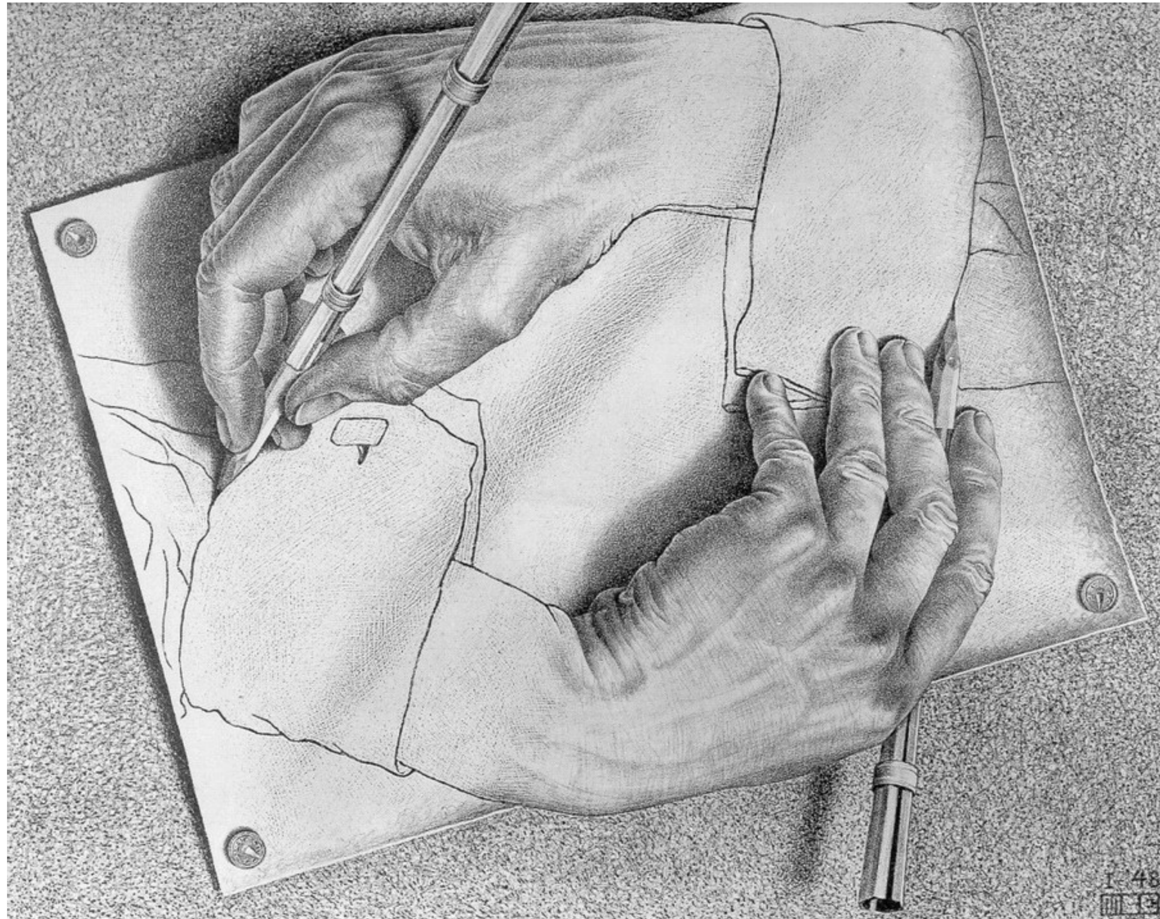
Building Tcl Extensions



A Brief History of Tcl

- Tcl began as an embed-able scripting language for larger applications
- Extending the language was intended from the very beginning

Early Tcl Extensions

- Grab a copy of the Tcl/Tk sources
- Copy public distributed extensions
- Add new functions
- Distribute your executable/library

myprog
- tcl
- tk
- tclx (and other extensions)
- myprog_internals

Pros/Cons

- Anyone with sufficient skill could write a Tcl-ish application
- Adding new functions is trivial

- Upstream changes to the Tcl core could lead to massive rewrites in derivative works
- Duplication of effort
- Key useful functions had radically different implementations

Put More in the Core

- The core team (Dr. Ousterhout and later Sun and Scriptics) would bang together new editions of the Tcl/Tk core every 6-12 months
- New features were contributed by the community, with final implementation by the core team

Pros/Cons

- Tcl picked up a lot of general purpose functions
- It's easier to code in C than hack in autoconf/Make

