TOAD:
Tips for Object Architecture for Development

By: Sean Woods

Presented to the 2009 Tcl Developer’s Conference
Abstract

Like a child at Christmas, Tcl developers everywhere are unwrapping all of the new functions of the Tcl8.6 core. The feature everyone wants to play with is, of course, TclOO. Our knight in shining armor. The holy grail of core team. Object Oriented programming is right.in.the.core.

And now that it's here we have to ask, “and what exactly are we going to do with it?”

This paper is Sean’s attempt to put together the “Tcl Way” of writing code for TclOO. Like everything else Tcl, it’s not about tab spacing and pascalNotation vs. underbars _uberalles. It’s about how not to hang yourself with some of the rope that the notation provides.

Along the way, Sean will provide a few tidbits and gee-whiz tricks he’s discovered/stole about TclOO.

Developer Biography

Sean Woods is a regular at the Tcl Developers Conference. Known as “The Hypnotoad” in the community, he is well known for his off beat way to present otherwise dry material.

Sean’s experience with Tcl goes back to 1996, where he worked on a large scale automation project for Kulicke and Soffa. Sean currently uses Tcl/Tk to generate and visualize simulations for the US Navy.

Introduction

Like every great innovation, this paper has a bit of a long story.

Fortunately, that story has already been written. It’s in my paper for the ’06 conference entitled “Tao: The Tcl Architecture of Objects”. So if you are interested in the history, check it out there.

This paper, however, is for the living. So, we have an official object system.

No more forging OO with our bare hands from still molten steel like in the “good old days”. You while you are still invited to walk to school uphill, both ways, our focus as a community should now be on how to use this new system competently.

I have been OO programming in Tcl for several challenging environments in the past:

- Web portals
- One-off game projects
- A canvas-based ship description editor
- An agent based fire-fighting simulation

And over the years I have honed a particular style. Everything presented are concepts I’ve invented, and likely reinvented several times, over the years.

Many of the concepts are stand-alone, so you can feel free to rifle through the bits you like and leave the rest.

All right, by this point you are either interested enough to read on, or not. So let’s just cut to the code.

The TOAD Way

The TOAD Way I like to think of as the “least energy path” for software development. Like all good Tcl techniques, after you see it, you’ll just find yourself doing it naturally. (Assuming you haven’t been doing things that way for years.)

Much of it protects you against many, many, pitfalls that can crop up from haphazard development techniques.

Use a Psuedo Language

My rule is if a snippet exists in three places or more, it should be re-cast as a
procedure, a method, or a macro of some sort.

Anyone who has ever had to clean up someone else’s code (Or even worse, look at your own code years later…) knows the horror of “copy and sorta paste.” You know what I mean, a 10 line routine that is scattered ALL over the code. (Usually complete with comment.) But 8/10 of the copies contain a subtle change that you completely overlook the next time you go to copy and paste.

Well that tendency gets worse with object oriented code. (And I speak from experience.) The biggest offenders are a pre-ambles we all seem to toss onto the top of methods to put all the variables we want in just the right place.

The example above is a common design template in web development. We get data in, in one form. We play with it. We format it back to something the webserver wants to see.

If you are building a webserver, each method could be a page. You’d have foo, bar, baz, bing, boom. Each does a different page function. But for all of them you have a common set of routines that govern input handling, session management, etc.

What I found helpful was to actually wrap the key working parts in each page around a procedure that added the “cut and paste” to the top and bottom. In the old days, I had to do it all up front with a wrapper. But TclOO includes a powerful set of tools that allow you do define

```tcl
method foo theunmungedarg {
    # Load our variables
    variable bar
    variable bat
    set arg [my munge $theunmungedarg]
    # Begin with a blank result
    set result {}

    # (the actual method)
    ...

    # Remunge our result
    return [my remunge $theresult]
}
```

```tcl
proc pageMethod {
    class methodname body
}
{
    # Build a buffer that starts
    # with our preamble
    set methodbody ::preamble
    # Tacks on our body
    append methodbody \n $body \n
    # Tack on code that
    # transforms the result
    append methodbody \n $::postamble \n
    # With the actual body built
    # define the method
    oo::class define $class \n $methodname \n theunmungedarg \n $methodbody
}
```

classes and objects on the fly:

So now, to build our pages instead of having to copy and paste a ton of code we simply:
This has an added advantage for development: you can call these `pageMethod` procedures again later to reload the code without having to destroy your original class first. For webservers, I put the main class in one file, and the page method in another so that I can re-load the page generating code in a still-running interpreter without having to completely re-start the server.

And, of course, when you find some funky fix than needs to be applied to the front end or back end, you can update your template generator instead of having to apply the same fix in a dozen places.

**Use dicts for arguments**

The basic idea here is that we are now in the 21st century. Software development no longer assumes that you know absolutely everything about everything before you start coding. Largely because all strategies evaporate on contact with the actual implementation.

