Quill:
A Development Automation System for Tcl/Tk

By William H. Duquette

Abstract

Leiningen is a sophisticated build system for the Clojure programming language, used by both novices and skilled developers. Leiningen creates new skeleton project trees, acquires and manages external dependencies (including Clojure itself), provides build and test services, and deploys projects into the larger Clojure ecosystem. This paper examines Leiningen and describes Quill, an attempt to begin to create a similar tool for Tcl/Tk development.

1. The Leiningen Advantage

Clojure [1] is a LISP-like language that targets the Java Virtual Machine. Unlike most LISPs it embraces its host environment; and like Tcl/Tk it has excellent support for metaprogramming and embedded domain-specific languages. While investigating Clojure, I soon discovered that the essential Clojure development tool is a build-and-automation tool called Leiningen [2]. The following is a genuine Leiningen dialog:

```
$ lein new app myapp
Generating a project called myapp based on the 'app' template.
$ cd myapp
$ lein run
Hello, World!
$ lein test
lein test myapp.core-test
...
Ran 1 tests containing 1 assertions.
1 failures, 0 errors.
Tests failed.
$ lein repl
...
myapp.core=> (+ 1 1)
2
myapp.core=> (exit)
Bye for now!
$ lein uberjar
Compiling myapp.core
Created /Users/will/github/myapp/target/uberjar/myapp-0.1.0-SNAPSHOT.jar
Created /Users/will/github/myapp/target/myapp-0.1.0-SNAPSHOT-standalone.jar
$ java -jar ./target/myapp-0.1.0-SNAPSHOT-standalone.jar
Hello, World!
$
```

In moments a naïve user can create a new project tree for the desired kind of project, complete with skeleton modules and an executable stub, execute the stub in development, run the skeleton test suite, do interactive testing at a Clojure REPL, (a Read/Eval/Print Loop, the equivalent of a Tcl shell), and deploy the new application as a standalone .jar file. The code must be written; but the skeleton is in place and is known to work.

The details of the project are captured in the project file, `project.clj`. This is a Clojure-formatted file that defines the project's name, version number, description, and a host of other information, including external dependencies. Here is the project file from a simple text adventure I was working on:

```
(defproject whd-advent "0.0.1-SNAPSHOT"
  :description "Will's Text Adventure"
  :dependencies [
     [org.clojure/clojure "1.6.0"]
     [org.clojure/core.unify "0.5.5"]]
  :javac-options ["-target" "1.6" "-source" "1.6" "-Xlint:-options"]
  :aot [whd-advent.core]
  :main whd-advent.core)
```

If I execute "`lein run`" for this project, Leiningen will verify that it has the referenced dependencies—which is to say, Clojure itself and a unification-matching library. If not, it will go out and retrieve them from a repository on the web. The external libraries can be written in any JVM language, and can be retrieved from any Maven [3] repository. (Leiningen is compatible with Maven, but is simpler to use.)

Once a Clojure package is complete, Leiningen will happily push it out to any desired Maven or Clojure repository (usually `clojars.org`):

```
$ lein deploy clojars
```

It can also install the package into a local repository on the user's own machine, for use with the user's other projects. Finally, Leiningen is extensible; plugins can add new Leiningen subcommands and new project tree templates.

The essential insight for Leiningen is that there is a nearly infinite number of ways to set up a project tree; and all that's needed is one way that's good enough. And then, a good project tree is the result of many small decisions, many of which have hidden long-term consequences; by setting up a standard "good-enough" project tree Leiningen can spare the user many troubles.

Virtually all of these capabilities are available for Tcl/Tk in form or another, but they are not collected together in one place, nor are they as straightforward to use.

I looked at Leiningen and said, "Tcl newbies need a tool like this." And then, after a moment, I said, "***I*** need a tool like this." Quill is the result. [4]

2. The Quill Vision

After examining Leiningen and my own development process and standards, I came up with the following list of requirements.

