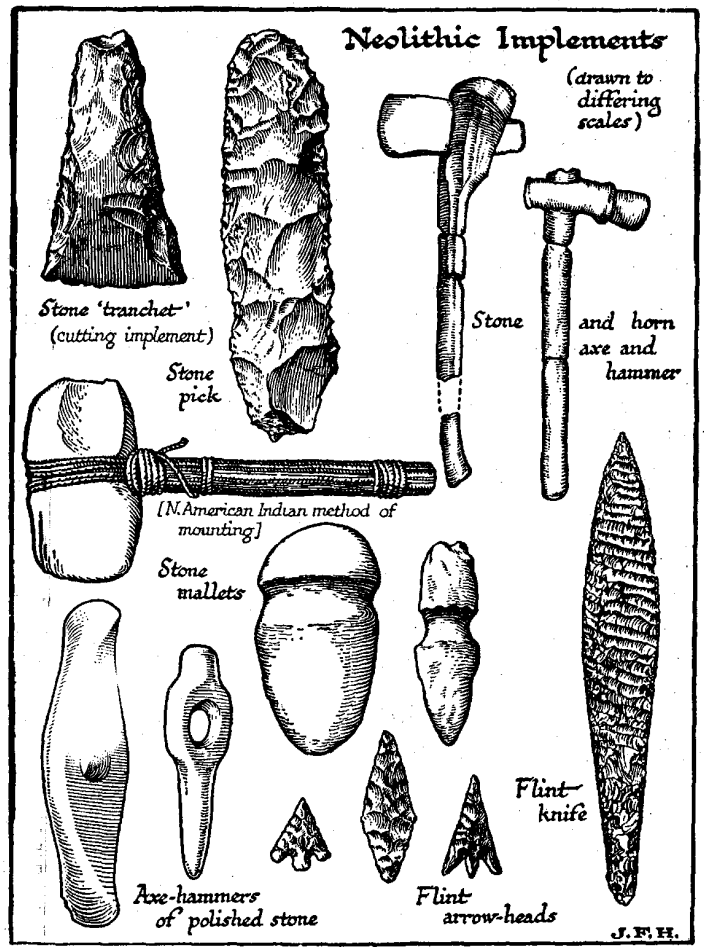


Introducing TOOL

The Tcl Object Oriented Library

Presented at the 22nd Annual Tcl Developer's Conference (Tcl'2015)
Manassas, VA
October 19 - 23, 2015

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

Abstract

With the advent of TclOO, the Tcl Community is in need of common design patterns on which to build applications and utilities. Rather then descend into a myriad of domain specific libraries; this paper presents the concept of a TOOL (Tcl Object Oriented Library). Borrowing many of the conventions and idioms from Tcllib, the goal of TOOL is provide a common suite of tools that are tested, documented, and ready for immediate deployment. Like Tcllib, TOOL will be broken into modules by subject area. This paper will focus on the core classes as well as tool::shed (a suite for building and distributing script and binary packages for Tcl.)

Introduction

We all use Tcllib. Tcllib has a lot of things going for it. It's a one-stop shop for ton of handy code. Compared to standard libraries in other languages, Tcllib's code is of really high quality. It has regression tests, and online documentation. Most of it even works!

We are entering a new era in Tcl development: Object Oriented (OO) code. OO Code is not built with libraries, but with frameworks. Frameworks are a different style of programming. And that different style of programming needs a different set of tools and practices than Tcllib currently provides.

Code Recycling vs. Inheritance

A large conventional pre-OO project in Tcl uses a style that I call "code recycling." Code recycling is a bit like putting together a Frankenstein monster. The developer pulls together parts from disparate sources. He/She then spends some effort grafting the different organs together.

A large-scale OO project in Tcl utilizes an inheritance-based style. Inheritance is a more like putting together a creature at Jurassic Park. The developer starts with frameworks and splices them together. Components share code, and combine holistically.

Forward Compatibility

Tcllib has been developed over a number of years across Tcl cores with a varying level of capability. Some parts of the library can't run as efficiently as is possible because it must maintain support for those older Tcl cores. Some packages rely on binary packages that clash with new core capabilities.

Removing that backward compatibility from Tcllib would be, if not a disaster, a crying shame. Rather than radically alter Tcllib, TOOL is designed to operate side-by-side with it.

Live Documentation

Documentation in Tcllib is currently a batch process. The developer writes a .man file which is then processed by the Swiss Army Knife (SAK) to produce the html file on the website. However, the output of that batch process must, itself, be checked into fossil in order to be published online. This has the potential to cause hate and discontent if two different branches auto-generate their documentation and attempt to merge together.

