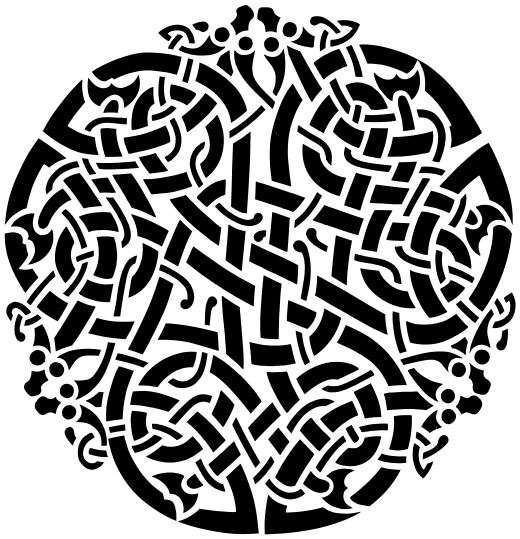


TAO/TK

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.

Interoperability with TclOO

Behind the scenes, TAO classes are really TclOO classes. They just have a few extra methods, and additional introspection via the TAO internal database. It is perfectly reasonable to have a TAO object list a TclOO object as a superclass, and *vice versa*.

```
oo::class create foo {
}
::tao::class bar {
  superclass foo
  method someop args {
    return some
  }
}
oo::class create baz {
  superclass bar
}
```

All of the TclOO introspection tools are usable on TAO objects and classes:

```
info class definition bar someop
> args {
>   return some
>}
```

In short, TAO is intended to be an additive extension to TclOO. It does not break or otherwise force the user to alter his/her way of doing things if they are already familiar with 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 an in-memory sqlite database. Wholly new keywords string together a series of TclOO calls and sql queries to produce TAO code.

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
  }
}
```

Unlike in TclOO, the command tao::class is not actually an object. It's a procedure that implements the parser. If you didn't know already, in TclOO the very keyword used to create a class is itself a class, thus why you have to give it a method to work on. (Either create or new.)

Operation of the Tao Parser

The TAO parser operates in 3 main phases:

Parse the incoming class

Code submitted to the **tao::class** command is passed to the **::tao::parser** namespace for digestion. Each keyword is read, and a picture of the new class is developed. At the same time, the raw TclOO form of the new class is being built using **oo::define**.

Apply dynamically generated methods.

Once the picture of the new class is developed, the parser adds dynamically generated methods. These methods include the **property** method and method ensembles.

Update affected classes

After the class is parsed, built, and the dynamic methods area applied, the parser looks to see if the new definition of the class will affect any already existing decedents of this class. It then regenerates the dynamic methods for all of those descendents.

Properties

```
::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 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.

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.

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

Inheritance

The inheritance mechanism for TAO mimics the behavior of inheritend in TclOO. Under TclOO, when searching for a method implementation, the last method defined is the method that is used. Because the parser runs outside of TclOO, TAO must recreate that process for database queries.

The routine essentially calls [**info class superclasses**] on the subject of inquiry. Because **[info class ancestors**] returns only the immediate heritage specified by the **superclass** keyword, TAO must repeat the process on every ancestor, and ancestor's ancestor.

Every class we have scanned is added to our result, but only the first time it is encountered. If an ancestor is inherited through multiple paths, subsequent references it is ignored.

The result is a list starting from the most advanced, and stepping back to the most primitive of every class that has had an impact on the present class.

The implementation is here:

```
proc ::tao::class_ancestors {
 class {stackvar {}}
} {
  if { $stackvar ne {} } {
    upvar 1 $stackvar stack
  } else {
    set stack {}
  }
  if { $class in $stack } {
    return {}
  }
  stack push stack $class
  if {![catch {
     ::info class superclasses $class
  } ancestors]} {
    foreach ancestor $ancestors {
      class_ancestors $ancestor stack
    }
  }
  if {![catch {
   ::info class mixins $class
  } ancestors]} {
    foreach ancestor $ancestors {
      class_ancestors $ancestor stack
    }
  }
  return $stack
}
```

