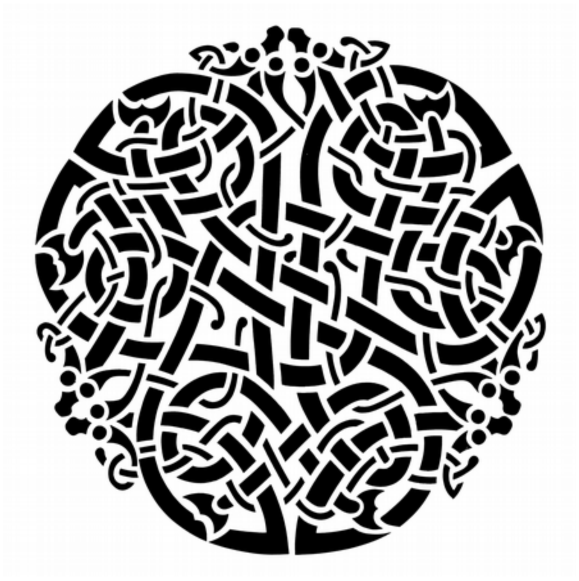


Practcl

Package Repository Automation for C and Tcl

Presented at the 23rd Annual Tcl Developer's Conference (Tcl'2016)
Houston, TX
November 14-18, 2016

Sean Deely Woods
Senior Developer
Test and Evaluations Solutions, LLC
400 Holiday Court
Suite 204
Warrenton, VA 20185

Version 1.0.1

Abstract

Practcl is a library of tools for assembling Tcl distributions, from within a Tcl script. It utilizes exec combined with an object oriented recipe system to download, configure, compile, package, and install extensions. Practcl also includes a build system, a markup language for generating C code, and facilities to link executables, dynamic libraries, and static libraries.

Problem Statement

Any sufficiently developed Tcl based application eventually turns into its own Tcl distribution. Even the Tcl core itself is a Tcl distribution[1]. Many of us are fortunate enough to be able to farm out the job of maintaining our distribution. Be it to the local Operating System, or ActiveState, or KBS, or KitCreator. But when you sit down to compile any of the big boys, Aolserver, BRL-CAD, Androwish, or the like, you'll see that they build everything from the core on up by themselves. The product I work on, The Integrated Recoverability Mode (IRM) does this as well.

Building extension distributions for Tcl is a dark art. In theory every extension builds the same way:

```
./configure ; make install
```

But that's only really useful for creating extensions to be installed in the local operating system. And only on Unix[2]. And at no point does the standard address publishing what the software is, nor what it depends on, nor a myriad of other questions a high level integrator needs to know.

More than information, much of the distribution maker's job is building a procedure. With a conventional distribution, that automation can (and usually does) extend into 3 or 4 different languages, and generally have to make quantum assumptions about the environment, the build tools available, and the intended target for the build.

A lot of institutional knowledge that goes into building a Tcl distribution. Practcl is my effort to assemble vocabulary for expressing that knowledge into a human edited, machine executable way.

What is Practcl?

The *Package Repository Automation for C and Tcl* (Practcl) is a model-based toolbox for assembling Tcl extensions. For portability it is written in Tcl. And for sanity it was written from scratch to utilize native support for TclOO. I set out to create something that fulfills the following goals:

1. Ships as a single file.
2. Is retrofit-able into existing TEA projects.
3. Plays well with others

Practcl ships with tools to perform the following tasks:

1. Download source code from Fossil, Git, Tarball, or Zip
2. Configure and compile extensions based on the following build systems: TEA3, Critcl, Swiss Army Knife (SAK)[3], Kettle, Practcl
3. Wrap Extensions for packaging in the TEAPOT.
4. Compile specially prepared extensions to be statically linked into a kit
5. Build self-contained Tcl based executables. (Using Zipvfs.)
6. Compile build products, libraries, and executables via exec calls directly from Tcl.

