The TCL Architecture of Objects (TAO)

Presented at the 13th Annual Tcl/Tk Conference, October 9-15, 2006,
Naperville, IL

Sean Deely Woods
Senior Network Engineer
The Franklin Institute Science Museum
222 N. 20th Street
Philadelphia, PA 19147
Email: yoda@etoyoc.com
Website: http://www.etoyoc.com/

Abstract

Technical design is only half of a successful product. User interface is an important, but oft overlooked component of design. This lack of design is particularly acute in products that are geared for developers. TAO is an attempt by Sean Woods, to develop an architecture that is more than a rapid prototyping system. It is a rapid developer immersion system as well. TAO is an programming architecture that is designed by developers for developers. It is primarily designed for ever-changing systems like web content. This paper will address both the technical and the psychological strategies employed by TAO to deliver web content to staff at the Franklin Institute Science Museum.

1. Introduction

In response to a complex project, meager to devote to it, and my own personal demons I have developed a programming strategy that I believe will help all programmers of TCL. I call this strategy TAO, the TCL Architecture of Objects. It's a little OOP, a lot of organization, and a bit of a wizard's touch in places.

TAO employs a strategy that I call "Naturalistic Programming." It shamelessly steals principles from Taoist philosophy. It exploits natural processes, manages action through inaction, and de-emphasizes ridged structure. TAO harnesses the power of the computer to chase details, and leaves the human free to be creative.

1.1 The Diamond Problem

If TAO had to overcome one problem and nothing else, it was the Diamond problem[1]. All object systems run into it eventually. And my code ran into it immediately.

Figure one illustrates the problem. We are trying to create class A from class B and C. The difficulty lies in that both B and C are decedents of a common ancestor, D. Our brains grasp the solution intuitively, "It's both". But a hierarchy of doesn't like ambiguity.

Many OOP systems only have a workaround for the Diamond Problem. C++ and Java punt the issue by making one go through and Interface[2] or virtual class. [Incr Tcl] simply curls up and dies. Many other systems mumble about it, and then proceed to act in their own way.[3]

One day I had an epiphany. The problem was not that my objects failed to fit into a simple hierarchy. The problem was that a simple hierarchy did not fit my objects.

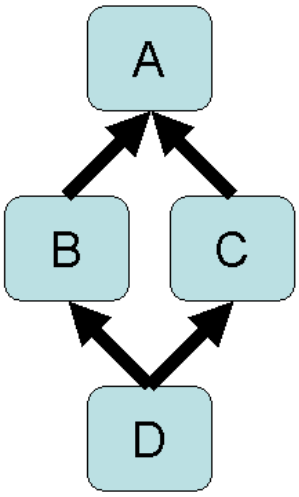

Figure 1 - The Diamond Problem

1.2 Yggdrasil

Developers routinely throw around tree structures as simple constructs of recursion. They have a single root that bifurcates into a myriad of trunks and branches. And it would work except for the fact the no structure like that exists in nature.

Things with a definite beginning and ending in nature don't branch. A worm has a beginning and an end. Life has a beginning and an end. A string has a beginning and an end. The two ends balance one another.

A tree structure in nature never branches in a single direction. What developers call the "root" is actually the dividing line between the branches above and the real roots

beneath. The roots themselves bifurcate into a myriad branches. Trees have a balance between what is above and below the ground.

The idea of organizing the world around a tree structure is not a creation of modern computer science. The Norse believed that the entire world is built on a giant tree, the Yggdrasil[4]. Its branches are so tall, and roots so deep, that they meet one another. Yggdrasil joined together the underworld with our world and the heavens above. Our world was a disk wrapped around the trunk. Similar "World Tree" myths exist in Hindu[5], Siberian[6], and even Mayan[7] culture.

Figure 2 – Yggdrasil

The computer science notion of a tree is primitive even by ancient standards. But the ancients were on to something. The human mind is very good at picking a start point, any point, and working its way out.

If I was to have any hope of mapping reality in code, I had to devise a way to pick an arbitrary starting point and work my way out in multiple directions at once.

One approach used by other systems to solve this problem is to implement the structure as a network stored in a relational database[8]. With a relational database, I could start with the class I intended to create, and let my search radiate out to all of the branches and roots of the class network. There is no structure inherent in a database. That is part of its charm. I only have to make up the parts of the "tree" that I would actually be using.

2 Overview of TAO

Building code from snippets stored in a relational database sounds cool. Except that developers need some frame of reference to be able to make sense of it all. We don't like free for alls. We want a cozy set of rules to work with.