TAO performs a database query on every property defined for each ancestor, in the order given by **tao::class_ancestors**. A new property is added to a data structure for the class we are parsing if no such property was defined before:

```
proc ::tao::dynamic_methods_property {
 class ancestors
} {
 set info {}
 foreach ancestor $ancestors {
   ::tao::db eval {
SELECT property,type,dict FROM property
WHERE class=:ancestor and defined=:ancestor
} {
   if {
     [dict exists $info $type $property]
   } continue
   dict set info $type $property $dict
  } ; # End of query
 } ; # End of foreach
 # At this point $info contains the complete
 # picture
```

Property Types

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
```

A complete table of supported types is as follows:

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 Signals.

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 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 submethods are inherited by 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"}
 }
}
```

Because Method Ensembles are dynamically generated, a method ensemble will trump a normal method. Any attempt to implement "has" as a bare method in a 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"
```

Method Ensemble Implementation

Ensemble submethods are tracked and catalogued by the TAO database. After the class is parsed, TAO builds a series of dynamically generated methods. Method ensembles are built during this stage.

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 that defined one.

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 listing above, you well see a call to **tao::dynamic_arguments**. **tao::dynamic_arguments** is a routine that reads the arguments beyond the method name given to the ensemble, and converts them into the local variables. The intent is to mimic the argument handling behavior of the **proc** command. If too many arguments, or not enough arguments are given, **tao::dynamic_arguments** will throw an error.

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 to *dictargs*, that argument is assumed to be a key/value list.

The following example demonstrates the various ways Method Ensembles will accept input via *dictargs*. The **::character::bio** method parrots back the input it is given.

```
::tao::class show {
 method character::bio {who dictargs} {
   if {[dict exists $dictargs phrase]} {
    set phrase [dict get $dictargs phrase]
   } else {
    set phrase {}
   }
   puts [list $subject says $phrase]
 }
}
# Not enough arguments
show create addams
addams character bio
> ERROR: Usage: who ?dictargs?

# Arguments given after the mandatory.
addams character bio fester \
  phrase {Light Bulb}
> fester says {Light Bulb}

# Arguments packed into a key/value list as
# the first args
addams character bio lurch \
  {phrase {Good Evening}}
> lurch says {Good Evening}

# And it even does dashes!
adama character bio mortisha \
  -subject {Mon Cher}
> mortisha says {Mon Cher}
```

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.

A modified version of the ***property*** method is included with all TAO classes. 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 if a class method 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:

```
# Example
taotk::frame .foo
```

```
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 also work if we did it such:

```
oo::class create Frame {
}
oo::define Frame method unknown args {
  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
    }
  }
}
```

The idea being that with either case, the class behaves like a Tk command:

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

The difference comes in when we assume this behavior is inherited by descendents:

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

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

Like properties and method ensembles, Class Methods are tracked in the database and applied with the other dynamically generated methods.

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 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. In the example above, 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, and the regenerate flag is rippled out to all descendents of descendents, and so on. When that is complete, the parser re-creates all of the dynamic methods for all classes with the **regenerate** flag. And the process of regenerating the dynamic methods marks each class as **regenerate=0** once more.

Modifications to a class are seemingly instant as far as the Tcl environment is concerned. If we add a property of *has_teeth* to carnivores:

```
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
```

Thomas, our **housecat** several examples, now has teeth:

```
Thomas property has_teeth
> 1
```

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

The Variable keyword

TAO also provides a keyword **variable** that works differently than the **variable** keyword in TclOO. In TclOO, **variable** allows a variable to magically appear in every method of that class (but not descendents.) In TAO **variable** declares a variable and sets it's default. That declaration and default will carry through to descendents. To access that variable, one still needs to use the **my variable** command in methods.

```
tao::class animal {
  variable hungry 0
  method is::hungry {} {
    my variable hungry
    return $hungry
  }
}
Thomas is hungry
> 0
```

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. A complete reference is located in the Appendix.

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 variables declared in the class definition as well as provides default values for all options. The next step is to run **configurelist** on the options fed 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, and configurations applied without having to recreate all of the steps in their own constructor or figure out the right incantations of **[next]** to call an ancestral constructor.

