Repositories, deployment factories, binary code, and CDROM's

Jean-Claude Wippler
Equi4 Software
June 2002

Introduction

Developing scripted applications with Tcl is fun. It is a high level and often very productive activity, full of pleasant surprises. Applications built this way tend to incorporate a high amount of functionality with little effort, covering all aspects of business logic, graphical user interfaces, databases, and networking, and interfacing to a lot of existing technology.

Best of all, scripting leads to portable applications which can run well on Windows, all common Unix variants, and Macintosh. Well... in theory. The Achilles' heel of scripting is compiled code. Once one needs it - whether for performance or for interfacing to existing software - all the headaches of builds, makes, compiles, and links come back full force. It is ironic that the success of scripting means fewer people have been exposed to tools such as compilers, or have lost their confidence to "go back to C". Binary code is in a way both the fallback option of scripting and its secret weapon, yet its practicalities can easily become a serious stumbling block.

When binary code is used, whether custom and application-specific or simply to use some of the common and popular "extensions", deployment instantly becomes a far more involved process. Even for single-platform use, the need to include and maintain binary code can become a tough hurdle.

This paper discusses the implications of using and deploying binary (i.e. platform-specific) code in scripted applications, and offers ways to resolve several of the issues, based on "scripted documents" (a packaging model) and "critcl" (a way to embed C into Tcl). It also explores the issues of binary distribution for sharing re-usable packages, and describes ways to set up and maintain repositories for such packages. A few examples will be presented to illustrate what the current state is and just how practical things have already become. Finally, an on-line archive site will be described which is being created right now - addressing issues of packaging, platform-specific builds, versioning, documentation, and updating.

Software: from concept to easily distributed and maintained code

A typical scenario for building applications with Tcl/Tk could be:

- Install Tcl/Tk, possibly also some common packages such as those in TclLib
- Grab a few other extensions, such as BWidgets or IncrTcl
- Start coding a main app script, edit/run/debug
- Consider using tools such as TkCon, TkInspect, and GUI builders
- Browse web/usenet/wiki, consider using more available packages
- Plow along, discard some approaches and packages, adopt others
- Hey, it works, let's wrap things up and give it to others to try out
- Hmmm... what do I do next?

At this point, a number of issues need to be dealt with:

- how to get Tcl/Tk set up on the target machines
- how to figure out what packages need to be included with the app
- how to pack/unpack - this is when most people will create an "installer"

When compiled extensions are needed, custom code or simply common extensions for which not all required binary distributions exist, things get trickier:

- how to get extension(s) compiled for all targeted platforms
- if compiled for static linking: how to rebuild all the app main's
- else: how to deploy these shared libraries along with the application

With a source distribution model (common on Unix for open source, but unusual for commercial projects), the deployment will have to include the following steps:

- obtaining the proper "tool chain" to compile/link the extension on each target
- installing the resulting object code so the application can find it

With a binary distribution model, the burden shifts to the developer/vendor:

- build extensions, either cross-compiling, or remotely and then collecting results
- embed all versions into the app, or package separately for each platform

Even with single-platform use, the builds and maintenance of builds, as well as testing that the completed application works properly, are all activities which can take a lot of effort. In more elaborate multi-platform scenario's, or when there is more than one deliverable (e.g. a client / server system), adequate validation and testing of the final product can become a nightmare.

Tcl/Tk boosts the development process - but why stop there?

One of the consequences of building scripted applications, is that the end result consists of a number of scripts plus a runtime environment. Since there is no final link step, these usually stay around as separate files. Evidently, the lack of a link step is a gain for developer productivity and flexibility during the creative phase of a project.

The flip side of this coin is that deployment becomes more complicated than when one ships a single final "executable". In some cases, i.e. when the Tcl/Tk runtime is known to be present on the target systems, deployment continues to be quite simple. As an example, TkCon and TkBiff are simply single large script files with everything they need, and which are designed to work with just about any sufficiently modern pre-installed Tcl/Tk release.

On Windows, Tcl/Tk is not normally present on the machine, however. One way to address this problem is to create an installer for Tcl/Tk, with the most recent and most complete one now being ActiveTcl, as produced and maintained by ActiveState.

