Odie and Sherpa

Tools to automate the building of Tcl/Tk and its most common extensions from bare metal

Presented at the 21st Annual Tcl Developer's Conference (Tcl'2014)
Portland, Oregon
November 10th-14th, 2014

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

Abstract

Many casual developers can leverage the power of tclkits and the teapot to assemble self-contained executables. But what happens when you need something that isn't included in the teapot? Or if you are building on an exotic platform that ActiveTcl doesn't support yet? What if you work in an environment where you have to certify everything you have built?

This paper describes Odie, an environment for performing automated builds of Tcl/Tk and its assorted packages. Included in this environment is Sherpa, a package management tool that combines a kit-building tool, automated documenter, and package retriever.

Odie

Odie is the Open Design and Integration Environment. Think of it as a self-contained build factory for all things Tcl/Tk. It consists of a pile of autoconf scripts, shell scripts, Makefiles, and Tcl scripts to automate the building of Tcl/Tk. Odie is designed to yank components from the web as fossil repositories. It also understands how to work with Git and plain old tarballs.

Major components

Odie Bootstrap

Odie bootstrap is the base Odie repository. Cloned from fossil, and unpacked in the place of the user's choosing. It contains all of the components to download and install:

1. A dedicated local copy of Tcl/Tk,
2. A self-contained seed executable called a basekit
3. The Sherpa build system.

Odielib

Odielib is my collection of Tcl and C code for building software. Embedded in the Tcl library are pure-tcl equivalents for most of the C functions, and a script to detect if the C library is present or not. Odielib is distributed in the same fossil repository as the Odie bootstrap.

Tao

Tao, the Tcl Architecture of Objects, is a dialect of TclOO. It contains several enhancements and core object functions that the rest of my code library has come to rely on. Chief among them: property handling. Tao is distributed as a separate repository, which is unpacked in the Odie sandbox, and installed in the local Tcl's auto_path.

Sherpa

Sherpa is a command line interface to the tools in Odie. It makes it really handy for integrating Tcl code into the Make/Automake process. Sherpa, and its supporting libraries, are written in Tao and assembled into a self-contained executable at install time. Sherpa builds as a self-contained binary, and thus it can work outside of the confines of Odie. Sherpa is distributed in the same fossil repository as the Odie bootstrap.

Why Build Tcl from Scratch?

Building Tcl/Tk is not often required by the end-user. The presence of Tcl is generally assumed in most Unix environments. On Windows, Activestate does a wonderful job of keeping us up to date.

However, being part of most operating systems, trying to upgrade to the latest version of Tcl is not without its headaches. Some commonly used packages rely on very old behaviors. OS maintainers keep versions of tools locked in time. Sometimes they throw an extra flag at the linker that means any extension you try to compile is going to lead to a nasty crash.

The instructions for building Tcl often defer to a default set of assumptions:

a) You are running on Unix
b) You have admin privileges
c) That installing your own Tcl/Tk in /usr/local won't cause chaos with other system tools
d) The defaults for ./configure produce a workable Tcl/Tk for your platform.

Odie stems from my own experience where one or more of the above isn't true:

- As an undergrad, I would often have to work on large multi-user systems that granted me access to a C compiler, but where I was restricted to my home directory.
- Working on OSX, the operating system has a resident Tcl/Tk. It is heavily modified, and locked in time at Tcl 8.5.9. ActiveTcl installs to /usr/local. The UNIX environment builder MacPorts had a different Tcl/Tk installed to /opt/local. All are in the system path. None like to be tampered with.
- Recently at T&E Solutions, I have had to support simultaneous builds on Linux, MacOSX, and Windows. One project required the coroutines from the then beta 8.6.
- Apple has released a series of upgrades that crippled the Cocoa based Tk port. We had to work with builds checked out directly from fossil. Oh yes… and we had to recompile all of the Tk-related extensions as well.

Now I can't say my experience is typical. Nevertheless, if I've learned anything as a developer, typical is a word in the dictionary between typhus and typocosmy[1]

Fundamental Assumptions of Odie

Odie has its own set of fundamental assumptions:

- The developer cannot trust any Tcl/Tk found in the path. Tcl scripts that are part of the build environment can rely on a dedicated Tcl.
- A C development environment is already present on the developer's machine.
- Any binaries produced by the developer will ultimately be run on a different machine, thus must be self-contained.
- Odie is not permitted to install tools in the system path.

[1] "typic." webstersdictionary1828.com/Dictionary/T, 1828. Web. 12 Oct 2014

Setting up Odie

Odie is distributed as a fossil repository. This allows the user to push modifications to all of their build environments in an orderly fashion. It also allows special branches to be made to tailor the process to particular projects or peculiar build environments. It is also handy to be able to pull a build system from back in time to reconstruct an environment for a bygone platform.

With a technical crowd, it is best to lead in with an example. To set up a garden variety instance of Odie:

```
$ mkdir -p ~/odie/sandbox/odie
$ mkdir -p ~/odie/download
$ fossil clone http://fossil.etoyoc.com/fossil/odie ~/odie/download/odie.fossil
$ cd ~/odie/sandbox/odie
$ fossil open ~/odie/download/odie.fossil
$ ./configure --prefix=$HOME/odie
$ make install
```

If you want to save a little time, be sure to copy the fossil repositories you received on the memory stick from the conference into the ~/odie/download folder. They'll save you a bit of time in cloning the repositories.

