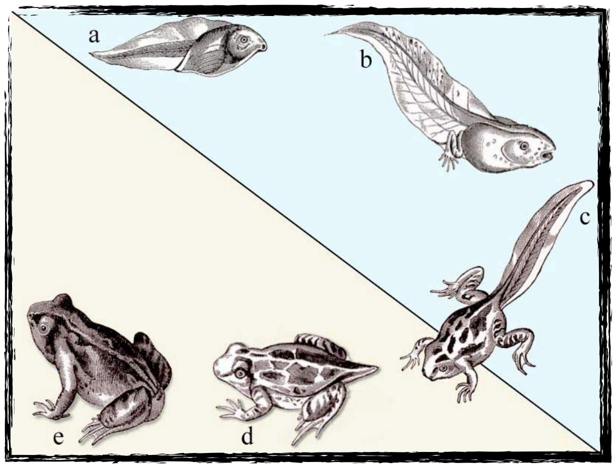

Lifecycle Object Generators (LOG)

Presented to the 19th Annual Tcl Developer's Conference (Tcl'2012)
Chicago, IL
November 12-14, 2012

Sean Deely Woods
Senior Developer
Test and Evaluation Solutions, LLC
400 Holiday Court
Suite 204
Warrenton, VA 22185
Email: yoda@etoyoc.com
Website: http://www.etoyoc.com

Abstract:

This paper describes a design concept call "Lifecycle Object Generators", or LOG for short. It involves a combination of coroutines, TclOO, and basic data structures to create objects that can readily transition from one class to another throughout the course of an application. This paper will describe the basic mechanisms required, and how this architecture can be applied to any complex problem from GUI design to Artificial Intelligence.

This paper is based on experience developing the Integrated Recovery Model for T&E Solutions.

Introduction

Most interesting computer models try to describe the actions and interactions of living, or at the very least animate, things. (The study of most dead an inanimate objects requiring a bit less computer power.) Living things have a tendency to change behavior. Until now modeling that change in behavior has required keeping track of state as variables and encoding every method with a patchwork of if/then/switch statements.

This paper will describe a new technique that exploits the ability of an object in TclOO to change class dynamically. TclOO is available as a package for Tcl 8.5, and is integrated into the core of the upcoming Tcl 8.6.

Style Guide

In this paper, I will be using the following style conventions:

Built in Tcl command/ keyword	`oo::class`
Name of an class, object, or variable	*`class_bar`*

Block of example code	`# Comment`
	`set foo bar`

Nickel Tour of TclOO

This paper exploits many advanced features of TclOO. But before we play with the advanced features, it may be helpful to go back over the basic ones.

A new class is declared with the `oo::class` command:

```
oo::class create classname {
 superclasses ancestor ancestor ...

 method methodname arguments {
  # Body of method
 }
}
```

Within the body, one declares the structure of the class. The keywords we'll be focusing on in this paper are:

`constructor`	Defines the constructor
`destructor`	Defines the destructor
`forward`	Forward calls for a method to another command
`method`	Define a method
`superclass`	Define the ancestors of this class

Once created, a class is a command. A command with several methods, the most important is **`create`**.

```
# Create a new object with a known name
classname create objectname

# Create a new object with a
# dynamically generated name
set obj [classname new]
```

And once an object is created, it lives as a command. To call a method:

```
objectname method $arg1 arg2 [arg3]
# Save a value returned from a method
set var [objectname method $arg]
```

If methods look and act a lot like procedures, that is by design. They can return a value, just like a standard Tcl proc. They can also call several built in commands, specific to the TclOO environment:

`my`	Exercise a method of the current object
`next`	Call on an ancestor's implementation of this class
`self`	Returns the fully qualified name of this object

The **my** command is an unambiguous way for the Tcl parser to discern what commands are local to the object, and what commands should be resolved globally. It also makes for easier reading on the part of the programmer.

```
proc noop {string} {
  puts “global - $string”
}
oo::class create noop {
  method noop string {
    puts “[self] - $string”
  }
  method test {} {
    my noop “Hello World”
    noop “Hello World”
  }
}
noop create testobj
testobj test
testobj - Hello World
global - Hello World
```

In addtion to the **oo::class** command, TclOO provides **oo::define** and **oo::objdefine**. **oo::define** is used to modify a class dynamically. **oo::objdefine** is used to modify an object dynamically. TclOO also enhances the **info** command with two new methods: **info class** and **info object**, As you can imagine, **info class** provides introspection for classes, and **info object** provides introspection for objects.