But this is not a general solution. For developers, and for corporate sysadmin-based contexts, such an installer works fine. For end users, having to install "Tcl/Tk" can be a show-stopper. No matter how streamlined it is, and no matter how much eye-candy one adds to it, the whole concept of having to set up some sort of "runtime environment" in addition to the application itself can evade the understanding of most end users - and rightfully so, given that Tcl/Tk is merely a vehicle chosen by developers to get their product implemented. How does the naive user respond to "Do you want to install Tcl/Tk in C:\Tcl?", for example - given that he/she is so eager to try out, say, an accounting application?

In the last few years, a number of "wrapper" tools have become popular, most notably ProWrap and FreeWrap. These address the issue that installing a run-time is so badly suited for end-user deployment. Instead, they take scripts (and sometimes also compiled extensions and other data files), and produce a single-file executable, which for all practical purposes then *is* the application. These tools have become very popular, because they can help make the end-user experience a pleasant and successful one. Not in the least because this wrapped environment is less likely to break due to differences between target machines, or even due to changes to the system later on.

Despite their benefits, such wrappers are not always appropriate:

- they do not address multi-platform deployment scenarios
- the only update option is full replacement of rather big executables
- wrapped apps tend to run in a different context, and need additional testing

That last aspect means that application development is still quite different from application use. There is still considerable room for improvement, as the next section will show.

Scripted Documents: a methodology, all the way to deployment

Imagine having a simple directory and file structure, where all application scripts, standard packages, compiled extensions, documentation, images, and other binary data resides. That's not so hard - in fact, it is most likely common practice among many Tcl developers already.

Imagine also having a well-defined way of storing both such an application and all its support files, and the Tcl/Tk system itself. Then deployment would become a matter of picking up all the relevant pieces, shipping it to the target machine somehow, and it would run out of the box.

Suppose furthermore that the structure could be wrapped into single self-consistent files, in a space efficient compressed form, and that these file could be "executed" without unpacking, and without altering a single line in the application.

This, in a nutshell is what Scripted Documents (the wrapped application), TclKit (the wrapped Tcl/Tk runtime), and VFS (the Virtual File System layer now in Tcl) are all about.

Scripted documents let you "seal" a complete directory tree into a runnable form. The application which can "run" such documents is called TclKit, which itself is little more than a wrapped version of Tcl, Tk, IncrTcl, and a few more extensions - plus all the necessary runtime library scripts.

The essence of scripted documents, is that they act like a file-system-inside-a-file. This file system contains a copy of all files that form an application, but in normal use it never gets unpacked at all. The TclKit runtime contains a complete Tcl/Tk system, and it too can function without ever being unpacked. The command "tclkit myappfile" is the scripted document way of "running the application", although on Unix the same effect can be achieved simply by making the scripted document executable ("chmod +x") and have tclkit on the search path.

Keeping things separated in two files has a number of benefits:

- tclkit is a generic runtime, the same file works with all scripted documents
- there is one tclkit build for each supported platform: long live simplicity
- if applications contain only portable scripts, the scripted document is portable

So if multi-platform deployment is an issue, one can deploy the application as a file which runs on all platforms - just by launching it with the appropriate TclKit system.

The philosophy behind scripted documents is that deployment is an integral part of the development process. Applications get built and extended over time, in whatever way a developer sees fit, but with a certain directory structure as storage convention. Development can take place using any Tcl/Tk installation (including TclKit!). Tcl commands such as "package require", "source", and "load" can be used - as always.

Then, when the time comes to deploy, all one does is wrap the entire application directory tree into a scripted document. The resulting file is compressed and ready for deployment. In fact, it is runnable. The effect is that deployed applications run in nearly the same way as they do during the development process. The Virtual File System "mount" concept makes running from a real file system and running from inside a scripted document almost indistinguishable.

Scripted documents are implemented on top of an embedded database called MetaKit (part of TclKit, but also available as Tcl extension). One of the key benefits of this approach is that scripted documents can be modified, and even be self-modifying and self-updating. There is no risk of an application modifying itself and then damaging the file, because MetaKit supports transactions. Even a power failure will not damage a scripted document, it will simply revert to a previously committed state (no repair is ever needed: this is fully automatic).