Grab some coffee; it will be a little while. So, while we are waiting, let me explain what is going on. Once all of these steps are completed you should see a directory structure of tools built under ~/odie

~/odie/sandbox	Contained within will be folders for the sources of Tcl, Tk, and all of its extensions built by Odie
~/odie/download	Contained within will be the fossil repositories that Tcl and its extensions where checked out of. For tarball-derived sources, the tarballs will be downloaded here. The teapot will save packages here. Later on, as Sherpa will cache packages as teapots here.
~/odie/bin	Where binaries are installed. The most important will be: tclsh86 – Tcl wish86 – Wish sherpa – A self contained executable for Sherpa tclkit86 – Tcl/Tk assembled as a self-contained executable.
~/odie/lib	Location where Tcl/Tk's shared libraries and extensions are installed for local use.

Configuring Odie

There are three main directories of interest to the developer:

- Downloads – Where tarballs are downloaded, where fossil repositories are cloned, and where teapot caches its files. This may (and probably should) be a central folder for all instances of Odie.
- Sandbox – Where source code is unpacked and built. This should be unique for every Odie instance.
- Prefix – The top-level directory where Odie installs its tools.

The location of each of these folders is configurable at ./configure time:

- --prefix Controls prefix. On Unix this defaults to $HOME/odie. On Windows, this defaults to C:/odie. (We can't rely on $HOME under Windows. See Notes.)
- --with-download Controls the location of the download folder. If not specified, it defaults to $PREFIX/odie
- --with-sandbox Controls the location of the sandbox folder. If not specified, it defaults to $PREFIX/sandbox.

In the above example, we have configured Odie with the standard prefix. ./configure will assume the sandbox will be located at ${PREFIX}/sandbox and the download folder will be at ${PREFIX}/download. It will also assume we want as standard Tcl/Tk build as is defined for this platform.

Other flags that developers may be interested in:

- --with-tclbranch – Specifies what version of Tcl to check out of the fossil repository. This can be a tag such as "core-8-6-2". Tag can also be the sha1 string of a particular checkout. If not specified, the default is "trunk." (i.e. the most recent semi-stable checkout of Tcl.)
- --with-tkbranch – Specifies what version of Tk to check out of the fossil repository. This can also be a tag, or sha1 tag. If not specified, the default is whatever branch was specified for Tcl. If the value of tag is "none", no Tk or Tk related extensions are built for this Odie.
- --enable-cocoa – On MacOSX, --enable-cocoa=no will cause Tk and its related extensions to target X11 instead of the native cocoa port. (When building Tk, this is equivalent to –enable-aqua.)

Multiple Instances of Odie

In my case, I usually find myself doing most of my work on my Mac. On my Mac, I keep two installations of Odie. One targets the native Cocoa port of Tk. The other targets the X11 port of Tk. (Some of the visuals I work with operate better in one or the other.) Odie will default to the Cocoa port, so if we want to build a second instance targeting X11:

```
mkdir -p ~/odie-x11/sandbox/odie
cd ~/odie-x11/sandbox/odie
fossil open ~/odie/download/odie.fossil
./configure --prefix=$HOME/odie-x11 --with-download=$HOME/odie/download --enable-cocoa=NO
make install
```

Both of these instances will live, happily, side by side in one's home directory. The ~/odie path will build Tcl/Tk and all of its extensions with bindings for Cocoa. The ~/odie-x11 path will build Tcl/Tk and all of its extensions with bindings for X11. Each will get its own Tclsh, and Wish, basekit, Sherpa, and extensions.

They *will* share a download folder. So I will have only one instance of the Odie fossil repository, and one instance of the Tcl repository, one copy of the SQLite source tarball, and so on. Two fossil checkouts sharing a common repository have a nifty side benefit: You can swap private checkouts between them.

```
cd ~/odie/sandbox/taolib
echo “Sean is cool!” >> README
fossil commit --private ; # Checking in evaluation version
cd ~/odie-x11/sandbox/taolib
fossil update private
UPDATE README
```

Using Odie

Configuration Files

When ./configure is run, Odie builds two files to assist any other product with integration: odieConfig.sh and odieConfig.tcl. Both contain roughly the same information. odieConfig.sh is in a format useful for Makefiles and shell scripts. odieConfig.tcl is in a format useful for Tcl. The configuration files capture vital information about the local environment. They tell Odie where it will find key executables it will need.

Basekits

Along with a standard dynamically linked Tcl/Tk, your Odie also built a modified Tcl shell called a basekit. It will be located at ~/odie/bin/tclkit86 on Unix, or c:/odie/bin/tclkit86.exe in windows. The basekit is a self-contained copy of Tcl/Tk with ZipVFS support. Concatenated to the standard Tcl shell is a zip file system, and within the zip, the file system contains the files for **tcl_library** and **tk_library**. As built, the shell behaves like any other Tclsh:

```
Mac:Odie hypnotoad$ ~/odie/bin/tclkit86
% puts “Hello World!”
Hello World!
% exit
Mac:Odie hypnotoad$ echo puts {Hello World!} > hello.tcl
~/odie/bin/tclkit86 hello.tcl
Hello World!
```

The only noticeable difference between Tclkit and Tclsh is that Tclkit will report /zvfs/boot/tcl as its location for **tcl_library**. /zvfs is the mount point the boot loader selects at startup.

You can turn a Tclkit into a self-contained executable by giving it a **main.tcl** file in the zip file system.

```
Mac:Odie hypnotoad$ cp ~/odie/bin/tclkit mykit
cp hello.tcl main.tcl
zip -A mykit main.tcl
./mykit
Hello World!
```

In addition to Tcl scripts, you can also embed images, dynamic libraries, and virtually any other type of file you may need. Well, anything you may need that works with read-only access.

Unless Tk was disabled during ./configure, it is packed into the Tclkit as a dynamically loaded library. Accessing Tk is as simple as “package require Tk.”