Destroying a class

Classes in TclOO are implemented as objects, with their own constructors, destructors, and methods.

If you destroy a class, you automatically destroy any classes or objects derived from that class. And of course for every class that is destroyed as a result of destroying a class you destroy all of its derivatives, and so on. Taking our example from above:

```
info command obj*
obja objb objc objd obje objf
a destroy
info command obj*

# ^ Empty ^
```

Be careful though, *destroying objects by destroying their class prevents the object destructor from being called.*

Multiple Inheritance

One matter that will come up as we develop complex hierarchies of classes will be multiple inheritance. Given a choice between method implementations, TclOO will always choose the latest one defined.

```
oo::class create a {
 method noop {} { return a }
}
oo::class create b {
 superclass a
 method noop {} { return b }
}
oo::class create c {superclass a b}
oo::class create d {superclass b a}
oo::class create e {
 superclass c
 method noop {} { return e }
}
oo::class create f {superclass a b e}
oo::class create g {superclass a b c d e}
oo::class create h {superclass a b d c e}
oo::class create i {superclass e d c b a}
```

a is a common ancestor to the rest, and it provides an basic implementation of a method called *noop*. *b* is a descendent of *a* that provides its own implementation of *noop*. *c* and *d* inherit both *a* and *b* explicitly, but in a different order. *e* is a descendent of *b* that provides its own implementation of *noop*. *f* is a descendent of all of the classes *a-e*. *g-i* demonstrate various combinations of *a-e*.

```
foreach class {a b c d e f} {
 $class create obj$class
 puts [list obj$class [obj$class noop]]
}
obja a
objb b
objc b
objd b
obje e
objf e
objg e
objh e
obji e
```

You will see that in every example, the latest version of the *noop* that is defined is the one that is used. Since *b* is a descendent of *a*, given a choice between *b*'s implementation of a method and *a*'s implementation of a method, *b* will always be preferred. Likewise, *e* is a descendent of *b*. *e*'s version of a method will always be preferred to *b*'s.

If we do the example differently, sans b inheriting a and e inheriting b, we would get a different results, and the order in which classes are specified in the superclasses keyword becomes more important:

```
oo::class create a {
 method noop {} { return a }
}
oo::class create b {
 method noop {} { return b }
}
oo::class create c {
 superclass a b
}
oo::class create d {
 superclass b a
}
oo::class create e {
 method noop {} { return e }
}
oo::class create f {superclass a b e}
oo::class create g {superclass a b c d e}
oo::class create h {superclass a b d c e}
oo::class create i {superclass e d c b a}
```

```
foreach class {a b c d e f g h i} {
 $class create obj$class
 puts [list obj$class [obj$class noop]]
}
obja a
objb b
objc a
objd b
obje e
objf a
objg b
objh a
obji e
```

Objects Changing Classes

Within the **oo::objdefine** command is the ability for an object to change class:

```
oo::objdefine $object class $newclass
```

An object can even alter it's own class from within a method:

```
oo::class create moac {
 method morph newclass {
   oo::objdefine [self] class $newclass
 }
}
```

To demonstrate this process in action, imagine two classes, *classa* and *classb*:

```
oo::class create classa {
 superclass moac
 method testfunc {} {
   return "I am a classa object"
 }
}
oo::class create classb {
  superclass classa
  method testfunc {} {
   return "I am a classb object"
  }
}
```

Both classes have their own implementation of *testfunc*. The value that *testfunc* returns isn't as important as the fact that the values returned are different for the two different classes. Now with the help of a sufficiently rigged demo:

```
classb create test
test testfunc
I am a classb object
# Change class with oo::objdefine
oo::objdefine testfunc class classa
test testfunc
I am a classa object
# Ask the system what class test is
info object class test
::classa
```

```
# Change class with the morph method
test morph classb
test testfunc
I am a classb object
# Ask the system what class test is
info object class test
::classb
```

You can see that **`oo::objdefine`** ***`$object`*** **`class`** takes effect immediately. And it doesn't matter whether the call to change class occurs from within the object or externally. We can even change class several times during the execution of a method:

```
oo::define classb {
  method confusing_demo {} {
    # Store our present class
    set myclass [info object [self] \
      class]
    puts "Start"
    puts "1 - [my testfunc]"
    # Become a different class
    my morph classa
    puts "2 - [my testfunc]"
    # Return to our original class
    my morph $myclass
    puts "3 - [my testfunc]"
    puts "Done"
  }
}
```

The *`classb`* class now has an additional method, *`confusing_demo`*. Note, that through the miracle of modern science, changes to the class automatically apply to all objects that are instances of that class. So we can now call on this new method from our existing *test* object.

```
test confusing_demo
Start
1 - I am a classb object
2 - I am a classa object
3 - I am a classb object
Done
```

The body of *`confusing_demo`* is simply calling the same method three times. In between the calls, we change the class of the object with the *`morph`* method. The different implementations of *`testfunc`* give different output.