Most of my class libraries were written in [Incr Tcl], and I rather liked the way itcl code looks. So I used itcl's style as the basis for TAO's programmer interface. Incr Tcl code will port over to TAO, but developers need to be aware of some important differences.

Firstly, TAO does not respect data hiding. I find that it makes problems hard to track down on large systems, so I don't use it. And I don't code what I don't use. This, or any other feature, can be added a later time if there is sufficient demand[9]. For now the *public* and *private* keywords are used by the TAO interchangeably.

Instead of data hiding, TAO has two types of non-volatile variables for use in object code. A **symbol** is a map between a local variable name to a globally stored variable. A **static** is a copy of a global value as a local variable. Because it is a copy, changes to that local variable are not reflected in the global value. See the developer's manual section of the paper for an in-depth example of how **symbol** and **static** work.

2.1 SQL Backend

TAO source code is never run directly. It goes through a decompile stage where it is torn apart, and the pieces stored in several sqlite tables. Because this digestion process takes a not-insignificant amount of time, TAO provides a shortcut. Instead of sourcing a file directly, TAO checks the time stamp on the file and compares it to a time stamp on record. If the two match, the system assumes that the information already in SQL is good and TAO moves on to the next file. This is essential on large systems with hundreds of class files.

Class meta-data is retained between runs of a program. And if that were not good enough, unused classes to not consume any memory in the active process, and multiple processes can all share the same database. Conversely, applications making changes to the class structure that are incompatible with one another can be given their own separate databases.

The exact file location for TAO's meta cache is presently hard coded for each platform TAO operates under. The location selected is one that is universally readable and writeable. If the global location is unavailable, TAO falls back to hard coded directory in the users's home path.

TAO creates two sqlite databases. One holds class metadata, the other hold object and environment metadata. Where collisions are expected between applications, temporary tables are specified in the definition. This keeps them in memory within the running application and not on the disk-based store.

2.2 Classes and Objects

Classes are not called into being until they are used by an object. Once "compiled" each class is given a unique namespaces populated with procedures that implement its methods.

The exact namespace for a class can be obtained with the ::tao::class_namespace command.

The secret ingredient that makes TAO work is the insertion of one line at the top of every method: ::tao::peek The function grabs the top item on the TAO object stack. It then runs a script that will map local variables to global ones.

TAO uses a stack so it can handle objects recursively calling other objects, even if multiple objects are of the same class.

Objects exist in three parts:

1. A global array that stores the non-volatile variables. Hashes are stored as separate global arrays.
2. A state script that is indexed and stored internally by the TAO system
3. A procedure that manipulates the stack and routes code to the proper class namespace.

```
::tao::clobject networkSql {
    inherit ::sqlcon::mysqltcl
} –db_user user –db_pass pass –database network

networkSql query “select now()”
> 2006-10-01 18:36:44
networkSql::query “select now()”
> 2006-10-01 18:36:45
```

Example 1 - clobject usage

"Singletons," classes that exist as only one object, are handled through a special construct called a clobject. clobjects get their own namespace (the same name as their object) and instead of referring to the stack their methods know to load only data for them. The neat part about singletons is that you can access them either through their interface procedure, or by making a direct call to their methods in their namespace.

2.3 TAOisms

Now that we have the basic operations of TAO under out belt, I would like to explain some of the tools that are unique to TAO.

2.3.1 Stacked Inheritance

Hierarchies are not static in TAO, they are built on the fly from a database search. Up until a class is called for by an object, it can be made and reshaped. The same class can mean different things in different applications, yet retain the same look and feel to you, the programmer. You can take any two classes, throw them in the blender, and get out a new class that is a mix of both.

How does TAO do it? It waits until the absolute last minute to implement a class. When it finally builds a class, it does not try to sort through ambiguity until all the cards are on the table.

Each competing definition for a method or variable is stacked on top of the another. The most distant ancestor on the bottom of the stack, and any code defined by the classes' own definition on top. When selecting building the body of a procedure, only the top level is used. The same goes for defining a variable.

2.3.2 *consecution* and *mesh*

What good is having your code in a stack if you can't do nifty tricks with it?

You have a new class that wants to add an extra behavior to an existing method of the parent class. In classic [Incr Tcl] you would have to copy and paste the old method definition and append your new code. Or call **chain** command. While **chain** is implemented in TAO, it was very expensive to call in early versions of the system.