Now for small projects, and simple functions, sure, you can always assume that the number of arguments for a function will never change. I have a function that takes in a string, and output another string, no brainer. If return a simple mathematical transform of a fixed number or parameters, sure.

But most functions involve the interoperable machinations of the system. And those change during the course of a project. A lot. For open ended design I recommend a complexly-simple style for arguments. Don’t bother. Take in a single argument, and that argument is a dict with the actual arguments.

Again, the worst offenders tend to be web portals. They love to pass you extra data. And the form that data takes is pretty free-form. And every once in a while, it comes in useful!

So, for argument’s sake, lets have an object “strawman”. Strawman generates an on-screen display of various nodes, and the redraw methods for each node type take in a nodeid, and a color

Now, at some point your marketing folks come back to you with a pile of other things they’s like to see displayed. Color. Stipple. Maybe even images. Ugh. Do you really want to add an argument for each one?

Oh sure, you could take in arguments the tk way. But then you are stuck adding the dashes, and then removing the dashes, and really there’s an easier way.

Dicts.
In the example snippet above, we take a dict defined by `global_defaults` and feed them into local variables. We then treat formatting as a dict, and dict with handily will read each key/value pair and load them in turn as local variables.

And so our drawing code can now happily call `$color` and `$stipple` and whatever else you find you need to describe the object. It will always have a value, as defined in `$global_defaults`.

Calls to this function would look like this:

```
strawManObj redrawFoo e10 {
    color red
}
strawManObj redrawFoo e11 [list \
    color [someColorFunction e11] \
    stipple grey25]
```

In the example snippet above, we take a dict defined by `global_defaults` and feed them into local variables. We then treat formatting as a dict, and dict with handily will read each key/value pair and load them in turn as local variables.

And so our drawing code can now happily call `$color` and `$stipple` and whatever else you find you need to describe the object. It will always have a value, as defined in `$global_defaults`.

Calls to this function would look like this:

```
strawManObj redrawFoo e10 {
    color red
}
strawManObj redrawFoo e11 [list \
    color [someColorFunction e11] \
    stipple grey25]
```

variable in your object. For example:

Now, some other method, calling up the `poorlychosenexample` variable will see `DEADBEEF` instead of it’s regularly scheduled value.

And, speaking from experience, this can be a real pain to diagnose. It can also get you into serious trouble in environments like web portals where you have data coming in from the outside.

To that end, I’ve devised a reasonably devious way of handling state data…

Ok, maybe to be play to the old school BASIC crowd I should have used “peek and poke”. But then when I started talking about using protection and wrapping your dict, nobody would stop giggling long enough for me to finish this paper.

The general idea is that your object has only one “variable”. That variable is a dict, and everyone accesses a copy of it through the `get` method. Changes to the state are done through the `put` method.

Because the state is a dict, it’s easy to apply to a body of code. And, because you are accessing a copy, you don’t care if your later self decides to name one of his local variables “table” which is used by other methods to track what sqltable a
There's also a handy side effect in that you can initialize your object's state with a single argument to the constructor. Here is a quick and dirty implementation:

I did throw in one creature comfort, if the user provides a fieldname, get will grab just that field. (Whether you check for its existence or not first is a matter of taste.) With no argument, you get the whole enchilada.

Of course, once you've wrapped the state of your object, there is really nothing that says it has to be stored in a local variable. Or in a variable at all! In many of my systems get and put actually talk to an SQL table.

By the by, because we are going to be doing a lot of merging of dicts let me go ahead and define a useful proc:

It simply takes N dicts, and applies them in order into one big dict, ensuring the later values for each field supersede the previous.

In practice, you'll see a lot of methods in this paradigm like this:

(info is the sum total of what is in the object's state, and what was given to us by the function. Somewhere along the line in either the state of be object or the argument to the method $bing, $ban and $baz are defined. We calculate $bar from them. If the result doesn't match $baz we store the new value. Silly function, yes, but it gets the concepts across.

Containers and Nodes

Ok, so let us expand a little on these devious little methods we have created, get and put. As I alluded to, once you get in the habit of accessing your state through these (or any other) methods, a new world opens up to you. I like to call them “disposable objects.” In webservers, I use them to pop on the scene, deliver some content, and then die with an arrow through the back.)
Basically this procedure calculates an object id and method from a combination of the URL and webform data passed in by the webserver. It then summons an object into being. It calls a method from the object, and stores a result. At the end it destroy the object, and deliver the result back to the caller.

Because the object isn’t actually storing any data, we aren’t actually losing anything by the object’s destruction. And the next time we call up that record/page/whathaveyou it is free to morph into another class entirely.

All of these classes used the same basic fields, but which field who could edit changed throughout the record’s lifecycle.

