option keyvaluelist
```

Keyword: option_class

```
Syntax:
 option_class fieldname keyvaluelist
```

The **option_class** keyword defines a set of attributes that can be inherited by other options.

This statement is equivalent to:

```
property fieldname option_class \
 keyvaluelist
```

Keyword: property

```
Syntax:
 property fieldname ?type? description
```

The **property** keyword defines a property that will be conferred to all objects of this class, and passed on to descendents of this class.

Keyword: variable

```
Syntax:
 variable fieldname defaultvalue
```

The **variable** keyword defines an variable that will be conferred to all object of this class, and passed on to descendents of this class. The difference between the TAO form of the keyword and the standard TclOO usage is that TAO guarantees the variable will be initialized with the default value within the **InitializePublic** method, which is normally called by the constructor.

This statement is equivilent to:

```
property fieldname variable default
```

Keyword: class_method

```
Syntax:
  class_method name arglist body
```

The **class_method** keyword defines a method for the class itself. If no **method** of the same name is defined of the class, the implementation will be copied to the objects of the class.

class_method is equivalent to declaring a local method for the class object via:

```
oo::objdefine class method arglist body
```

A method defined by **class_method** will be inherited by descendents of a class, whereas the above example would not.

The Mother of all Classes

Static Methods

The static methods of the **moac** are methods declared in tao/moac.tcl file, using the conventional manner. They can be superseded and/or replaced by descendents.

Method: cget

```
Syntax:
  cget field ?default?
```

Returns the current value for option *field*. If "default" is given as the second argument, return the default value for option *field*.

Method: configure

```
Syntax:
  configure field
  configure field value ?field value…?
```

If one value given, return the current value for *field*. If two or more arguments given, write new values to options.

Method: configurelist

```
Syntax:
   configurelist {field value field value…}
```

Write new values to options.

Method: event cancel

```
Syntax:
  event cancel handle
```

Cancel any timer events created by **event schedule**. **handle** can be of any form acceptable to [string match].

Method: event generate

```
Syntax:
  event generate event args…
```

Generates an event to be published to other objects. Args are intended to be a key/value list describing the event.

Method: event publish

```
Syntax:
  event publish who event
```

Add a notification subscription for events matching **event** to recipients matching **who**. Both **event** and **who** can be of any form acceptable to [string match].

event schedule handle interval script

Schedule for the Tcl event loop to run *script* on the object's behalf after *interval*. Any input valid for [after] is acceptable for *interval*.

event subscribe *who event*

Add a notification subscription for events matching **event** to senders matching **who**. Both **event** and **who** can be of any form acceptable to [string match].

event unpublish ?*event?*

Remove all subscribers for this object's notifications matching pattern **event**. **event** can be of any form acceptable to [string match]. If no event is given, all publications for this object are removed.

event unsubscribe ?*event?*

Remove all subscriptions for this object to notifications matching pattern **event**. **event** can be of any form acceptable to [string match]. If no event is given, all subscriptions for this object are removed. (NOTE: It is still possible for the object to still receive notifications if the object matches another object's **publication**.)

graft stub object

Create a link to another object as a forwarded method. Two methods are created: $**stub** and <$**stub**>.

InitializePublic

Designed to be the first method called by the constructor. This method reads the properties of the object an ensures the default value is loaded for all declared variables and options.

initialize

Designed to be called by the constructor after the object has initialized all of its variables. The default implementation is empty, it is reserved for developers to perform any higher level initialization that an object may require within the constructor.

morph newclass

Convert this object to be of class *newclass*.

organ all

Return a key/value list of all stubs for this object and what objects they point to.

organ *stub*

Return the object directed to by *stub*.

private method args…

Exercise a private method.

signal signal ?signal…?

Register a signal to be executed during the next *Signal_pipeline*. An "idle" signal will trigger an [after idle] call to **Signal_pipeline.**

Signal_pipeline

Develop an execute a pipeline based on all signal received since the last call to **Signal_pipeline**.

Method Ensembles

action actionname ?args…?

Every submethod is a response to an "event". Actions are expected to be immediate. Developers can feel free to define their own events.

action pipeline_busy

Commands to run at the start of **Signal_pipeline**, but before the pipeline begins.

action pipeline_idle

Commands to run at completion (or failure) of **Signal_pipeline**.

lock active

Return a list of all locks currently active on this object.

lock create lock ?lock…?

Create one or more locks on the object. Returns zero if the all of the locks specified are new. Returns 1 if one or more of the locks was already active.

lock peek lock ?lock…?

Returns 1 if one or more of the locks specified is active. Returns 0 otherwise.

lock remove lock ?lock…?

Remove one or more locks on the object. Returns zero if other locks are still present on the object. Returns 1, and calls **lock remove_all** if there are no more locks on the object.

lock remove_all

Remove all locks on the object and call the **Signal_pipeline** method.

notify event ?args…?

This ensemble is present to allow object to respond to notifications. The only defined notification is *default*, which quietly ignores any event that wasn't already processed.

Option_set option newvalue

Each submethod is the name of an option. The default handler mirrors any option set with an existing internal variable.

SubObject *stub*

Return the name of an object to create for a particular stub. The default hander returns [namespace current]::Subobject_generic_*$stub*

Class Methods

property property

Returns the value of constant property for the class. Note: This version of the property method performs a database query to the TAO db backend. It does not reflect properties inherited from ancestors.

property type property

Returns the value of the *property* of type type. Note: This version of the property method performs a database query to the TAO db backend. It does not reflect properties inherited from ancestors.

Dynamic Methods

Dynamic methods are generated by the TAO parser. They are custom produced for each class, and replaced with the next call to **tao::class**.

Method: property *field*

Returns the value of property *field*. If multiple types for *field* are given, the line of succession is as follows: signal, option, variable, subst, eval, const.

i.e. if a class has a signal named *foo* and a constant named *foo* the data returned from **property** will be the description for the signal.

Property defines the following virtual fields:

Virutal	Description
type	Returns the
list_*type*	Return a list of properties defined for type *type.*
list	Returns a list of all properties.
signal_order	Returns a list with the computed order in which the signal pipeline will be evaluated.
signals	Returns a dict describing all signals defined for this class.
options Or option_dict	Returns a dict describing all options defined for this class.
publicvars	Returns a list of all variables tracked by the property system.

Method: property type field

This second form or property takes in the name of a property type, as well as a field. Each property type has two dynamic fields:

Virutal	Description
list	Returns a list of all properties of type *type*
dict	Returns a dict with the complete defintions of all properties of type *type*

TAO DB Schema

Table: ancestry

TAO has to independently track the chain of heredity for classes. It does this by replicating the rules TclOO uses, and then recording the results as a sequence of ancestors from the most advanced, to the most primitive.