OO Development often requires a glimpse into the internals of the framework. And if two different branches have substantially altered the behavior of a class, the documentation needs to be able to reflect that difference side by side.

For TOOL I have selected Markdown as the de-facto standard for embedded documents. We have a Tcl implementation already. Fossil and GitHub can render Markdown on the fly. External editors exist for it. And it's not overly hostile to store as an Sqlite field.

Growth

Tcllib is undergoing some growing pains. Just the installed modules for version 1.17 weigh in at nearly 10mb. Many distributions either dissect Tcllib into individual packages, or leave it out of default installations to save on size. Which leads to the problem of modules in tcllib requiring other modules in tcllib. In the end, to use any random part of Tcllib requires either:

1. The entire Tcllib needs to be installed locally
2. The local system has the ability to grab packages on demand
3. A packaged application has sorted out all of the dependencies ahead of time.

New Targets

Until now, Tcllib only supports pure Tcl packages, and packages which can be enhanced by Critcl. In the wild, however, TOOL will need to support additional design architectures:

1. Pure Tcl Packages
2. Critcl enhanced packages
3. C Dynamic Libraries
4. C Static Libraries
5. Enhanced shells with extra binary capabilities
6. Self contained executables

Design Assumptions

TOOL begins with the assumption that the user is running Tcl8.6 or greater, and the facilities present to read and create Zip files. The facilities are available now in Tcllib to build and extract zip files. The ability to mount a zip file as a file system is available with the VFS extension. If TIP 430 ever gets out of "Pending" status, that same capability will be available in the core.

TOOL Goals

So, what will TOOL be like then? TOOL is a suite of repositories, and not a single "one distribution to rule them all." The Goal for tool is that each project:

1. Focus on a set of related tasks.
2. Implement one or more packages.
3. Organize interdependent packages into modules.
4. Have its own ticket system and code maintainer team.
5. Utilize a common installer/integrator
6. Provide live documentation online
7. Support multiple versions of a framework simultaneously
8. Have curated metadata maintained by the actual developers

Distribution

TOOLs are distributed as some form of web-hosted repository. The initial implementation will focus on distributing code via fossil repositories. If there is community interest (read that someone else will take the time to write and maintain the code) the architecture could easily support GitHub, SubVersion, CVSTrack, etc. The goal is that, in the end, developers can utilize whatever hosting system meets their needs.

tool::core

Part of any good library is a foundation of tools worth using. TOOL the movement will be include **tool** the framework. The **tool** framework introduces several programming style innovations for Tcl programming. These keywords, methods, and notational shorthand add commands to the **oo::define** namespace, and methods to **oo::class** and **oo::object.**

Extensions to [dict]

TOOL relies on extending the **dict** ensemble with several new commands.

dict getnull ***dictValue ?key…?***

This command works like **dict get**, but instead of failing when asked for a non-existent location it returns an empty list.

dict is_dict ***dictValue***

Returns true if the data in *dictValue* contains a key/value list that could be interpreted as a dict. Internally it uses a catch around **dict size** in an effort to prevent the value from being shimmered into a list.

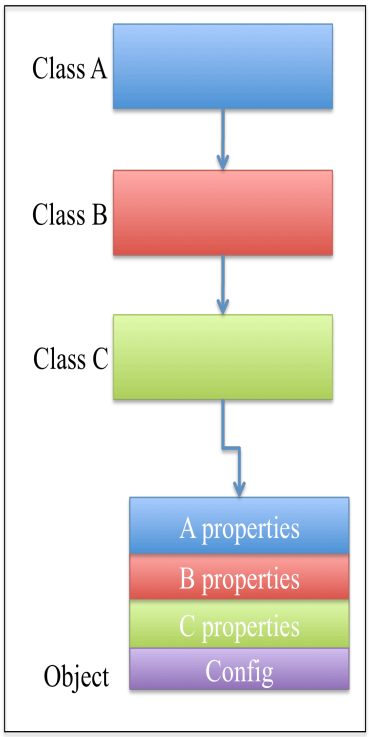
dict rmerge ***dict dict ?dict…?***

Performs a recursive merge on two or more dicts. A value from the rightmost argument overrides any value from an argument to the left. Unlike **dict merge**, this command will descend into the leaves and branches of each dict. Any field which ends in a colon (:) is understood to be a leaf node, and its value will not be broken down further.

