Opening Statement: GoLUG Programming Practices Panel Discussion

by Steve Litt

10/4/2010

Copyright (C) 2010, Steve Litt. All rights reserved. Steve Litt consents to and gives you the right to copy and distribute verbatim copies of this document without further consent. NO WARRANTY, USE AT YOUR OWN RISK.

Contents

1 Overview **2**

2 User and Owner Needs **2**
- 2.1 User Friendly . . . 2
- 2.2 Productive for the user . . . 3
- 2.3 Security . . . 3
- 2.4 Minimal bugs . . . 4

3 The need for speed (programming speed) **4**
- 3.1 Beware of the slog . . . 4
  - 3.1.1 The Rewrite Debacle . . . 4
  - 3.1.2 The Powerbuilder delay . . . 5
  - 3.1.3 The Slog: Bottom Line . . . 6
- 3.2 In Praise of the Quickie . . . 6
  - 3.2.1 The Instant Lookup . . . 6
  - 3.2.2 The Library Stamper . . . 7
  - 3.2.3 The SNMP Controller . . . 7
  - 3.2.4 Time Tracking On the Cheap . . . 7
  - 3.2.5 The Quickie: Bottom Line . . . 8
- 3.3 Rapid Application Development (RAD) Techniques . . . 8

4 There's no "one way" to program **9**

Programming is the act of giving instructions to the computer enabling the computer to later and repeatedly perform a service for the user. Some folks draw a distinction that programming with a "scripting language" isn't programming, but this is like saying that travelling on a bicycle isn't travelling or saving money in a piggy bank isn't saving. In all the preceding cases, the end result is the same and the process is essentially the same.

1 Overview

This is my (Steve Litt's) opening statement for the GoLUG programming practices panel discussion. This opening statement will cover the following:

- User and owner needs
- The need for speed (programming speed)
- There's no "one way" to program

2 User and Owner Needs

An owner (corporation, organization or person) asks programmers to write a program to perform a service for the users. Sounds simple enough, but it actually encompasses a vast array of situations.

The owner, user and programmer can be one single person. Or the owner can be a multi-billion dollar corporation, the users can be a large subset of all the earth's computer users, with hundreds of programmers writing the code.

Users can be employees of the owner, the owner's customers, or casual users of the owner's services. This makes a huge difference. The owner can compel its employees to train on the software, but casual users require an absolutely obvious user interface, and the case of a customer is somewhere in the middle.

2.1 User Friendly

All other things being equal, all computer programs should be obvious and user friendly. But all other things are never equal. Here are some typical

tradeoffs:

- User friendly vs. user productivity (once trained)
- User friendly vs. programmer time and expense
- User friendly vs. program simplicity

In other words, user friendly is not free. Factors like user turnover, naive users and a large, hard to train user community favor user friendly. Factors like user longevity, intelligent or experienced users, and training programs favor less user friendliness.

The owner determines the user friendlieness in his specs. Hopefully the owner has considered the cost.

2.2 Productive for the user

Programmers must respect the user's time. Don't make the user look up a filename and type it in, but instead provide a filepicker. Don't alternately ask for user input and then run a 2 minute process, but instead ask for all info up front and then run all the 2 minute processes.

Mousing is a productivity killer. It's OK on tasks done seldomly, but the user should be given keyboard ways to do common tasks. This is why the keyboard centric Vim editor is so much faster to use than mouse driven editors. You can hit five to seven keystrokes in the time it takes to reach for a mouse, click something, and return your hand to the keyboard. You can press three keystrokes in the time it takes to move the mouse and click something.

Mice are great for user friendly, but ugly for user productivity.

2.3 Security

Hacks cost big money. Unless the program is used exclusively by one or two known, long term users, always validate user intput. Regardless of users, always check for errant pointers, out of bounds array accesses, and buffer overflow (once more, user input validation). Never use strcpy(), always use strncpy().

There are times when security must be enhanced, and times when it can be relaxed. Running a web app, especially a popular one, accessible by all

calls for the strictest security measures. On the other hand, if you write a little program to help yourself keep track of your books, you can afford to relax security, but be careful — if this becomes free software used by many, you need to replace the lost security.

2.4 Minimal bugs

With the possible exceptions of Knuth's TEX program and DJB's Qmail, all software has bugs. Strive for continuous improvement in bug removal, and try very hard that no bugs cause data loss.

Debugging takes time and costs money. In some circumstances, such as limited users who are knowledgeable, it might be cost effective to "let it fly" even though known bugs have not been fixed, as long as those bugs can be conveniently worked around by the user.

3 The need for speed (programming speed)

The user and owner pay, and pay dearly, for programming time. As a project gets more and more behind schedule and over budget, the owners think more and more about not throwing good money after bad. And of course, if you get too far behind, a new product makes the whole thing unnecessary and makes you look like an idiot.

Oppositely, if you can deliver a working program before anyone expects it, you'll be a hero and maybe can get on the team that delivers the better iteration.

3.1 Beware of the slog

Projects that slog along become ever more likely to get cancelled, with all the slash and burn that entails. Try very hard never to be on a project that's behind.

