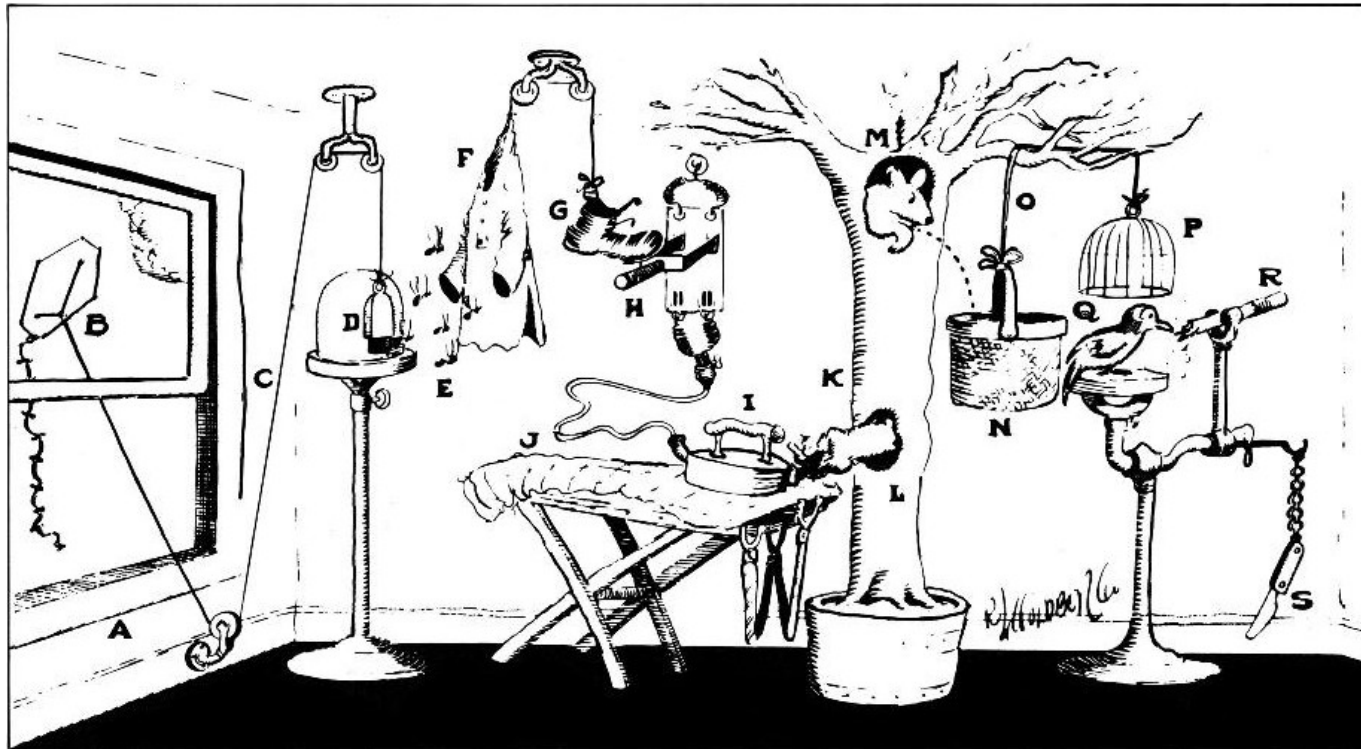


# Advanced TcIOO and Megawidgets



Presented by:  
Sean Woods

# What do you need for this class?

- Basic understanding of TclOO
- Some experience with Tcl
- A working grasp of OO concepts
  - Inheritance
  - Data Encapsulation
- A standard measure of Tclicr masochism

# Advanced TclOO

- Customizing `oo::class`
- Adding new keywords to `oo::define`
- Metaclasses
- Adding new methods to `oo::object`
- Defining and inheriting metadata
- Changing objects from one class to another
- Grafting objects together
- Process pipelines
- Replacing “`vwait forever`”

# Megawidgets

- Building meta classes
- Tk-like option handling
- Tk-like constructors
- Destructor handling
- Message passing
- Object hierarchy

# Wax on, Wax off



# Be patient

- Building megawidgets requires a lot of plumbing, wires, and ducts behind the walls
- The five words of ruin:
  - “I don't need a framework!”