dictobj *methodname varname ?cases?*

The **dictobj** keyword defines a method ensemble as well as an internal variable for the class. This ensemble allows the internal variable *varname* to be manipulated in a matter that is similar to the **dict** command in Tcl. Internally; the ensemble is implemented as a **switch** statement. To allow the developer a means of providing custom functions, if a key value list may be given as a third argument. The cases defined by that list are injected before the standard cases. The implementation uses a templating mechanism in which %VARNAME% will be substituted with he variable name and %METHOD% will be substituted with the new method name.

```
oo::define myclass dictobj shed shed {dump {return $%VARNAME%}}
myclass create myobj
myobj shed set name: {The nameless name}
myobj shed dump
> name: {The nameless name}
myobj shed exists name:
> 1
```



meta *submethod ?arg…?*

The meta keyword ensemble allows properties to be defined for a class. Properties are inherited by descendant classes and imparted onto objects of that class. Each object sees all of the properties from its class, and it can overlay data from a local variable *config* over it.

```
oo::define myclass meta set property color: blue
myobj meta get property
> color: blue
oo::class create myclasschild {superclass myclass}
mychildclass meta get property color:
> blue
mychildclass meta set property color: orange
mychildclass meta get property color:
> orange
```

Notifications

TOOL objects can subscribe to receive events from other objects. To facilitate that, all objects have and two additional methods “notify” and “subscribe”.

notify *eventtype cliendata*

Send a notification to all subscribed objects that an event of *eventtype* has occurred. *clientdata* is a free-form block of data that notifier produces and the subscriber is assumed to know how to process.

subscribe *senderpattern eventpattern script*

Arranges to trigger *script* whenever a sender matching senderpattern sends an event with a type matching eventpattern. The following patterns will be substituted into the call:

%sender%	The object which sent the notification
%type%	The type of event
%clientdata%	The block of clientdata generated by the sender to describe the event
%self%	The object which received the event

```
mychild subscribe * heard {puts “%self% heard %sender%”}
myobj notify heard {}
> mychild heard myobj
```

NOTE: Notifications do not use the Tcl event loop. Any script specified by an object is executed directly by the *notify* method. So avoid calling {while 1 {}}, “my destroy”, etc.

Delayed Gratification

When building GUIs, business logic, and otherwise coding a project more complex than “HELLO WORLD!” objects often need to change state. Changing state may simply update an internal variable. But many times altering state requires a major costume change, the reconstruction of an onerous data structure, or setting off a cascade of other state changes.

If these state changes are processed as a pure reflex, the results can be chaos. Especially if altering A triggers changing B, which changes C, which changes A again. To mitigate the Chaos, TOOL borrows a mechanism from Tao called “Signal Pipelines.”

With signal pipelines, the developer builds chains of events, with markers for causes and effects. When an event is received, rather than immediately act on the information, and object logs that event in a list of signals recieved.

When **tool::do_events** is called, objects take turns processing their signal pipelines. Each stage in the pipeline is only processed once per event. When processing the pipeline, the object knows what stage triggers what other stage and what order to execute them in. It then builds a coroutine to exercise every subroutine that was specified in the signal description.

As action scripts form a portion of a coroutine, it is possible to have a pipeline span multiple calls of **tool::do_event** by invoking **yield**. Values returned by **yield** are ignored. A return will terminate the coroutine. Objects are restricted to a single pipeline coroutine at a time. New signals are collected until the current coroutine exits.

Imagine if we were to implement an autoconf style build system.

```
cd $srcpath
./configure
make all
install
```

The signal pipeline for that would be:

```
oo::define buildclass {
  meta set signal {
    configure {triggers: compile action: {puts configure}}
    compile   {triggers: install action: {puts compile} follows: configure}
    install   {action: {puts install} follows: compile}
  }
}
myobj signal configure
tool::do_events
> configure
> compile
> install
myobj signal install
tool::do_events
> install
myobj signal configure install
tool::do_events
> configure
> compile
> install
```

Object Grafting

TOOL objects utilize a technique of deputizing other objects to manage related functions. To do this, every object has two methods **graft** and **organ**.

object **graft** *methodname objname ?methodname objname…?*

Forward all method calls for **<methodname>** to the object **objname**.