Sherpa

Alongside of the basekits and Tcl and wish in your ~/odie/bin folder you will see another executable **sherpa**. Sherpa has two jobs:

1. Build, install, and maintain packages for the local Odie environment.
2. Build and package extensions for inclusion in virtual file systems.

Sherpa is a tool for building the Virtual File Systems (VFS) of self-contained applications. It interacts with Odie and the teapot to either download or build Tcl extensions. It will then assemble the VFS and index all of the packages properly for loading.

Included with Odie are the ingredients for several Apps. Let’s focus on the “toadkit”. Toadkit is a modified basekit with packages needed for gaming. You can find that project in ~/odie/sandbox/odie/apps/toadkit. That project consists solely of a Makefile:

```
#!/usr/bin/make
include ../../odieConfig.sh
APPNAME=toadkit
PACKAGES=sqlite tcllib taolib odielib udp tclvfs vectcl tkimg canvas3d tkhtml tklib

all:  ${APPNAME}${EXE}

clean:
   $(SHERPA) rmdir ${APPNAME} ${APPNAME}.exe ${APPNAME}.vfs $(irmtarget)

${APPNAME}${EXE}: ${BASEKIT} ${APPNAME}.vfs
   cp ${BASEKIT} ${APPNAME}${EXE}
   cd ${APPNAME}.vfs ; $(ZIP) -rAq ../${APPNAME}${EXE} .
   chmod a+x ${APPNAME}${EXE}

${APPNAME}.vfs:
   $(SHERPA) rmdir ${APPNAME}.vfs manifest.txt plugin.zip
   mkdir -p ${APPNAME}.vfs
   $(SHERPA) vfs-install ${APPNAME}.vfs ${PACKAGES}
   $(SHERPA) vfs-mkIndex ${APPNAME}.vfs

install: ${APPNAME}${EXE}
   cp -f ${APPNAME}${EXE} ${LOCAL_REPO}/bin/${APPNAME}${EXE}
```

As you can see, the Makefile is remarkably simple. All of the heavy lifting is done by Sherpa. The call to include **odieConfig.sh** loads the environment with everything the script will need to know. **$(SHERPA)** is the call to the local build of the Sherpa toolkit. We even go so far as to tell the Makefile whether an executable will need a .EXE extension.

Using Sherpa

The Sherpa executable is a flexible tool. It has embedded within in the TclVFS extension, the entire Tcllib, and a complete list of recipes to build most of the common Tcl/Tk extensions. All of those functions are accessible from the command line.

Package Utilities

These routines maintain packages within the local Odie. To manage packages in a virtual file system (i.e. when building a self contained executable) use the **vfs** series of commands.

sherpa package-fetch packagename ?packagename…?

Download and unpack into the sandbox the packages specified.

sherpa package-install packagename ?packagename…?

Install one or more packages to Odie. Sherpa will try to download, compile, and install from source when possible. When not possible, it will attempt to obtain the package from the Teapot.

sherpa package-list

Output to stdout a list of all packages supported by this copy of Sherpa. Output is in the form of:

```
modulename class
modulename class
…
```

sherpa package-reinstall packagename ?packagename…?

Skips checks to see if a package is already present, and performs a fresh compile and install. For Tcl packages, this function will also rebuild the teapot cache of a package.

sherpa package-uninstall packagename ?packagename…?

Remove one or more packages from Odie. Any local files related to the teapot are also removed.

sherpa package-upgrade packagename ?packagename…?

Check with the distribution systems for the packages specified and detect if a new version is available. If a new version is available, download, compile and install it.

Teapot Access

Sherpa utilizes Roy Keene’s teapot clone *teaparty* to locate any extension for which it does not have a recipe. While functions from *teaparty* are utilized internally by Sherpa, these functions provide direct access to the teapot.

sherpa teapot-get vfsroot *os cpu package ?package…?*

Retrieve one or more packages from the teapot that are compatible with the **os** and **cpu** combination, and install them to the path specified by **vfsroot**.

sherpa teapot-info

Output the following block of text to stdout:

```
OS: $VALUE
CPU: $VALUE
SERVERS: $VALUE
```

Where:

- OS is the operating system as computed by Odie
- CPU is the architecture as computed by Odie
- SERVERS is a list of the teapot servers that Sherpa is configured to use

sherpa teapot-list *?os? ?cpu?*

Return a list of all packages available for the combination of **os** and **cpu**.

Virtual File System Utilities

The suite of routines to package virtual file systems.

sherpa vfs-install vfsroot packagename ?packagename…?

The call to **sherpa vfs_install** downloads, compiles, and installs into the folder specified all of the listed extensions. The extensions will be placed in the “teapot” folder of the VFS. Each extension will be stored in a folder named after the md5 hash of the extension combined with its platform and version.

```
   ${SHERPA) vfs_install ${APPNAME}.vfs ${PACKAGES}
ls ${APPNAME}.vfs/teapot
67368B11EFB262C049F7B484389295C6 B8F5129A8E937A249A33920287689053
B2C6534EE22ACE3EE02317854940A3C1 FF86E501D7C25BDDF4F2C2BBFE6739AA

ls toadkit.vfs/teapot/B8F5129A8E937A249A33920287689053/
pkgIndex.tcl libsqlite3.8.5.dylib teapot.txt
```

We utilize hashes to eliminate name collisions, spaces in names, funky characters, capitalization, or whatever may cause us grief down the road.

sherpa vfs-mkIndex

While it would be possible to initialize our program by adding the “teapot” folder to the auto_path, up until recently ZipVFS could not reliably glob for files. To get around that limitation, I was always in the habit of pre-indexing my extensions. To do this, Sherpa will descend into the VFS folder, and locate every **pkgIndex.tcl** file. It will then build a script called **packages.tcl**. Sourcing this file will have Tcl systematically source each of the pkgIndex.tcl files in their appropriate path. For folders without a pkgIndex.tcl, Sherpa will examine each file ending in .tcl for **package provides** statements, and add the appropriate **package ifneeded** to the **packages.tcl** script.

