DISGUISING FORTH

Presented by Bob Nash at the November 20th, 2004 Forth Day

Abstract

Although Forth is one of the first choices for new applications with challenging requirements, this early success is often forgotten and the application is later targeted for conversion to a more ”mainstream” or ”maintainable” programming language. Based on a recent conversion effort, this paper discusses some ways that Forth applications can be disguised so that they can survive a conversion push (putsch?).

Haiku Abstract

Forth on the windy path
Precious Words in a Windows frame;
Embrace the stormy change

Haiku Rules as I understand them:

1. A classic Haiku poem must have a seasonal element (e.g., mention of cherry blossoms for Spring) - this rule is sometimes violated in ”phony Haiku”
2. The first line must be 5 syllables, the second 7 syllables, and the third 5 syllables. Some variation from this form is permitted, usually as needed to accomodate languages other than Japanese.
3. The last line must express a separate thought but must also be able to be interpreted such that it relates to the thought expressed in the first two lines.
4. Ideally, the last line, when considered in the context of the first two, should introduce a meaning or perspective that is completely different than those given by either the first two lines or the last line of the program.
5. If you think it is easy to compose an ”adequate” Haiku, you probably don’t understand it (I don’t, hence the ”phony Haiku”).

Introduction

Forth is often the language of choice for initially implementing applications with short implementation times, demanding real-time requirements or interactive development with new hardware.

As useful as Forth may be for leading edge projects, many Forth professionals find themselves in the unfortunate position of seeing their programs converted to "more mainstream" programming languages such as C, C++, Java or Visual Basic.

One approach to combating an illogical response to imagined problems is to present a reasoned argument showing how Forth is not only the right initial choice, but is also the best long-term choice. This approach may be intellectually satisfying but, as often as not, it is unsuccessful.

This paper describes another approach to preserving Forth's presence by minimizing its application profile. This is accomplished by using Forth's traditional modular programming techniques and by presenting straightforward ways for Forth components to interact with each other and users.

This approach also uses tools such as HTML and browsers preserve the "look and feel" of mainstream programming while using Forth to do the real work.

This author is presently using this approach to rework an existing application that consists of three standalone Windows programs, all written in Swift Forth. This application has been in continuous unattended service since its initial deployment in June 2001 but is now being targeted for conversion to "a more maintainable mainstream programming language."

Doomsday

When first told that my application was to be converted to another language, my response was entirely predictable: at first I denied that it would really happen, which was followed by anger that my programs were being targeted.

Predictably, I went through all of the rest of the emotional gamut, finally reaching the acceptance stage. I had to face reality: the conversion was probably inevitable because the replacement project was already budgeted and there was a scope statement including the words “mainstream” and “maintainable.”

After finally coming to terms with user and management concerns, I decided not to expend any more effort explaining why the applications should not be converted. Instead, I tried to figure out how my moderately complex application could be converted without exceeding budget or compromising reliability and functionality.

A Thin Veil

My first attempt at retaining Forth while making the existing applications look more “mainstream” was to convert the user interface from a native Windows API to a browser.

Initially, this was just an attempt to easily reduce the size of Forth executable programs while using HTML and an HTTP Server for the user interface. I felt that by reducing the size of three standalone Forth-based Windows programs, it would be easier to write a functional specification for the conversion project.

I also hoped that several smaller executable programs, each with limited functionality, would be less attractive conversion targets. The expectation was that many of the smaller programs would be dismissed as “insignificant” in the conversion.

Unexpected Results

I did not expect the user interface conversion to have much visibility. In fact, to disguise the scope of the changes, I tried to minimize the significance of any new functionality. New capabilities were casually noted in emails, but were not touted as major improvements.

Unexpectedly, the converted user interface was not only very successful from a user and management standpoint, but it also helped me see how to more effectively retain Forth for the more essential parts of my application.

The users appreciated the conversion because they could now access many functions remotely with a browser. Previously, many functions could be performed only at the host computer using program menus and the Windows API.

One of the less-used functions of the application was to display a daily summary report. The report could be displayed at the host, but it couldn't be easily printed because there was no local printer. With the new browser interface, users can use a browser to both view and print the report from any PC within our corporate computing domain.

Management's response was also gratifying. Because of the positive user comments, the initial stages of the conversion were perceived as being very effective.

More importantly, I was perceived to be a "team player" when I exhibited the converted user interface code and it was seen to be mostly HTML statements. When queried about the "WEB server", I replied that it was just a public domain program that I had downloaded. Of course, I didn't mention that the server was programmed in Forth (actually, Swift Forth).

Enthusiasm for Apathy

The lack of interest in the source and nature of my HTTP server was the beginning of my first big discovery: nobody was interested in how a standalone Windows program was written, provided that it performed all of its functions reliably.

Why then was there concern about the programs in my application being written in Forth? They had been continuously and reliably performing their functions for more than four years and yet they were perceived as being a risk. If only the functionality of the programs was important, why was the fact that they were written in Forth of any significance?

At first, I thought that there was a perception that Forth itself was an unreliable or risky language, but this did not seem to be confirmed in my conversations with management. The source of concern was the difficulty of performing changes.