3.1.1 The Rewrite Debacle

As a junior programmer in 1985, had a contract to work on a medical management company's medical management software, which they put in doctors offices throughout the LA area. In 1985 this software was still the flagship

of the industry, ran all the finances and logistics at a doctor's office, large or small. Trouble was, our software was increasingly long in the tooth, too glitchy, too hard to modify. The company wanted a rewrite.

I submitted a plan to rewrite in 9 months, but the owners, the consulting company, and the lead programmer all said 9 months was too much time, so the lead programmer made a basic interpreter with built in relational database access, and we tried rewriting it in that interpreter. No luck, too slow, too crashy.

The next year we rewrote in Focus. Too slow, too difficult to program our special ways of handling things.

The next year we rewrote in some startup development environment. I was fired in July 1987 for not supporting that decision, and I'm proud of that firing.

By 1989 the new system was almost finished, but they'd run out of time, run out of money, fired all their programmers, and were trying to sell the system for pennies on the dollar. Their business had been taken by one of their former clients who now had a better program. As far as I know, the company went out of business.

By the way, looking at it in hindsight, the way our original program was built, we could have replaced it piecewise in very little time, because the system was comprised primarily of individual executables connected with menus. Whoops!

3.1.2 The Powerbuilder delay

In the 1990's I was a contractor on a highly skilled team of 3 writing a very visible enterprise app in Powerbuilder. Everyone including me felt that three people wasn't enough to make the deadline necessitated by the rapid aging of their current enterprise app. So they hired a new Powerbuilder guy, who promptly set the schedule back even more by running to the three of us every 15 minutes asking how to write basic Powerbuilder.

The project got behind, and as it got behind we had less time to debug, and with a cast in stone delivery date, we ended up delivering something buggy and slow just for lack of time. The project's leader found another job after seeing the writing on the wall. I left soon after. The bad programmer was fired a couple years later.

Had we finished that app on time with our normal quality, we would have gotten a tickertape parade and would have had credentials to program

anywhere in Powerbuilder. Instead, all of us had to pull ourselves up by our bootstraps all over again.

3.1.3 The Slog: Bottom Line

When you get behind, bad things happen, to your employer, yourself and your customer. When trying to deliver a Rolls Royce, be careful you don't deliver a half-assembled Rolls Royce a year too late. Delivering a Pinto would be a much better idea. You can always get the Rolls Royce later.

3.2 In Praise of the Quickie

During my 1984-1998 full time programming career, I got a lot of tickertape parades for little quickie programs that did what the user needed, and only what the user needed.

3.2.1 The Instant Lookup

In 1985 the DP manager came to me surrepticiously, around the chain of command, to ask me to create a program to merge all the 25 separate doctor office databases in order to match incoming checks to accouts. Until this time, incoming checks were handled by several clerks with daily printed fanfold account listings for each doctor. It was error prone, expensive, and used a box of paper a day.

For three days I worked on her request in parallel with the work I was supposed to have been doing, so nobody knew I was doing unauthorized work. Slightly modifying the binary search algorithm in Nicholas Wirths "Data Structures and Algorithms" book so it pulled up the first several of many same-named items, I wrote a program that the user types in the first few letters of the last name, hits Enter, and twenty matching names that sort equal or above that popped up immediately (even on an overloaded PDP11, binary search is fast). Then I wrote a program to concatinate all accounts from all doctors offices and sort them by account name, and that was what my binary search was run against.

The result, it cut down the number of clerks to 1, eliminated the box a day of paper, and produced more accurate results. The DP manager let everyone know I wrote the program (that was part of the deal), and my name was spoken lovingly by all the executives.

3.2.2 The Library Stamper

Contracting for a law firm in 1988, the DP manager asked me to give the law librarian what she needed, but do it quick — we had lots more important stuff to do. I queried the librarian for what she needed, which was basically a prompt to type a number and have an algorithm convert it to a number to print on a book, and print it out. A half hour later in Turbo Pascal, her command line utility was ready.

In 1998, with the whole law firm windowz-ized, I saw the library was still using my Turbo Pascal quickie. I said "don't you want something more sophisticated?" They said "No, this does exactly what we want."

3.2.3 The SNMP Controller

2000 rolled around and an Orlando company called me for a quickie Perl project. It was the height of the dot-net boom, and they were selling a system to have your own building wide IT infrastructure complete with Cisco Routers in your high rise. They'd sold a bunch of them and their crew of Java programmers was working hard on providing the program to control the Cisco routers, but the Java team was several months from completion.

The IT director, who was an old school guy, called me up and asked me to do a quickie so customers wouldn't revolt waiting for the Java guys. He even told me the Perl module to use. Three days later I delivered what they needed, and took the heat off. The Java programmers got the time to do their project right because of my quickie.

3.2.4 Time Tracking On the Cheap

Some time in the early 21st century, I was working on a couple very hot deadline contracts, and needed a time tracking program. But with all my billable work, I had almost no time to either make a time tracking program or research one to download or buy.