For **pkgIndex.tcl** files, you will see a line like this:

```
set dir [file join [lindex $::PATHSTACK end] \
  teapot/67368B11EFB262C049F7B484389295C6/lib/tcllib1.16/wip] ; \
source [file join [lindex $::PATHSTACK end] \
   teapot/67368B11EFB262C049F7B484389295C6/lib/tcllib1.16/wip pkgIndex.tcl]
```

For standalone files, you will see a line like this:

```
package ifneeded codebale 0.2 [list source [file join [lindex $::PATHSTACK end] lib/codebale
index.tcl]]
```

The net result will be that your executable will have instant knowledge of the entire library of code available within the VFS in one shot, and without having to undergo a discovery process. This is particularly helpful, because if Tcl doesn't have to resort to "**package unknown**" it won't look outside of the VFS for software.

General Purpose Tools

Sherpa contains a few creature comforts that use Tcl's facilities to replicate commonly needed system utilities.

sherpa diff *filename filename*

Invoke a resident copy of tkdiff to compare two files.

sherpa edit *filename*

Invoke a resident copy of Richard Hipp's "e" file editor. Great for times when you just need to hack an ASCII file, but you really don't have (or don't want to find) the resident text editor. Also handy for cases like under Windows when the resident text editor either doesn't understand line breaks (notepad.exe) or wants to turn your document into a word processing document (wordpad.exe)

sherpa http-get *url destination*

Download a file via http. This implementation is based on the **http** package, with an enhancement that understands how to follow server document redirects. It should be noted that the content redirects used by sourceforge and the like still confuse this command. Nevertheless, it is handy for those simple downloads when you don't have **wget** or **curl** handy.

sherpa module *modulename method ?args…?*

Exercise the specified method of the specified module and express the output to stdout.

sherpa odie_config *format*

Output to stdout the Odie environment used to build Sherpa. Sherpa will output the configuration in two formats:

- sh – Compatible for inclusion in shell (/bin/sh) scripts and Makefiles
- tcl – Compatible for inclusion in Tcl scripts.

sherpa rmdir *path ?path…?*

A way to recursively delete files through the power of Tcl. Needed for cases where MinGW creates a path that it, itself, cannot erase. Particularly when building Critcl. Which, at one point at least, seemed to like putting : characters into path names.

sherpa shell

Open Sherpa with an interactive GUI prompt. All of the functions accessible directly from the command line are located in the **::command** namespace.

sherpa source *filename ?encoding?*

Invoke a series of commands located in the file **filename**. This is implemented internally as:

```
proc ::command::source args {
  uplevel #0 [list source {*}$args]
}
```

sherpa unzip *archive destpath*

Unpack a zipfile to the destination path specified. The command is implemented using the **zipfile::decode** package included with Tcllib.

sherpa untar *archive destpath*

Unpack a tarball to the destination specified. This command is implemented using the host operating system's **tar** command. (Future versions will use the **tar** facilities in Tcllib and/or TclVFS.)

sherpa zip *archive sourcepath*

Create a zipfile from the path given by *sourcepath*. This command is implemented using the **zipfile::encode** package included with Tcllib.

Sean's Private Spell Book

Sherpa also includes a few tools that I use that may or may not be generally useful. These tools center around the technique I use for generating automated help systems, and a system I use to reformat Tcl source files to suit my ideal of what a library of code should look like.

sherpa autodoc *path ?path…?*

Index all source code in the paths specified, and place the output into the folder **/autodoc**, relative to the module's root path. For any Tcl source file that doesn't conform to "Sean's Own Standard", prepare a conforming implementation as "sourcefile.tcl.new". The products in /autodoc are helpdoc.rc and helpdoc.sqlite. The SQLite file is in the Tao's Yggdrasil schema. The rc file is the same information, but encoded in a script that would rebuild the SQLite file in a local file system. (SQLite doesn't like working inside of many VFS implementation, or off network shares.)

sherpa autodoc_changes

Find each Tcl source files with "corrections" made from a pass by **sherpa autodoc**. Present the user with a diff and the option to keep the changes, delete the correction, or skip.

sherpa autodoc_scan *?path …?*

Perform the same indexing as **sherpa autodoc**, but without creating corrections.

sherpa scm-move *source destination*

Move files in a repository from *source* to *destination* and record this move with fossil at the same time. This tool only supports fossil repositories, currently.

Developing for Sherpa

Sherpa recipes include a collection of workarounds to get packages to build and install on the platforms Odie supports. If it seems a little complicated, that is simply a reflection of how difficult it really is to roll one's own Tcl distribution. The process used by Sherpa is by no means unique. It just happens to be (somewhat) documented.

Sherpa has two major concepts: modules and recipes. A module is a reusable block of code. A recipe is machine executable process to download, install and/or package Tcl extensions and other programs. Recipes are made of modules.