# What's a Framework?

- The conventional definition?
- Um... quick someone who went to school for this:

# What is a Framework in Tcl

- It's somewhere between a design pattern and a library
- Rather than responds to calls thrown over the wall, a framework meshes into inner workings of the program



# Using a Framework in Tcl

- Most frameworks load as a package
- They can include:
  - Definitions for base classes
  - New keywords for TclOO
  - Language automation tools
- In Sean's perfect world:
  - Frameworks are loadable with “package require”

# The TOOL Framework

- I like to teach with a concrete example
- TOOL is a base framework for a new TclOO community standard library
- It is published via fossil at:
  - <http://fossil.etoyoc.com/fossil/tool>

# Modifying oo::class

- You can do that?
  - Yes
- Should you do that?
  - Depends
- When should I not?
  - When your enhancement changes the behavior of TclOO
  - (Not that changing OO's behavior it is a problem, it just makes your software incompatible with anyone else's)

# Modifying oo::class

- oo:class is actually a TclOO object itself
- New methods can be defined for it with oo::define
- Any method created in this manner is available to ANY instance of oo::class
- Remember: All classes are objects too

# Example

```
oo::define oo::class {  
  method is_a args {  
    return false  
  }  
}
```

```
oo::class is_a frog  
> false
```

```
oo::class create frog {  
  self method is_a args {  
    if {"frog" in $args} {  
      return true  
    }  
    return [next]  
  }  
}
```

```
frog is_a frog  
> true
```

# self method

- self method allows a class object to define its own method.
- self method is *not* inherited by descendants
- self method is *not* imparted to objects of the class

# self method

```
oo::class create toad {  
  superclass frog  
}  
toad is_a frog  
> false
```

```
frog create kermit  
kermit is_a frog  
>unknown method "is_a": must be destroy
```

# Adding new keywords to oo::define

- oo::define is a Tcl namespace
- Any command in oo::define is available within the body of an oo::create statement
- The body of a proc in oo::define can exercise all of the other commands as if they were part of the class definition itself
- The name of the current class is accessible with `[lindex [::info level -1] 1]`



# Adding to oo::define

```
proc ::oo::define::macro {name patterns arglist body} {  
  set patterns [uplevel 1 [subst $patterns]]  
  lappend patterns @CLASS@ [lindex [::info level -1] 1]  
  lappend patterns @CLASS_TAIL@ [namespace tail $class]  
  method $name [string map $patterns $arglist] [string map $patterns $body]  
}
```

```
set macro_isa {  
  is_a {} args {  
    if {"@CLASS_TAIL@" in $args} return true  
    return [next]  
  }  
}
```

```
oo::class create amphibian {  
  macro {*}$macro_isa  
}
```

# Meta-Classes

- Classes which re-define what classes are
- There are only 9 methods to override:
  - constructor
  - destructor
  - filter
  - forward
  - method
  - mixin
  - superclass
  - variable

# Meta-Classes

```
oo::class create Class {
  superclass oo::class
  method constructor {argList body} {
    ::oo::define [self] constructor $argList $body
  }
  method destructor {body} {
    ::oo::define [self] destructor $body
  }
  method filter {args} {
    ::oo::define [self] filter {*} $args
  }
  method forward {name cmdName args} {
    ::oo::define [self] forward $name $cmdName {*} $args
  }
  method method {name argList body} {
    ::oo::define [self] method $name $argList $body
  }
  method mixin {args} {
    ::oo::define [self] mixin {*} $args
  }
  method superclass {args} {
    ::oo::define [self] superclass {*} $args
  }
  method variable {args} {
    ::oo::define [self] variable {*} $args
  }
}
```

**You went full Meta-class**



**You never go full meta-class**

# Extending oo::object

- oo::object is just an object like any other
- Methods can be added or modified with oo::define
- Any modification to oo::object is imparted onto EVERY object in TclOO
- *Even the ones you have already defined*

# Extending oo::object

```
oo::define oo::object {  
  method is_a args {  
    return [[info object class [self]] is_a {*}$args]  
  }  
}
```

```
kermit is_a frog  
> true
```

# Putting it all together

- We will now use all of the concepts we have developed thusfar to implement meta-data tracking for TcIOO

# Meta-Data

- Has nothing to do with meta-classes
- Or that nasty business with mobile phones, the NSA, or the Bildeberg Group.
- (Or at least that's what want the Illuminati want you to believe)



# Meta-Data

- Meta-data is “data” which is imparted by the class onto the descendents of that class, and onto objects of that class.
- It is essential for a properly designed option database
- Or database schemas
- And 1e6 other things that you didn't realize you could do before you had the tool