- Leiningen-like simplicity. It must be easy for a novice to use.
- Power and flexibility. Hard things must still be possible.
- Automation based on a single project metadata file.
- Creation of project trees based on templates.
- Management of external dependencies.
- Ability to run project applications with minimal effort (i.e., no need to set TCL_LIB_PATH).
- Ability to run arbitrary scripts in the project context (i.e., with easy access to the project code base).
- Interactive testing and development in the project context in a shell window.
- Test suite management.
- Ability to build (or invoke the commands to build) non-Tcl build targets.
- Building Tcl libraries and applications for deployment (e.g., starkits, starpacks).
- Building distribution .zip files based on user file patterns.
- Formatting of project documentation for use with web browsers.
- Installing applications and libraries on the user's machine for general use.
- Deploying libraries to a remote repository for use by others.
- Cross-platform tool (at least Linux, Mac OSX, and Windows).
- Command-line tool, for power users.
- GUI tool for casual users and project visualization, including code browsing.

All of these goals are possible; some are trivial, some are easy, some are doable, and some will require help from other projects (e.g., deploying libraries to a remote repository in an automated way).

3. Available Pieces

Practically speaking, a tool as ambitious as Quill will succeed only by riding on other's coattails, at least in the near term, much as CVS began as a collection of scripts built on top of RCS. I chose to base Quill on the following pieces of technology:

3.1 Zip File Management

Zipping and unzipping .zip files is a generally useful capability for a tool like Quill. Tcllib includes two useful packages, `zipfile::encode` [5] and `zipfile::decode` [6], that handle most of the work.

3.2 Teapots and Teacups

The *de facto* system for retrieving Tcl packages over the Internet is ActiveState's teacup/teapot system. One of the essential features of Quill is the automatic retrieval of external packages, and since development of a new network-accessible package repository is out-of-scope, and since I generally use ActiveTcl [7], I standardized on teacup/teapot. This has been generally fruitful, with only a few unpleasant warts to work around.

The teapot repository at `teapot.activestate.com` not only provides a source for popular packages, but the local teapot repository on the user's own machine provides a place to install the user's packages for general use by his other projects. It turns out to be straightforward to wrap up an existing Tcl package as a .zip file that can be installed in the user's local teapot: just add a manifest file called `teapot.txt`, and zip up the package directory.

The most serious issue with using the `teacup` tool is its interaction with the host privilege system. When you install ActiveTcl, you get a local teapot repository; and when ActiveTcl is installed for all users this repository is located in a directory that requires admin or root privileges to update. That means that on Un*x-like systems you generally can't install packages without using `sudo`, which is a significant usability problem.

Quill escapes this by creating a second teapot in the user's home directory, and installing packages into that. Setting this up still requires some use of `sudo` in many cases, and is rather ugly. Then, the user's teapot must be linked to the `tclsh` (or the `tclsh` will not look at it); and this links the user's teapot for all users on the machine.

The solution, as I see it, is this: ActiveTcl should automatically and always look for a teapot in the user's home directory in a specified place; and unless directed with options, `teacup` should always operate on the teapot in the user's home directory, leaving the teapot included in the installation as a purely global resource. If this were done, no functionality would be lost; with an option, an admin could install a package for global use (using `sudo` if need be), while individual users (and Quill!) could use `teacup` freely without need for `sudo`.

3.3 Building Starkits and Starpacks

There are a number of solutions available for building stand-alone Tcl/Tk executables. I chose to standardize on the `tclapp` tool from ActiveState's TclDevKit [8] for two reasons. First, it works well with teapot repositories; it will pull packages from both local and remote teapots as needed. Second, I've been using `tclapp` for years, and I'm familiar with it.

In time I would expect Quill to provide other methods as well.

3.4 Choice of Tcl Shell

Given all of the above, it's no surprise that Quill assumes that the user is using some flavor of ActiveTcl. ActiveTcl provides `teacup` and the access to teapots; it also provides the basekits needed for building applications with `tclapp`. Quill can indeed be used with arbitrary Tcl shells, but features that require access to teapots or `tclapp` will be unavailable.