Each recipe for a package or tool is implemented as a TclOO object. Similar objects are grouped into classes. Each is defined with a **sherpa::recipe** statement:

```
sherpa::recipe tclvfs {
  fossil_url {http://fossil.etoyoc.com/fossil/tclvfs}
  package_binary 1
  package_binary_tk 0
  destroot_capable 0
} sherpa.module.tea_fossil
```

The arguments for a Sherpa recipe are thus:

sherpa::recipe *module_name property_dict implementation*

The implementation can be either the name of an existing implementation class, or the definition of a new class as a block of TclOO code. In the example above, we are building the TclVFS extension. The TclVFS extension is of the *sherpa.module.tea_fossil* variety. That class is the fusion of two modules: **sherpa.buildsystem.tea** and **sherpa.distribution.fossil**. That combination has several configuration options:

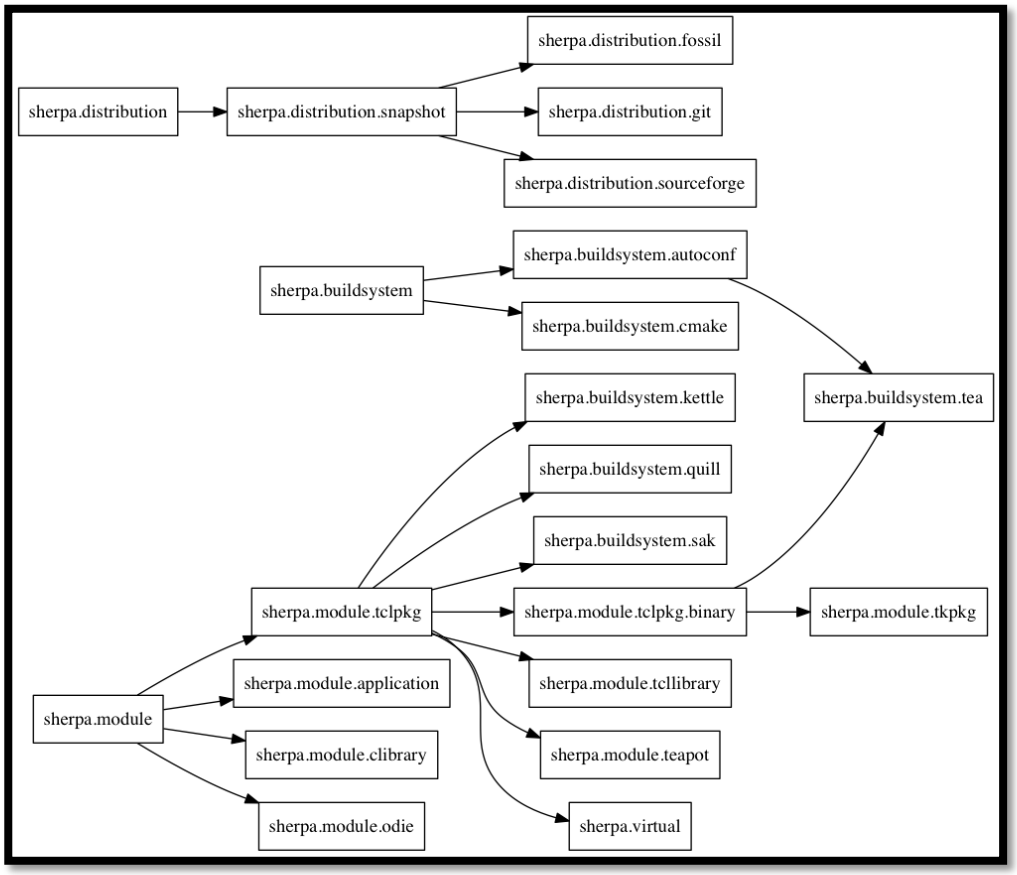
fossil_url	Tells fossil where to clone the repository from
package_binary 1	Tells the tea build system to expect the recipe to produce a binary Tcl extension
package_tk 0	Tells Sherpa that the binary Tcl extension does not link to Tk
destroot_capable 0	Tells the build system that this particular Makefile does not understand the convention make DESTROOT=$PATH install. What it will do instead is install the package to the local system and then snapshot the folder under $PREFIX/lib that is produced.

Packages that require custom code to build or download can have a more elaborate definition:

```
sherpa::recipe sqlite {
  aliases tclsqlite
  package_name sqlite
  package_version 3.8.5
  package_binary 1
  package_binary_tk 0
  tarball_dir sqlite-autoconf-3080500
  tarball_url {http://sqlite.org/2014/sqlite-autoconf-3080500.tar.gz}
} {
  superclass sherpa.distribution.tarball sherpa.buildsystem.tea

  method build_path {} {
    set path [file join [my module_path] tea]
    return $path
  }

  method native_vfs_install path {
    copy_path [file join $::Odie(local_repo) lib [my property package_name][my property
package_version]] [file join $path lib [my property package_name][my property package_version]]

  }
}
```

In this case, the SQLite extension has a non-tea standard location for compiling. Instead of compiling in the top-level directory, or the platform directory, this package compiles from the **tea** directory. There is also a slight complexity in the way that SQLite is distributed. It arrives as a tarball that we will unpack. The folder it unpacks as will have to be renamed.

Sherpa contains many ready-made classes that can be combined to build most extensions in the Tcl/Tk ecosystem. Modules also exist to build external tools like fossil and zip. The class hierarchy of Sherpa's classes looks roughly like this:



Module Classes

A “module” is a complete set of instructions for building and installing a Tcl package, executable, or external tool. All Sherpa modules must implement the following functions:

Class: sherpa.module

Method: sherpa_clean

Performs an operation to clean out the module path in preparation for a fresh build. Any existing copy of the software is removed from Odie. All tracking records mark the package as not installed.

Method: sherpa_build

As applicable, this command downloads the code, auto detects properties, and compiles it.

Method: sherpa_detect_properties

Combine the configuration fed into the recipe with data derived from the build system and distribution system into a gestalt.

Method: sherpa_install

Perform the steps needed to install this package into the local Odie environment.

Method: sherpa_present

Return a true if the package is already installed in the local Odie, and false otherwise.

