The Stories of Software

Online Edition

Bits Versus Electrons

The Stories of Software

By Bob Frankston

Today, there is a lot of interest in teaching people to code as a basic skill. This seems to make a lot of sense in a world that is increasingly being redefined by software. Yet, I can't help but think that it's akin to teaching typing rather than teaching people how to write or, more importantly, how to communicate. The consumer product and services industry (no longer just consumer electronics) is creating a demand for people with programming skills who can take an idea and turn it into executable code. But the concept of software is much broader and deeper than simply substituting software for gears and levers. It is a way of telling stories and creating agents that can take on a life of their own.

The Internet is a byproduct of this new concept. Rather than depending on scribes or telecommunications companies to carry our messages, now we use software to program around those once necessary intermediaries. Mastering the skills and language of programming gives us the vocabulary for understanding how software is rewriting the world. And then, we can begin the conversation about what I call the *New Literacy*. All that said, it's fun to build things that take on a life of their own that are actually useful and that I can share with others. With software, I don't need to build an entire factory to create products (and services) that I can sell or simply distribute to the world.

With software, I don't need to build an entire factory to create products (and services) that I can sell or simply distribute to the world.

THE AGE OF PROGRAMMING

The idea of storing a series of instructions in a computer's memory, just like any other data, dates back to the 1940s and is often referred to as the Von Neumann architecture. Stored programs had a major advantage over wired logic because you could change the behavior of the computer by loading different instructions. At the time, computers were programmed by wiring them for a given task. Plug boards represented one approach to wiring for simple card processing systems (Figure 1). Stored programs were more flexible, but it was still tedious to carefully assemble the very detailed instructions necessary for each task. One of the first breakthroughs was automatic programming in the 1950s. This avoided the need to employ programmers. Instead, mathematicians and scientists could write in their normal language of formulas:

- PI = 3.14156
- SIN = SIN(PI).

FIGURE 1. An IBM 407 plug board.

Digital Object Identifier 10.1109/MCE.2016.2556741
Date of publication: 10 August 2016

You can read this article online.

Preface

Today there is a lot of interest in teaching people to code as a basic skill. This seems to make a lot of sense in a world that is increasingly being redefined by software. Yet I can't help but think that it's akin to teaching typing rather than teaching people how to write or, more importantly, how to communicate.

The consumer product and services industry (no longer just consumer electronics) is creating a demand for people with programming skills who can take an idea and turn it into executable code.

But the concept of software is much broader and deeper than simply substituting software for gears and levers. It is a way of telling stories and creating agents that can take on a life of their own.

The Internet is a byproduct of this new concept. Rather than depending on scribes or telecommunications companies to carry our messages, now we use software to program around those once necessary intermediaries.

Mastering the skills and language of programming gives us the vocabulary for understanding how software is rewriting the world. And then we can begin the conversation about what I call the New Literacy.

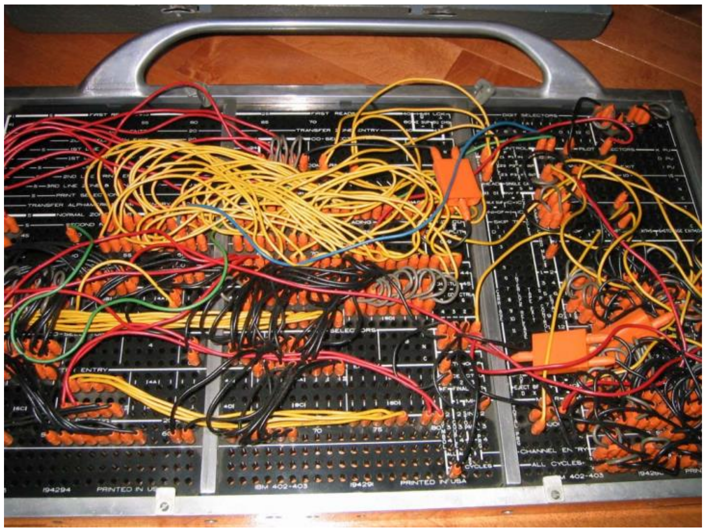
All that said, it's fun to build things that take on a life of their own that are actually useful and which I can share with others. With software, I don't need to build an entire factory to create products (and services) that I can sell to or simply distribute to the world.

The Age of Programming

The idea of storing a series of instructions in a computer's memory just like any other data dates back to the 1940's and is often referred to as the Von Neumann architecture.

Stored programs had a major advantage over wired logic because you could change the behavior of the computer by loading different instructions. At the time computers were programmed by wiring them for a given task. Plug boards represented one approach to wiring for simple card processing systems.

Stored programs were more flexible but it was still tedious to carefully assemble the very detailed instructions necessary for each task.



One of the first breakthrough was automatic programming in the 1950s. This avoided the need to employ programmers. Instead mathematicians and scientists could write in their normal language of formulas:

```
PI = 3.14156
SIN = SIN(PI)
```

What could be simpler and more readable? Stored program computers could do their own programming! The program was the Formula Translator or, more familiarly, the Fortran language. To today's programmers Fortran is just another language and it still has a community of users[i].