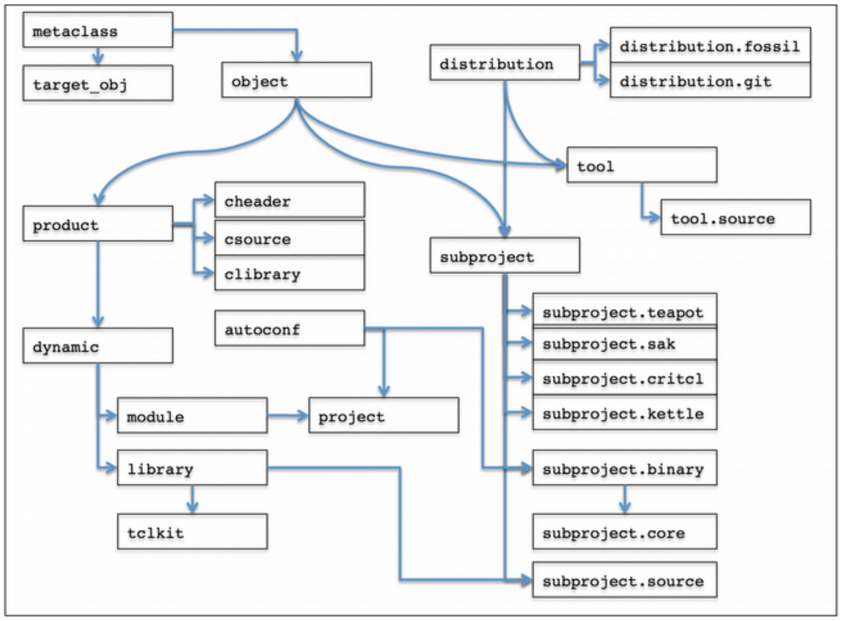
1 See: TIP#50, TIP#364, TIP#376

2 Native windows builds use something called *nmake*.

3 The installer for Tcllib, Tklib, Taolib

Practcl Internals

Practcl is made up of 67 procedures and 26 classes, expressing 179 methods. At last count, about 4632 lines of code. That may seem like a lot, but the Tcl.m4 file is around 4300 lines. The configure files that tcl.m4 produces are on the order of 11000 lines. And the *clock* command in the Tcl core is actually implemented in a little over 5000 lines of code. So its not small, but as useful utilities go, it's not overly huge either.



Practcl Class Hierarchy

My first effort to jot it all down everything there was to know about Practcl was about 30 pages (and counting). So I cut it down a bit. Parties interested in the nitty gritty details can download the complete “Manual In Progress” at:

```
http://www.etoyoc.com/tcl/Practcl-Manual.pdf
```

The main operating principle of Pratcl is that an elephant is eaten one bite at a time. Each object spawned by Practcl is a bite out of the elephant. The most complex behaviors are exhibited by the most primitive of elements. The Practcl Metaclass, the mother of all objects, has lot of methods for expressing how objects relate to one another. Because all of the objects share a common architecture, and expectation of behavior, coordinating them is somewhat easier.

The classes that users call in scripts are the leviathans at the top of the hierarchy. They are seemingly simple because their job is mostly to set in motion actions carried out by their subordinates. Many actions in Practcl involve a top level object calling a herd of subordnates, who recursively call their subordinates, and so on until an answer begin to bubble from the end points of the system back to the trunk.

Another principle of the Practcl architecture is divine procrastination. Practcl waits until a question is asked before it starts assembling the answer. And it waits as long as possible before asking a question. The effect is that vague problems that require vague answers are quick, easy, and computationally cheap to solve. And when a complex problem requiring a complex answer is answered, the system works from the bottom up to produce a complete and holistic answer.

In the two examples I have prepared (and a third that was left on the cutting room floor), you will see a lot of object calling objects, searches through relational hierarchies, and aggregating data from disparate sources.

Example 1: Implementing TIP#453

TIP#453 is a suggestion on my part that rather than relying on the contents of *pkgs/* to be well behaved TEA extension, we provide for the means for each package to include its own build instructions. And thus open the world to non-TEA extensions. Included in these instructions will be routines for re-constituting the *pkgs/* file system for fresh Fossil checkouts.

The TIP also simplifies a complexity in the core's source code. Currently the facility to assemble and install the contents of the *pkgs/* directory is repeated three times in the Core's build system. It appears once in the Unix makefile, once in the MinGW makefile, and once again in the nmake build file.