Method: sherpa_skip

Return a true if the package cannot be installed in this environment, false otherwise.

Method: sherpa_uninstall

Remove a package from Odie and the local teapot.

Method: sherpa_upgrade

Detect, download, recompile, and reinstall a package if a new version is available.

Method: sherpa_vfs_install ***base***

Arguments: **base** – Top-level directory of a new virtual file system

Perform the steps needed to install this package into a virtual file system. If a cached version is available in the teapot, that version is used. Otherwise, a call to the build system’s **build_vfs_install** is made.

Interaction with Teapot

Sherpa uses teapot as a medium of exchange between installed packages and packages in a VFS. This interaction also allows Sherpa to draw on the vast library of packages available through the teapot that it does not have a recipe for, or for which the build tools are not present in the local environment to make. These methods are implemented only for the subset of modules that expect to produce Tcl extensions.

Class: sherpa.module.tclpkg

A class that builds a Tcl package. This class consists mostly of workarounds to shoehorn non-conforming packages to produce teapot compatible extensions.

Method: teapot_arch

Generate a string that represents what architecture this package would have in the teapot.

Method: teapot_build

1. Build a version of this package suitable for distributing in the teapot.
2. Stuff that VFS into the module directory as **teapot.zip**
3. Prepare a **teapot.txt** file with as much data as we can collect.

Implementations with their own tools for building place a teapot call their own code here. The default is to ***build_vfs_install*** into a temporary location, and hammer that file structure into shape with ***teapot_vfs_structure***.

Method: teapot_cachefile

Return the fully qualified, normalized, path name to the cache file this package either generated or will generate. For use in such wonderful functions as "file exists [my teapot_cachefile]", or "command::zip [my teapot_cachefile] $teapotvfs", or "command::unzip [my teapot_cachefile] $appvfs/teapot"

Method: teapot_info

For Tcl packages without a ready-made **teapot.txt** file making facility, return a key/value list of all of the information needed to build one. This method produces a pre-amble, appends the results of **build_teapot_data** and **distribution_teapot_data**, and concludes by exporting all of the other meta data tracked by Sherpa as "Meta sherpa_property" lines.

Method: teapot_package_fqn

Return the fully qualified name for a package, as it would be found on a teapot server. i.e. in the form of:

```
package/name/$PACKAGENAME/ver/$PACKAGEVER/arch/$PACKAGEARCH/file
```

Method: teapot_package_name

Return the output of **teapot_package_fqn**, using a cached copy. Some of the lookups used by **teapot_package_fqn** have the potential to be expensive in the wrong circumstances.

Method: teapot_vfs_structure

Massage a product of *build_vfs_install* to conform to the structure as befitting teapot.

Class: sherpa.module.teapot

This class is a placeholder for packages located in the teapot, but for which no local recipe is available, or for which Sherpa is aware of a problem keeping this recipe from working in the current environment. This class is essentially a driver to allow access to the teapot client. At present, the class only supports downloading and installing pre-built packages.

Distribution Classes

Distribution classes tell Sherpa where and how to download source code. Some packages are distributed as fossil repositories. Others use Git. Others are still shipped as tarballs. Each of those distribution systems has its own behaviors, capabilities, and settings. Optionally, this class may provide additional metadata for documentation and/or formulating teapot archives.

A distribution must implement the following methods:

Class: sherpa.distribution

Method: distribution_download

This method performs all of the steps necessary to download the software from the Internet and unpack the raw source to $SANDBOX/$MODULENAME.

Method: distribution_meta_data

This method extracts meta data from the source control system. Data is returned as a dict. The default is to return an empty dict.

Method: distribution_teapot_data

Return a chunk of text describing this package to be included in the **teapot.txt** file.

Method: distribution_upgrade

This method:

1. Goes to the version control system and tried to determine if a new version is available for download and installation.
2. If a new version is available, perform a download.

If a new version was downloaded, the method will return true. If no new version was downloaded, the method will return false. This method is currently implemented for fossil distributions only.

Class: sherpa.distribution.snapshot

A class that defines behaviors utilized when code is distributed via a tarball or zipfile downloaded from the Internet:

- snapshot_url – The URL from which the archive is downloaded
- snapshot_dir – The name of the directory created after unpacking the file. If null, the archive is unpacked directly in the sandbox. Used for cases like SQLite, where the tarball unpacks to “sqlite-030805-autoconf”, and we really want the directory to be “sqlite.”

Class: sherpa.distribution.fossil

A class that defines behaviors utilized when code is distributed via fossil. This class defines the following two properties:

- fossil_url – The URL from which the fossil repository is to be cloned
- fossil_tag – The fossil tag to use for builds. If not specified the default is “trunk”

If a fossil executable is not available, the class will fall back to a snapshot. The URL of the snapshot is auto-generated from the repository URL.

Class: sherpa.distribution.git

A class that defines behaviors utilized when code is distributed via Git. This class defines the following properties:

- git_url – The url from which the Git repository is populated
- git_branch – The branch of the code used for builds. The default is “HEAD”

If a Git executable is not available, the class will fall back to a snapshot. The URL of the snapshot is auto-generated from the repository URL.

Build Classes

Build classes tell Sherpa how to compile and install raw source code downloaded to $SANDBOX/$MODULENAME

A build must implement the following methods:

Class: sherpa.buildsystem

Method: build_compile

Arguments: None

Perform any steps necessary to compile the code.

Method: build_install

Arguments: None

Perform any steps necessary to install this package in the local Odie.

Method: build_meta_data

Arguments: None

This method that extracts meta data from the build system. Data is returned as a dict. The default is to return an empty dict.

Method: build_path

Arguments: None

Return the top-level directory where builds are performed.