Fortran solved the problem of programming. Actually it solved one problem with programming. It was just the start of a process of developing vocabularies that bridge our understanding and the descriptions computing devices can "understand". What does it mean to understand? That's part of the challenge.

By the time I took a class in the history of programming languages in 1967 our professor, Saul Rosen, had published a thick collection of papers on *Programming Systems & Languages* as people experimented with many approaches to programming.

We started out with languages specialized for various purposes such as COBOL for business and Lisp for AI. Today's languages are more general purpose but we continue to develop vocabularies appropriate to classes of applications.

The Art of Programming

I've remained fascinated by programming languages over the years as we better understood the art of programming. I say art because deciding how to tell a story is indeed an art. This is why I chose the name Software Arts for the company Dan Bricklin and I founded in 1979.

An important part of the art of programming is an articulate understanding of technique. One of the big challenges is understanding how to express concepts in terms the machine can understand. Even harder, perhaps, is understanding what the code is really doing as the number of interactions increases.

As I write this I'm trying hard to avoid jargon but there are limits as the very concepts require their own language. We tend to use anthropomorphic language to describe programs but doing so runs the risk of inferring too much understanding on the part of the computer.

Object-oriented program is an important concept but stories and engineering don't work if you lose perspective or, more to the point, perspectives. Objects are useful in organizing programs but it's important that we're working with representations. A number is a number that we may interpret as a temperature in Celsius or Fahrenheit depending on the context.

Ambiguity is essential and fundamental. This is why we don't do serious programming by drawing diagrams. Instead we use words to represent concepts not just operations. We might ask a program to print out a date but the details depend on the context.

Programming can be tedious and error prone and, in the 1970's we longed for a programmer's assistant that could keep track of all the housekeeping tasks. Today's IDEs (Integrated Development Environments) are a realization of that dream.

The growth of languages such as Python and JavaScript with a more relaxed view of objects has demonstrated the value of flexibility. Personally I like the approach taken by languages such as TypeScript which adds type annotation to JavaScript. This allows my assistant, the IDE, to help me. This works well with my personal style of kneading code or, to use more common term, refactoring code, by helping warn me about conflicts as I evolve the code.

Programs aren't just for web pages. They also do creative tasks on our devices (AKA smartphones) to turn them into sensors, cameras or whatever.

We need to do more than continue to improve on the idea of automatic programming. Software is more about ideas than putting together programs.

Programming and Programmers

Today there is a burgeoning demand for programmers who can take business rules and represent them in code. Programmers are also needed to handcraft websites.

It is a very labor-intensive practice and recalls the early days of automatic programming. We still face the challenge of translating the concepts into working code.

A lot has changed. Programs have to continue to evolve over time. It's not so much that they are written as that they are constantly being rewritten and evolve as the requirements evolve.

As with any new technology we start out by substituting programs for older technologies. Today much of programming is the new form of manual labor building web sites or making things smarter. The focus is on creating products and the use cases.

To get more of the benefit of software we need to provide interfaces (APIs or Application Program Interfaces[ii]) that enable users to create their own solutions.

This shifts where value created. We can see this with the Internet which changes the business of networking from providing solutions (voice and reliable deliver) to enable technologies (best efforts packet transport). We're still navigating the transition.

Symbiosis and Empowerment

In the 1950's John McCarthy proposed[iii] the idea of multiple people sharing a single computer – time sharing. He recognized that there was a key difference between writing a program and submitting to run and actively interacting with the computers.

The idea of a man/machine symbiosis continued to evolve at MIT's Project MAC which co-founded by an acoustic psychologist, JCR Licklider. Lick played a key role in ARPA's funding of computer science research which gave us the Arpanet and, eventually, the Internet.

My first job, in 1966, was helping to build an online service that would allow analysts to explore financial data. The system, an SDS-940 was developed at Berkeley and one of its developers, Butler Lampson consulted for us and helped develop a tool, FFL or First Financial Language, intended for use by people who didn't consider themselves programmers. Users would specify rows and columns. The rows might be company names (like GM) and the columns might be an item like (SLS.Y66). And the value would be shown at the intersection.

I learn best by doing and in August 1966 I took home a computer terminal (top of a teletype) and was able to play with what was, in effect, a personal computer. I've been online ever since. I didn't have to worry about the cost so I could use a million-dollar computer as an alternative to a $100 typewriter. In effect I lived a future in which computer was just a mundane tool and capabilities such as email were assumed in my community.

In 1978 my friend Dan Bricklin designed a tool for his own use while in business school. This became VisiCalc, the first electronic spreadsheet. It gave people the ability to, in effect, create programs by working on their own solutions while the computer, in effect, took notes. A spreadsheet formula may look like Fortran but referring a cell as A1 isn't so much giving a name as a way of pointing and saying "that". As some users get more into the programming side they look beyond the surface can start manipulating the formulas as text.

VisiCalc didn't know anything (or at least not much) about finance or any particular application but instead empowered the user by providing language that provided a bridge between the way they thought about their problems and the computer's mindless computational capabilities.