Using practcl, we are going to replace about 30 lines of Makefile automation (x3 files) with three lines:

```
packages:
   $(TCL_EXE) $(PKGS_DIR)/make.tcl compile $$builddir

packages-install:
   $(TCL_EXE) $(PKGS_DIR)/make.tcl install $$builddir $(INSTALL_ROOT)

packages-tests:
   $(TCL_EXE) $(PKGS_DIR)/make.tcl tests $$builddir
```

The first step is figure out where we are in the operating system, load the practcl library, and nominate a toplevel object:

```
set ::CWD [pwd]
set ::SRCDIR [file dirname [file dirname [file normalize [info script]]]
set ::TCLDIR [lindex $argv 1]
source [file join $::SRCDIR library practcl practcl.tcl]
::practcl::project create MAIN [::practcl::config.tcl $::TCLDIR]
```

The `::practcl::config.tcl` statement performs a scan for build tool information from *autoconf*, a cross compile script, and/or the local operating system. There are some items our project does need to discover about the local environment. For instance: on windows we need to append **.exe** to any executable we generate. And also, are we building for the local OS, or are we cross compiling?

We will then discover what packages are bundled with this distribution.

```
foreach path [glob [file join $::SRCDIR pkgs *]] {
  if {![file exists [file join $path configure]] && \
      ![file exists [file join $path prac.tcl]]} continue
  set name [file tail $path]
  set obj [MAIN add_project $name {}]
  if {[file exists [file join $path configure]]} {
    $obj define set [list install 1 class subproject.binary srcdir $path]
  }
  if {[file exists [file join $path prac.tcl]]} {
     $obj source [file join $path prac.tcl]
  }
}
```

The block of code above:

1) Ensures the directory contains at least a *configure* script or a *prac.tcl* driver file.
2) Build an object to represent the extension.
3) If the *configure* file is present Assume:
    a. The srcdir of the object is that path
    b. The extension is intended for installation,
    c. The extension uses standard Tcl binary extension behaviors
4) If the *prac.tcl* file is present, have the object read the instructions within the file

With our packages defined, we can now go about the business of doing something with them, in response to what the user requested via the command line. We will define them as procs in the ::do namespace, so that exercising them and argument handling are as simple as defining a new proc.

```
namespace eval ::do {}
proc ::do::compile {TCLDIR args} {
  foreach pkg [MAIN link list project] {
   $pkg define set config_opts -with-tcl=$TCLDIR --with-tclinclude=[file join $::SRCDIR generic] {*}$args
   $pkg compile
  }
}
proc ::do::install {DESTDIR} {
  foreach pkg [MAIN link list project] {
   $pkg install $DESTDIR
  }
}
proc ::do::test {} {
  foreach pkg [MAIN link list project] {
   $pkg test
  }
}
```

And we can just keep tacking on ::do namespace procedures as we want to add more features. In the end we place a command line handler:

```
# Provide a polite handler
if {[info commands ::do::[lindex $argv 0]] eq {}} {
  puts “Unknown command: [lindex $argv 0]
  set list {}
  foreach cmd [info command ::do::*] { lappend list [namespace tail $cmd] }
  puts “Valid: $list”
  exit 1
}
# Pass the arguments to the ::do namespace
::do::[lindex $argv 0] {*}[lrange $argv 2 end]
```

I realize line count isn't everything, but in 45 lines of Tcl code we have eliminated 60 lines of shell script, 30 lines of nmake, and managed to keep the user experience across all of our platforms uniform. It's a slight improvement for managing the tasks that the core's build system already does.

Where the effort pays off is when we have to start adding packages that aren't TEA compliant. Let add tcllib. Tcllib is not a TEA extension. It has it's own build system called SAK. So we'll now have to explain to the core just how to install it. In addition to the Tcllib sources, we will add an additional file call *prac.tcl*. Assuming we didn't have a driver already for SAK, we could write one on the fly:

```
my define class subproject

oo::objdefine self {
  method install DEST {
    set pkg [my define get pkg_name [my define get name]]
    my unpack
    set prefix [string trimleft [my <project> define get prefix] /]
    set srcdir [my define get srcdir]
    ::practcl::dotclexec [file join $srcdir installer.tcl] \
      -pkg-path [file join $DEST $prefix lib $pkg]  \
      -no-examples -no-html -no-nroff \
      -no-wait -no-gui -no-apps
  }
}
```

And even better, that same driver would be useful to any Practcl application.

Example 2: Building a Self-Contained Executable

In this section we will use Practcl to wrap a self-contained executable. This is normally black magic, but with Practcl its… well easy isn't quite the word I'd use to describe it. Perhaps grey alchemy?