For cases where one wants to extract the contents of a scripted document as separate dirs and files again there is a utility called “sdx” (it's a scripted document too, so it runs with tclkit). For cases where tclkit cannot be used, there is also a pure-Tcl script called "ReadKit", which can list the contents and unpack any scripted document without relying on TclKit or MetaKit. To actually uncompress files, it needs either the Trf or the Zlib extension. If those are not available, readkit can convert a scripted document to a ZIP archive, which can then be further unpacked using widely available standard utilities (such as WinZip on Windows).

Note how scripted documents are essentially transparent to the application, to the point that all files can easily be extracted, even with Tcl's "file copy" command in fact. For commercial use, this is not always desirable. To prevent people from extracting scripts and seeing their contents one will need to add some form of encoding system. This has not been addressed in a generic way yet, but it can be added on top of the existing functionality (e.g. one could define an "xsource" command which decodes and evals a specified script). This approach has been used in at least one comm ercial application.

An interesting, but not yet deeply explored, option is the deployment over various media. For some time now, scripted documents have been used in applications whereby some part (sometimes in fact all) of the application logic was distributed over the net, in a client-server configuration. Due to the fact that scripted documents are modifiable (yet fail-safe), it is quite easy to use a local scripted document as a cache of application logic and data. This means that software updates can be made fully automatic and transparent, and changes stored locally. Taken to extremes, an application can be deployed as a small "placeholder", which on first startup simply first downloads all the necessary logic, saving it locally inside the scripted document for subsequent use. One commercial application uses this approach to provide a user interface that evolves over time – based on the user interaction – but with no involvement from the user apart from establishing a network connection.

Another deployment scenario which becomes feasible with scripted documents, is to distribute the application and possible also large datasets on a CD-ROM. When copied to a local hard disk, such an application can then again turn to net-based schemes for updates, code as well as data.

Binary code and compiled extensions are easy (nowadays)

Until now, adding compiled code to Tcl has always been a complex issue. Not only did one have to deal with tools such as autoconf and make, it also meant one had to have a developer environment set up with Tcl/Tk's header files and "stub libraries", for example. The conceptual distance between "writing a bit of C code" and having an extension which is actually callable from a Tcl script, can be daunting. Yes, "real programmers" will point out that this is what it takes to do "true" programming. But for many who are used to the productivity of Tcl it can easily become a frustrating and time-wasting road to... failure.

But there is hope. A new package and tool has been implemented, called "CriTcl", which takes a lot of the details away from the developer. CriTcl can take care of creating binary code, hiding all issues of including the standard Tcl headers, calling the compiler in the right way, making the result stub-enabled, even setting up a fully TIP 55 compliant package in a directory, with "pkgIndex.tcl" ready to go.

To give you an impression of how CriTcl fits into Tcl, here's a trivial example:

```
package require critcl
critcl::cproc cube {int value} double {
    return (double) value * value * value; /* this is C */
}
puts "The cube of 12345 is [cube 12345]"
```

A paper titled "CriTcl: Beyond Stubs and Compilers" by Steve Landers and co-authored by yours truly will be presented at the Tcl/Tk 2002 conference in Vancouver, next September. This paper goes into much more detail regarding the concept, use, and packaging support of CriTcl.

Suffice to mention here that you can transfer a script to multiple platforms, run critcl on each, and collect the results. The resulting extension will run on those platforms with any Tcl/Tk installation (provided it supports stubs, and that the extension does not call functions present only in the newest releases of Tcl/Tk).

Although CriTcl makes things simple, it is definitely not a simplistic tool. CriTcl has been used for dozens of extensions already (many are now in "critlib" and "kitten"). It has been used to re-build complex existing C extensions such as "Tkhtml". It has even been used to re-build the entire regular expression subsystem of Tcl, to show that one could gradually modularize substantial parts of the Tcl/Tk core itself. The latest version of the "critcl.bin" scripted document - designed and implemented by Steve Landers - is a fully self-contained utility, complete with on-line help. It is already a very practical way to add C and C++ code to Tcl applications, whether for performance or for interfacing purposes, and whether for new code or to simplify building of existing extensions. And of course, it plays well with scripted documents - so the same steps can be used to create extensions for use in deployed applications.