Spreadsheets represent one approach to empowering users, or, more to the point, people to tap into the power of computing.

Developing (or creating) web pages is another way to tap into the capabilities of computers. Simple markup is a form of coding but it isn't very powerful. Over time the sophistication has grown as designers learned to create a dynamic experience very different from preparing a static printed page. As browsers have evolved from simply presenting pages to becoming programming environments more people are learning to program.

More important perhaps than the mechanics of programming is understanding the concepts. The goal of learning to program isn't about getting jobs (though that can be one result) as much as learning the languages of software. Just as learning a foreign language helps one understand one's own language, learning to program a computer requires an articulate understanding of what one is trying to say or do.

One of my mentors, Seymour Papert, explained that the goal of education should be to learn how to learn. A part of that is debugging our understanding. If you do badly on a test it doesn't mean you are stupid. You just need to figure out what you did not understand. This is in sharp contrast to the notion that education is about learning arbitrary facts and rules.

In this context, teaching programming is not so much about a job skill as giving people a vocabulary for sharing their understanding with computers and, in the process, a better way of articulating their understanding so they can share it with other users.

The Internet and Relationships

This brings us to the Internet that happened when we learned to program around intermediaries. But what does that mean? In a prior column[iv] I wrote about how we discovered that in order to interconnect disparate systems we had to exchange packets without relying on the network to assure the meaning was preserved. Instead we had to solve problems and interpret the packets outside of the network in our devices.

As programmers our job is to see what we can do with the capabilities of the transport. If we have high packet loss, we can still do simple messaging. If we have high capacity, we can have video conversations. This is very different from the traditional engineering paradigm of building layers of dependency. As devices become more capable (programmable) we can shift the perspective from depending on layers to seeing them as resources we can choose to repurpose. This ability to reinvent is part of what has driven Moore's law.[v]

The key here is a shift from thinking about the mechanics of the task of messaging to focusing on the relationships between the end points. (http://rmf.vc/IEEERelationships). This allows those with domain expertise to use their expertise without having to negotiate with a third party merely to make a connection between two end points.

It also benefits those without domain expertise because they can choose from a wide variety of experts rather than being limited to a single expert who controls the path between the two end points.

Traditional engineering can now be extended to include elements without concern about physical proximity.

Focusing on relationships represents a sharp departure from traditional electrical engineering and networking which has a separate cable and protocol for each application going back to the days of special phone wires and video wires and, more recently USB, Bluetooth, HDMI, DVI and on and on.

Soft

Taking advantage of these new opportunities is a departure from traditional engineering which builds systems in layers. Instead we focus on each application and view available facilities as resources and opportunities. We solve problems by creating new abstractions and changing relationships (or bindings).

Here's where the focus shifts to the word "soft" in software. At a simple level we can rapidly evolve software solutions as opposed to hardware which needs a long design cycle. It can take years to produce a new chip whereas we can rework software in minutes.

Even subtler is our ability to solve problems by finding, or inventing, new representations. This is not entirely new. The invention of the solar system not only simplified navigation but also gave us the insights that led to Newtonian physics.

There is no algorithm for finding the right representation or architecture because we aren't solving a single problem. This is why it's useful to think about an available facility as a resource or opportunity without being confined to a provider's view of how it's supposed to be used. A strand of copper may be used as a networking medium or it can carry power or we can use it to hang a poster. Or it can be all of these at once depending on factors entirely outside the wire itself.

With hardware we build the one function into the device but with software we can constantly reinvent what something is and what it does. Even better, we can share the same physical resource for entirely different purposes as when we share a single Internet.

This also applies to understanding how systems work. I don't like the term complex adaptive systems. That's backwards because the function, and thus the complexity, depends on the point of view. Instead we need to find the many simple systems or perspectives.

With software we're not substituting bits for electrons but instead have a completely different conceptual framework.

Traditional programming gives us some of the tools we need to start to explore this new landscape of abstractions. But yet we're still in the early stage of substituting software for mechanical systems. The next stage will seem like magic as we effect solutions merely by taking a fresh perspective. OK, not merely but it seems magical in the same way that when connecting to a website you don't think about all that has to go right for you to simply type in a URL and connect. The secret is that a lot has to go right but it's OK for a lot to go wrong as long as you're not particular about exactly what goes right.

It may be annoying if Skype isn't working for a while but in return for accepting some risk we get the ability to do video much of the time. And, unlike hardware systems, software can evolve in place. But it will take time for people to accept some risk in return for the ability to create, and share, their own solutions.

Welcome to our new reality. We solve problems by finding ways to tell stories that our computing devices can understand using a vocabulary appropriate to the task.

[i] https://en.wikipedia.org/wiki/Fortran#Fortran_2015
[ii] http://rmf.vc/IEEEAPIFirst

[iii] http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html and http://ethw.org/Archives:The_Computer_Pioneers:_Switched_Output:_Time-sharing_at_MIT

[iv] http://rmf.vc/IEEERefactoringCE

[v] http://rmf.vc/BeyondLimits