object **organ all|***methodname*

Return the object that is handling calls to **<methodname**>. If **all** is specified, return a key value list of **methodname objname** for all methods being forwarded.

```
# Example
myobj graft db ::shed::sqliteobj
myobj <db> onecolumn {select uuid from human where name='hypnotoad'}
> 3b8b8d62-5cc0-48bf-b589-f0d948941d0e
myobj organ db
> ::shed::sqliteobj
```

noop

As a convenience TOOL defines a global command **noop**. **noop** takes any arguments, performs no actions, and returns an empty string. **noop** is designed to be a safe "object" to graft for edge cases where a particular organ would be meaningless. (For example the <parent> of a root node.)

tool::main

While TOOL can interact with the Tcl event model, large complex interactions tend to confuse Tcl. Especially in the case of nested calls to **update**. To this end, TOOL provides a never terminating while loop, which:

- processes scheduled events
- processes object notifications
- processes object coroutines
- calls "update" to allow the Tcl event loop to process

In place of the typical **vwait forever**, simply call **tool::main**. If you prefer to implement your own loop, be sure to make periodic calls to **tool::do_events**.

tool::shed

To meet the need for a common installer interface, a new framework implements the Simple Handy Extension Descriptor (SHED). SHED tackles the problem of understanding the structure of software development project. SHED objects exist in 3 states:

1. As live objects in an interpreter
2. As static data within a SHED descriptor file
3. As static data posted to a URL
4. As static data within an Sqlite database

SHED Descriptor Files

The first and obvious choice for storing objects and data would the JavaScript Object Notation (JSON). It is ready made for this sort of application. However, the JSON notation uses a lot characters that are special to Tcl. To operate in a live interpreter requires conversion to a form more digestible to tcl, namely dicts. XML would be another good choice, but again, to work inside a live Tcl interpreter it too would have to be converted to a dict.

```
trunk {
  leaf: value
  branch {
    leaf: value
  }
  objects/ {
    element {…}
    element {…}
    element {…}
  }
}
```

SHED cuts out the middleman. It is a dict, a stylized dict, but a dict nonetheless. SHED uses the last character of a field to indicate its function.

- Elements that represent subordinate pieces of a data structure have no suffix.
- Elements that are a terminal leaf end with a colon (:)
- Elements that are a list of objects end with a forward slash (/)

Knowing which elements are terminal leaves is essential when combining dicts. It prevents blocks of literal data from being interpreted as key/value lists.

Every SHED compliant project will include a **project.shed** file in the top-level directory of the source code. This file is machine generated by software utilizing the **tool::shed** package. The shed package looks for guidance from human edited scripts located at **shed.tcl**, **support/shed.tcl**, or **script/shed.tcl**. These scripts are actually invoked within a live **project** object, and can employ **my** to exercise the object's methods. When running the **::PROJECT_ROOT** variable will indicate the path of the project being indexed.

SHED Database

Having data running in a live interpreter can be handy for examining data from a single project at a time. But the next great Teapot or Gutter is going to need to assemble all of these views into a multiverse. And to that end, SHED has a database form. Each class of object defines a chunk of a database schema.

SHED Classes

SHED breaks the world into 7 varieties of objects:

project	A top-level project
distribution	A means of distributing a *project*, *products,* or both.
release	A distinct version of a project, packaged by a *distribution*
package	A package callable from **[package require]**
module	A collection of source files or installation products
human	A human point of contact for any of these objects. Humans may be responsible for authorship, maintenance, spiritual guidance, testing, etc.
product	A human or machine-readable block of data. Examples: images, game maps, documentation.

All classes for the SHED are contained in the **tool::shed** namespace. Non-exclusive variations of classes are nested with dots (.). Exclusive variations of classes are nested with slashes (/). examples:

Class	Description
::tool::shed::distribution	Base class for distributing software
::tool::shed::distribution.http	Class which implements mechanisms for http distribution
::tool::shed::distribution/fossil	A specific of distribution mechanisms that employs fossil
::tool::shed::distribution/github	A specific of distribution mechanisms that employs github

The goals of this convention are threefold.

1. Avoiding murky relationships between classes
2. Allow for machine-readable shorthand
3. Allow SHED to be run inside of other programs

If we know a database property contains the name of a class, we can avoid having to store the entire path to the class. Users can fill in simply “fossil” instead of ::tool::shed::distribution/fossil.