You’ll notice there was an object I didn’t properly explain called *webConObj*. It is of a class I like to call a containers. The idea is that every object system needs some permanent objects. Something for everyone else to call, and who will always “be there.” If it can handle a few other jobs as assigned, even better!

A container’s principle job, however is to spawn of “nodes”. In most implementations they are also the node’s primary way of accessing the data back end. And they do this by providing two methods that complement the node’s “get” and “put”. They are “nodeget” and “nodeput”.

**nodeget** and **nodeput**, as you see, look and act just like the node’s own get and put methods, but they take an addition argument that tells the container which node.

Now this example isn’t particularly clever because all we do is give our spawned nodes a copy of the data we have stored. To be really powerful, we need to redirect their get and put statements to address the container directly.

Now I’ve tried a few different techniques, but the one the works best takes two complimentary classes. One the container, one the node. The container passes it’s name and a reference id to the node. The node uses this to bootstrap itself back into the container.

You’ll note, that I’m using a parlor trick from TclOO called “forward”. Forward allows you to redirect a method call to somewhere else. Essentially, a call

```tcl
proc pageDeliver {url webformdata} {
    getWho $url \ $webformdata wObject wMethod
    set obj [webConObj spawn$Object]
    set content [\ $obj $whichMethod $webformdata]
    $obj destroy
    return $content
}
```

```tcl
oo::class create wall {
    superclass strawbail
    method spawn {nodeid} {return [::brick create \ [self]/$nodeid \ [self] $nodeid] }
    method attach {object nodeid} {
        oo::objdefine $object method nodeid {} \ [list return $nodeid]
        oo::objdefine $object forward \ get [self] nodeget $nodeid
        oo::objdefine $object forward \ put [self] nodeget $nodeid
        oo::objdefine $object forward \ containerObj [self]
    }
}
```

```tcl
oo::class create ::brick {
    superclass strawman
    constructor {conobj nodeid} {
        $conobj attach [self] $nodeid
    }
}
```

```tcl
oo::define strawbail {
    method spawn {nodeid} {
        set dat [my nodeget $nodeid]
        return [::strawman create \ [self]/$nodeid $dat] }
    method nodeget {
        nodeid {fieldname {}}
    }
}
```

```tcl
variable objNodes
```
to the brick’s “get” method is actually a call to the wall’s “nodeget” method, with the argument that tells “nodeget” which node is there.

Observe:
That same brick class will also work happily if it’s tied to an SQL backend.

\[\%
\text{wall create wallContainer}
\%
\text{wallContainer nodeput 1 \{somevalue 10\}}
\%
\text{set brick1 \[wallContainer spawn 1\]}
\%
\text{\$brick1 put \{someothervalue 20\}}
\%
\text{\$brick1 get somevalue}
\> 10
\%
\text{wallContainer nodeget 1}
\> somevalue 10 someothervalue 20
\%
\text{sqlite3 db :memory:}
\%
\text{db eval \{}
\%
\text{create table store \{}
\%
\text{nodeid integer,}
\%
\text{field string,}
\%
\text{value string,}
\%
\text{primary key(nodeid,field) \}}
\%
\text{\}}
\%
\text{oo::class create sqlwall \{}
\%
\text{superclass wall}
\%
\text{method nodeget \{nodeid \{fieldname \{\}\}\} \{}$
\%
\text{if \{ \$fieldname != \{\}\\} \{}$
\%
\text{return \[db one \{}
\%
\text{select value from store where}
\%
\text{nodeid=$nodeid and field=$fieldname }
\%
\text{\]}
\%
\text{return \[db eval \{}
\%
\text{select field,value from store where}
\%
\text{nodeid=$nodeid }
\%
\text{\]}
\%
\text{\}}
\%
\text{method nodeput \{nodeid keyvaluelist\} \{}
\%
\text{variable objNodes}
\%
\text{foreach \{key value\} \$keyvaluelist \{
\%
\text{db eval \{}
\%
\text{insert or replace into store}
\%
\text{(nodeid,field,value) VALUES}
\%
\text{($nodeid,$key,$value) \}
\%
\text{\}
\%
\text{\}}}
\%
\text{\}}}
\%
\text{sqlwall create wallContainer}
\%
\text{wallContainer nodeput 1 \{somevalue 10\}}
\%
\text{set brick1 \[wallContainer spawn 1\]}
\%
\text{\$brick1 put \{someothervalue 20\}}
\%
\text{\$brick1 get somevalue}
\> 10
\%
\text{wallContainer nodeget 1}
\> somevalue 10 someothervalue 20
\%
\]

sqlwall is a modified wall. All it changes is where nodeget and nodeput get and store their data. In this case, instead of a dict, they are storing data to an sql table. As a “wall” sqlwall will still spawn off nodes of the brick type. I have not modified the brick class in any way. Nor have I modified how the wall class initializes a brick. Let’s see how it behaves:

(Tadaa) Exactly the same! Thank you

Tales from the front:
I found disposable objects a very useful state of affairs for a workorder system at the Franklin Institute. A record started off as a “report”. Once a report had been reviewed, it became an “assignment”. The completed assignment became a “completed assignment.” A completed assignment could order could be filed away into the archives as “closes”, or re-punted as an “assignment” if some issue with the workmanship needed addressing.