Beware of Disappearing Methods

There are plenty of ways to confuse matters by swapping an object's class. In this scenario, we have an event that is programmed to go off when an object changes class.

```
oo::class create  baz {
 method do_something {} {
   puts "Meh"
 }
 method morph newclass {
   oo::objdefine [self] class $newclass
   my do_something
 }
}
oo::class create  fubar {
 method event_morph {} {
   puts "I have morphed"
 }
 method morph newclass {
   oo::objdefine [self] class $newclass
   my event_morph
 }
}
```

Now, suppose we convert this object from *`fubar`* to *`baz`*:

```
fubar create test
test morph baz
error: Unknown method "event_morph"
```

We get an error! And we get that error because the object assumes the new class instantly. We just happened to pick a class that doesn't implement the *`event_morph`* method, which the script the object is running through tries to call on the next line.

Note, even though we encountered an error, `test` remains class *`baz`*. So if we run the *`morph`* method again:

```
test morph baz
Meh
```

It runs successfully. We can even make `test` back into a *`fubar`*:

```
info object class test
baz
test morph fubar
Meh
info object class test
fubar
test morph fubar
I have morphed
```

What Happens [next]

Another interesting wrinkle in changing classes is how the **next** keyword resolves within a method that changes the object's class. Lets say we have an class that uses **next** to exercise the ancestral implementation of the same method.

```
oo::class create a {
 superclass moac
 method testfunc {} {
  puts "a - [info object [self] class]"
 }
}
oo::class create b {
 superclass a
 method testfunc {} {
  next
  puts "b - [info object [self] class]"
 }
}
oo::class create c {
 superclass b
 method testfunc {} {
  my morph a
  next
  puts "c - [info object [self] class]"
 }
}
```

For interactions between *a* and *b*, things are quite straightforward.

```
a create test
test testfunc
a - a
test morph b
test testfunc
a - b
b - b
```

c is our complex case. Its implementation of *testfunc* changes the class of the object. And worse, it changes the class to one in which there is no ancestor for the **next** operator to hop to.

You would expect the system to die horribly along the lines of:

```
c create test
test testfunc
no next method implementation
    while executing
"next " ...
```

Instead we see:

```
c create test
test testfunc
a - a
b - a
c - a
# Note, the object really has changed
# class
info object class test
a
```

The pathway through the **next** calls is computed before the method is invoked.

Design Patterns

Now that we have covered the basics, it is time to start to develop the **LOG** framework.

Storing Properties

When an object expects to change class, there is often information specific to that class that we would like to access. A variable isn't a good fit for this purpose as its value doesn't change when the class changes. So I like to employ methods that return hard coded values.

The simplest way would be to declare a method for every value we would want to return:

```
oo::class create a {
 method color {} { return green }
 method flavor {} { return lime }
}
oo::class create b {
 method color {} { return green }
 method flavor {} { return apple }
}
oo::class create c {
 method color {} { return red }
 method flavor {} { return cherry }
}
```

For the lazy programmer this system has several drawbacks. First, it is difficult to distinguish between a method that is a property and a, shall we say, livelier method. Second, the notation is verbose. It introduces the temptation to cut and paste. Third, we have no fallback mechanism should a part of the system call for a property that has not been configured yet, or is simply not applicable to the object in question.

