Customizing Think C and Symantec C++ with Frontier

Dave Winer
UserLand Software

Introduction

Think C is a wonderful development environment. Lots of commercial and in-house developers use Think C because of its incredibly fast builds, integrated project management and text editing features. It's a very rationally designed system, it works, and it's well supported. But it's always been one major missing feature — there was no way to automate it. So there was a major penalty for organizing your work as reusable libraries of solved problems — when you made a change to one of your base libraries, you had to manually rebuild all projects that depend on it.

When we heard that Think C and Symantec C++ 6.0 would support scripting, we got very excited. We wanted to see how using scripts to drive Think Project Manager, or TPM, would impact our development work.

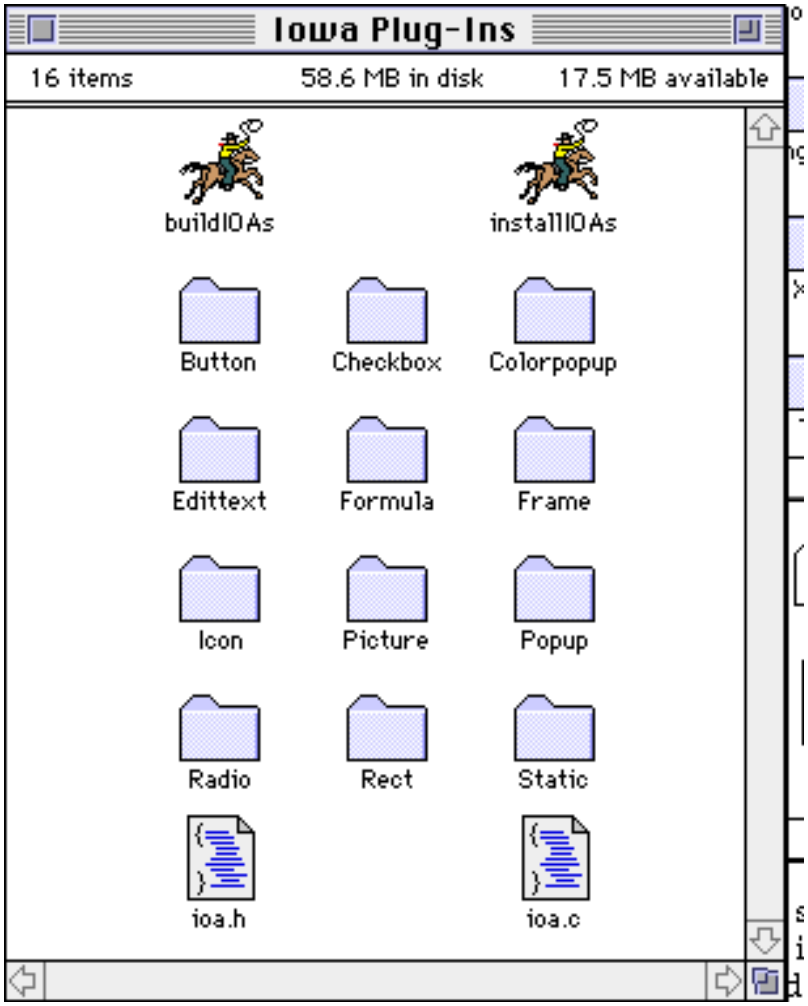
We learned that there are two types of scripts: ones that add features to TPM that everyone can use, and customizations that streamline work for individual programmers and development teams.

For scripts that everyone can use, check out the Scripting Starter Kit for Think C. You can download it from any of UserLand's on-line services, listed in the last section.

This article is a case study in customization. It takes you thru the development process of a single script, starting with a first proof-of-concept version, all the way to a very useful and complete script that saves us time every day.