Object Methods:

Each class fills out a template of public access function.

constructor / destructor

SHED objects are implementing a JSON-like document with nested objects. To that end, other SHED objects most often spawn them. Rather than try to spell all of this out in English, it’s probably better to express in script:

```
constructor {superiorObj ObjName {script {}}} {
  tool::object_create [self]                          # Register an object with the TOOL Framework
  my variable subordinates                            # Initialize the dict which tracks
  set subordinates {}                                 # subordinate objects
  my shed set name: $ObjName                          # Initialize the name: field
  if {$superiorObj eq {}} {
    my graft superior ::noop                          # This object has no parent
  } else {
    my graft superior $superiorObj                    # This object has a parent, graft it
    my graft {*}[my <superior> exported_objects]      # And graft in any organs the parent exports
  }
  my Shed_Script $script                              # Read this object's shed descriptor
}
destructor {} {
  tool::object_destroy [self]                         # Unregister the object from TOOL
  catch {my <superior> subordinate_unregister [self]} # Unregister the object from its parent
}
```

compute_uuid

Generate a UUID appropriate for this class. For most classes, this is a call to **[uuid::uuid generate]**. However, some varieties of objects may choose to generate UUIDs based on fingerprints provided by file object hashes, SCM tags, etc.

export

Return a snippet of SHED that describes this object and all subordinate objects.

exported_objects

Returns a key/value list of objects that any subordinate would need to graft to itself in order to operate.

identify

Returns the name by which this object would like to be known. Usually the **name:** field from shed.

search_compare *other_object*

Compare the current object to *other_object*. Return:

-1	This object should appear before *other_object*
0	This object is equivalent to *other_object*
1	This object should appear after *other_object*

search_match *args…*

Return true if this object matches the search criteria specified by the arguments. False otherwise.

Shed_Links

Called by **Shed_Script**. Generate any implied links between this object and other objects based on the spec parsed by **Shed_Script**.

Shed_Script *SHED*

Parse a block of SHED data and generate properties and subordinate objects.

Shed_Script_Level *path SHED*

Invoked from **Shed_Script**. Parse the topmost level of keys and values from SHED, farming any subordinate elements to recursive calls to **Shed_Script_Level** or to objects generated by the rules of **SHED**.

sql_delete *dbobject uuid*

Deletes the SQL representation of this object and its links.

sql_export *dbobject*

Generates an SQL description of this object and its links and injects that information into the Sqlite connection at *dbobject*.

sql_import *dbobject ?uuid?*

Populates this object with data from *dbobject*, and generates and/or populates any subordinate objects with data from *dbobject.* If *uuid* is specified, it will replace any stored *uuid* in this object. If not specified, a previously configure *uuid* will be used.

subordinate_unregister *object*

Called by destructor of objects generated by this object. Destroy any links that this object may have shared with the other object.

Class Methods:

Some rules of a class need to be applied without creating an object of that class. SHED classes provide the following methods:

scan *filepath*

Return a SHED description of the file or directory specified by *filepath*. Returns an empty list if :

- the file path does not exist
- the content is not appropriate for describing in SHED
- the path contains no data
- this class has no concept of files or directories

Dynamically Generated Methods

To implement the parser, the SHED module builds a container method into for each shed class to manage children of the other classes. This method is an ensemble that provides the following sub-commands:

class add *objname SHED*

Add a new object and link it to this class as *objname*, and impart on it the descriptor given by *SHED.*

class best *args…*

Return the identity of the subordinate of type *class* which best matches the search specified in *args*.

class best_object *args…*

Return the object of the subordinate of type *class* which best matches the search specified in *args*.

class **delete** ***objname***
Destroy a subordinate object of type *class* and named *objname*.

class **export**
Generates a SHED descriptor for all subordinate object of type *class*.

class **exists** ***name***
Returns true if an object of type *class* and named *name* exists.

class **find** ***args…***
Return a list of identities of objects that matches the description in *args*.

class **import** ***SHED***
Generates objects of type *class* from the SHED descriptor *SHED*.

class **info** ***args…***
Find the best object described by *args* and return a dump of its **shed** data dict.

class **link** ***name object***
Generate a link of type *class* to an existing object *object* and call it *name*.

class **list**
Generate a list if identifiers for all links to objects of type *class*.

class **object** ***args…***
Return a list of objects that matches the description in *args*.

class **scan** ***filepath***
Scan *filepath* and generate any subordinate objects that would represent *filepath*.

tool::shed::human

Introduction:

A *human* is a reference to a human being or organization that authors, maintains, or distributes code.