Single-file deployment, two examples

A while back, the "AlphaTk" editor was turned into a scripted document. This very large pure-Tcl application by Vince Darley turned out to be a serious test case, and brought out many issues. In fact, this was before VFS support was present in the core, and it is probably one of the reasons why Vince started implementing the new VFS code that has ended up becoming part of Tcl/Tk 8.4. AlphaTk now works fine as scripted document and runs on multiple platforms. It takes advantage of the fact that scripted documents can be modifiable file systems, so it can store and manage preferences and caches inside itself. At the Tcl coding level, this is fully transparent - AlphaTk simply reads and writes files, and creates directories.

A more recent example of a scripted document is the "NewzPoint" application, by Michael Jacobson. This Win32 application embeds the MSIE browser in a Tk window - using the magic of COM and Optcl. Though not originally designed for deployment as scripted document, it took only minutes to turn it into one.

The rest of the application has not been altered, though scripts which open or source other files will need to be aware of the fact that the current directory is not necessarily the location where scripts are located.

The initial release of NewzPoint used the two-file scripted document + tclkit approach, but TclKit also supports wrapping a scripted document with TclKit to produce a single platform specific executable. Now NewzPoint has also been made available as single Windows ".exe". This is a much preferred alternative for many Windows users, and they need not know they are using TclKit, Scripted Documents or indeed a scripted application. Scripted documents, even single-file versions, can be unpacked using the **sdx** utility (or readkit, if TclKit isn't available to run sdx). Once you have an unpacked directory tree containing the files of an application, you can launch that application in the classical way, simply by starting the main application script:

```
wish myapp.vfs/bin/main.tcl
```

Which goes to show that scripted documents are a great convenience, but also an open-ended technology. It is a wrapping system that can greatly simplify deployment, but if needed all the classical tools and techniques can still be used - as before.

A digression: portable and searchable on-line help

Now that deployment has been "solved" in a cross-platform way, and now that truly portable applications can be packaged, the benefit of keeping as much of an application scripted in Tcl increases quite dramatically. When everything is coded in Tcl, your applications will automatically become usable on over a dozen platforms, i.e. all platforms for which there is a TclKit build. Many more platforms even, if one uses the "readkit" script to unpack and run the application with a classical Tcl/Tk installation.

Not all applications can be made so portable. The moment even just a single compiled extension is used, that nice total portability often has to be sacrificed. There are three ways to maintain maximum portability:

- recode C-coded logic as Tcl - Don Libes' MD5 is an example of this
- include pure-Tcl fallback code, even if the functionality is reduced
- work towards a growing repository of compiled "standard" extensions

This section will focus on the first option, with as example the goal of providing on-line help as part of an application.

There are several ways to provi de help. Richard Hipp's "Tkhtml" widget is a wonderful way to display rich HTML content. But unfortunately it is compiled C code, and hence not available on all platforms, such as the classic Macintosh. Another option is to embed a small HTTPD server, and use whatever browser is available to present documentation. The difficulty here is to make sure such an approach works in all contexts, and across the huge variety of browsers out there. If on-line help fails or is too hard to get right, then many users will quickly give up.

Fortunately, there are more ways to skin this cat. There is now a solution which is 100% portable and which takes full advantage of Tk's power, of existing open source software, which supports searching and hyperlinks, and which is based on a simple but well-known and proven technology: the "wiki".

This on-line help facility adds fast search capability on titles and on full text contents. It is portable, it runs inside and outside scripted documents, and it adds just a few dozen Kb to the application. Support for inline images is likely to be ready by the time you read this.

The picture is starting to shape up. We have now "solved" the issue of packaging, of multi-platform embedded C code, of portable on-line help, all of which go a long way towards dealing with multi-platform deployment.

Why binary code make sense

Clearly, finding solutions which do not require binary code, as in the previous section, continue to be by far the most attractive way to reduce deployment complexity and overhead.

But there will always remain a need for compiled code. Some applications simply cannot do without the added speed, and some will always need access to existing extensions and C code bindings. For a wide range of applications, there is often simply no way around the need for having compiled code. The question is whether everyone who uses it needs to become an autoconf / make / compiler / linker expert as well.