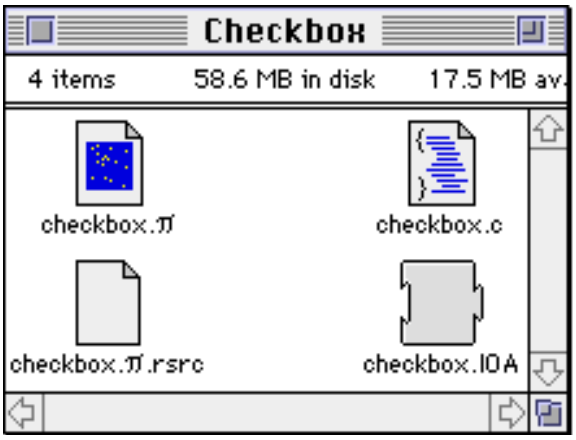
Background: The Iowa Plug-Ins folder

At UserLand, we have a set of Think C projects that share a common .h and .c file. They all live in the same folder:



There are two Frontier desktop scripts; one that rebuilds all the projects, producing code resource files, and another that installs the code resources in the System Folder. ioa.c and ioa.h are the common files used in all the projects. When these files change, all the projects must be rebuilt.

Looking inside one of the sub-folders:

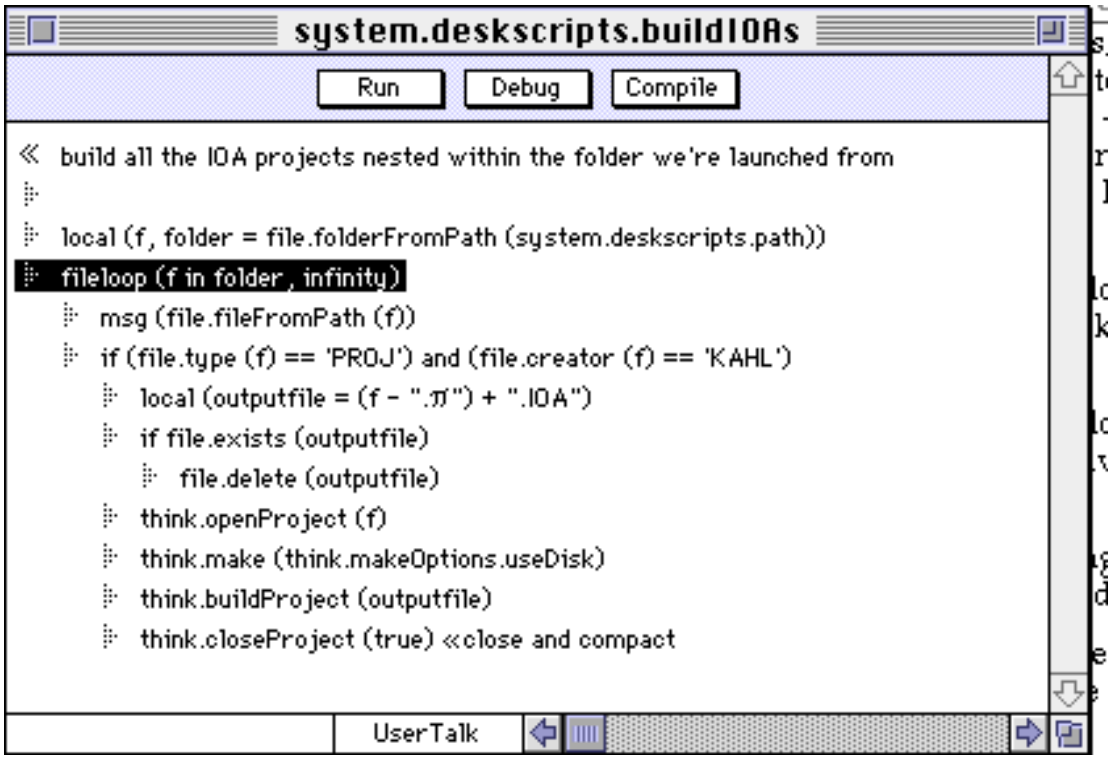


Each sub-folder has a common set of files: a project file, a C source file, a Resorcerer resource file and a code resource file produced by building the project.

In the following four sections we develop the script, starting with a basic script, then enhance it to include error checking, and then producing a final version that maintains a log of the compilations it does.

Version 1 -- The basic script

Here's the first version of the buildIOAs script, viewed in a Frontier script editing window:



The script loops over all files contained in the folder it was launched from. system.deskscripts.path contains the full path to the script file. It uses that path to determine which files it's supposed to operate on.