3.5 Interactive Development

The standard `tclsh` command line is notoriously unpleasant to use, given that it lacks "readline"-style command editing; and the usual work-around is to use a home-grown replacement or `tkcon` [9]. Happily, `tkcon` is delivered with ActiveTcl, and Quill makes full use of it.

3.6 Formatting HTML Documentation

There is no agreed standard for formatting Tcl/Tk documentation. The Tcl/Tk core uses one system that has not caught on (so far as I can tell) with the wider community. Tcllib uses `doctools` [10], which is based on my own `textutil::expander` [11] package; but similarly, `doctools` hasn't caught on outside of Tcllib. I've been using Tcl scripts to format HTML pages since at least 1999; my first major Tcl project was `expand` [12], the precursor to `textutil::expander`, and looking back on it I seem to have spent an inordinate fraction of my life, both at work and at home, on different systems for formatting HTML software documentation using `textutil::expander` macros.

Quill includes my latest iteration of this system. The input is HTML-like, but not HTML, as all tags are actually macros implemented in Tcl. It includes two distinct targets: software manual pages, and classic documents with sections, section numbering, and tables of contents.

At present the system produces HTML only, with CSS used to control the appearance. The CSS styles are currently fixed, but could in time be made available for modification by the user. The system is not implemented to support multiple back-ends; but the documentation markup is designed so that in principle this could be done.

3.7 Object System

Because Quill draws on quite a lot of infrastructure I'd written for other personal projects, I'm quite naturally using Snit [13] as my object system.

4. Quill at Three Months

I began working on Quill in early July of 2014, about three months before time of writing; already it meets a surprising number of the requirements laid out in Section 2. It's a young project, but parts of it (notably the infrastructure layer and the documentation code) draws on my own previous work; and of course it outsources many things to the tools listed in Section 3.

Quill is built as a standalone executable (a starpack). I'm developing it on Mac OSX, and building it for OSX, 64-bit Linux, and 32-bit Windows. It is a command-line tool with subcommands *à la* `cvs`, `svn`, `fossil`, `git`, `lein`, *et al*.

4.1 Philosophy

The classic Tcl/Tk usage pattern is this: you install Tcl/Tk on the system in a central location, and then you just write scripts. If you want access to third-party packages, you acquire them, install them in a central location, possibly rebuild Tcl to automatically look in that location, and then you just write scripts.

This is fine for *ad hoc* scripts, and it's friendly for novice users who can sit down at a computer for which all of this has already been set up. It's less friendly for users who have to set it up themselves; and it isn't friendly at all for novices or other users who have written a script and want to share it with others, especially if the script is complicated enough to warrant breaking it into multiple files or has any significant external dependencies.

Quill embodies a different model of development. You install Tcl on your system and create a project. For each project, you specify in the project metadata file what its external dependencies are. Quill goes to the net to acquire those dependencies for local use, and builds them into the finished application. Quill also supports the creation of new libraries that can be deployed for use as external dependencies by other Quill projects. Each project is its own world.

And then, Quill makes it easy to package up your finished application and give it to others to use, with a fair degree of confidence that it will work.

4.2 Creating a Project

To create a new project tree, use the `quill new` command, specifying the template, the project name, and template specific parameters. The following creates a new project that defines a skeleton application, to be delivered as a starkit. The project will be called "my-project", and the application will be called "myapp".

```
$ quill new app my-project myapp
Creating an "app" tree at /Users/will/my-project/...
$ cd my-project
```

4.2.1 The Project Tree

The resulting project tree looks like this:

`project.quill`	The project metadata file
`README.md`	A skeleton README file
`bin/`	Application loader scripts and ancillary tools
`myapp.tcl`	Application “myapp”’s loader script
`docs/`	Root of the project’s documentation tree
`index.quilldoc`	Index page for project documentation, in quilldoc(5) format.
`man1/`	Man page section 1: applications
`myapp.manpage`	“myapp”’s man page, in manpage(5) format.
`lib/`	Root of the project’s Tcl source tree
`quillinfo/`	Project metadata package
`pkgIndex.tcl`	Package index file
`pkgModules.tcl`	Package loader script
`quillinfo.tcl`	Package code
`app_myapp/`	“myapp”’s application implementation package
`pkgIndex.tcl`	Package index file
`pkgModules.tcl`	Package loader script
`main.tcl`	Contains the “`main`” for “myapp”.
`test/`	Root of the project’s test suite tree
`app_myapp/`	Test directory for package `app_myapp`.
`all_tests.test`	Tcltest(n) test script for all tests in directory
`app_myapp.test`	Tcltest(n) test script for one module (edit and copy as needed)