Constructor Option Syntax

The **::tao::args_to_options** procedure 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

Option Handling

Options are tracked as a special kind of property. The "value" for Options are specified in Dict format, because we need to track a lot more than a constant value. For convenience, the **option** keyword was added to the TAO parser as a shortcut.

```
::tao::class housecat {
  superclass felis
  property gender option {
    widget select storage string default {}
    values {{} male female}
  }
  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 Thomasina \
  -gender female -weight 8
```

And we can modify an existing object via the **configure** command:

```
housecat configure Thomas \
  -gender male -weight 10
```

Because in large systems one may need to perform a dump of information from a database or some other source, TAO is pretty flexible about the format of options.

```
housecat create Thomas {
  gender male weight 10
}
housecat create Thomasina \
  gender female weight 8
# After a year of snacking
Thomas configure weight 11
# Showing off
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 lieu of storing the value internally: See: `Option Substitution`.
`set-command`	Script to set the value externally in lieu 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)

```
::tao::class housecat {
  option favorite_food {
    set-command {
 puts {%self%'s favorite %field% is %value%}
    }
  }
}
Thomas configure favorite_food tuna
> ::Thomas's favorite food is tuna
```

Option Classes

Developers often find that options follow a certain template. TAO provides for option templates with the **class** property of options. **class** can specify the name of another option. To create an option which is simply a prototype, use the **option_class** keyword.

```
::tao::class animal {
  option_class variable {
    default {}
    set-command {
   my variable %field% ; set %field% %value%
   puts "%field% is now %value%"
     }
     get-command {
   my variable %field% ;return [set %field%]
    }
  }
}
tao::class housecat {
  option color {
    class variable default black
  }
  method color {} {
    my variable color ; return $color
  }
}
Thomas configure color orange
> color is now orange
Thomas color
> orange
```

Option Event Processing

To keep the outcome of events tied to modifying 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***.

For calls to the configure command, but not during the constructor, the **Option_set** ensemble is called for all incoming values.

If we want an event to fire off after we set the weight for instance:

```
::tao::class housecat {
   method Option_set::weight newvalue {
     if {$newvalue > 10 } {
       puts "This cat is fat!"
     }
   }
}
Thomas configure -weight 12
> This cat is fat!
```

By default, normal descendents of **moac** will accept unknown options. Descendents of **::taotk::meta::widget** will throw an error on unknown options. The developer can control the behavior of their class with the **options_strict** property.

```
::tao::class housecat {
  property options_strict 1
}
Thomas configure unknownoption 10
> Error unknown option -unknownoption.
Valid: gender weight color favorite_food
::tao::class housecat {
  property options_strict 0
}
Thomas configure unknownoption 10
# No error, and we can retrieve the value
Thomas cget unknownoption
> 10
```

Forwarding and Grafting

The **moac** also provides a set of functions to manage forwarding methods. The simplest is **forward**. Forward is just a simple call to **::oo::objectdefine forward**.

```
Thomas forward puts ::puts
Thomas puts "Hello World"
> Hello World
```

Often times, though, we need to recall what it is we are forwarding to. While TclOO will tell you that a method is a forward vs. any other type of method, it won't tell you where the call is going.

TAO provides the **graft** method, which forwards a command while at the same time recording where it's going. The destination can be retrieved later via the **organ** method.

```
tao::class supertextbox {
  constructor {tkpath} {
     toplevel $tkpath
     my graft toplevel $tkpath
     text $tkpath.text
     my graft text $tkpath.text
  }
  method title newtitle {
    set tl [my organ toplevel]
    wm $tl title $newtitle
  }
  method replacetext {newtext} {
    my <text> delete 0.0 end
    my <text> insert end $newtext
  }
}
```

In the above example, we graft both the toplevel window and the text widget. In the **newtitle** method, we need to pass the path to the window to **wm**. we use the **organ** method to retrieve the value for *toplevel*. In **replacetext** we need to manipulate the text widget. We use the grafted name to address

the text widget as though it were one of our own methods.

The **replacetext** method could also have been written:

```
tao::class supertextbox {
  method replacetext {newtext} {
    my text delete 0.0 end
    my text insert end $newtext
  }
}
```

Graft creates two forwarded methods, the original name, and the <original name>. I find marking which methods are going out to a forward somewhat useful. However, we also need the plaintext version because TclOO will not allow a method to be accessed from the public interface if it does not start with a lower case letter.

Locks, Signals, and Notifications

The **moac** defines several methods, and parser keywords to manage locks, signals, and notifications.

Locks

Locks were designed as a guard against an object recursively calling pipeline routines. **lock create** can behave like the classic "up" operator in a semaphore. When you call the lock the first time, the operation returns false. If the lock was already present, it returns true.