Two locals are set up: f is the loop parameter of the fileloop, folder contains the full path to the folder that contains the desktop script. Each time thru the loop, f contains the full path to one of the files contained within the folder. Frontier's fileloop construct takes an optional value determining the number of levels to traverse. We ask it to go infinitely deep, so we get to look at all files contained in all sub-folder levels.

The script displays the name of each file in Frontier's main window. It only operates on files that are Think C project files, whose creator id is 'KAHL' and file type is 'PROJ'.

For each project file the script calculates the name of the output file:

```
local (outputfile = (f - ".π") + ".IOA")
```

The variable outputfile is created locally to the script. f is a string containing the full path to the project file. The .π extension is subtracted, using string artithmetic, and then .IOA extension is added. If f's value is "System:Iowa Plug-

Ins:Checkbox:checkbox.π" the value of outputfile would be "System:Iowa Plug-Ins:Checkbox:checkbox.IOA".

We check to see if the output file exists; if it does, it is deleted. If the build failed due to a syntax error, we won't be left with an incompatible code file. The script then opens the project, dirties all changed files, recompiles the dirtied files, creates the output file and closes the project. These Frontier verbs work exactly as their interactive counterparts work. But they take variables as parameters, and as we'll see in the next version of this script, they can return error values.

Version 2 -- Add error checking

The first version of the script was useful, but what if one of the projects didn't compile because of a syntax error, linker error, or if we ran out of disk space? We wouldn't be totally out of luck, because the script deleted each output file before building it. Even so, we could be left with an inconsistent setup if we weren't watching the script run. And of course, that's one of the major benefits of using a script to do the dirty detail work for you. You can read the paper, return a phone call or play with your dog while the script is running.

So, in the second version of the buildIOAs script, we add some rudimentary error checking, so at least the script will stop running and leave an error message explaining why it stopped. In the next version, we'll make the script even better.

Here's the second version of the buildIOAs script:

```
local (f, folder = file.folderFromPath (system.deskscripts.path))
fileloop (f in folder, infinity)
    msg (file.fileFromPath (f))
    if think.isProjectFile (f)
        local (outputfile = (f - ".π") + ".IOA")
        if file.exists (outputfile)
            file.delete (outputfile)
        think.openProject (f)
        if not think.make (think.makeOptions.useDisk)
            scriptError ("think.make failed on " + file.fileFromPath (f))
        if not think.buildProject (outputfile)
            scriptError ("think.buildProject failed on " + file.fileFromPath (f))
        think.closeProject (true) «close and compact
```

Several things have changed in this version. First, we changed:

```
if (file.type (f) == 'PROJ') and (file.creator (f) == 'KAHL')
```

to:

```
if think.isProjectFile (f)
```

This takes advantage of a standard verb that makes it easier to determine if a file is a project file. Scripts that work with Think C seem to do this a lot, so we made it simpler to do.

We also check the returned value of each of the verbs where errors are important:

```
if not think.make (think.makeOptions.useDisk)
    scriptError ("think.make failed on " + file.fileFromPath (f))
if not think.buildProject (outputfile)
    scriptError ("think.buildProject failed on " + file.fileFromPath (f))
```

If there's a syntax error in checkbox.c, the Error Info window opens in Frontier:

Since the script is terminated by the scriptError call, Think C will be left pointing to the error when you come back from playing with the dog. If you click on the Go To button, the script editor opens the line that called scriptError, with Frontier's debugger active.

Version 3 -- Maintain an audit trail

It would be nice if the script did as much work as possible, reporting any errors in a list window, and then moving on to the next project.