Expected Properties:

Field	Format	Description
handle:	String	Canonical handle of this person or organization within the community (i.e. hypnotoad)
description:	String	Human readable description of this person or organization
uuid:	GUUID	A completely fictitious global UUID to track this individual by. Helpful because names are not unique and are subject to change.
full-name:	String	The full name of the individual or organization
email:	Email address	An email address
url:	URL	A URL to this individual or organization's home page

Database Schema:

```
create table human (
  uuid uuid primary key,
  handle string,
);
create table human_property (
  human uuid,field string, value value,
  PRIMARY KEY (human,field),
  FOREIGN KEY(human) REFERENCES human(uuid) ON UPDATE CASCADE
);
create table human_object (
  human uuid,
  linktype string,
  objtype string, --application, human, package, etc.
  objuuid uuid, --Key in foreign table
  FOREIGN KEY(human) REFERENCES human(uuid) ON UPDATE CASCADE
);
```

::tool::shed::project

Description:

A *project* represents a project, online or on a developer's local machine. The main **project.shed** file defines a project object.

Expected Properties:

Field	Format	Description
description:	Markdown	A human readable description
distribution:	String	The name of the currently selected distribution
generated:	UTC Time code	Time stamp of the release that generated this shed file
release:	String	The name of the currently selected release
uuid:	GUUID/SHA1 Hash	A unique identifier for this project.
name:	String	Name for this project
short-name:	String	Shortened version of the name for the purposes of building file paths, zip files, etc.

Database Schema:

```
create table project(
  uuid uuid primary key,
  name string
);
create table project_property (
  project uuid, field string, value value,
  PRIMARY KEY (project,field),
  FOREIGN KEY(project) REFERENCES project(uuid) ON UPDATE CASCADE
);
```

::tool::shed::distribution

Introduction:

A *distribution* is means of getting the source code or compiled product to the user.

Expected Properties:

Field	Format	Description
class:	Class name	Type of distribution. (i.e. fossil, github, tarball, teacup, zip)
url:	URL	Location for the project online
mirrors:	URL list	List of alternate locations

uuid:	GUUID	For official distributions, this UUID will be identical to the UUID of the project. Alternative incarnation of a project will have a different UUID.
name:	String	A name describing this distribution. The default distribution for most projects should be called “official.”

Database Schema:

```
create table distribution (
  uuid uuid primary key,
  project uuid, name string,
  FOREIGN KEY(project) REFERENCES project(uuid) ON UPDATE CASCADE
);
create table distribution_property (
  distribution uuid, field string, value value,
  PRIMARY KEY (distribution,field),
  FOREIGN KEY(distribution) REFERENCES distribution(uuid) ON UPDATE CASCADE
);
```

::tool::shed::release

Introduction:

A *release* is a bundle of source code, executables, and/or data sets. Each release is considered a child of a distribution. Distributions can have multiple releases. Releases are also given a version number so that can be selected in numerical(ish) order.

Expected Properties:

Field	Format	Description
checkout:	SCM Tag	The tag from the SCM this release is derived from
class:	Class name	Type of release. Valid values are descendants of the ::tool::shed::release class.
distribution:	List of distribution objects	Which distribution this release is associated with.
profile:	Teacup Profile	For binary release, the teacup standard profile name
status:	Selection	A description of the vitality of this release. Valid: development, release, stable, deprecated.
timestamp:	UTC Time stamp	The latest check in from this release as of when the SHED description was generated.

Database Schema:

```
create table release (
  uuid uuid primary key,
  version version, distribution uuid, name string, profile string,
  FOREIGN KEY(distribution) REFERENCES distribution(uuid) ON UPDATE CASCADE
);
create table release_property (
  release uuid, field string, value value,
  PRIMARY KEY (release,field),
  FOREIGN KEY(release) REFERENCES release(uuid) ON UPDATE CASCADE
);
create table release_object (
  release uuid,
  linktype string,
  objtype string, --application, module, package, etc.
  objuuid uuid, --Key in foreign table
  FOREIGN KEY(release) REFERENCES release(uuid) ON UPDATE CASCADE
);
```

::tool::shed::module

Introduction:

A *module* is a suite of re-usable code bundled into a *release.* A *module* can contain one or more packages.

Expected Properties:

Field	Format	Description
author:	Human UUID	The original author of this module
class:	Class name	Type of module. Valid values are descendants of the ::shed::class::module class.
description:	String	Human readable description of the module
maintainer:	Human UUID	Point of contact for support
name:	String	Name of module
origin:	Release UUID	If this module is a copy from another project, which release
path:	Local file path	Path relative to the PROJECT_ROOT where this module can be found
pkg-provides:	{PKG VER} …	List of packages and versions provided by this module
pkg-requires:	Package List	List of packages that are called on by this module
sources:	List of files	List of source files that implement this module
version:	Version string	Version number or sequence of this release

Database Schema:

```
create table module (
  uuid uuid primary key, name string
);
create table module_property (
  module uuid,field string, value value,
  PRIMARY KEY (module,field),
  FOREIGN KEY(module) REFERENCES module(uuid) ON UPDATE CASCADE
);
create table module_object (
  module uuid,
  linktype string,
  objtype string, --application, module, package, etc.
  objuuid uuid, --Key in foreign table
  FOREIGN KEY(module) REFERENCES module(uuid) ON UPDATE CASCADE
);
```

tool::shed::product

Introduction:

A *product* is a human or machine-readable block of data bundled into a *release*.