LOG adds two new methods: *`property_define`* and *`properties`*. *`property_define`* creates a single value. *`properties`* allow us to specify a key/value list. We can do this easily within TclOO because, behind the scenes, classes are merely a special kind of object. The just happen to be of class **`oo::class`**.

```
oo::define oo::class {
  method property_define {field value} {
    oo::define [self] method prop_$field \
      {} [list return $value]
  }
  method properties dict {
    foreach {var val} $dict {
      my property $var $val
    }
  }
  method property {field args} {
   set methods [info object methods [self] \
       -all -private]
   if {"prop_$field" in $methods } {
     return [my prop_$field {*}$args]
   }
  }
}
```

We also need to configure all of our client classes with a version of the property method.

```
oo::class create moac {
  method property {field args} {
   set methods [info object methods [self] \
       -all -private]
   if {"prop_$field" in $methods } {
     return [my prop_$field {*}$args]
   }
  }
}
```

So to configure a class:

```
oo::class create a {superclass moac}
a property_define color green
a properties {
 flavor lime
}
a create test
test property color
green
```

At the same time, if I ask for an item that is not configured (or configured yet), I get back an empty list instead of an error.

```
test property speed

# ^ Empty List ^
```

And if we are modeling a system worthy of a Lewis Carroll novel, we can alter the property of a class on the fly too.

```
a property_define speed very_fast
test property speed
very_fast
```

Using Classes to Represent State

State machine code becomes notoriously complex when there are more than a handful of states. I am going to introduce an easier way: create a separate class for each state an object can be in. Thus, if a method has to behave differently, we can just define that change for the particular state.

Let us begin with a few ground rules for changing an object's class. Even better, let's have a library of base classes that enforce those rules. All classes that are eligible to change class will be descendants of a common baseclass: *state_machine*.

state_machine provides several methods:

state_change	Change the class (and this state) of an object. Takes an additional argument which can pass additional data to event scripts.
state_current	Return the current class (thus state) of an object
state_enter	Script to run when an object enters the configured state
state_exit	Script to run when and object exits the current state

```
oo::class create state_machine {
  superclass moac ; # For "property" method
  constructor {} { my state_enter {} }
  # Return the current state
  method state_current {} {
    return [info object class \
     [self object]]
  }
  # Actions when we exit state
  method state_exit {} {}
  # Actions when we enter state
  method state_enter {} {}

  # Returns 1 if state changed
  # Returns 0 otherwise
  method state_change {newstate} {
   if { $newstate eq {} } { return 0 }
   set oldstate [my state_current]
   if { $newstate eq $oldstate } {
    # In the desired state, do nothing
    return 0
   }
   # Run cleanup from old state
   my state_exit
   oo::objdefine [self] class $newstate
   # Run setup from new state
   my state_enter
   return 1
  }
```

Example: Lifecycle of a Frog

Let is show off our newly developed *state_machine* with a demonstration: The lifecycle of a frog.

```
oo::class create frog {
  superclass state_machine
  method state_exit info {
    puts "Leaving [my state_current]"
  }
  method state_enter info {
    puts "Entering [my state_current]"
    next $info
  }
}
frog properties {
  has_tail 0
  respiration lung
  state_next {}
  color green
}
```

The baseclass *frog* is a series of general assumptions one could make about any frog, stored as properties. One of those properties *state_next* tells us what developmental state follows the current state. For an adult *frog*, we have no *state_next*, so we configure an empty set.

To model our frog's lifecycle, a program can simply walk from one state to another, reading the properties as it goes.

```
oo::class create frog.egg {
  superclass frog
}
frog.egg properties {
  has_tail 0
  respiration none
  state_next frog.tadpole
}
oo::class create frog.tadpole {
  superclass frog.egg
}
frog.tadpole properties {
  has_tail 1
  respiration gill
  state_next frog
}
```

```
frog.egg create hypno
set changed 1
while {$changed} {
 foreach fld {
   has_tail respiration state_next
 } {
   puts " * $fld [hypno property $fld]"
 }
 set newstate [hypno property state_next]
 set changed [hypno state_change $newstate]
}
Entering ::frog.egg
* has_tail 0
* respiration none
* state_next frog.tadpole
Leaving ::frog.egg
Entering ::frog.tadpole
* has_tail 1
* respiration gill
* state_next frog.tadpole
Leaving ::frog.tadpole
Entering ::frog
* has_tail 0
* respiration lung
* state_next frog
```

Discrete Time Phases

Discrete time simulations are similar to tabletop games. Actors (or players) take turns. And the rules of the game govern which interactions are valid during which part of a game turn.

In [1]Risk™ , each turn has three phases: placing reinforcements, attack, and fortifying. Players are only allowed to add troops to the battlefield at a certain time. There is only one phase in which we would expect troops to be removed from the battlefield (as casualties.) And there is only one point in the turn where troops can move. Phases make the outcome of a series of events more consistent.