We'll keep that list in a Frontier outline window. Every time the script runs, it adds a major headline to the outline:

Building all IOA projects on 4/28/93; 12:43:29 PM

And for every build it attempts, the script will report the result:

colorpopup.π: clean build

When it's done, not only will we know which projects built and which ones didn't, but we'll also accumulate a record of all the builds done. Here's what the log outline looks like after the first run:

Building all IOA projects on 4/28/93; 12:43:29 PM

- **button.π: clean build**
- **checkbox.π: make failed**
- **colorpopup.π: clean build**
- **edittext.π: clean build**
- **formula.π: clean build**
- **frame.π: clean build**
- **icon.π: clean build**
- **picture.π: clean build**
- **popup.π: clean build**
- **radio.π: clean build**
- **rect.π: clean build**
- **static.π: clean build**

Here's what version 3 of the script looks like:

```
local (log = @think.log)
local (dir = right)
on startLog (headline)
    if not defined (log^) «the outline isn't there, create it
        new (outlineType, log)
        editmenu.setFont ("geneva")
        editmenu.setFontSize (9)
    edit (log) «open the log in a window
    target.set (log) «all outline verbs apply to this window
    op.firstSummit () «move to the first line in the outline
    op.go (down, infinity) «move to the last top-level line in the outline
    op.insert (headline, down)
on addToLog (subhead)
    msg (subhead) «display string in Frontier's main window
    op.insert (subhead, dir) «insert it in the outline
    dir = down
on closeLog ()
    target.clear ()
local (f, folder = file.folderFromPath (system.deskscripts.path))
startLog ("Building all IOA projects on " + clock.now ())
fileloop (f in folder, infinity)
    msg (file.fileFromPath (f))
    if think.isProjectFile (f)
        local (outputfile = (f - ".π") + ".IOA", result)
        if file.exists (outputfile)
            file.delete (outputfile)
        think.openProject (f)
        if think.make (think.makeOptions.useDisk)
            if think.buildProject (outputfile)
                result = "clean build"
            else
                result = "build failed"
        else
            result = "make failed"
        addToLog (file.fileFromPath (f) + ": " + result)
        think.closeProject (true) «close and compact
closeLog ()
```

Several new techniques are being used in this version. It adds two new locals to support the log outline, and defines three local subroutines, startLog, addToLog and

closeLog. In Frontier, as in Pascal, scripts can have local subroutines, nested to any level that makes sense to the programmer. startLog creates a new log outline if one doesn’t exist, and opens it in a window. It adds the main headline for this script. addToLog adds a new headline subordinate to the main headline. closeLog does whatever cleaning up is necessary (for this version not much cleaning up is needed).

Calls to startLog, addToLog and closeLog have been added in the main body of the script.

By the way, this script is much easier to read in Frontier's script editor because you can collapse and expand headings to see as much or as little detail as you like.

Version 4 -- Factor the script

Here's the final version of the script:

```
local (f, folder = file.folderFromPath (system.deskscripts.path))
think.notebook.start ("Building all IOA projects on " + clock.now ())
fileloop (f in folder, infinity)
    msg (file.fileFromPath (f))
    if think.isProjectFile (f)
        local (outputfile = (f - ".π") + ".IOA", result)
        if file.exists (outputfile)
            file.delete (outputfile)
        think.openProject (f)
        if think.make (think.makeOptions.useDisk)
            if think.buildProject (outputfile)
                result = "clean build"
            else
                result = "build failed"
        else
            result = "make failed"
        think.notebook.add (file.fileFromPath (f) + ": " + result)
        think.closeProject (true) «close and compact
think.notebook.close ()
```

We've made the script a lot simpler this time by moving most of the code for maintaining the log outline into a sub-table of the Think table, called Think.notebook. Of course this makes the script simpler, but it also makes it possible to use the notebook scripts from other Think C-related scripts.

That's it! There may be other improvements we could make to the script, but there has to be a time to stop and this is it. Having invested about an hour in creating thIs script, we reap the benefit every time the header file changes and all the plug-ins need to be rebuilt. In the first week, we recouped much more than the hour we invested. Plus, the cost to make a change is much smaller, so we make more changes. The result is more powerful, more reliable and easier to use software.

Finally, if you have any questions, comments or suggestions, please get in touch through one of UserLand's on-line services. If you're an AppleLink user, check out the UserLand Discussion Board under the Third Parties icon. On CompuServe, visit the UserLand Forum in the Computing Support section, or enter GO USERLAND at any ! prompt. On America On-Line, enter the keyword USERLAND.