# Implementing Meta-data in TOOL

- TOOL implements a meta-data system through a combination of:
  - Adding a new method to oo::class
  - Adding a new method to oo::object
  - Adding a new keyword to oo::define

# Examples

```
# Really generic class
oo::class create tetrapod {
  meta set limbs: 4
}
```

<b>limbs:</b>	<b>4</b>
---------------	----------

```
# More specific class
oo::class create amphibian {
  superclass tetrapod
  meta set habitat: land
  meta set diet: insects
}
```

limbs:	4
<b>habitat:</b>	<b>land</b>
<b>diet:</b>	<b>insects</b>

```
#really specific class
oo::class create frog {
  superclass amphibian
  meta set habitat: swamp
}
```

limbs:	4
<b>habitat:</b>	<b>swamp</b>
diet:	insects

# Examples (on the object side)

```
frog create kermit  
dict print [meta dump]  
> limbs: 4  
> habitat: swamp  
> diet: insects
```

```
kermit meta cget limbs  
> 4
```

```
kermit meta set favorite_color: {Green}  
kermit meta set habitat: {The stage}  
kermit meta set diet: {The finer things}  
kermit meta cget diet  
> {The Finer Things}
```

# Implementation

- TOOL Creates:
  - a `::tool::meta` command ensemble, which stores the values
  - a “meta” keyword to `oo::define`
  - a “meta” method to `oo::class`
  - a “meta” method to `oo::object`
- Magic behind the scenes builds composite dicts with the sum-total of all has been defined for a class
- These caches are destroyed if the class or one of its ancestors alters its meta data

# Implementation (Objects)

- Objects maintain a local dict **config**
- When meta data is requested, if a value is present in config it is returned
- When a value does not exist in config, the **meta** method then searches for data from the class

# Other oddities...

- To properly handle recursive data structures meta dicts follow the standard developed for shed:
  - Elements which end in “:” are considered a leaf node
- The “cget” ensemble method automatically searches for a leaf node, and will check for the presence of the value as either “field:” or “field”

# Implementation

- In the “tool” fossil repository, meta-data is implemented in:
  - `/modules/tool/meta.tcl`



# Object morphology



# Object Morphology

- Objects in UI often change behavior depending on their stage of evolution
- A change in class solves this problem nicely
- Changing the class of an object in TclOO is as “easy” as:

```
oo::objdefine $OBJECT class $NEWCLASS
```

# Object Morphology

- `oo::objdefine` changes the class of an object instantly.
- If the performed within an object's own method, the body of the method (as defined by the old class) runs to completion
- **HOWEVER:**
  - any calls to external methods will be of the new class
  - The constructor of the new class is never run

# Object Morphology

- TOOL adds three methods to all classes:

<code>morph <i>newclass</i></code>	Change an object's class
<code>Morph_leave</code>	Behaviors to exercise as an object leaves the current class, prior to <b>oo::objdefine</b>
<code>Morph_enter</code>	Behaviors to exercise as an object enters a new class. Invoked following <b>oo::objdefine</b>
<code>InitializePublic</code>	Behaviors which will initialize any expected variables, as well as populate expected options with default values if not already defined.

```

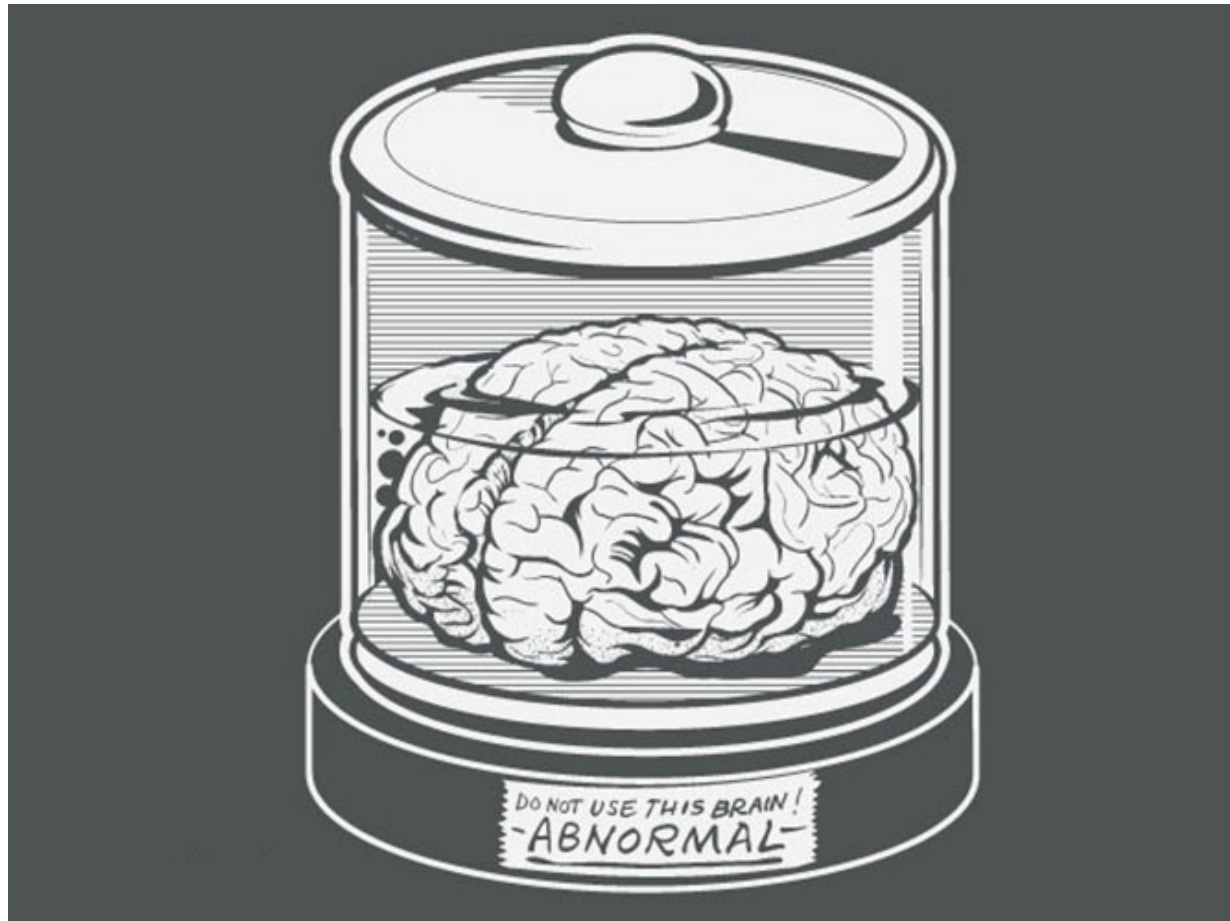
oo::define oo::class {

method morph newclass {
  if {$newclass eq {}} return
  set class [string trimleft [info object class [self]]]
  set newclass [string trimleft $newclass :]
  if {[info command ::$newclass] eq {}} {
    error "Class $newclass does not exist"
  }
  if { $class ne $newclass } {
    my Morph_leave
    oo::objdefine [self] class ::${newclass}
    my variable config
    set savestate $config
    my InitializePublic
    my configurelist $savestate
    my Morph_enter
  }
}

method Morph_leave {} {}
method Morph_enter {} {}
# InitializePublic is defined in the option handing module
}

```

# Grafting Objects



# Grafting Objects

- Megawidgets must coordinate the activity of several subject widgets
- These subjects can often have similar (or identical) names for methods to one another
- Trying to have the megawidget intercept the subject's methods and play them off as their own is messy
- Tool provides a framework for this in `/modules/tool/organ.tcl`

# Grafting Objects

```
oo::object create editwidget {
  superclass took::tk::meta::text

  constructor {w dbobj} {
    my TextHull $w ; # ← grafts the text widget to <text>
    my graft $dbobj record
    my load
  }
  method load {} {
    my <text> delete 0.0 end
    my <text> insert end [my <record> get]
  }
  method save {} {
    my <record> put [my <text> get 0.0 end]
  }
}
```



# Grafting Objects

```
oo::object create editwidget {
  superclass took::tk::meta::text

  constructor {w dbobj} {
    my TextHull $w ; # ← grafts the text widget to <text>
    my graft record $dbobj
    my load
  }
  method load {} {
    my <text> delete 0.0 end
    my <text> insert end [my <record> get]
  }
  method save {} {
    my <record> put [my <text> get 0.0 end]
  }
}
```