I developed the **mesh** method type to eliminate the need to call **chain** for incremental changes to the body of the parent. Mesh is like a method, but instead of replacing the code on the top of the stack, it appends to it. The developer can control whether the new code is inserted at the beginning or the end of the parent method. In *Example 2 - mesh usage and resulting body* I demonstrate the process of creating a **mesh** as well as the resulting body of the final method. *Example 3 – [Incr Tcl] equivelent to mesh usage* shows the same code as Example 2 using [Incr Tcl] notation.

```
tao::class ::foo {
   method validate {a b} {
      if { $a > 100 } {
         error “$a is out of bounds”
          }
      if { $b < 10 } {
         error “$b is out of bounds”
      }
   }
}

tao::class ::bar {
    inherit ::foo
    mesh validate {a b} {
        if { $a > 10 and $b < 30 } {
          error “Invalid combo of $a and $b”
      }
   }
}
```

```
>::tao::compile_class ::bar
>info body [::tao::class_nspace
::bar]::validate
{
::tao::peek
if { $a > 100 } {
   error “$a is out of bounds”
}
if { $b < 10 } {
   error “$b is out of bounds”
}
}
if { $a > 10 and $b < 30 } {
   error “Invalid combo of $a and $b”
}
}
```

Example 2 - mesh usage and resulting body

```
itcl::class ::foo {
   public method validate {a b} {
      if { $a > 100 } {
         error “$a is out of bounds”
          }
      if { $b < 10 } {
         error “$b is out of bounds”
      }
   }
}

itcl::class ::bar {
   inherit ::foo
   public method validate {a b} {
     chain
       if { $a > 10 and $b < 30 } {
         error “Invalid combo of $a and $b”
      }
  }
}
```

Example 3 – [Incr Tcl] equivelent to mesh usage

mesh isn’t perfect for every situation. If a parent class returns a value the child’s code will never be called. **mesh** is good for methods that don’t return a value, or simply error check.

To return a value from a **mesh**, I created the **consecution** method type. **consecution** is a stylized form of **mesh** where the results of the child class are appended to the end of common variable, result. Note that a return statement or uncaught error in the parent method will prevent the child’s code from ever running, just like the mesh. But, you can at least build on the value that the parent created.

Like a mesh, a chain definition can take two bodies. If a second body is given, the first is knitted in before the ancestor main bodies. This is particularly helpful to map new arguments to old names should they change.

Example 4 – consecution usage and resulting body demonstrates a sample consecution construct, as well as the body of the method that results. *Example 5 – [Incr Tcl] equivalent to chain usage* shows the equivalent code to *Example 4* in [Incr Tcl] notation.

```
tao::class ::foo {
   method bard {} {
      set result “It was the best of times”
   }
}

tao::class ::bar {
   inherit ::foo
   consecution bard {} {
      append result \n \
        “It was the worst of times”
  }
}
------
>::tao::compile_class ::bar
> info body [::tao::class_nspace
::bar]::bard
{
   tao::peek
   set result {}
   set result “It was the best of times”
   aooend result \n \
       “It was the worst of times”
   return $result
}
```

Example 4 – consecution usage and resulting body

```
itcl::class ::foo {
   public method bard {} {
      return “It was the best of times”
   }
}

itcl::class ::bar {
   inherit ::foo
   public method method bard {} {
     set result [chain]
    append result \n \
        “It was the best of times”
     return $result
   }
}
```

Example 5 – [Incr Tcl] equivalent to chain usage

3 Programming Practices for TAO

All of this technology is for naught if we don't practice a few sensible strategies for optimal code. These guidelines have been collected from my own experience, and have no doubt been published elsewhere[10]. I list them here because TAO is designed specifically to exploit them.

These tips are not specific to TAO. They are a good idea in any environment. They make your code easier to read, easier to debug, and easier to maintain in the future.

1. Keep the details fuzzy

Unless you have a specific performance reason for doing so, never use a hard coded value where a variable will do. Every one of those values, even physical constants, should be changeable from the developer interface. If practical, provide a mechanism to input changes to the assumed values with a configuration file.

2. Start general, add specifics

Make as few design assumptions as possible. Start with a general case, and build onto it the specifics. You don't need to implement every possibility of the general case, only what you are using. Unimplemented cases should be documented and throw a polite error to the developer if he or she tries to access something that hasn't been defined yet.

3. All code is a singular project

All of your code must have a common protocol for data exchange, especially for conversations between modules. The best case is the most general. Calls that are expected to converse with outside code takes one and only one argument: a key/value list. This key value list can then be sanity checked. Any missing detail can filled in. Extra information can be discarded. The calling function does not need to bother itself with having to remember which argument goes in what sequence.