Method: build_teapot_data

Return a chunk of text describing this package to be included in the **teapot.txt** file.

*Method: build_vfs_install **base***

Arguments: **base** – Top-level directory of a new virtual file system

Perform the steps needed to install this package into a virtual file system.

Class: sherpa.buildsystem.autoconf

A class which implements a build system using Gnu-style autoconf tools. Sherpa assumes:

1. The software builds from the root directory of the sources
2. If *autogen.sh* is present, it should be run with /bin/sh
3. If *autogen.sh* is not present, and *configure* is not present, and *configure.in* is present, autoconf should be run.
4. If no *Makefile* is present, running *./configure* will fix that. (When run, the configure script specifies $ODIEROOT as the prefix and $ODIEROOT/bin as the bin dir.)
5. “make all” will build everything
6. “make install” will install everything

Class: sherpa.buildsystem.cmake

This class implements a build system based on cmake. The implementation is mainly a placeholder. I welcome (read that I am begging for) input from someone more knowledgeable in cmake. A naïve first pass of an implementation is present, but no recipes use it yet.

Class: sherpa.buildsystem.kettle

This class implements a build system based on Andreas Kupries’ kettle. It knows how to pass the right commands to the module’s **build.tcl** file to get what it needs. It is currently used in the recipes to build cmdr, critcl, fx, and tcl-linenoise

Class: sherpa.buildsystem.quill

This class is a placeholder for interaction/coordination/participation with William Duquette’s Quill build system. It doesn’t do anything as of the writing of this paper. However, it probably will by the time you are reading this paper. Moreover, it certainly will by the end of the Tcl 2014 conference.

Class: sherpa.buildsystem.sak

This class implements a build system based on the “Swiss Army Knife” (SAK) style installer developed by Andreas Kupries, and deployed in Tcllib, Tklib and Taolib. Like the kettle driver, the SAK driver is mainly an adapter to get what Sherpa wants out of the resident Tcl based installer.

Class: sherpa.buildsystem.tea

A descendent of **sherpa.buildsystem.autoconf**, this class adds the additional assumptions that:

1. The end product is a Tcl extension
2. The product conforms to the “norms” of the TEA specification.
3. If a tclconfig folder is missing from the root folder, one should be provided.
4. If a teapot.txt file exists, it is gospel.

Tea builds can be further configured with the following options:

- destroot_capable – Boolean, when true the Makefile understands how to redirect an install with “make intall DESTROOT=$DESTROOT”. This option is ignored under Cygwin. When false (or ignored) a VFS install image is synthesized from whatever the package installs to $ODIEROOT/lib/$PKGNAME$PGKVERSION

Future Directions

Cross Compiling

Presently Odie supports one target at a time. With the level of control over the build process that Odie allows, the focus on building Tcl and its extensions from scratch, etc., it should be fairly straightforward to permit cross compiling from one platform to another. To enable this properly, I would have to go back through the Sherpa system and make clear delineations between Odie (the host operating system) and Todie (The target operating system.) While I brewed up a few ideas in the process of writing this paper, the implementation is not ready for prime time.

Better CMAKE support

Sherpa's support for cmake is, at best, theoretical. As cmake is a popular build system, and many Tcl applications and extensions are built with it, Sherpa will have to understand it better. At this point, no extensions I have needed build with cmake, but there are many supporting libraries from the c++ world that do. If you are a user of cmake, I welcome any input or code contributions.

Microsoft Visual Studio Support

On the Windows platform, Odie uses MSYS/MinGW. There are many extensions, the latest TWAPI in particular, which require Microsoft Visual Studio. I am not currently a Visual Studio user. Any suggestions or code contributions are welcome.

Teapot Uploading

With all of the power to build extensions from scratch, the ability to download teapot archives, and generate them, it would be nice to capture the teapot products produced by Odie and upload them to a teacup server. Even nicer: in an automated fashion. The level of support would be nowhere near what ActiveTcl provides, but for experimental builds, exotic platforms, or regulatory environments that demand tight control over the build process, this would be an excellent tool in the hands of the Tcl distribution maintainer.

JimTcl Integration

Odie currently uses autoconf and sh scripts to perform its low level bootstrapping. Sherpa was written in Tcl8.6 and Tao out of expedience. Now that it has crystallized into a semi-stable form, there is nothing that Odie or Sherpa does that requires a full-up installation of Tcl/Tk. With some, albeit substantial, modifications, Odie and Sherpa could be re-engineered to operate using Steve Bennet's JimTcl. JimTcl has the advantage of running in a small memory footprint. It builds as a single C file. Using JimTcl would eliminate the need to download the Tcl sources separately to do a basic build.

Another fringe benefit is that it gets me out of having to hack in SH scripts and M4.

Aside from bootstrapping, it would also be useful to have an "Odie" like environment for embedded Linux distributions. Many of them target JimTcl because of its small footprint. They often compile on exotic platforms. In addition, usually, they have a need for some niche extensions.

Quill Interoperability

William Duquette’s Quill is a brilliant high-level interface for developing standalone executables, but it relies on externally produced basekits and packages. Odie is a system for producing basekits and packages, but it lacks a high level interface to produce executables. Both systems use the teapot as a currency of exchange.

FreeWrap/KitGen/KitCreator/Etc…

The primary focus of Odie is as a package builder. Odie happens to have a kit creator because:

1. I know how to do it
2. Building them under 8.6 is somewhat easier than under earlier Tcl releases.
3. There were issues under MinGW that required wrapping the entire tool in a self contained executable to solve.

That said, Odie currently supports a very limited market in the Tcl universe: Namely bleeding edge users on 8.6. To support older versions of Tcl, or kit building facilities not based on ZVFS, the kit building facilities for Odie would have to be an order of magnitude more complex.