# Grafting Objects

```
oo::object create editwidget {  
  superclass took::tk::meta::text  
  
  constructor {w dbobj} {  
    my TextHull $w ; # ← grafts the text widget to <text>  
    my graft record $dbobj  
    my load  
  }  
  method load {} {  
    my <text> delete 0.0 end  
    my <text> insert end [my <record> get]  
  }  
  method save {} {  
    my <record> put [my <text> get 0.0 end]  
  }  
}
```

Both the record object and the text widget have a method named “get”

# Grafting Objects

```
oo::object create editwidget {
  superclass took::tk::meta::text

  constructor {w dbobj} {
    my TextHull $w ; # ← grafts the text widget to <text>
    my graft record $dbobj
    my load
  }
  method load {} {
    my <text> delete 0.0 end
    my <text> insert end [my <record> get]
  }
  method save {} {
    my <record> put [my <text> get 0.0 end]
  }
}
```

We can exercise all of the methods of the *text* widget in an obvious way

# Grafting Objects

```
my <masterframe> <record> <db> eval {  
  select * from datatable  
} {  
  puts "$field: $value"  
}
```

Other objects can exercise  
those methods too

# Grafting Objects

```
my <masterframe> <record> <db> eval {  
  select * from datatable  
} {  
  puts "$field: $value"  
}  
puts "The last field was $field"
```

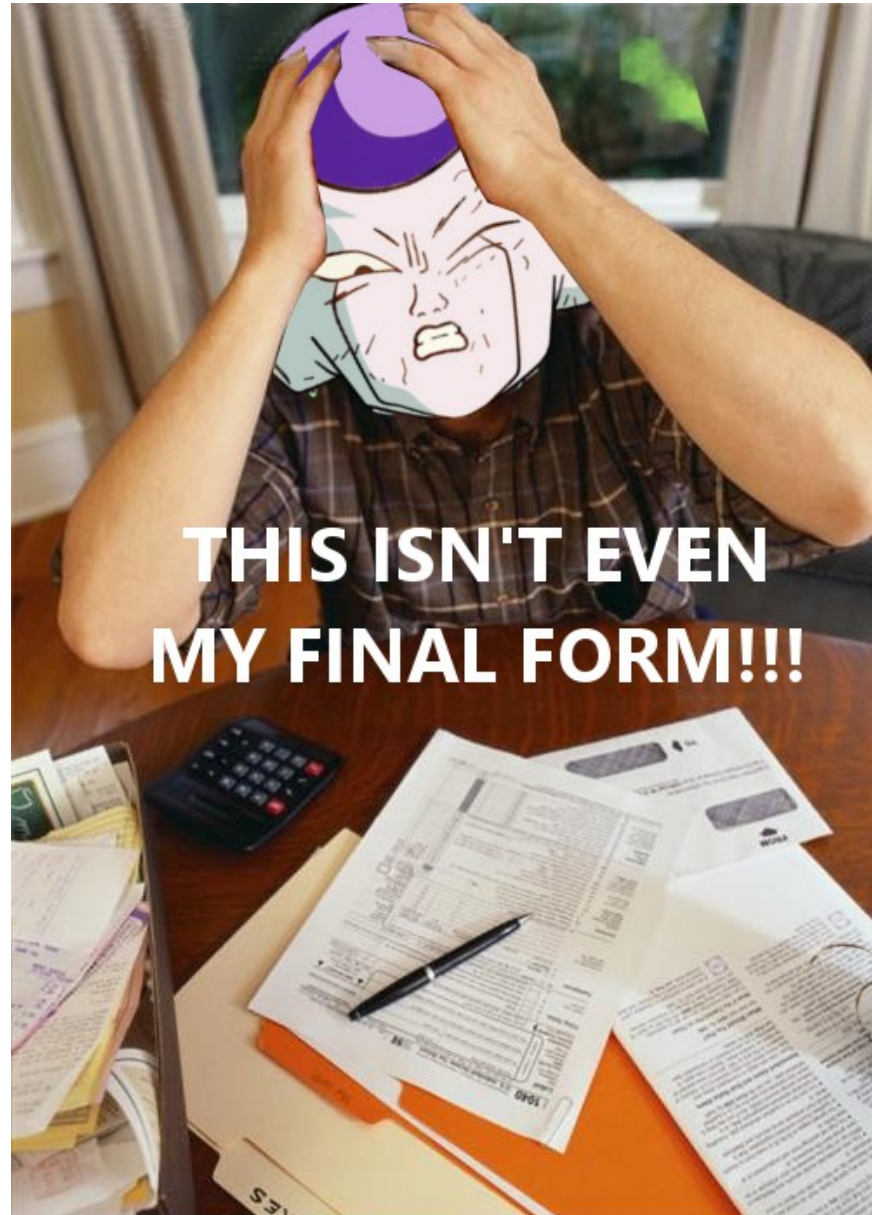
This is exercising a control structure. The loop actually ran in the current context, as if the caller had addressed Sqlite directly. **BECAUSE IT DID.**