4. Differentiate between interface and process

To properly handle the big picture, all code in a library should be segregated into two types: Interface code and Process code. Interface code should do one and only one thing: translate outside protocols to your internal protocol. Process code should speak only in your internal protocol. If process code is returning data to the outside it should produce its output in your internal protocol and then pass the result to an Interface.

5. Write once, Use everywhere

If you find yourself copying the same routine in multiple places, *stop*. Code the sequence as a subroutine and replace all the copies with a subroutine. This reduces mistakes. It reduces line count. It has a negligible impact on TCL performance. But most of all, it allows later developers to follow your code easier if they aren't having to glance through pages and pages of similar looking code. As a practice I regularly patrol my library for code snippets that can be consolidated.

6. Be the Programmer, not the Computer

Computers are good at repeating things. If you find yourself tapping out repetitive sounding code, *stop*. One of the supreme powers of TCL over any other code paradigm is the ability to machine to write its own code on the fly. If there is a pattern, teach the computer the pattern and let the machine do your work.

I have written many a system that uses a foreach loop with code snippets to assemble procedures on the fly. This has a twofold impact on your productivity. First: you don't have to sit down and write out 10 different iterations of the same procedure. Second: when you inevitably have to modify this subroutine you only have to alter one snippet.

Having the computer code it's own procedures has no impact on TCL performance. The result is code that is just the same as if you had written it yourself.

*Remember**: **proc** is a command like any other in TCL.*

4 Applications

TAO is a pretty useful general purpose tool. Being able to just sit down an tap out object code without having to learn to tapdance around limitations opens TAO up to many potential applications.

4.1 Present Uses

4.1.1 Taohttpd

Taohttpd is a dynamic web content engine used for several websites, including several Intranet sites at the Franklin Institute and the Camping Committee for the Philadelphia Folk Festival. Taohttpd extends the Tclhttpd to intelligently wrap a web server around TAO objects.

4.1.2 Preen

Preen is a Unix cluster management suite I use to manage our servers and workstations at the Franklin Institute. Preen exists as a library of standalone scripts that use TAO as a common library and interface toolkit.

4.1.3 TFINET

TFINET is in the early stages of development. It is intended to deliver a dashboard interface to Intranet content, RSS, Instant Messaging, and directory information to staff at the Franklin Institute. It will provide a Tk gui interface to many of the same resources that are accessed through the Taohttpd.

TFINET will also push updates to workstations and track the movement of users and machines around the network.

4.2 Future Uses

4.2.1 Toy Languages

TAO code is simple to write, almost to a fault. The fact that its notation is written from a human perspective with the computer playing autopilot makes it ideal for teaching. Instructors can provide ready-made classes for students to manipulate, and the details of operation can be obscured. As a student becomes more advanced, the instructor can lift the veil and expose him or her to different levels of the inner working of the system.

4.2.2 Game Scripting

Like Toy Languages, game developers often want to provide a simple way for uses to customize a system, while not overwhelming the modder with detail. If a user wants to code a game AI, they often have a specific behavior they want to modify. Most modern API’s require the developer to begin with a copy and pasted template that includes detail he or she is not interested in dealing with.

Under TAO a user can say “start here, and build up.” At the core changes the user code is less likely to break. They aren’t copying and pasting a lot or code that will become outdated in the next release.

Finally, other hackers can use the original mod as a springboard for their own work.

4.2.3 Artificial Intelligence

Because TAO code is built form database searches, it is ideal for open-ended development projects, particularly where the “Developer” is in fact a machine. TAO provides a consistent framework on which to combine ideas together into a meaningful way without imposing structure. Because the “code” is stored in a database, the computer can easily add to it’s programming based on experience, or manipulate its data store to produce novel solutions.

Suppose we want to have an expert system that reads a paragraph of data, and then answers questions about what it has learned. Each noun in the paragraph can be an object. As the object is described, it inherits attributes from a host of concept classes. In addition to data, host concepts can include rules for answering specific questions.

5 Limitations

TAO is a leap forward in software development. But it may not be for everyone. There are a few wrinkles inherent in the design that may not work for your situation.

5.1 TAO requires sqlite3

TAO requires sqlite3[11] to operate. All of it's indexing is built around the search features of sqlite. It makes extensive use of sqlite's "INSERT OR REPLACE" function to keep line count low. The TAO class system makes extensive use of direct calls to the sqlite engine, and sqlite3 in particular.

If your application environment does not have access the most recent sqlite, TAO will not work for you. While I find the package to be ubiquitous, I can understand situations where having to load sqlite is either operationally impossible or a logistical nightmare.