In fact, it seems plausible that the use of binary distribution will increase further, not decrease. Here's why:

- The core of Tcl/Tk is solid, stable, and has an excellent build system - and though that makes it easy to build, it also means that setting up an automated build system is simple - delivering a continuous stream of up-to-date binaries for a large range of platf orms. Both TclKit and ActiveTcl come pre-built for a number of platforms.

- If people don’t need to build Tcl/Tk themselves, they are less likely to learn about autoconf and make. In addition, prebuilt Tcl/Tk distributions lower the barriers to using Tcl/Tk -– and we are increasingly seeing programming, and scripting in particular, no longer limited to those who have a Computer Science degree.

- Windows and Macintosh machines have all but conquered the desktop. It is very easy to support all those systems with just a few binaries. Portability is great, but in terms of numbers one can do a lot with just one or two binaries. Developers on Windows are used to binary code (both closed- and open-source), because "it just works".

- Even Linux systems are evolving to binary distribution, a concept which was unheard of in the Unix world before the arrival of solutions such as RPM.

- Binary code makes a lot of economic sense - i.e. in terms of reduced effort: why build the same thing over and over again, just to end up with the same result, when a vendor (or a packager) can do the job, and save everyone else a lot of effort and trouble?

- Compile farms are widely available to developers nowadays. So as far as standard extensions go, there really is no reason to assume people will continue to spend energy over and over again, trying to figure out how to build a widely used extension. The hurdle is just too high.

Yet for custom-specific compiled code, the picture is completely different. Code which only matters to a certain developer, which is in active development in fact - and which is more often than not also proprietary -will never benefit from public distribution, from people hacking around to get a good autoconf/make setup, or from compile farms.

This is where CriTcl comes in. It lets developers focus on the C code, instead of the logistics of creating an autoconf/make setup and stubs declarations and bindings. It forces them to keep things absolutely simple, and not try to solve it all in C - instead delegating as much as possible to portable Tcl scripts, while bringing just a minimum into C, for performance, or to create a Tcl binding to other C/C++ code.

One goal of CriTcl is to make it as easy as possible to convert a bit if C into binary code on all platforms one has access to, for which there is a tclkit built plus a C compiler. The availability of TclKit is not essential, but it greatly simplifies the multi-platform build process.

Lastly comes the task of doing what this paper is all about: creating the deliverables one needs for one or more platforms. This is where scripted documents can shine. They wrap things up, taking advantage of the fact that VFS "mount" makes their use transparent, all the way to having compiled shared libraries included as well as on-line help. The distributed file acts like a little self-contained file-system "world", which conveniently happens to function just fine without getting unpacked at all.

In the case of CDROM distribution, one gains the ability to run the packaged application off the CDROM, without the need to install anything. Such a distribution can contain tclkit builds for a wide range of platforms.

Due to the fact that single-file executables (i.e. tclkit and the application scripted document merged into a single file) can be generated cross platform, it is now feasible to develop a pure-Tcl application on a single platform, finish that development process up to the final release, and then - as last step - generate all wrapped executables from the development machine.

Distribution of standard runtimes and extensions

Various ways have been developed so far to try and help get standard code out, sometimes also in compiled form.

In the scripting world, Perl's "CPAN" is legendary, and the name says it all: Comprehensive Perl Archive Network. This repository provides an incredible amount of code and functionality. It uses Perl itself to automate all the work of fetching (source) code, driving a build, and installing build results in a standard spot. But in the context of application deployment, it has a number of serious deficiencies:

- it is a source distribution model, unsuitable for end-user deployment
- while amazingly powerful, it does not always succeed in building things properly
- module dependencies tend to lead to huge download cycles
- adding modules to an RPM-centric world can easily lead to conflicts

In other words, it is a technical solution to a technical problem designed by technologists.

All of the above issues can be (and usually are) show-stoppers, in the context of getting an application out to users, whether in a large-scale or in a commercial context. Yours truly has never in the past years succeeded in getting even relatively simple applications based on CPAN modules to work. Oh sure, it does a lot, and it impresses deeply - but it hasn't delivered.

Another trend is the creating of installers for increasingly extensive scripting environments. There are such installers for Windows for Tcl, Python, and Perl (the best known all come from a single supplier: ActiveState).