# Grafting Objects

Child objects can also inherit stubs from parents

```
oo::class create child {  
  constructor parent {  
    my graft {*}[$parent organ all]  
    my graft parent $parent  
    my <text> insert end \  
      “[self] has latched on”  
  }  
}
```

# Pipelines and Signals



# Pipelines and Signals

- Many megawidget applications are for forms and layouts
- Both require coordinating the activity of several (dozen) objects
- There can (and often is) a gap between when an OO object is created and when it expresses itself in Tk
- Pipeline and signal handling are in `/modules/tool/pipeline.tcl`



# Pipelines and Signals

- Objects declare
  - snippets of code
  - what snippets are triggered in response
  - which snippet should precede this snippet
  - which snippet should follow this snippet

# Pipelines and Signals

```
oo::object create editform {  
  superclass editwidget  
  
  signal display {  
    action {my TextHull [my organ <hull>]}  
    triggers content  
  }  
  signal content {  
    action {my load}  
    follows display  
  }  
  constructor {w dbobj} {  
    my graft hull $w  
    my graft $dbobj record  
  }  
}
```

And Finally...



# Mega-Widgets

- A mega-widget framework already exists in the Tk core
- See: `(TK Source)/library/megawidget.tcl`

# Meta-Class: tk::Megawidget

```
::oo::class create ::tk::Megawidget {
  superclass ::oo::class
  method unknown {w args} {
    if {[string match .* $w]} {
      [self] create $w {*} $args
      return $w
    }
    next $w {*} $args
  }
  unexport new unknown
  self method create {name superclasses body} {
    next $name [list \
      superclass ::tk::MegawidgetClass \
      {*} $superclasses] \; $body
  }
}
```

# tl;dr

- The `tk::Megawidget` metaclass is a class which uses the “unknown” handler to generate an object with the name of a tk path.

# Class: tk::MegawidgetClass

- Instance objects of tk::Megawidget
- The constructor performs a bait and switch with the object's name
- Major Methods:
  - constructor
  - destructor
  - configure
  - cget
  - GetSpecs
  - CreateHull
  - Create
  - WhenIdle
  - DoWhenIdle

# tk::MegawidgetClass constructor

```
constructor args {
  # Extract the "widget name" from the object name
  set w [namespace tail [self]]

  # Configure things
  set OptionSpecification [my GetSpecs]
  my configure {*}$args

  # Move the object out of the way of the hull widget
  rename [self] _tmp

  # Make the hull widget(s)
  my CreateHull
  bind $hull <Destroy> [list [namespace which my] destroy]

  # Rename things into their final places
  rename ::$w theFrame
  rename [self] ::$w

  # Make the contents
  my Create
}
```



# tk::MegawidgetClass destructor

```
destructor {
    foreach {name cb} [array get IdleCallbacks] {
        after cancel $cb
        unset IdleCallbacks($name)
    }
    if {[winfo exists $w]} {
        bind $hull <Destroy> {}
        destroy $w
    }
}
```

# tk::MegawidgetClass configure

- Meh... process command line options
- BOOORRRRING

```
method configure args {
    tclParseConfigSpec [my varname options] \
        $OptionSpecification "" $args
}
method cget option {
    return $options($option)
}

method GetSpecs {} {
    return {
        {-takefocus takeFocus TakeFocus {}}
    }
}
```

# tk::MegawidgetClass CreateHull

- Not provided by the default class
- Builds the frame or toplevel for the widget

# tk::MegawidgetClass CreateHull

- Not provided by the default class
- Fills out the widget underneath the hull

# WhenIdle DoWhenIdle

```
method WhenIdle {method args} {
  if {![info exists IdleCallbacks($method)]} {
    set IdleCallbacks($method) [after idle [list \
      [namespace which my] DoWhenIdle $method $args]]
  }
}
method DoWhenIdle {method arguments} {
  unset IdleCallbacks($method)
  tailcall my $method {*}$arguments
}
```

# Improvements in TOOL\_Tk

- Utilizes the meta subsystem as an option database