4.2.2 The Project File

The Quill project metadata file, `project.quill`, is a Tcl-syntax file that defines the project’s metadata. The initial content of the project file is this:

```
project my-project 0.0a0 "Your project description"
homepage http://home.page.url
app myapp
require Tcl 8.6.1
```

The `project` statement defines the project’s name, current version number, and a short description. The project’s version number is also the version number for all of the project’s applications and libraries; it can be changed in this one location, and the changes flow to where they need to go.

The `homepage` statement defines the project home page for inclusion in documentation.

The `app` statement defines an application, "myapp" in this case; options to the statement determine whether the application is GUI or non-gui, and whether it should be built as a starkit or a starpack. By default, the application is a non-gui starkit.

The `require` statement identifies an external dependency. Tcl is always required; the version number was taken from the development `tclsh` installed on the system. Users can require other packages by name and version as found at `teapot.activestate.com`, or as installed into the local teapot. In the latter case the `-local` option indicates that the package is locally built and cannot be retrieved over the net.

The following statements may also be used.

The `provide` statement indicates that the project defines a library package for external use:

```
provide mylib
```

The library must reside in `lib/mylib/`, and must have a `pkgIndex.tcl` and `pkgModules.tcl` file as described below.

The `dist` statement defines a distribution set, giving it a name and listing the file patterns to be included in the distribution:

```
dist install-%platform {
   %apps
   %libs
   docs/*.html
   docs/*/*.html
   docs/*.md
   LICENSE
   README.md
}
```

The name of the set is "`install-%platform`", where "`%platform`" will be replaced by current `platform::identify` string. This is appropriate for a project that provides a starpack or includes external dependencies that are platform-specific. The patterns include straightforward glob patterns relative to the project's root directory and special patterns like `%apps` and `%libs` that expand to the application and libraries appropriate for the "`%platform`". In particular, `%libs` expands to the teapot .zip files for the `provided` libraries, rather than to the source `lib/` subdirectories.

4.2.3 Library Architecture

A Quill project may contain three kinds of library package: provided packages, i.e., packages to be used outside the project; application implementation packages; and application infrastructure packages. All three of these have the same structure. A library packaged called *name* is represented on the disk by the following artifacts:

lib/*name*/	The package's directory
pkgIndex.tcl	The package index file
pkgModules.tcl	The package's module loader script
module.tcl	The package's modules
…	
test/*name*/	The package's test directory
all_tests.test	A tcltest(n) script for running all of the package's tests
module.test	A tcltest(n) script for lib/*name*/*module*.tcl
…	

In a Quill project, the project version number and the version numbers for all external dependencies are recorded in `project.quill`. However, Tcl also expects to find them a number of other places. This file architecture allows Quill to insert the required version numbers where they belong, so that the application runs correctly whether it is running in development as a plain family of Tcl scripts or as a starkit or a starpack, and so that provided libraries carry the project version number with them.

The `pkgIndex.tcl` file looks like this (shorn of boilerplate comments):

```
# -quill-ifneeded-begin DO NOT EDIT BY HAND
package ifneeded app_myapp 0.0a0 [list source [file join $dir pkgModules.tcl]]
# -quill-ifneeded-end
```

Whenever `project.quill` changes, Quill automatically looks in all `lib/pkgIndex.tcl` files and updates the code between the "`-quill-ifneeded-*`" marks to match the project version number and library name.