I would rather like users of Odie be able to call up these more sophisticated kits as recipes. I am currently working on an Odie recipe for FreeWrap. I would be more than happy to either work with, or graciously accept contributions from, other kit manufacturers.

Notes:

No Place Like $HOME on Windows

On the Windows platform, Odie is built under MSYS/MinGW. MSYS builds its own posix environment, including its own $HOME directory for the user, all relative to: `c:/Mingw/msys/1.0`. However, once Tcl is built, it uses the operating system's notion of where files exist, relative to `c:`.

Odie involves a lot of interchange between MSYS and Tcl. There's the potential for confusion with respect to file paths. (More than potential in my experience.) To make matters worse, many autoconf tools do not handle spaces in path names properly.

I recommend creating your Odie in `c:/odie` because that folder is not subject to any Windows or MSYS path magic. Both Msys utilities and Tcl can address a file care of `c:/odie`.

Sherpa Teapot Cache

Cached versions of packages are stored in the "download" folder of Odie. For every package built there will be two files present: local.*MD5HASH* and local.*MD5HASH*.txt. The md5 hash is built from the package name, package version, platform, and fossil checkout. If you have multiple installations of Odie sharing a download folder, a package with the same name and version will have two different hashes for the two different platforms. The .txt file next to the hash of the same name is the contents of the teapot.txt file. If you are curious as to which package goes with which hash, that information can be retrieved from there.

An index of all teapot downloadable packages is stored at $ODIEROOT/var/sherpa/teapot.rc.

Sherpa's Database

Sherpa maintains a database of what packages have been installed, as well as meta data collected from them, in a cached location under *$ODIEROOT/var/sherpa*. The SQLite file is a master index of sherpa's picture of the state of various packages. The .txt files are a backup copy of the state and metadata from every package. The .txt files are present for the initial stages of the build when SQLite may or may not be present. They also serve as a handy data backup format if the Sherpa SQLite database should become corrupt, or if the database schema should change. The schema for this database is rudimentary:

```
CREATE TABLE module_info (
  package string,
  field   string,
  value   string,
  primary key (package,field) on conflict replace
);
```

The contents of this database are accessible with the **::sherpa::meta_get** command. It is currently only used to record if a package was installed or not.

Extending Sherpa

Sherpa will source the "$ODIEROOT/etc/sherpa.rc" file if it exists. This file can be used to define your own set of one-off Sherpa recipes, feed additional configuration information to existing recipes, or act as a boot loader for an entire build suite. Use your powers for good.

Acknowledgments and External Links

As usual, I'm not so much developing new ideas as bundling the works of far more talented individuals. The following projects are referenced in this paper:

Autoconf, Automake, Gnu Tools	Free Software Foundation	https://www.gnu.org/software/software.html
Cmake	Kitware	http://www.cmake.org
FreeWrap	Dennis LaBelle	http://freewrap.sourceforge.net
JimTcl, Autosetup	Steve Bennett	http://jim.tcl.tk
Kettle	Andreas Kupries	http://core.tcl.tk/akupries/kettle/index
KitCreator	Roy Keene	http://kitcreator.rkeene.org/fossil/index
KitGen	Pat Thoyts	https://github.com/patthoyts/kitgen
MinGW/MSYS	MinGW Team	http://www.mingw.org
Quill	William Duquette	https://github.com/wduquette/tcl-quill
SQLite	D. Richard Hipp	http://www.sqlite.com
Tcl, Tk, Tcllib	The Tcl Core Team	http://core.tcl.tk
Teaparty	Roy Keene	http://www.rkeene.org/projects/info/wiki/189
Teapot/Teacup	Jeff Hobbs, ActiveState	http://docs.activestate.com/activetcl/8.4/tpm/toc.html
TOBE	D. Richard Hipp	http://www.hwaci.com/sw/tobe/index.html
TWAPI – Tcl Windows API	Ashok P. Nadkarni	http://twapi.sourceforge.net

Special Thanks Yous

I wish to express a very special thank you to Andreas Kupries, Gerald Lester, Joe Mistachkin and Will Duquette for reviewing early drafts of this paper. I promize to hav less tpyos next thyme.[sic]

Cover Art

The cover image is entitled "*Conceptual drawing of a "typical lab" for Oak Ridge National Laboratory's Central Research Building (4500 North) prior to construction in the early 1950s*.", and was downloaded from:

http://knoxblogs.com/atomiccity/2012/03/28/fancy_smancy_science_lab_1950s/

More Information

If you would like to know more about Odie, Taolib, or any of my other software tools, they are available for download at:

http://fossil.etoyoc.com

Odie is at: http://fossil.etoyoc.com/fossil/odie

Taolib is at: http://fossil.etoyoc.com/fossil/taolib

For many of the extensions used in Odie, I have created a fossil mirror to collect any patches, bug fixes, or build system reforms I have come up with. Where possible those changes are passed back to the original projects. In many cases, there is no one actively maintaining those projects. My fossil mirrors are located at:

http://fossil.etoyoc.com/fossil

I have also created what I dub "the sandbox" as a place where anyone can collaborate on maintaining patches to Tcl and its extensions. The sandbox is a copy of every fossil respository I maintain or mirror. In the sandbox, every repository is open to anonymous checkins. The repositories also allow users to self-register, manage tickets, and alter the Wiki. That project is located at:

http://fossil.etoyoc.com/sandbox

If you have any questions, comments, or contributions, I can be reached:

- on the TkChat app as "hypnotoad"
- via email at: yoda@etoyoc.com (For personal correspondance)
- via email at: swoods@tnesolutions.com (For business corresponadance)