```
::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 objects 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.

Internally, locks are simply a list stored in a variable *ActiveLocks*. The **lock create** method ensures that only one copy of a lock is present in the list at any given time.

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 during this stage in the pipeline.
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, as well as the order in which they need to be executed. It then executes that plan before returning.

Notifications

Notifications are a message passing system for objects. They are akin to Tk bindings. Objects can both emit and receive notifications.

In the case of emitting an event, it simply passes the message to the TAO message handler. 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** ensemble 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 dictargs} {
   set caught [dict get $dictargs caught_by]
   if { $caught ne [self] } continue
   set fish [dict get $dictargs fish]
   my action catch_fish $fish
 }
}
```

For the fisherman, we need to set up a handler for **fish_on** messages. Because messages are 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 us 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 catch_criterial $fisherman]} {
        # No fish caught
      continue
     }
     # Generate the fish
     set species [my random_species]
     set fish [$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

Object names are tracked by the TAO message handling system. As such, the system needs to be notified if an object is destroyed or renamed.

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 {}]

TAO developers need to take care when binding objects for events. Objects can, and do, change names.

Let's suppose we are binding a command to a button. In pure Tcl, this is easy, we tell the system to call back at a given command name:

```
button .foo -command some_command
```

With namespaces, we need to be a little more elaborate.

```
namespace eval ::foo {
   button .foo -command ::foo::some_command
}
```

Inside of a class method, if the object wants to exercises one of its own methods, it needs to provide a path. The technique that immediately comes to mind for most developers is the **[self]** command.

```
oo::class define myobject {
 method makebutton {} {
   button .foo -command \
    [list [self] some_command]
 }
}
```

I am here to tell you there is a better way. Many of you may not be familiar with the **namespace code** command. **namespace code** captures the environment in which it was called, and provides a globalized return path.

```
oo::class define myobject {
 method makebutton {} {
   button .foo -command \
    [namespace code {my some_command}]
 }
}
```

While an object can change names, it will never change namespaces. (Besides, this is TAO, and the name that can be stored isn't the real name anyway ;-)

Here is a rigged demo of the phenomenon. We have a silly class that can delay calling one of its methods through the Tcl event loop. We will use two different approaches to mapping the global path back to our intended call.

- The **delayed_nspace** method takes in a method as an argument and schedules a callback to that method using **namespace code**.
- The **delayed_self** method takes in a method as an argument and schedules a callback to that method using **self**.

They both also advertise which method sent the event via the **[self method]** mechanism, so we can track what is going on.

```
tao::class silly {
  method gratification {{how {}}} {
    return "[self] Ahhh $how"
  }

  method delayed_nspace m {
   after idle \
 [namespace code [list my $m [self method]]]
  }

  method delayed_self m {
    after idle \
      [list [self] $m [self method]]]
  }
}
```

For normal purposes, both techniques work the same way.

```
silly create whosit
whosit gratification
> ::whosit Ahhhh
whosit delayed_nspace gratification ; update
> ::whosit Ahhhh delayed_nspace
whosit delayed_self gratification ; update
> ::whosit Ahhhhh delayed_self
```

Now we will trigger the events, but change the name before we give the event loop a chance to respond.

```
whosit delayed_nspace gratification
whosit delayed_self gratification
rename whosit whatsit
update
> ::whatsit Ahhhh delayed_nspace
> BGERROR: Invalid command "whosit"
```

namespace code has other advantages. Unlike calls to **[self]**, the event can call private methods.