Expected Properties:

Field	Format	Description
class:	Class name	Type of product. Valid values are descendants of the tool::shed::product class. Examples: application, image, teapot
description:	Markdown	Human readable description of the product.
module:	UUID	Which module this product is derived from
name:	String	Name of product
pkg-provides:	Package List	List of packages this product provides when installed
pkg-requires:	Package List	List of packages that need to be present in the environment for the product to be interpreted or displayed.
profile:	Teacup Profile	Which teacup profile does this product target.

Database Schema:

```
create table product (
  uuid uuid primary key,
  release uuid, module uuid, name string, profile string,
  FOREIGN KEY(release) REFERENCES release(uuid) ON UPDATE CASCADE
  FOREIGN KEY(module) REFERENCES module(uuid) ON UPDATE CASCADE
);
create table product_property (
  product uuid,field string, value value,
  PRIMARY KEY (product,field),
  FOREIGN KEY(product) REFERENCES product(uuid) ON UPDATE CASCADE
);
create table product_object (
  product uuid,
  linktype string,
  objtype string, --application, module, package, etc.
  objuuid uuid, --Key in foreign table
  FOREIGN KEY(product) REFERENCES product(uuid) ON UPDATE CASCADE
);
```

tool::shed::package

Introduction:

A *package* is a tcl package that is callable with **[package require]**. Packages are linked to products and modules for installation and dependency resolution.

Expected Properties:

Field	Format	Description
author:	Human UUID	The original author of this application
class:	Class name	Type of application. Valid values are descendents of the tool::shed:package class.
description:	Markdown	Human readable description of the application
file:	Local file path	For scripts, a path relative to the PROJECT_ROOT which represents the top-level script to be invoked
maintainer:	Human UUID	Point of contact for support
name:	String	Name of executable
pkg-requires:	Package List	List of packages this package depends on

Database Schema:

```
create table package (
  uuid uuid primary key,
  release uuid, name string, version string,
  FOREIGN KEY(release) REFERENCES release(uuid) ON UPDATE CASCADE
);
create table package_property (
  package uuid, field string, value value,
  PRIMARY KEY (package,field),
  FOREIGN KEY(package) REFERENCES package(uuid) ON UPDATE CASCADE
);
```

Getting TOOL and SHED

tool::core, **tool::shed**, and **tool::scgi** are developed as part of the **sherpa** project. (http://fossil.etoyoc.com/fossil/sherpa) All of the modules for **tool** are mirrored to Tcllib. Sherpa also mirrors modules from tcllib upon which it depends. Tool code is checked into the **odie** branch of tcllib, which is periodically merged into the trunk branch.

Using TOOL

Tool ships with a **shed** application script. The shed script maintains a configuration in the OS user's home directory. Shed understands several commands:

shed configure *setting ?value?*

Retrieve of set the local preference of *setting*. Setting can be one of the following:

profile	Teacup compatible profile to assume if not stated Default: Auto-detect
prefix	Top-level location where SHED operates Default: $HOME/tcl
sandbox	Location where source code is unpacked Default: $prefix/sandbox
download	Location where repository databases, tarballs, etc should be downloaded Default: $prefix/download
database	Location of the master shed index Default: $prefix/var/shed.sqlite
app-path	Location to install applications Default: $prefix/bin
lib-path	Location to install libraries Default: $prefix/lib
tclsh	Tcl Shell to invoke for scripts Default: Auto-Detect
user	Configure the handle of the local user. Default: anonymous

shed database-rebuild

This function will search for any project.shed file located within the *$sandbox*. Each file will be read, and indexed, and compiled into a fresh copy of the *$database*.

shed env-build

Place an alias in the current OS user's ~/.aliases or ~/.bashrc file to allow him/her to invoke **shed** as a local command. This command will also inject *$lib-path* into the *auto_path* of the user's .tclshrc file.