```
create table ancestry (
  class string references class,
  ancorder integer,
  parent string references class,
  primary key (class,ancorder)
);
```

Table: class

Each class that has been processed by the TAO parser has an entry. ***name*** is the name of the class. ***package*** is for future expansion. ***regen*** is set to true when an ancestor of the class is modified, and the dynamic methods for the class need to be regenerated.

```
create table class (
  name string primary key,
  package string,
  regen integer default 0
);
```

Table: class_alias

As projects grow and evolve, the names of classes can change over time. The ***class_alias*** table is consulted in cases where a new class tries to refer to an ancestor whose name has changed.

```
create table class_alias (
  cname string references class,
  alias string references class
);
```

Table: ensemble

TAO captures information about method ensembles in the ***ensemble*** table.

```
create table ensemble (
  class string references class,
  method string,
  submethod string,
  arglist string,
  defined string references class,
  body text,
  primary key (class,method,submethod)
  on conflict replace);
```

Table: method

TAO captures information about methods in the ***method*** table.

```
 create table method (
   class string references class,
    method string,
   arglist string,
   body text,
   defined string references class,
   primary key (class,method)
   on conflict replace);
```

Table: object

Each object that has been spawned by a tao class has an entry. ***name*** is the name of the class. ***package*** is for future expansion. ***regen*** is set to true when the class is modified, and the dynamic methods for the object need to be regenerated.

```
create table object (
  name string primary key,
  package string,
  regen integer default 0
);
```

Table: object_alias

Objects can occasionally change names throughout the course of the program. This has an entry for all former names an object may have possessed.

```
create table object_alias (
  cname string references class,
  alias string references class
);
```

Table: object_bind

TAO has an independent event handling system, The **object_bind** table is where an object designates which script to call when an event is triggered.

```
create table object_bind (
  object string references object,
  event  string,
  script blob,
  primary key (object,event) on conflict
replace
);
```

Table: object_schedule

TAO has an independent event handling system, The **object_schedule** table is where an object can schedule an event to occur in the future.

```
 create table object_schedule (
   object string references object,
   event  string,
   time   integer,
   eventorder  integer default 0,
   script string,
   primary key (object,event) on conflict
replace
 );
```

Table: object_subscribers

TAO has an independent event handling system, The **object_subscribers** table is where an object can subscribe which events from which objects it wishes to respond to. Note: ***sender***, ***receiver*** and ***event*** are matched using the same rules as [string match].

```
create table object_subscribers (
  sender   string references object,
  receiver string references object,
  event string,
  primary key (sender,receiver,event) on
conflict ignore
);
```

To make the "example" object listen to all events from "appmain":

```
insert into object_subscribers (
sender,receiver,event
) VALUES (
'appmain','example','*'
);
```

Table: property

TAO stores the input given by the parser's option, property, and signal keywords as records in the ***property*** table.

```
create table property (
  class string references class,
  property string,
  defined string references class,
  type string,
  dict keyvaluelist,
  primary key (class,property,type) on
conflict replace
);
```

Table: typemethod

TAO captures information about class method classes in the ***typemethod*** table.

```
 create table typemethod (
   class string references class,
    method string,
   arglist string,
   body text,
   defined string references class,
   primary key (class,method) on conflict
replace
 );
```

Tao/Tk Meta Classes

Class: taotk::meta::widget

The base class for all Tao/Tk objects.