The `pkgModules.tcl` file looks like this:

```
# -quill-provide-begin DO NOT EDIT BY HAND
package provide app_myapp 0.0a0
# -quill-provide-end

# -quill-require-begin INSERT PACKAGE REQUIRES HERE
package require snit 2.3
# -quill-require-end

namespace eval ::app_myapp:: { variable library [file dirname [info script]] }
source [file join $::app_myapp::library main.tcl]
...
```

Again, Quill updates the marked sections automatically when `project.quill` changes. The code between the "`-quill-provide-*`" marks is updated automatically to reflect the library name and project version number. The code between the "`-quill-require-*`" marks is scanned, and for any packages listed in project.quill the package version number is updated to match that in `project.quill`. Finally, the `pkgModules.tcl` file ends with one source command for each of the project's modules.

The library's modules can of course contain any desired code, but package require statements should generally appear only in `pkgModules.tcl`.

4.2.4 Application Architecture

A Quill application "*name*" consists of an application loader script, `bin/`*`name`*`.tcl`, and an application implementation package called "`app_`*`name`*". The loader script is a boilerplate file provided by the template; it sets up the `auto_path`, requires Tcl (and Tk if the application is a GUI application) and the application implementation package, and then calls

```
main $argv
```

The `quill new app` command creates the application implementation package with one module, `main.tcl`, which defines a stub `main` proc. In all other regards, the application implementation package is a normal package with the architecture and constraints described in the previous section.

4.2.5 The `quillinfo` package

For projects that define an application, Quill automatically creates a package called `quillinfo`. This package contains commands allow the project to query the project metadata, and particularly the project name and version number. The API is defined in the Quill documentation tree.

4.2.6 Documentation Tree

Quill makes the following assumptions about the contents of the documentation tree:

- Files with the .manpage extension in the `docs/man1/`, `docs/man5/`, `docs/mann/`, and `docs/mani/` subdirectories are in manpage(5) format and should be processed as manual pages. (The sections are Applications, File Formats, Tcl Commands, and Tcl Interfaces.)
- Files with the .quilldoc extension in `docs/` and its non-`man*` subdirectories are in quilldoc(5) format and should be processed accordingly.

The `quill docs` command will process any or all of the files in the tree.

4.2.7 Test Tree

Quill assumes that projects use tcltest(n) to organize their test suites. The project can define any number of test targets; each target is represented by a subdirectory of `test/`. Typically, test targets are associated with library packages, but this is up to the user. Each subdirectory contains these files:

`test/`*`target`*`/`	The test target directory
`all_tests.test`	A tcltest(n) script for running all of the target's tests
`name``.test`	An individual tcltest(n) script
…	

The all_tests.test script is usually generated by Quill. It is pure boilerplate, and looks like this (shorn of boilerplate comments):

```
if {[lsearch [namespace children] ::tcltest] == -1} {
    package require tcltest 2.3
    eval ::tcltest::configure $argv
}

::tcltest::configure \
    -testdir [file dirname [file normalize [info script]]] \
    -notfile all_tests.test

::tcltest::runAllTests
```

There is nothing remarkable about it; it simply directs tcltest(n) to run the other test scripts in the same directory, each in its own instance of `tclsh`. In addition, it passes any command-line options on to tcltest(n). The individual test scripts, especially those for a specific library package, have the following contents:

```
if {[lsearch [namespace children] ::tcltest] == -1} {
    package require tcltest 2.3
    eval ::tcltest::configure $argv
}

namespace import ::tcltest::test

source ../../lib/app_myapp/pkgModules.tcl
namespace import ::app_myapp::*

# Setup: TBD

test dummy-1.1 {dummy test} -body {
    set a false
} -result {true}

::tcltest::cleanupTests
```

Note that we load the package's code using `source` rather than `package require`; otherwise, each test script would need to be updated with the project's version number each time it changed.

4.3 Quill Capabilities

At time of writing, Quill has the following capabilities:

`quill build`
Builds particular build targets. By default it wraps all provided libraries as teapot .zip archives and all applications as either starkits or starpacks, as indicated in `project.quill`. The `quill build all` command checks for up-to-date external dependencies, runs the test suite, formats all documentation, builds the libraries and applications, and zips up all defined distribution sets.

