Ideas for Crafting Software for an Internet-Centric University

C. Frank Starmer and Josh D. Starmer

Paradigm

The main idea is that you, the user, should be empowered to rapidly access information in order to synthesize solutions to your problems. Said another way, our goal is to provide tools that facilitate access to and transport of information from remote resources without demanding you to install lots of techy stuff (that often does not work) possibly requiring a "certified" specialist to install the stuff (like ODBC drivers).

Remember the early Mac days when you "discovered" copy and paste, a legacy from Xerox Parc and MIT and X windows. Popularizing the intuitively obvious click, drag, copy and paste opened an entirely new world for porting information across applications on the same desktop. Cut and paste augmented our memories and improved the rate we could develop documents (either programs or traditional documents).

Our goal at MUSC and the IT Lab is to build an IT software infrastructure, i.e. browser-based tools, that not only provide access to information but provide you with simple tools for manipulating that data. Why? Because thats what you do anyway in the paper and pencil world. So for today's problem, we must enable you to rapidly synthesize a solutions to your problems, i.e. enabling you to innovate. Rapid synthesis of solutions is facilitated by a good memory. What is new in our approach is that we explicitly realize that internet connectivity + commodity computing creates a new memory tool - facilitating access to information and thus transferring time spent remembering and accessing information to time spent thinking and synthesizing problem solutions. Our approach builds on the hardware/network internet connectivity by placing software tools on your desktop that enable you to to identify and transport information from resources to your desktop - in as transparent manner as possible.

What is a Tool?

Our approach is tool based. What is a tool? A tool is a software process that does one thing. It takes input data, performs some transformation and generates an output data stream. I first saw "tool" described in early UNIX documentation (see Bell System Technical Journal 57:6 July-August 1978 (Special Issue on Unix Time-sharing system) - Tools are mentioned in the Forward, page 1901). But to my mind, the major contributions from Ritchie, Thompson, Kernighan, McIlroy and Bourne that made UNIX (and Linux) withstand the test of time without breaking was:

1. input/output device independence (data streams, not card images)
2. command-line parameters (that fine tuned the "what the tool does")
3. pipes (permiting composition of tools (grep "something" file.txt | more))
4. unix file structure

Why tools? Because we can often synthesize a solution by realizing the solution as a sequence of data transformations. But building the solution as a composition of tools, we reduce the likelihood we'll break something when repairing a bug in the overall solution process.

How do you know if you have a tool? An extreme test is to describe what the tool does. If the description contains a conjuction (and) then its not a tool! Why is this an issue? Because problems are solved with compositions of tools (often described in shell scripts) and if the tool does more than one thing, then its reusability may be seriously limited. Some may reply that there really are no tools by

this definition. For example, "cat " is what I would classify as a tool but has many "options". With no command line options, it displays all printable characters while "cat -A" displays both printable and unprintable characters. The appended command line argument acts to fine tune the tool's data transformation and taken together, makes a primitive tool we described above.

A Test of the Tool Paradigm in a setting where change was the norm

During the 70s at Duke, we had crafted a large database for Cardiology that was built on the "system" paradigm, not the tool paradigm. It was a large collection of procedures wired together into a system that provided terminal management, data collection, report generation, database management and statistical analyses. It was a good project - but difficult to maintain and evolve. In 1980, I switched from the database group to working with Gus Grant, and developed software for managing experiments and the follow-on data analysis. In the lab, things change every day and the "system" approach would make me crazy with continuous evolution and horrible debugging and testing nightmares. Moreover, Gus was an acid test. He was not comfortable with using anything related to a computer. I decided to try the tool paradigm and package tools into compositions imbedded in shell scripts. Then, when he changed an experimental protocol, we alted the command line parameters of the components of a shell script. Starting in 1977, we had a UNIX system running and understood about pipes and shell scripts and redirection. I thought that if we could manufacture the right collection of tools, then we could rapidly adapt to changes in experimental protocols and Gus would not be overwhelmed with the computer side of the lab. The analogy I was working under was that of the local garage auto mechanic. He/She has a toolbox with a limited number of tools - yet they can attack most any flavor of automobile. Why not try the same approach to cardiac electrophysiology. We did it, and with Marge Dietz, wrote a suite of tools, lablib, that continues to be used today. We've not added a tool for 8 or 9 years, and find that we can manage any experiment Gus decides to run.

Inductive Software Engineering: How are tools identified?

Where do the tools come from? This is the inductive part of our strategy. We (the friendly folks at the IT Lab), help you solve your problem, with an eye on repeating themes and similarities to other problems we have faced. When the number of repeats exceeds a threshold, a tool may be useful. The inductive component arises from extending the problem base that is used to identify repeating themes from 1 to 2 to 3 ... n. The "art" of inductive software engineering is identifying the repeating themes with a degree of abstraction that reveals an underlying simplicity. Its like the series of classics Don Knuth wrote: The Art of Computer Programming. Its our ability to understand the problem, to abstract it by identifying the essential processes, and to see repeating themes from our abstractions. This is the art of programming, the art of software engineering, the art of making ice cream by the IT Lab.