Unlike CPAN, installing such systems is far simpler, more likely to succeed, and quite a quick way to get a lot of the infrastructure in place. In fact, Python has always pushed for the model of delivering itself with "batteries included" - and it's thriving.

But again, these systems fall short in the context of end-user deployment. They do not offer a way to get a simple application based on scripting out to a user (all downloads end up being several MBs, sometimes even over a dozen MB). The request for getting a "small" installer out is not being addressed. Evidently, there is a lot of functionality to be deployed when one wants to present all of Tcl, Tk, and more - but why does it have to be more than fits on a floppy disk? It looks like this trend will only get worse, as these systems accumulate more and more code in their quest to be an all-inclusive "batteries-included" solution. Again, this misses the point that while developers may love it, it just leads to tedious and impractical end-user scenario's.

What remains, and what all scripting languages offer now in some way, is the "wrapped app", i.e. a single executable file that carries the scripting environment with it as well as the application itself. Some solutions end up acting as installers, leaving an "installed package" behind when run, others unpack on the fly to a temporary area, and clean up on exit, still others - and this includes TclKit - run in packed format.

This is where Tcl/Tk is currently at a considerable advantage: it has "stubs". The stub mechanism means that one can create a statically linked executable which runs as is, yet which still supports dynamically loade d

extensions. This is also why TclKit is able to run scripted documents with shared libraries inside (unpacking just the shared library file, loading it, and then discarding it again).

The following section presents a new approach for the distribution of applications, based on the model of scripted documents, binary repositories, and the TclKit runtimes.

A Standalone Executable Assembly Line

So far, there seem to be preciously few solutions for collecting and sharing compiled builds, other than the base system builds themselves.

Wouldn't it be nice if one could go to a website, pick a couple of packages, select a couple of platforms, and end up with an archive or an executable to which only the application-specific parts need to be added?

In a way, much of this is possible with scripted documents and tclkit today, albeit manually. Pick the extensions you need, grab one or more builds of tclkit, add your application code, and wrap it all up into an set of deliverable executables?

The missing piece is ... "convenience". As developer, I don't want to be an assembly-line worker, I want to set up a factory, and be the factory manager. Specifying the goal, defining the required components, and filling in the missing pieces - being the essence of the application itself. There's no fun in having to assemble a deliverable over and over again as the development cycle progresses (and often well beyond, into the next releases). It seems odd, given the fact that so much can be automated with computers.

Welcome to "SEAL" .... the **S**tandalone **E**xecutable **A**ssembly **L**ine.

Right now, an archive is being created (called "SDarchive" for now) to help simplify the task of binary distribution. The goal of this archive is to explore possible options and to start a new trend. The key areas of focus for this archive is not to collect binaries like crazy, only to end up with yet another repository of stale and unmaintained data, but to work towards a structure which can both technically and socially maintain itself for a long time to come.

The initial aims of SDarchive are modest: just get a collection off the ground which suits a few people, including your truly. Taking full advantage of the scripted document model, i.e. making the exploration, use, maintenance, and deployment of complete applications, small utilities, and individual packages simpler than ever before.

Three dimensions are particularly important for SDarchive:

- functionality: a selection of apps, tools, and packages
- platform: a selection of platform to be supported
- context, i.e any mix of: code, documentation, sample code, test suites

Several aspects can be explored to turn the initially passive SDarchive repository into what will ultimately be an assembly line for applications:

- a basic CGI interface to browse and make selections for downloading
- a web-based interface to manage submissions, updates, and build results
- later on : a TK-based local interface (over HTTP) to improve the interaction
- tying into the Tcl'ers Wiki as a way to comment, suggest, and contact authors
- a wiki-based way to tie documentation and package dependency graphs together

Although some of the above is just in the "blue sky" stages of development, it is fair to say that once a bit of this is in place much progress can probably be made in a relatively short time - after all, this is all done through Tcl/Tk. Scripting is not just the goal - it is also the tool with which all this can be realized.