5.2 TAO uses disk-based storage

TAO's sqlite database is written to disk. It is a simple matter to switch to memory based tables, but in my experience the RAM usage is prohibitively high.

5.3 TAO does not support data hiding

The present version of TAO does not support data hiding. While it will accept the "public" and "private" keywords from Itcl, it does not honor them. It could be added, but it does not do it today.

5.4 Compatibility with Itcl is superficial

While TAO code is made to look like [Incr Tcl], and it implements generic design patterns quite easily, if your code makes a lot of calls to the Itcl internals TAO is not going to work without a substantial rewrite. While I found that TAO is much simpler to write than the equivilent code in Incr Tcl, if you have a pile of existing Incr Tcl code you may find it challenging to port.

TAO uses a different database backend, so it is not possible at the present time to have Incr Tcl code inherit TAO classes or *vice verse*. That said, there is nothing that would keep you from running TAO and [Incr Tcl], or SNIT and XOTcl for that matter, inside the same application.

5.5 TAO was written for Tcl 8.4

TAO was written to utilize features and methodologies present in the Tcl 8.4 core. It uses the *file normalize* and *string map* commands that were not present in earlier versions of 8.x. It uses namespaces which were not present before Tcl 8.0. I am aware of several operating environments that bundle 8.3 with the base distribution. For TAO to operate in them they must be upgraded to Tcl8.4, or I have to sit down and backport TAO.

6 Future Directions

The [Incr Tcl] model is just one of the many syntactic systems in use to bring objects to TCL. I chose Incr Tcl based on personal preference, but there is nothing that would prevent a developer from writing a class parser that digested code written in the style of other OOP languages.

7 Acknowledgments

If I can see far, it is because I stand on the backs of giants. There were many influences in the creation of TAO. I owe a huge debt of gratitude to the developers of [Incr Tcl]. While I may not have used their code, the interface and feel of TAO is taken right from Itcl.

I would also like to thank Pete Stein. He was a volunteer and later intern at the Franklin Institute who was patient enough to learn TAO and use it in his own code. Pete was also an excellent foil to evaluate ideas throughout the development of TAO. I learned a lot by watching how he used the system.

I would like to also than Swapna Saireddy, my former assistant at the Franklin Institute. Like Pete, she too developed applications for TAO. She got things to work without the benefit of a manual, and had to listen to me explain a lot of “insanely GREAT!” explanations as to why the system had suddenly and unexpectedly been redesigned.

I would also like to thank my employer, The Franklin Institute for giving me the freedom to develop TAO and use it for our in-house applications. I would also like to thank all of the users of my system, both that the Franklin Institute and the Philadelphia Folk Festival. There was a lot of trial, error, and transition during the development process of TAO. Without their patience as I worked problems out, along with their expectation that the system actually work, TAO would be just an idea.

I would also like to thank Brent Welch. His book *Practical Programming in Tcl/Tk* was an excellent reference for both me and my development team. Having a printed manual and tutorial that I could plop down in front of new staff and say “learn this” was invaluable.

Finally, I would like to thank my wife, Sara for putting up with me disappearing into the basement for days at a time putting this paper together, as well as proofreading the early versions of this text.

8 Availability

TAO and its supporting documentation are available for download from:

http://www.etoyoc.com/

References

[1] The Diamond Problem, Wikipedia, http://en.wikipedia.org/wiki/Diamond_problem
[2] Designing with Interfaces, Bill Venners, http://www.javaworld.com/jw-12-1998/jw-12-techniques.html
[3] The Diamond Problem, Answers.com, http://www.answers.com/topic/diamond-problem
[4] Yggdrassil: The World Tree, Encyclopedia Mythica, http://www.cauldronfarm.com/nine/yggdrasil.html
[5] Parallels Between Norse and Indic Creation Myths, *Michèle P. Rousseau*, http://www.rousseau-writer.com/myths.htm
[6] Parts of the true story of a world picture, http://www.nbi.dk/~natphil/Siberian.html
[7] Raising The Sky: The Maya Creation Myth And The Milky Way, http://members.shaw.ca/mjfinley/creation.html
[8] A similar approach was described by Tony Martson in his essay "A flexible Tree Structure" http://www.tonymarston.net/php-mysql/tree-structure.html
[9] Sufficient Demand: (n) the desire of a user to have a feature coupled with a patch submitted by said user to implement it.
[10] Further tips to enhance Tcl can be found at, http://wiki.tcl.tk/348
[11] Sqlite, and embeddable SQL engine, can be found at: http://www.sqlite.org/