shed package-install-local *package ?version?*

Find a Tcl package as well as its dependencies in the local shed database, and install them to *$prefix*.

shed shed-build *filepath ?shedscript?*

Explores a code repository rooted at *filepath*, figures out its distribution mechanism, takes a guess at its structure, and writes a **project.shed** file into the root. If a *shedscript* is given, the main project object sources that script. If not *shedscript* is given, the discover tool will look for one at **shed.tcl**, **scripts/shed.tcl**, or **support/shed.tcl**.

```
#Simple shed script
my human add hypnotoad {
  role: author
}
my application scan $::PROJECT_ROOT/apps/shet.tcl
foreach path [glob $::PROJECT_ROOT/modules/*] {
  my module scan $path
}
my release add autosetup {
  description {An excursion built around autosetup}
  checkout: autosetup
}
```

A shed script written to operate within the project object, and provide a picture of the file structure for this repository, as well as any human curated metadata that would be impossible (or impractical) to generate automatically.

Prior to invoking the shed script, the discovery system will automatically extract data from fossil repositories, and auto-generate a placeholder distribution (official) and release (whatever the current checkout is tagged as).

shed shed-discover *shedfile|url*

Read the contents of a local file or URL, and translate that data into records for the local SHED database.

shed update

Download and apply updates to the SHED database from the teapot as well as any fossil repositories unpacked into local *$sandbox*.

shed vfs-index *vfsroot*

Produce a *pkgIndex.tcl* file that contains a master index of every package loadable within *vfsroot*.

shed vfs-install *vfsroot profile {package ?version?} ?{package ?version?}…*

Find a Tcl package as well as its dependencies in the local shed database, and install them to the *shed* folder within a virtual file system under construction at *vfsroot.*

Submitting New Tools

TOOL does not depend on a centralized management system. Which is fortunate, because I still haven't worked out how to build one. My tentative scheme is to provide a central directory where developers and distributers can register, and can post URLs to the SHEDs they maintain. Better ideas or help to that end are welcome and appreciated.

Work still to be done

At the time this paper was written, I have finished the **tool::core** and **tool::shed** packages, and their associated test suites. There is a lot left to do. And any help would be appreciated!

Teacup Integration

Through a combination for working with ActiveState and Roy Keene's "Teaparty", provide a bilateral exchange of data between Teacup and SHED. At present, "Teaparty" lacks a database backend, and SHED lacks a "search and install a package" capability. (Project codename: **tool::store**.)

Sherpa Integration

The ODIE project has an automated build tool called Sherpa. This tool is already OO based, and provides a lot of ready-made recipes for building common Tcl extensions on command. Recipes are manually built, and distributed by checking scripts into a fossil repository. SHED could provide a better mechanism for both generating recipes and delivering them. (Project codename: **tool::kit**.)

NetTOOL

NetTool is envisioned as a framework for network protocol development. The goal is to provide an assortment of ready-to-use design patterns for mail exchange, web services, and UDP discovery. Much of this work will involve porting existing Tcllib code to the TOOL framework. (Project codename: **tool::socket**.)

WebTOOL

WebTool is a framework for building dynamic web content, mainly through SCGI. Work has begun on this with the upcoming SCGI package for Tcllib. The SCGI package is already TclOO based, and ready to be made into a TOOL. The key will be to backstop it with a suite of ready to use functions. (Project codename: **tool::shiny**.)

COOL

The ODIE project has another bit of technology that could enhance/be enhanced by **tool**. The C and Tcl Hashtable Language for High Level Understanding (CTHULHU). This project uses Tcl to build C code as well as to operate the underlying build system. The intent is to adapt CTHULHU's notation to utilize SHED, and also re-engineer Cthulhu to utilize the C API of TclOO. (Project codename: **CTHOOLHU**)

Feedback

Ideas, comments, code submissions, and large denominations of charitable contributions can be sent to me, Sean Woods: yoda@etoyoc.com

Credits

Cover Image: "Neolithic Implements, Figure 79", *The Outline of History*, H.G.Wells
URL: http://www.ibiblio.org/pub/docs/books/sherwood/Wells-Outline/Images/0079img.htm

Projects Mentioned:

Fossil	http://www.fossil-scm.org
Odie:	http://fossil.etoyoc.com/fossil/odie
Sherpa	http://fossil.etoyoc.com/fossil/sherpa
Sqlite	http://www.sqlite.org
Tcl Core	http://core.tcl.tk/tcl
Tcllib:	http://core.tcl.tk/tcllib
Teaparty:	http://teaparty.rkeene.org/fossil