Even though we are building a kit, our Pratcl script starts off very much like in Example 1:

```
set ::SRCDIR [file dirname [file normalize [info script]]]
source [file join $::SRCDIR tclconfig practcl.tcl]
set CWD [pwd]
::practcl::tclkit create BASEKIT [::practcl::config.tcl $CWD]
BASEKIT define set name toadkit
BASEKIT define set srcdir $::SRCDIR
BASEKIT define set builddir $::CWD
BASEKIT define set installdir [file join $CWD PKGROOT]
BASEKIT define set prefix   /zvfs
BASEKIT define set sandbox  [file dirname $::SRCDIR]
BASEKIT define set download [::pratcl::LOCAL define get download]
BASEKIT define set teapot   [::pratcl::LOCAL define get teapot]
```

Our main object this time is called **BASEKIT**. This object is both of class `::practcl::tclkit` which is a descendant of `::practcl::library` which is a descendent of `::practcl::project`. The *tclkit* imbues the class with the ability to link an executable and wrap a VFS onto it. The *project* class allows this object to take on deputy objects and have them download, compile, package, etc external software.

This project is a project of projects, and is running without *autoconf*. As such we have to feed it a lot more information than what was fed to our project in Example 1.

The **define** method is an access function to a per-object internal array. Each object can store its own private configuration data, and because **define** is a public method, it can be accessed from other objects as well.

You will see that some of the data that is going into BASEKIT are from methods of an command called `::pratcl::LOCAL.` The `::pratcl::LOCAL` command is actually an object that was created when Practcl was loaded. It's an object of type `::pratcl::project`, but it is rigged to only deliver data that was auto-discovered about the local operating system. The construction process for `::practcl::LOCAL` includes integrating user preferences preferences stored in a platform appropriate user directory.

Tclkits are a modified Tcl shell. We generate our shell from the Tcl/tk sources. Our first deputy we need is one to manage a static Tcl core:

```
BASEKIT add_project TCLCORE {
  class subproject.core
  name tcl
  fossil_url http://core.tcl.tk/tcl
  tag release
  static 1
}
```

Each call to the *add_package* method creates a new deputy object. The first argument will be the name of the project. The second is a key/value list of data that will go into the new project's *define* store. An optional third argument is interpreted as a call to `::oo::objdefine` to modify the object. It's a handy place to replace methods with on-off implementation tweaks.

We can then exercise the new object's methods to build our static Tcl:

```
# Find the platform specific flags needed to make Tcl
set os [BASEKIT define get TEACUP_OS]
set tcl_config_opts [::practcl::platform::tcl_core_options $os]
lappend tcl_config_opts --prefix [BASEKIT define get prefix]
BASEKIT project TCLCORE define set config_opts $tcl_config_opts
BASEKIT project TCLCORE compile
```

Every command that starts with `BASEKIT project TCLCORE` is actually being performed by the object representing the Tcl core.

For this example, we will be statically linking Tk into our shell, but not loading it on startup:

```
BASEKIT add_project TKCORE {
  class subproject.core name tk
  fossil_url http://core.tcl.tk/tk
  tag release
  static 1     autoload 0
  pkg_name Tk  initfunc Tk_Init
}

tk_config_opts  [::practcl::platform::tk_core_options $os]
set _TclSrcDir [BASEKIT project TCLCORE define get localsrcdir]
BASEKIT define set tclsrcdir $_TclSrcDir
lappend tk_config_opts --with-tcl=$_TclSrcDir
BASEKIT project TKCORE define set config_opts $tk_config_opts
BASEKIT project TKCORE compile
```

In our definition for the *TKCORE* you can see that it too is an object of type `subproject.core`. It we also send it the *static* flag. Buy we also introduce three new flags: *autoload*, *pkg_name*, and *initfunc*. Our kit builder will use this information later to compose the bootstrap for our executable. The effect will be linking this extension to our executable statically. But instead of loading it at startup, we will advertise this extension as a statically loaded package, and rig matters such that a `package require Tk` will ultimately trigger the C function **Tk_Init**.