`quill config`
Allows the user to query and modify the Quill configuration file. At present, this is mostly used to configure which external tools Quill is to use (i.e., which `tclsh`.)

`quill deps`
Manages external dependencies. By default, it checks whether the external packages required in `project.quill` are present in the local environment. Alternatively, it can attempt to download all missing packages, or refresh particular packages.

`quill dist`
Build distribution .zip files. The user can define any number of distribution sets using the "`dist`" statement in project.quill, i.e., a source distribution, a documentation distribution, or an installation distribution. The user can control precisely which files go into any distribution. This package builds any or all of the distribution set .zip files.

`quill docs`
Formats the project documentation. This consists of quilldoc(5) files in `docs/` and its subdirectories, and manpage(5) files in the optional `man1/`, `man5/`, `mann/`, and `mani/` subdirectories.

`quill env`
Describes the development environment. In particular, it lists the external tools (e.g., `tclsh`, `teacup`, `tkcon`) that Quill has found in the environment, or that have been explicitly configured using `quill config`.

`quill help`
Help for each Quill subcommand, in the usual way.

`quill info`

Displays the project metadata to the console. Because this command also updates metadata in the project's files (as do most) it is a good practice to use `quill info` to verify any edits to the `project.quill` file.

`quill install`

Install applications and libraries into the local environment for use by scripts and other projects. Applications are installed into `~/bin`, and libraries are installed into the local teapot.

`quill new`

Creates new project trees. At present there is a single template, "app"; this is an area for near-term future work.

`quill replace`

Global text replacement across files. Because I often want this and it was easy to do.

`quill run`

Runs the primary application (the first listed in project.quill), passing it any arguments. This is equivalent to entering "`./bin/myapp.tcl`" in the project root directory, but can be done from anywhere in the project tree.

`quill myscript.tcl`

Runs the named script using the development `tclsh`, first setting TCL_LIB_PATH to make the project's code base accessible.

`quill shell`

Opens a `tkcon` window using the development tclsh and first setting TCL_LIB_PATH to make the project's code base accessible. The primary application's application package is required automatically. This command will likely be made more flexible in the future.

`quill teapot`

Manages the local teapot repository. In particular, `quill teapot fix` outputs a script that can be run with `sudo` to link the user's teapot to the `tclsh` and fix up any related permissions problems.

`quill test`

Executes the project test suite. Depending on the arguments, this tool can run all tests, all tests in a particular `test/` subdirectory, or a particular module in a particular `test/` subdirectory, passing options to tcltest(n) in each case. When running all tests, the results are summarized unless the `-verbose` option is used.

`quill version`

Displays the Quill tool's version to the console, and also points the user at the Quill issue tracker.