Everything is Browser-based

The Internet opens a whole new world of accessible data resources. Perhaps one of the major contributions was that the internet freed us from the world of video terminals that were more or less tightly coupled to specific computing systems. The internet looked to me like a big distributed switch that made it possible to access, in parallel, resources located in different places and move information between resources as needed to solve a problem. Its the movement of information across applications that I believe is important. X windows and the Mac showed how addictive the GUI copy/paste operation can be for applications running on your desktop. At MUSC we have created not only data manipulation tools, but a neat web interfaces to institional databases that makes database access intuitively obvious. The my- database interfaces provide both a clean,access to databases but also a way to transport data into a local spreadsheet for further processing (click on the myGrants at

mySites). Here is another example, a tool to build a web presentation around a database: mySiteMaker.

The Zurich airport recently convinced me of the importance of moving to browser-based applications. Walking down the concourse, there are free web -appliances available that are running either Netscape or IE. Not only could I check my email - but I could look at the operational status of our network and check our grants database - while waiting for a flight to Sarajevo.

Web-tools: Extending the action of a tool from desktop to the web

Where is the next frontier for tool development? We have recently identifyed a very nice web-motivated tool extension for moving data between applications. Traditionally, tools work fine for objects residing locally on your desktop. With browsers, we have been slow to figure out uniform ways to make objects imbedded within a browser display the targets of a tool action. Now we think we have it: make the object a URL and have a script that parses the URL for the object you wish to attack with the tool. Frequently used tools such as copy/paste, join, select appear suitable for extension to the web. Here, the idea is that you sit at your desk and invoke a tool where the target is either a local object or a URL. Check out a web-copy/paste tool . Recently, we've added another bookmarklet like table export - a spell check service, ispell accessible via a SOAP call. Check out a spell check utility

Why do folks resist this strategy and why surviving the needs of tomorrow demands this strategy?

Here, I'll speculate about the world of IT professionals and points of resistance that conflict with the change life cycle introduced by the internet. There are those that trained before the 1980s and those trained after the 1980s. Prior to the 1980s, the command line was the primary method for executing programs. During the 1980s, the GUI came into vogue and the double click became the way to fire up programs on a computer. Both methods have their pros and cons. The command line lent itself to maximum flexibilty while the GUI lent itself to maximum ease of use. On the other hand, command line programs tended to be, for the average user, cryptic while GUI software tended to have very limited flexibility.

The command line, with such structures as pipes and file redirection, shaped the way many people thought about how software should be used. These stuctures made it possible to string together many programs to give the functionality of a large scale integrated program (what we will refer to as "package software"). During the 1980s, GUIs became popular and these also shaped the way poeople thought about software. Since double clicking the mouse did not lend itself to stringing together many programs to get a task done, package software began to flurish. While limited in their flexibility, package software seemed to solve most of the problems typical users were having. Users whose needs went beyond what the package offered were forced to wait for that pie in the sky typically referred to as: *The Latest Upgrade*.

With the 1990s came the popularization of the internet via the world wide web. The world wide web presented a whole new way of distributing data and software throughout networks of computers. This easy way to communicate led people to have vast expectations of what could now be done that could never have been done before. This potential, however, has had difficutlty being fully realized because of how heterogenus the world wide web is and how quickly it can change. There is no single operating system that all computers use on the web and there is no single suite of applications that you can expect to find on each machine. Because there is so much variety and things are constantly changing, waiting of the latest upgrade stopped being an even remotely realistic way to solve problems. The

problems were changing faster than *The Latest Release* could be made available. The flexibility offered by the command line was required to stay afloat. The problem then was how can one capture the flexiblity of the command line and still allow typical users to be able to understand how to use the software. The key to this is by putting all of the user interface in a web browser and finding a clever way to distribute the command line scripting between the client and the server. (And if Napster and Gnutella have their way, there will be no distinction between client and server.)

The problems that we are having today is that most software developers are either unfamiliar with or not willing to try to solve problems using the best features acquired during the past three decades. People who are trying to solve problems with only command line applications are neglecting the effectiveness that a GUI can have in terms of making the software usable. People who are still trying to solve problems using a package software approach haven't yet realized how futile this is becoming.

The futility is a product of the internet. Now that the internet provides dynamic connectivity with many different data sources, the time constant of adapting a package to a new web-related feature (e.g. browsers) is simply too long. Scripts, simple pipe compositions, packaged within a browser page or cgi represent a very effective way of leveraging programming effort. Its a natural way to construct solutions to data manipulation problems. They represent a very efficient way to rapidly prototype a tool so that a composition can be created that addresses a user's problem. Its a very effective and efficient way to go with the flow of web development. Now, the visual interface environment has provided a way to make composition more or less a point and click exercise. From my perspective, the best tools available today reflect contributions from the command-line era of the 70s, the gui and package era of the 80s and the web era of the 90s.