The longer-term goal of SEAL is to become a resource which maintains itself, with those who contribute and maintain tools and packages getting feedback from their users, while application developers see an actively maintained repository which greatly eases their job of getting top notch scripted products out. Prior experience with the Tcl'ers Wiki and with the "SAX" (Shareware Author indeX) show that when all parties involved have the right incentives, then magic sets in and collaboration, synergy, and community awareness take over. The economics of this equation are simple: "win-win", while the cost and expenses needed to achieve this are very very low. The SEAL resource won't come about overnight, but at the same time one has to conclude that all the technologies exist right now to make this eminently feasible.

A bit about motivation...

The motivation for SDarchive and SEAL is, as in so many open source projects: to scratch an itch. Scripting, although quite well established by now, continues to stumble along on the path of application software

development. It is not enough to have a good tool and be able to code up lots of solutions - one has to address the full lifecycle of software development.

With the predominance of Windows, and solutions that come out from just a few major players in the software industry, we have come to accept for a fact that computers are enormous time wasters when it comes to getting out solutions to people. The promise of productivity-enhancing anc creativity-stimulating new "killer" applications is completely overshadowed by the trouble of "installation".

If you think a bit about it, the whole concept of "installation" is artificial and unnatural. When was the last time you installed an appliance like a toaster or a kettle? Why does one need to jump through hoops to start trying out something new? Why does one need to go through a brain-wash style labyrinth of jargon words and acronyms to do fun things with a computer?

Scripting delivers on the promise of making programmers productive. Scripted documents can deliver on the promise of making it easy to get their work into the hands of users.

The achilles heel of scripting has always been the deployment of compiled code, even after the invention of "dynamically loadable extensions" and "shared libraries".

With scripted documents and tclkit, there is at last a way out. With a bit of concerted effort, the hurdle of sharing a huge range of binary packages can be solved for good.

But the surprise of scripted documents, is that they have in fact turned out to be far more general purpose. Even with single Tcl scripts, deployment as a scripted document has the benefit of convenience: the file is smaller, because it is compressed, and it behaves identically. Scripted documents can store software, optionally also user preferences, or even entire datasets. All in a single, self-consistent, file. It often works cross platform. When backed up, it very conveniently and naturally keeps application logic and data together, thus making it far more likely to be usable at a later date - even if only to extract data for unforeseen uses.

The "fun" part of scripted documents is hard to exaggerate. There is simplicity and convenience in all this, and it's all driven by the power of the Tcl/Tk scripting system. There is a steadily growing group of people who see the potential of what is happening, and who are becoming enthusiastic about the fact that they can at last focus on application content and functionality, knowing that "getting the app out" is now a solved issue.

In the light of all the other technologies out there, Tcl/Tk and scripted documents are little known dwarfs, or as one tends to say: "a well kept secret". Yet in the hands of those who use it it, they become a magic wand. It is exciting and very rewarding to be able to participate in this 100% open source development and help take deployment further. And as you can see, it's no rocket science - just common sense, and the joint effort of a growing number of Tcl'ers.

You are welcome to participate, or simply watch and benefit, as much as you like. For pointers, see the reference at the end of this article.

Acknowledgements

Many thanks to Steve Landers for the numerous discussions in the past months that led to this paper, and for his on-going support, insightful comments, and patient reviews during the course of writing this document.

Many thanks also to those who have implemented the technology on which all of this is founded, including: Jeff Hobbs, Vince Darley, Paul Duffin, Matt Newman, Jan Nijtmans, John Oosterhout, and many many others. As so often in software development, no part of TclKit would be of use without the underlying foundation code and the blood, sweat, and tears of many programmers.

It's nice to be able to stand on the shoulders of giants...

References

Scripted documents home page
http://www.equi4.com/scripdoc/
TclKit home page
http://www.equi4.com/tclkit/
The CritLib home page
http://www.equi4.com/critlib/

Scripted Document Archive:
http://mini.net/sdarchive/
Steve Landers' company homepage
http://www.digital-smarties.com/

The "official" Tcl Portal
http://www.tcl.tk/ (or equivalently: http://tcl.activestate.com/)
The Tcl SourceForge development site
http://tcl.sourceforge.net/

The Tcl'ers Wiki - a vibrant community meeting place
http://wiki.tcl.tk/ (or equivalently: http://mini.net/tcl/)
The Shareware Author indeX (SAX)
http://mini.net/sax/
The Comp. Lang. Tcl usenet discussion forum:
news://comp.lang.tcl/