```
tao::class silly {
  method Gratification {{how {}}} {
    return "[self] AHHH $how"
  }
}

whatsit delayed_nspace Gratification
update
> ::whatsit Ahhhh delayed_nspace
whatsit delayed_self Gratification
update
> BGERROR: unknown method Gratification must
be…
```

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.

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. Meta classes are method and properties that are inherited by other classes, but which don't make a complete product in their own right.

For coding consistency, TAO/TK meta classes are located in the **taotk::meta** namespace.

User Widgets

User Widgets are designed to be readily useable as a Tk-Like command. Borrowing from the tradition started by tile, TAO/TK widgets are located in their won namespace, **::taotk.** They operate 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 **EventDestroy** method is a sanity check. When <Destroy> goes off, 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 to a child window is that the toplevel window to was destroyef. It took a bit of trial and error to get 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 the developer may have his/her own destructor logic, the smarts for this process have been packed into a private method. It is the destructor's job to call that private method at some point.

Before we destroy the native Tk widget, we use the **unbind_widget** method to remove our bindings to prevent the *<Destroy>* binding for the Tk object from calling the destructor yet again.

```
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 tracked is an field in a global array
2. For every element, a key/value list of metadata is provided to the constructor.
3. The path specified will become a frame containing the tk objects that implement the UI representation of the element.

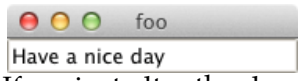
The syntax boils down to:

```
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 and entry box (the default if we have no other data):



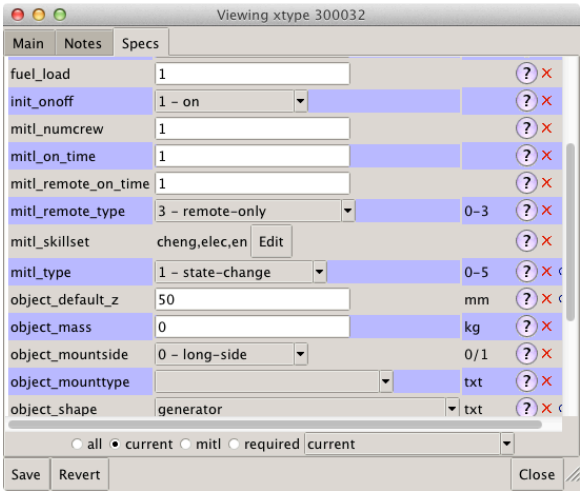
If we just alter the description, we get different behavior.

```
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 a combobox:

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



On the project I work with, 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.

Needless to say, figuring out which properties go on a given screen is hard enough without having to generate the GUI.

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 TAO/TK:

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.

Building Custom Dynamic Widgets

To register a new class of dynamic widget, place it in the ::taotk::dynamic namespace. The name you give it in that namespace is the name you reference in the **widget** property of the object description.

The constructor for dynamic widgets is as follows:

```
constructor {
  window fieldname arrayname args
} {
    my InitializePublic
    my configurelist \
      [::tao::args_to_options {*}$args]
    my variable field arrayvar
    set field $fieldname
    set arrayvar $arrayname

    my graft mainframe $window
    my graft nativewidget $window
    my BuildDynamicMethods
    my Build_topframe $window
    my build_widget $window
    my bind_widget $window
}
```

The **BuildDynamicMethods** method adds hooks to TAO/TK's style sheet handler. The **build_topframe** method builds the frame at *window*. The **build_widget** method builds the UI for this element. The **bind_widget** ensures the destruction of the Tk path calls the destructor for the object.

For every dynamic widget developed thus far, I've only had to modify the **build_widget** method.

Implementing an entry box is simplicity itself:

```
tao::class taotk::dynamic::entry {
 superclass taotk::dynamic
 option width {default 0}

 method build_widget window {
  set readonly [my cget readonly]
  set varname [my GlobalVariableName]
  set opts [list \
     -textvariable ${varname} \
     -width [my cget width] \
     -style [my Style label] \
  ]
  if { $readonly } {
    ::ttk::label $window.native {*}$opts \
      -takefocus 1
  } else {
   ::ttk::entry $window.native {*}$opts
  }
  my graft nativewidget $window.native
  grid $window.native -sticky news
 }
}
```

Conclusion

I've gone over quite a bit of material in this paper. In fact, there is quite a bit more on the cutting room floor, including sample projects, and demonstrations. That material as well as the complete sources for TAO/TK and all of its supporting libraries and documentation will be copied to the USB sticks that will be distributed by the conference. I will also be publishing the material online at:

http://www.etoyoc.com/tao2.0

Acknowledgements

This paper would not be possible if it weren't for the work of Donal Fellows to bring TclOO to life, convince the community to use it, and explain its inner workings to me patiently over several years.