4.4 Sample Session

The following is a sample session with Quill, involving a new project.

```
$ quill new app my-project myapp
Creating an "app" tree at /Users/will/github/test/my-project/...

$ cd my-project
$ quill info
my-project 0.0a0: Your project description

Project Tree:
    /Users/will/github/test/my-project

Applications:
    Name   Mode     ExeType
    -----  -------  -------
    myapp  Console  kit

Required Packages:
    Tcl  8.6.1

$ vim project.quill
$ quill info
my-project 0.1a0: A tool to do something.

Project Tree:
    /Users/will/github/test/my-project

Applications:
    Name   Mode      ExeType
    -----  -------   -------
    myapp  Console   exe

Required Packages:
    Tcl   8.6.1
    snit  2.3

$ quill deps
Dependency Status:
  snit 2.3                     (OK)
  basekit.tcl                  (OK)

$ quill env
Quill 0.2.2a0 thinks it is running on Mac OSX.

Local Teapot: /Users/will/.quill/teapot

Helper Tools:
   tclsh        /usr/local/bin/tclsh (v8.6.1)
   tkcon        /usr/bin/tkcon
   teacup       /usr/local/bin/teacup (v8.5.15.1.298288)
   tclapp       /usr/local/bin/tclapp
   basekit.tcl   /Library/Tcl/basekits/base-tcl8.6-thread-macosx10.5-i386-x86_64
   basekit.tk    /Library/Tcl/basekits/base-tk8.6-thread-macosx10.5-i386-x86_64
   teapot-pkg    /usr/local/bin/teapot-pkg

!  - Helper tool could not be found on disk.
+  - Path is configured explicitly.

$ quill docs
Writing /Users/will/github/test/my-project/docs/man1/myapp.html
Writing /Users/will/github/test/my-project/docs/man1/index.html
Writing /Users/will/github/test/my-project/docs/index.html

$ quill test
Summarizing test results.  Use 'quill -verbose test'
to see the details.

app_myapp:   Total  0  Passed 0  Skipped       0     Failed 1

$ quill run
my-project 0.1a0

Args: <>

$ quill run a b c
my-project 0.1a0

Args: <a b c>

$ ./bin/myapp.tcl a b c
my-project 0.1a0

Args: <a b c>

$ quill build
Building Console app myapp as myapp-0.1a0-macosx10.9-x86_64

$ ./bin/myapp-0.1a0-macosx10.9-x86_64 a b c
my-project 0.1a0

Args: <a b c>

$ vim project.quill
$ quill info
my-project 0.1a0: A tool to do something.

Project Tree:
    /Users/will/github/test/my-project

Applications:
    Name   Mode     ExeType
    -----  -------  -------
    myapp  Console  exe

Required Packages:
    Tcl   8.6.1
    snit  2.3

Distribution Sets:
    install-%platform

$ quill dist
```

```
Making my-project-0.1a0-install-macosx10.9-x86_64.zip...

$ unzip -l my-project-0.1a0-install-macosx10.9-x86_64.zip
Archive:  my-project-0.1a0-install-macosx10.9-x86_64.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
  4257824  10-12-14 10:08   my-project/bin/myapp-0.1a0-macosx10.9-x86_64
     2077  10-12-14 09:56   my-project/docs/index.html
     3477  10-12-14 10:04   my-project/docs/man1/index.html
     3965  10-12-14 10:04   my-project/docs/man1/myapp.html
       49  10-12-14 09:56   my-project/README.md
 --------                   -------
  4267392                   5 files

$
```

5. Future Work

Quill is still a young project; not all of the "standard" practices it embodies are completely fleshed out, nor is it as flexible as it needs to be. Easy things are simple to do; hard things are not always possible.

This section discusses the most pressing needs for future Quill development.

5.1 Feedback and Discussion

In setting up Quill and its notion of project trees I've naturally drawn on my own personal practices. Many of my decisions are debatable by reasonable people; "good enough" is in the eye and experience of the beholder. I would love to get feedback about my choices and discussion about how to provide what is lacking. For Quill to be successful as a tool for the broader community, it needs to support the needs of the community.

5.2 Stability

By its nature, Quill interacts heavily with the user's environment, and every user's environment is different. Assumptions that are met on the systems I use might not be met on your systems, leading to bugs and frustration. The number one priority in Quill development at this time is usage and testing by many users on many platforms, so as to shake out as many bad and hidden assumptions (and bad architectural decisions) as possible.

5.3 Near-term Features

The following are features I'd like to add to Quill in the near term:

Add new elements to existing project. The user should be able to add new project elements to an existing project using predefined templates (i.e., new applications, new libraries).

More standard project templates. Leiningen comes with templates for console applications, GUI applications, games, web applications, libraries, and so forth. Quill needs to do better.

Tool plugins. Quill should support user-defined plugins that add new subcommands to Quill on a system-wide or project-wide basis. Such tools should be able to access the project metadata and query (and possibly manipulate) the project tree. Plugins would be Tcl packages; they would be installed via the `quill config` mechanism for use by all projects or within `project.quill` for use by the specific project.

Template plugins. Quill should support user-defined plugins that add new project and project element templates for use with the `quill new` and `quill add` commands. These plugins would be Tcl packages installed as described above.

Arbitrary build targets. A Quill project should be able to contain directories whose content gets built via user-specified scripts as part of an overall project build. The scheme I have in mind is to add "`src/`*`name`*" directories to the project tree; statements in `project.quill` would name the relevant directories and optionally specify two system commands, one to "build all" and one to "clean". These would default to "`make clean all`" and "`make clean`" respectively.