Table games are engineered to have a definite "winner". The actor with priority is allowed to have a significant impact on the outcome of the scenario.

```
turn 1
 Player 1 - Reinforce Phase
 Player 1 - Attack Phase
 Player 1 - Fortify Phase

 Player 2 - Reinforce Phase
 Player 2 - Attack Phase
 Player 2 - Fortify Phase
```

With scientific simulations, we don't want a "winner." We want to devise a series of rules such that we get the same outcome whether the actors are run in sorted order, reverse sorted order, random order, or whatever that subtle, non-random, but sufficiently inscrutable order we get from `[array names]`.

We also want to create the illusion that all of the actions in a given time phase occur simultaneously. So rather than let one actor run through all of the phases, followed by another, we give each actor an opportunity to act during every phase.

[1] Risk™, Trademark Parker Brothers

```
turn 1
 Player 1 - Reinforce Phase
 Player 2 - Reinforce Phase
 Player 1 - Attack Phase
 Player 2 - Attack Phase
 Player 1 - Fortify Phase
 Player 2 - Fortify Phase
turn 2
  ...
```

In simulators which allow objects to change class, I found it best to restrict any such changes to a specific phase in the time step. Preferably one in which nothing else is going on.

```
Agent Timestep
 phase_physics
 phase_observe
 phase_plan
 phase_action
 phase_reaction
 phase_morph
```

When an object wants to change state, the new state is recorded as a local state variable. The actual change does not take place until the morph phase comes around.

```
oo::define state_machine_discrete {
 method state_change newstate {
  if {[my state_current] eq $newstate } {
    return 0
  }
  my variable next_state
  set next_state $newstate
  return 1
 }
 ###
 # Called by the driver of the simulation
 ##
 method phase_morph {} {
  my variable next_state
  if { $next_state eq {} } {
   return
  }
  my state_exit
  oo::objdefine [self] class $next_state
  my state_enter
  set next_state {}
 }
}
```

Example: Agent Based Modeling

The Integrated Recovery Model simulates a ship and her crew during a shipboard catastrophe. Part of the simulation entails crew members changing roles. In the model, each role is represented by a distinct class.

Any number of events can lead to a crew member changing role. The most common role changes are in response to an order. Some orders are direct. For instance, a leader telling a crew member under his/her command “You do this.” Other orders are indirect. When a crew member hears the call to go to General Quarters, he/she switches from whatever they were doing to their assigned role at GQ.

But the hardest ballet to choreograph by far was the transitions that occur when a crew member is assigned to a fire team. Most crew don’t wear a fire suit as part of their regular duties. Thus a crew member newly assigned to a fire team must find a set of gear, put it on, and connect with a team that may already be on scene. Those behaviors were complex enough to merit a separate role.

```
Crew starts as role human
Crew receives order to join Team
 > Crew becomes role team.prospect
Crew member gathers equipment
Crew walks to location of Team leader
Crew joins Team
 > Crew becomes role team.member
Team battles fire
Team dissolves
 > Crew becomes team.dismissed
Crew returns equipment
Crew walks back to assigned station
 > Crew becomes human
```

In IRM, each agent is configured with a property that lists what tasks they want to perform, and in what priority. Each task, in turn, has criteria that govern when it should activate, when it should abort, and a coroutine to carry out once activated.

Our `team.prospect` class has the following task list:

action-station	Gather tools, report to action station
safety-check	Reflexes for fleeing from danger
join-team	Join the team we are assigned to
go-home	Return to action station (only called if join-team fails)

Every agent has an `action-station` task. It has a method that produces a list of equipment required for the role assigned. It checks to see that the agent has a working version of each. And if a device is missing, exhausted, or damaged, the agent gets a new one.

Normally agents produce their own list of needed equipment, based on information configured by the model maker. For this paper, the pseudocode uses a simple property.

```
agent::class human {
 method ensemble {} {
  return [my property equipment]
 }
 method ensemble_missing {} {
  set result {}
  foreach device [my ensemble] {
   if {[my device_working $device]!=1} {
     lappend result $device
   }
  }
  return $result
 }
 task action-station {
  begin {
    return [llength [my ensemble_missing]]
  }
  ... # Define the rest of the task ...
 }
}
```

```
agent::class fireteam {
  superclasses human
  properties {
   equipment { nfti scba ppe radio }
   member_equipment {scba ppe}
  }
}
agent::class rescueteam {
 superclasses human
 rescueteam properties {
   equipment { radio stretcher scba }
   member_equipment {scba medkit}
 }
}
# One team.prospect class suffices
# to join either team
agent::class team.prospect {
 superclasses human
 method ensemble {} {
  my variable team
  return [$team property member_equipment]
 }
}
```