I would also like to thank my employer, T&E Solutions, for allowing me to share this research with the community, plus the time to prepare this material and present it at the conference.

I would like to thank my wife, Ginger, for being a code-widow for the past few weeks while I put this paper together.

I would also like to thank Victoria Andrews for proofreading early drafts of this paper.

The random squiggles that I may or may not have completely eliminated from this manuscript are early works of my son, Xavier. Never too early to start them coding, I guess. I just wish he wouldn't have picked Tcl instead of Perl for his first project.

The graphic on the front cover was taken from Mallory Pearce's "Ready-To-Use Celtic Designs" published by Dover Clip Art.

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:
  OBJ 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:
  OBJ configure field
  OBJ 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:
   OBJ configurelist {field value …}
```

Write new values to options.

Method: event cancel

```
Syntax:
  OBJ 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:
  OBJ 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:
  OBJ 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].

Method: event schedule

```
Syntax:
  OBJ 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*.

Method: event subscribe

```
Syntax:
  OBJ 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].

Method: event unpublish

```
Syntax:
  OBJ 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.

Method: event unsubscribe

```
Syntax:
  OBJ 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**.)

Method: graft

```
Syntax:
  OBJ graft stub object
```

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

Method: InitializePublic

```
Syntax:
  my 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.

Method: initialize

```
Syntax:
  OBJ 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.

Method: morph

```
Syntax:
  OBJ morph newclass
```

Convert this object to be of class *newclass*.

Method: organ

```
Syntax:
  OBJ organ all
  OBJ organ stub
```

If the argument **all** is given, return a key/value list of all stubs for this object and what objects they point to. Otherwise, return the object directed to by *stub*.

Method: private

```
Syntax:
  OBJ private method args…
```

Exercise a private method.

Method: signal

```
Syntax:
  OBJ 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** if not locks are active.

Method: Signal_pipeline

```
Syntax:
  my Signal_pipeline
```

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

Method Ensembles

Ensemble: 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.

Method: action::pipeline_busy

```
Syntax:
  OBJ action pipeline_busy
```

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

Method action::pipeline_idle

```
Syntax:
  OBJ action pipeline_idle
```

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

Ensemble: lock

The **lock** ensemble manages object locks.

Method: lock::active

```
Syntax:
  OBJ lock active
```

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

Method: lock::create

```
Syntax:
  OBJ 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.

Method lock::peek

```
Syntax:
  OBJ lock peek lock ?lock…?
```

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

Method: lock::remove

```
Syntax:
  OBJ 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.

Method: lock::remove_all

```
Syntax:
  OBJ lock remove_all
```

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

Ensemble: notify event dictargs

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.

Ensemble: Option_set option newvalue

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

Ensemble: SubObject *stub*

Return the name of an object to create for a particular stub. The default hander returns:

```
[namespace current]::Subobject_generic_$stub
```

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

```
Syntax:
  OBJ property property
  OBJ property type property
  OBJ property type <list>
  OBJ property type <dict>
```

This method has several functions depending on the content and number of arguments. A modified form is also created for the class itself, with a subset of all properties that do not depend on the state of an object instance.

```
Usage:  OBJ property property
```

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.

```
Usage: OBJ property type property
```

Return the value of property *property* of type *type*.

```
Usage:  OBJ property type <list>
```

Return a list of all properties of type *type*.

```
Usage:  OBJ property type <dict>
```

Return a dict 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 in this table. The ***name*** field is the name of the class. The ***package*** field is for future expansion. The ***regen*** field is set to true when an ancestor of the class is modified. See Class Regeneration.

```
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 table 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
 );
```