Different state had different ways of being displayed. They also had different work rules about who was allowed to edit what.

In the end it was easiest to represent each state as a class. Every pageview the record would save, and the next pageview, which “class” the record would be next was recalculated.

very much ladies and gentlemen.

Other tricks to remember in TclOO

There are some other things you need to know in TclOO. They are random,
capricious, and really would to have had
to have been there to understand they
whys and hows.

**Little Letter First**
By convention, TclOO treats all
methods that start with a small letter as a
public method. It’s actually a very nice
convention. It will get you into trouble if
you copy and paste Itcl code. (Not that

The way Itcl handles it is to exhaust
its local repertoire of methods before
going out to the world. Having built my
own object systems from scratch (and
who here in the crowd hasn’t?) I know
that this process is somewhat expensive.
Especially if you are doing for every line
of method code.

By using a “my” operator, TclOO
manages to avoid all this overhead and as
an added bonus run all of your method
code more or less bare inside the
interpreter. A call to a global command
costs the same a call to a local method.

**My Little Core Hacks**

**Constant Strings**
This is more a dict hack than a TclOO
hack, but because I’ve gone on at great
length about how Dicts can save the
world, it’s not a bad place for it.

I have a little trick I use for large
simulations to conserve memory. It’s in C,
and if I’m not careful will probably be
TIPed by the end of the conference.

I noticed that I was storing the same
strings over and over again as field names.
And thought I, “how many copies of those
do I actually need?”

So, with a little bit of playing, I came
up with a quick new command
“constant_string” constant_string will
take in a string. It searches through a list
of strings objects it already knows. If it
finds it, it increments that reference count
of the matching tcl object, and returns the
pointer to that tcl object as the return for
the function. If it does not find that string,
it makes a new tcl object, and adds it to
the list with a refcount of 1.

One simulation of mine, with about
35,000 nodes went from consuming 95mb
of ram to a little over 19mb in one go.
While I do use the same technique on the
C level, it’s also handy on the tcl level.

```
dict set inmemdb \ 
    [constant_string $field] $value
```

And it’s usage is as simple as:

What happens behind the scenes is that the value of $field is replaced by a pointer to an existing Tcl_Obj. It’s still there, but after the first copy, it’s no longer taking up any space. Have 30 records with the same field, and the 30 copies will be pointing to the same Tcl_Obj data structure.

**Conclusion**

Well, I hope you found something useful in all of this. As for me, I’m realizing there is a need for a library of these design pattern in TclLib. But in the meantime, all of the code, and examples as to how they are used are available on my website:

http://www.etoyoc.com/tao
Program Listing: C implementation of a `constant_string` command

```c
int constantCount;
Tcl_Obj *ConstantList;

/* Return a constant version of a string */
Tcl_Obj *constant_stringObj(Tcl_Interp *interp, const char *newName) {
    int nStrings, i, result;
    Tcl_Obj **stringObj, *newObj;
    char *zName;
    Tcl_ListObjGetElements(interp, ConstantList, &nStrings, &stringObj);
    /* Search through our list, drop off when we get past what string is alphabetical */
    for(i=0; i<nStrings; i++) {
        zName=Tcl_GetStringFromObj(stringObj[i], 0);
        if(strcmp(newName, zName)==0) {
            Tcl_IncrRefCount(stringObj[i]);
            return stringObj[i];
        }
        if(strcmp(newName, zName) > 0) break;
    }
    ConstantList->refCount=0;
    newObj=Tcl_NewStringObj(newName, -1);
    Tcl_IncrRefCount(newObj);
    /* Give me an extra one... just in case */
    Tcl_IncrRefCount(newObj);
    result=Tcl_ListObjReplace(interp, ConstantList, i, 0, 1, &newObj);
    ConstantList->refCount=100;
    if (result != TCL_OK) return 0;
    return newObj;
}

static int constantMapCmd(
    void *pArg,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *CONST objv[])
{
    char *newName;
    Tcl_Obj *result;

    if(objc != 2) {
        Tcl_WrongNumArgs(interp, 1, objv, "string");
    }
    newName=Tcl_GetStringFromObj(objv[1], 0);
    result=constant_stringObj(interp, newName);
    if (!result)
        return TCL_ERROR;
    Tcl_SetObjResult(interp, result);
    return TCL_OK;
}
```