Documentation macro plugins. manpage(5) and quilldoc(5) macro sets are easy to define; a plugin architecture would make it easier for users to tailor these formats to their needs.

5.4 Long-term Features

The following are features I'd like to see added to Quill in the longer term.

Support for building non-`tclapp` executables. The `tclapp` tool is a powerful and useful tool, especially as regards its interaction with teapot repositories. It also costs money; and as one of Quill's goals is to make Tcl/Tk more attractive to novices we need a wrapping solution that's effectively free-as-in-beer. This is certainly possible; Stephan Effelsberg has added one solution (based on Critcl's build script) to the Quill issue tracker. But it begs the question of where to reliably acquire basekits for different platforms, and raises legal issues about including packages from ActiveState's teapot in potentially commercial applications built without a TclDevKit license.

Deployment of Quill packages to a remote repository. I'd very much like to see this. The existing teapot infrastructure doesn't support remote deployment, so far as I know; and there's currently nothing else on offer. In addition, the teapot has a major lack compared to Clojure and

Maven repositories: it does not give access to package documentation or to package applications. This is a place where we need to step up our game as a community.

A Quill GUI. Building a cross-platform GUI for Quill is reasonably straightforward; but I'd like to let the command-line tool stabilize before going there.

Multiple back-ends for `quill docs`. Quill documentation is now produced only as HTML. It would be great to support multiple back-ends, especially including PDF output. Perhaps Quill could support documentation plugins for this purpose. HTML is generally good enough for my own purposes, so this is a place where other developers could make a contribution.

5.5 Blue Sky Ideas

The following are some of the wilder ideas I've had.

Embed Critcl in Quill. Critcl [14] is a tool for embedding C code in a Tcl script and including it in a finished application. It seems to me that Critcl could be embedded in Quill in such a way that C code could be included easily and transparently in any project. Naturally, Critcl requires an environment with a C compiler.

Embed a C compiler in Quill. If Quill included its own C compiler, many extensions could be built transparently. The "TEA" infrastructure would be part of Quill itself. This is likely a bridge too far, unless…

Make Quill part of the core distribution. The core team has a laudable goal of keeping Tcl/Tk's footprint small, since it is often used in embedded environments. Still, there is room for including development tools in the core distribution, tools that would not be deployed to production systems. Something like Quill could be quite useful in this regard. First, of course, Quill needs to prove itself in the field.

References

[1] Clojure Programming Language, `http://clojure.org`.

[2] Leiningen build system for Clojure, `http://leiningen.org`.

[3] Apache Maven Project, `http://maven.apache.org`.

[4] Quill Home Page, `https://github.com/wduquette/tcl-quill`.

[5] `zipfile::encode`, Generation of zip archives, `http://core.tcl.tk/tcllib/doc/trunk/embedded/www/tcllib/files/modules/zip/encode.html`

[6] `zipfile::decode`, Access to zip archives, `http://core.tcl.tk/tcllib/doc/trunk/embedded/www/tcllib/files/modules/zip/decode.html`

[7] ActiveTcl Tcl Distribution, `http://www.activestate.com/activetcl`

[8] TclDevKit, `http://www.activestate.com/tcl-dev-kit`

[9] TkCon Tcl Shell GUI, `http://tkcon.sourceforge.net`

[10] `doctools` documentation processor, `http://core.tcl.tk/tcllib/doc/trunk/embedded/www/tcllib/files/modules/doctools/doctools.html`

[11] `textutil::expander`, Macro Expander, `http://core.tcl.tk/tcllib/doc/trunk/embedded/www/tcllib/files/modules/textutil/expander.html`

[12] `expand` macro processor, `http://www.wjduquette.com/expand/index.html`

[13] Snit Object System, `http://core.tcl.tk/tcllib/doc/trunk/embedded/www/tcllib/files/modules/snit/snit.html`

[14] Critcl, `http://andreas-kupries.github.io/critcl/`