Let us add a few other extensions into our basekit:

```
BASEKIT add_project sqlite {
  tag trunk   class subproject.binary   pkg_name sqlite3   autoload 1
  initfunc Sqlite3_Init   static 1   vfsinstall 0
  fossil_url http://cyqlite.sourceforge.net/cgi-bin/sqlite
}
BASEKIT add_project vectcl {
  tag master class subproject.binary vfsinstall 1 static 0
  git_url https://github.com/auriocus/VecTcl
}
BASEKIT add_project tcllib {
  tag trunk class subproject.sak vfsinstall 1
  fossil_url http://core.tcl.tk/tcllib
}
```

With our packages defined, we can now loop through them, exercising the methods that retrieve the source, compile and install. The extra argument for *define get* allows us to provide a default value if internally it is null.

```
foreach item [BASEKIT link list package] {
  $item unpack
  $item compile
  if {[string is true [$item define get vfsinstall 0]]} {
    $item install [BASEKIT define get installdir]
  }
}
```

And at last we need to construct our new kit's bootstrap and link it:

```
BASEKIT implement $CWD
::practcl::build::static-tclkit \
   [file join $CWD toadkit_bare$::project(EXEEXT)] BASEKIT
```

After that command is run, you will see that Practcl has generated a .c and .h file named after your project. The files are a custom boot process for your new shell. In addition to generating the bootstrap, the *static-tclkit* file takes car of linking the final executable.

When we are ready to wrap our executable we can invoke:

```
file mkdir src/
set fout [open src/main.tcl w]
puts $fout {
puts {Hello World!}
package require Tk ; label .hello -text {Hello World!} ; pack .hello; vwait forever
}
close $fout
BASEKIT wrap $CWD hello hello.vfs src [BASEKIT define get installdir]
```

If we execute this script on the command line:

```
> tclsh make.tcl
> ./hello
Hello World!
```

Distribution

Practcl started as a branch in the Tclconfig project (http://core.tcl.tk/tclconfig), as a possible basis for TEA 4. At present, the master copy is maintained there, and it is currently mirrored to Tcllib and the *tip_453* and *core_zip_vfs* branches of the Tcl core.

Practcl Users

Practcl is currenly employed to build the Integrated Recoverability Model. That system is, unfortunately proprietary. However two of the open source projects IRM relies on also use Practcl:

http://fossil.etoyoc.com/fossil/odie – The Odie Kit builder

http://fossil.etoyoc.com/fossil/odielib – A C implemented accelerator for geometry problems

In addition to those, demonstration of Practcl are working their way into other projects:

http://fossil.etoyoc.com/fossil/sample - A *practcl* branch of the Tcl Sample Extension has been created to demonstrate replacing makefiles with Practcl.

http://fossil.etoyoc.com/fossil/mmtk – An expermiment to amalgamate several major Tk extensions into a single library

http://fossil.etoyoc.com/fossil/tkimg – A rewrite of the TkImg extension to eliminate the TEA extension calling sub-TEA extensions.

http://core.tcl.tk/tcl – The Tcl Core. Practcl is being evaluated in feature branches for tip#453 and tip#430

Conclusion

Practcl is one of those illogically logical solutions to a complex problem. Prior attempts to “normalize” Tcl extensions depended on trying to express them as data structures, or simply trying to express the process of building them as a sequence of instructions. Practcl splits the difference. It stores data in objects, and lets objects implement the procedures. With TclOO, we finally have a language ~~perverse~~ expressive enough to state the similarity between extensions on a high level, yet really get our hands dirty in the nasty details an workaround on the low level.

This is a fun toy, and I can't wait to see what people do with it.

Aknowledgements

Donal Fellows	Without his extensive work on TclOO, Practcl in its present form would not have been possible
Steve Landers, Gerald Lester, Christian Werner	This paper would have been longer, less engaging, and downright offensive to Critcl users everywhere without their review and creative feedback on earlier drafts of this paper
Kevin Kenny	Practcl's facilities for generating TclOO implementations in C are inspired/reverse engineered his work on TDBC.
My Wife and Kids	For giving me things to do in life even more fun the Tcl programming. (As hard as that is to imagine…)
Andreas Kupries	For his tireless efforts on keeping the island of misfit toys together over the years.
Ron Fox	Who is probably going to find and help me fix a dozen ~~tpyos~~ typos in this paper between now and publication

Contact Information

Questions, comments, or winning lotto numbers can be addressed to my personal email: yoda@etoyoc.com