Because the `team.prospect` role is it's own class, we can override the standard `ensemble` method with one that queries the team this agent will join.

```
# One team.prospect class suffices
# to join either team
agent::class team.prospect {
 superclasses human
 method ensemble {} {
  set team [my knowledge get team]
  return [$team property member_equipment]
 }
}
```

Thus:

```
fireteam create crew1
rescueteam create crew2
team.prospect create crew3
team.prospect create crew4
crew3 knowledge put team crew1
crew4 knowledge put team crew2
crew3 ensemble_missing
scba ppe
crew4 ensemble_missing
scba medkit
oo::objdefine crew3 human
crew3 ensemble_missing

# ^ Empty we are back to the human class
```

Application State

When designing a GUI, we also wrestle with state. Whether it be a megawidget, or a toplevel object that is managing the application, LOG can help.

In IRM our principle display interface is managed through a Tk canvas. Onto that canvas, we draw objects, color them, and respond to mouse gestures.

We divide our model's world into drawing layers. There are specific rules for rending a wall that are different than, say, a piece of equipment. Likewise, a user double clicking on a wall expects a different dialog box if clicking a crew member versus a portal.

Window objects call out which layers are active and in which state as a method of the window:

```
irm::class modelwindow {
 superclasses [redacted]
 method active_layers {
   return {
    wall  layer.wall.basic
    compt layer.compt.basic
    portal layer.portal.basic
    eqpt  layer.eqpt.basic
    crew  layer.crew.basic
   }
 }
}
```

When devising a set of visuals, I put together two sets of classes. One is the application window, the other is a drawing layer that is modified to produce the visual.

```
irm::class modelwindow.damage {
 superclasses modelwindow
 method active_layers {
   return {
    wall  layer.wall.damage
    compt layer.compt.damage
    eqpt  layer.eqpt.damage
    crew  layer.crew.damage
    portal layer.portal.damage
    holes  layer.holes
   }
 }
}
```

In this case we are putting together a special mode that highlights damaged objects with a special color.

When applying a new state, the window object will call forth into being an object to represent each layer, and configured with the appropriate class. If the layer already exists it simply changes class.

In our example, the modified drawing layer colors all damaged components red.

```
irm::class layer.eqpt.damage {
 superclasses layer.eqpt.basic
 method node_is_damaged nodeid {
   # test for damage that returns 1 or 0
 }
 method node_style {nodeid} {
  if {[my node_is_damaged $nodeid]} {
    return {-fill red -outline -red}
  } else {
    return {-fill grey -outline grey}
  }
 }
}
```

Application window states can also specify bindings for the canvas. In the next example, upon entering the new state the canvas gets new bindings. Once the user clicks on an object the window translates motion to drag actions. When the user releases the dragged object, the window reverts back to its normal state.

```
irm::class modelwindow.drag {
 superclasses modelwindow
 method active_layers {
   return {
    wall  layer.wall.basic
    eqpt  layer.eqpt.editor
    crew  layer.crew.editor
   }
 }
 method state_enter {} {
   set canvas [my get canvas]
   bind $canvas <B1> \
     [list [self] drag_start %x %y]
   bind $canvas <B1-Motion> {}
   bind $canvas <B1-Release> {}
   my redraw
 }
 method drag_start {x y} {
  set obj [my object_at $x $y]
  if { $obj eq {} } { bell ; return }
  set canvas [my get canvas]
  bind $canvas <B1-Motion> \
    [list [self] drag_do $obj %x %y]
  bind $canvas <B1-Release> \
    [list [self] drag_done $obj %x %y]
 }
 method drag_done {obj x y} {
   set layer [my object_layer $obj]
   $layer move_to $obj $x $y
   my morph modelwindow
 }
}
```

Conclusion

Lifecycle Object Generators are not the solution to every problem in Object Oriented programming. But they are quite useful for complex state-based logic. I am developing these concepts into a fully featured toolkit, which is available for download at:

http://www.etoyoc.com/tcl

Image Credits:

Cover Image:
“Entwicklung des Krötenfrosches”, By Meyers Konversations-Lexikon [Public domain], via Wikimedia Commons, accessed 17 October 2012, <http://commons.wikimedia.org/wiki/File%3AMetamorphosis_frog_Meyers.png>