- Core Developers had to review and reimplement every one of them
- Other extensions still had to track with changes in the core

TEA

- Starting in 1999 an effort was made to standardized build systems and the Tcl API
- 8.1 introduced stubs
  - Extensions were no longer tied to a specific tcl version
  - Minimized the need for access to the Tcl internals
- Scriptics developed TEA: the Tcl Extension Architecture
  - Compiles Extensions for Unix, Window, and Mac

Features of TEA

- Extensions were built as a microcosm of the core build system
- The same autoconf (and later nmake) systems in the core were replicated into TEA
- The core and extensions would publish their API as part of the build process

Pros/Cons

- Building a Tcl extension was exactly like building the Tcl core
- Plenty of shims for developers to modify the build system to suit their needs
- Built around autoconf, a dependible, portable...

- Building a Tcl extension was *exactly* like building the Tcl core
- Plenty of rope for developers to hang themselves with
- ...and unix-only shell script

Critcl

- A build system for Tcl extensions
- Runs inside Tcl
- Can extend Tcl on the fly
- Can build its own wrapped executable from its own source

Cons

-

(Sound of Crickets)

Why Don't More Projects use Critcl?

- It's new.
  - (As much as 14 years old can be called “new”)
- Outside of Tcllib, it doesn't power many extensions that are household names
  - Mostly because those extensions were written pre-Critcl, and many, pre-TEA
- The “Book Store Effect”

The Book Store Effect

- New Users to the language generally buy “a book”
  - Tcl and the Tk Toolkit (Ousterhout & Jones)
  - Practical Programming in Tcl/Tk (Welch, etc. al)
  - Tcl/Tk, A Developer's Guide (Flint)
- These works, the techniques, and design patterns make a lasting impression
- The only mention of Critcl I've found in either of them is a paragraph on page 200 or so in the 4th edition of Practical Programming

This Tutorial

- Installing Critcl
- Building a new project with Critcl
- Building an Extension with Practcl
- Building an Extension with TEA
- Building an Extension with a Bow Drill and Kindling

Installing Critcl

- Available as a ready to run kit or can be built from sources
- Binaries at:
  - http://equi4.com/starkit/critcl.html
- Source code:
  - https://github.com/andreas-kupries/critcl

To Install:

```
git clone https://github.com/andreas-kupries/critcl

cd critcl

tclsh ./build.tcl install
```

Building a C library on the fly

```
package require critcl       3.1.11
critcl::cproc triple {int i} int {
    return i * 3;    /* this is C code */
}
puts "three times 123 is [triple 123]"
```

Building a C library for packaging

Building a Project

TEA

- The Unofficially Official standard for Tcl Extensions
- Make Developers support two independent build systems:
  - Autoconf/Makefile (For Unix)
  - Nmake (For non-mingw builds of Windows)

What is there to hate?

- Many extensions in the wild have either a fleshed out Nmake or a fleshed out Gnumake build system
- Both are built on platform specific batch files (sh in unix, nmake+batch file Windows)
- All suffer from the very failing that Tcl we designed to prevent: spawning a process to check that 1+1 does in fact equal 2

Why do we have to keep supporting it?

- Every book in Print in Tcl describes how TEA is how extensions are built
- Many extensions **do** require the level of control that TEA allows over the build environment
  - Rewriting them to Critcl would be … painful
- Breaking something that works (albeit awkwardly) is not the Tcl Way

Practcl

- A migration path between TEA and Critcl
- Allows chunks of a TEA extension to be ported to a Tcl build system
- Ultimate goal is to:
  - reduce the role of autoconf
  - replace Makefiles
  -

- Hear the lamentations of their build automation

Warning:

The next slide contains semi-bigoted opinions from a marginally competent programmer.

Developers with strong opinions may find the next slide disturbing.

Viewer digression is advised

Why the hate on autoconf?

- It runs only on Unix
- Yes, yes, Cygwin and MSYS provide a compatibility layer.
  - But it sucks, and if you don't believe me try building something serious in it
- Microsoft is promising "sh" for Windows, but:
  - It's only on Windows 10
  - It requires a suite of developer tools
  - It's a collaboration between Microsoft and Ubuntu. I've never equated either with "reliability"

INGSOC
INGSOC

We Now Return to our Regularly Schedule Tutorial

The F*cking Magic Effect

- Autoconf performs substitutions to build the Makefile and pkgIndex.tcl
- A Tcl programmer building a C library must learn Shell and Nmake to produce an extension