So I took a couple hours to make a Perl program that did the very basics. No user interface — it was controlled entirely by command line arguments. No problem, I made a few scripts to call it, and then a week or so later bolted a UMENU front end on it. Whatever, writing that program took only 2 hours out of my billable working time.

3.2.5 The Quickie: Bottom Line

The quickie app is not always appropriate. When hundreds of users depend on it, and the company reputation is on the line, a quickie is appropriate only as a temporary detour until the real app is built. When users are naive and need the ultimate of user friendly, quickies fail. When you have the manpower and manpower for an enterprise quality app, and that's what the powers that be demand, that's what you produce.

But quickies are appropriate far more often than people think. All but one of the preceding quickie examples happened at companies with over 50 employees and one with well over 1000.

Writing a quickie app is a low risk, high potential reward activity.

Even though a quickie app by definition is not the highest quality app, it can be rewritten with little loss because it had little investment, and that little investment is usually compensated by the increased problem domain knowledged garnered by deploying and using the quickie.

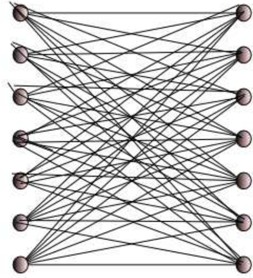
When doing a quickie, languages like Perl, Python, Ruby and Bash can be outstanding parts of the project due to their use of small executables that can keep bugs contained and can be tested and measured independently.

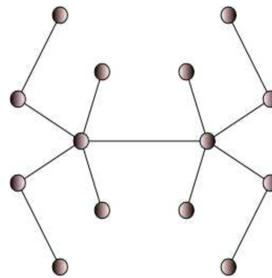
3.3 Rapid Application Development (RAD) Techniques

One of the simplest RAD techniques is to keep expectations modest. This is especially handy on version 1 of a program, when nobody has a handle on what it's going to be like when actually used. It may not be "pretty", and some may call it "unprofessional", but if it's an internal app it eliminates most potential for loss.

Another RAD technique is developing the system from small executables. When you double the source code size of an executable, you much more than double its complexity and time to completion, because complexity in a single executable depends not on a multiple of the components, but the factorial of the components.

Yes, yes, I know, I know, it doesn't have to be that way. A programmer could design a monolithic app so modularly that it complexity is close to proportional to its size. It's possible, but in real life it seldom happens, and as maintenance programmers make improvements, even the best laid out program starts to get messy. Small executables *enforce* modularity.





Another RAD technique involves drag and drop construction of forms, reports and database accesses. What's wonderful is that forms, reports and database accesses are among the most time consuming programmer activities, so automating them nitromethanes your work. Delphi, Lazarus, Powerbuilder, and a host of others offer this quick programming technique.

Intelligent defaults can double or triple your programming speed. Rails and Clarion are two that do this well.

4 There's no "one way" to program

Variables in programming include user intelligence, user knowledge, user patience, user training, user turnover, user expectations, owner expectations, programmer training, programmer skills, number of programmers deployable on the project, budget and deadline.

Two more variables are whether the software is a product sold to customers, a web app rented to customers, a web app offered free to people, an application to be used in house for many users, or something used by just a few users.

Commercial apps need to be pretty, in-house stuff doesn't. Apps used by externals needs to be user friendly, in-house users with longevity and training don't need user friendly and need productivity a lot more. IT crews with just a few programmers and a low budget must sacrifice ultimate quality for time and budgetary considerations, large IT departments with many programmers and large budgets do not.

Shellscripts encourage small executables and therefore modularity. They're not pretty, and they're usually not acceptable for commercial software sold to others, but combined with other types of programs they can be a powerful

force for maintenance of schedule and budget.

Perl has the handiest regular expressions and is therefore an obvious tool for parse-heavy tasks. Like shellscripts, Perl can be used as a glue language to bind other small executables. Python and Ruby are similarly productive languages, though not quite as easy with respect to Regex. Perl, Python and Ruby programmers are easy to find.

C# and Java are wonderful tools for building large, monolithic applications, especially if Internet connected. They're wonderful for making commercial apps, as is C++. These programmers are fairly easy to find, though good ones are expensive.

Frameworks speed work by giving autocompletion on right clicking and providing an object model to partially convert programming to a fill-in-the-blanks type activity, at least the run of the mill programming. That way you can reserve your hand coding for where it's needed — special programming that takes some intelligence.

If you want a web app yesterday, Rails is your friend, always assuming you know Rails well. Use its intelligent defaults intelligently, and it partially builds itself.

For thick-client GUI apps built fast, especially if done as small executables in a bigger system, stuff like Delphi, Lazarus are wonderful.

There's no law against building a system with many or all of these languages. You could acquire user input and perhaps database data with Lazarus, parse with Perl, and maybe wrap the whole thing with a simple shellscript to provide input in the form of environment variables. You can write big iron C# executables with C routines to get right next to the metal or speed up a particularly CPU intensive process.

Bottom line: Use the best tools and techniques for the job at hand.