Now I understood that the problem was not with Forth itself, but with the perceived availability of the expertise needed to change the way that the application worked.

There was also some concern that there might be undetected latent defects that were not discovered in testing and that they would be difficult to correct by a programmer not familiar with Forth.

Revalidating Forth

I now understood that, in addition to converting my applications' GUIs, I also needed to break them down into programs that interacted with each other in simple, reliable and testable ways. On examining my programs, I found that many of them did multiple tasks and did not have a clearly defined purpose.

For example, I had one program that monitored encoded ASCII text over a serial port from a radio receiver. Because of this monitoring function, I named the program "MONA" (MONitor off Air transmissions). However, this same program also sent ASCII text and control mnemonics to a signboard through a second serial port. The signboard application was included in MONA to avoid duplication of some common Windows API code.

I decided that, in addition to stripping out the user interface, the MONA program should have been split into two different programs, each with their own purpose.

This decision was validated when the user representative informed me that the signboard display function would be eliminated because my browser interface now allowed signboard information to be displayed in a browser window on user desktops. Thus, users could now see the signboard information regardless of where their desks were located in relation to the signboard display.

After re-factoring MONA to be smaller with a single purpose in life, I felt a little smug: my browser interface now provided an on-demand display and printout of recent radio transmissions. Also, MONA was now a smaller single-purpose program that was well validated and reliable. It is likely that MONA will not be touched in conversion effort.

My pride was soon wounded, however, when I discovered that I had just re-learned one of the most basic principles of good Forth programming: every Word had to have a well-defined purpose (e.g., it could be named) and the inputs and outputs should be simple, well defined and testable. I had failed to learn this important lesson, at least as it applied to programs as well as Forth Words.

Other Benefits

I also discovered that there were other benefits to modular programs in addition to the well-understood advantages of modularly designing Words and applications.

One benefit was that that the mechanism I most often used to exchange information between my re-factored programs was a text file (static string stack?). This was the same as one aspect of Unix programming that I appreciated: many Unix programs and utilities took text files as input and produced text files as output.

This decoupling with text files made it much easier to produce functional specifications and run validation suites. Unfortunately, these benefits also applied to re-writing existing programs in a different language. But, it also made it easier to determine if a conversion is successful (often it is not).

The Challenge of Change

Although the approach described above makes the application more robust, reliable and easier to develop, two problems remain: program changes and latent defects (bugs, to the unsophisticated). Remember, the fear that a program would have to be changed was at the heart of the threat to my current application.

The best way I have found to address these challenges is to accommodate changes with text-based parameter files or with scripting languages, a traditional strength of Forth.

This effort is the most difficult aspect of producing standalone turnkey programs. Users must be convinced that the scope of changes that they will make will fall within the available configuration parameters or within the capabilities of the scripting language. This is a hard sell. But, it is generally easier to sell the flexibility of a program than to argue that it will never need to be changed or that, if it does need change, the Forth expertise will be available to change it.

It is surprising that users and managers will readily accept a variety of scripting languages but will balk at programs written in Forth. Perhaps it is felt that scripting languages are inherently more understandable or easier to program.

Bugaboos

What about bugs? The best answer I have is: testing. If it can be demonstrated that the program behaves reliably with a comprehensive test suite, then users can be assured that the defects are at least outside the scope of normal, or at least critical, operation.

This is both good and bad news.

The bad news is that the application designer must develop and validate comprehensive test suites.

The good news is that, if test suites are developed incrementally and modularly, this is not such an onerous task.

Also, the testing imperative forces applications to be broken down into easily tested modules. This too is good news.

The best news is that, once developed, these test suites can be used for many purposes, including functional specifications, program validation and regression testing.

Lessons Learned

The most important lesson learned in the conversion effort was that Forth programming principles apply at many levels and can simplify many tasks, even the task of converting to a different language.

I now recognize the true wisdom of the trite maxim: every challenge is also an opportunity. I started out contemplating the destruction of my precious programs and ended up discovering a better way to build and to extend the life of Forth applications.

Although the project to convert my Forth application has not yet started, I feel confident that some of my programs will survive in their re-factored form. They will survive mostly because their functionality is relatively static and they have been proven reliable with extensive testing.

Although it is inevitable that new programs written in a different language will supplant some existing functionality, a modular design at least makes it easier to provide functional specifications and ensure that they are met.

A Parthing Shot

In discussing the re-writing of existing applications, I have found it useful t to make users and management aware of the relative roles of languages and applications.

I observe that being an expert programmer is of limited use without being familiar with the requirements of the application. I emphasize that an expert C programmer cannot easily change any C application without being familiar with the application's functional requirements and with the programming algorithms used to implement them. This is something that most programmers assume is well understood but, often, it is not.

Considering the above, I also find it useful to ask where the programming and application expertise will reside. This expertise can be maintained in-house or provided by a contractor.

Forcing a conscious choice of one of several development and maintenance alternatives, such as in-house/contractor or contractor/contractor, can often help users and management make choices that best fit the available contract and organizational resources.