You have advocated a program that offers a Master of Fine Arts in software, similar to programs that offer MFAs in creative writing. You say, "Traditions of computer science and software engineering have tried to turn all aspects of software creation into a pure engineering discipline, when they clearly are not. The MFA in software would begin to repair this error." Tell us about the philosophy behind this proposal.

Writing software should be treated as a creative activity. Just think about it -- the software that's interesting to make is software that hasn't been made before. Most other engineering disciplines are about building things that have been built before. People say, "Well, how come we can't build software the way we build bridges?" The answer is that we've been building bridges for thousands of years, and while we can make incremental improvements to bridges, the fact is that every bridge is like some other bridge that's been built. Someone says, "Oh, let's build a bridge across this river. The river is this wide, it's this deep, it's got to carry this load. It's for cars, pedestrians, or trains, so it will be kind of like this one or that one." They can know the category of bridge they're building, so they can zero in on the design pretty quickly. They don't have to reinvent the wheel.

But in software, even with something such as Java™ 2, Enterprise Edition or the Java implementation (or almost any of the APIs we define), we're rolling out -- if not the first -- at most the seventh or eighth version. We've only been building software for 50 years, and almost every time we're creating something new. If you look at software developers and what they produce, if you look at their source code, the programs they make, and the designs that they end up creating, there is real variability. And some people are really good and others are not so good.

So, because you can program well or poorly, and because most of it is creative (in that we don't really know what we're doing when we start out), my view is that we should train developers the way we train creative people like poets and artists. People may say,"Well, that sounds really nuts." But what do people do when they're being trained, for example, to get a Master of Fine Arts in poetry? They study great works of poetry. Do we do that in our software engineering disciplines? No. You don't look at the source code for great pieces of software. Or look at the architecture of great pieces of software. You don't look at their design. You don't study the lives of great software designers. So, you don't study the literature of the thing you're trying to build.

"The idea behind the MFA in software is that if we want to get good at writing software, we have to practice it, we have to have a critical literature, and we have to have a critical context."

- Richard Gabriel, Distinguished Engineer at Sun Microsystems
Second, MFA programs create a context in which you’re creating while reflecting on it. For example, you write poetry while reading and critiquing other poetry, and while working with mentors who are looking at what you’re doing, helping you think about what you’re doing and working with you on your revisions. Then you go into writers’ workshops and continue the whole process, and write many, many poems under supervision in a critical context, and with mentorship. We don’t do that with software.

I was talking to Mark Strand, who is one of the first poets who mentored me, and he said, more or less, that how good you are depends on how many poems you’ve written in your life. About two and a half years ago, I started writing a poem a day, and I’ve gotten way better since I started doing that. And so, I’ve probably written about 1000 poems in my life so far, almost all of them in the last two years.

Compare that to how many programs someone has written before they’re considered a software developer or engineer. Have they written 1000? No, they’ve probably written 50. So, the idea behind the MFA in software is that if we want to get good at writing software, we have to practice it, we have to have a critical literature, and we have to have a critical context. It looks like we may be able to start a program like that in the next year or so at a major university that I’m not free to name. It’s probably going to be called a Master of Software Arts.

You wrote a poem called "Unnormalized Models" that seems to be about software, with a slightly personal touch thrown in towards the middle. Here is the beginning of the poem:

"Unnormalized Models" is sort of a found poem. A lot of it is taken from an abstract from a talk I attended about certain types of reasoning in artificial intelligence systems. It takes a left turn in the middle in the parentheses through the use of dissociation or association.

Has writing poetry influenced the way you write code?

Writing code certainly feels very similar to writing poetry. When I’m writing poetry, it feels like the center of my thinking is in a particular place, and when I’m writing code the center of my thinking feels in the same kind of place. It’s the same kind of concentration. So, I’m thinking up possibilities, I’m thinking about, well, so how do I reinvent the code, gee, you know, what's the simplest way to do this. I'm thinking about things like simplicity -- how easy is it going to be for someone to look at it later? How well is it fulfilling the overall design that I have in mind? How well does it fit into the architecture? If I were writing a very long poem with many parts, I would be thinking, "Okay, how does this piece fit in with the other pieces? How is it part of the bigger picture?" When coding, I'm doing similar things, and if you look at the source code of extremely talented programmers, there's beauty in it. There's a lot of attention to compression, using the underlying programming language in a way that's easy to penetrate. Yes, writing code and writing poetry are similar.

I have given a keynote called "Triggers and Practice." In writing, a trigger is something that enters your mind and causes you to start a poem, or a story, or whatever you want to write. The trigger could be a scene, a line, an image that comes to you, or just something remembered. The poet Richard Hugo believes that all writing is creative writing in that you don't know what’s going to appear on the page until you write. All writing relies on triggers where you write something and respond to it.
Writing Software: Interacting with End Users

What do you most want developers to be aware of?

I think developers already know but are a little afraid to admit that writing software is a creative activity that requires a lot of interaction with the people who are going to use it. Writing software is a highly iterative, dynamic process requiring user feedback. Again, it's like writing poetry in that you write some of it, and then you respond to it, which triggers more creativity, and you keep going. Then, you show it to someone whose opinion you respect, and you see it differently.

Some of the more agile methodologies like Extreme Programming use this process. Extreme programmers discuss with end users what to build first. Then they spend a couple of weeks building, and show it to the end user. The end user uses it for a while, and then might say, "Oh, now I understand." Customers might not have a good understanding of what needs to be done since they're not programmers. Something they might not even have imagined might be doable.

"I think developers already know but are a little afraid to admit that writing software is a creative activity that requires a lot of interaction with the people who are going to use it."

- Richard Gabriel, Distinguished Engineer at Sun Microsystems

But you can't plan it all out in advance and then implement it. If you could, there would be no need for multiple releases of products.

Computing has existed for only 50 years. We don't really know what the principles are. True, we know the principles of algorithmic programming reasonably well, but we don't know the principles that govern the software that we're trying to build, and we need to be open-minded about how to do it.

I went to Amazon and searched for all the books containing "learn" or "teach yourself," and then "hours" or "days" in the title. Such as, "Teach Yourself X in 21 Days." There are hundreds of them and almost all of them are computer books. "Teach yourself object-oriented programming pronto," and so forth. Writing software is an art, and it takes about ten years to really get good at it. But there's an underlying attitude that programming and software development are easy. We have this attitude that programming is auxiliary, when it's a very central, difficult, human, sociological activity.

Returning Diversity to Computing

You have started something called the Feyerabend Project, (pronounced "FIRE-ah-bend") named after the maverick philosopher, Paul Feyerabend, who argued that for knowledge to progress, people must violate some of their methodological principles. You write, "Computing is based on utility, performance, efficiency, and cleverness. Where are beauty, compassion, humanity, morality, the human spirit and creativity?" So, what do you hope to accomplish with the Feyerabend Project?

It always comes back to creativity for me. Creativity thrives through diversity. So, if you look at creativity at large, lots of poets, painters, playwrights, fiction writers, and so on feed off of each other's creativity, but selectively. It's like biology, with its great diversity, where what really works well in particular environments is selected for survival.

Beginning in the early 1990s, much of the diversity in programming languages, programming methodologies, different computing paradigms, and so forth, disappeared. Why? Probably because object-oriented languages, methodologies, designs, and architectures became so commercially successful.

In the 1990s, the Internet changed the scale of things dramatically. By the middle of the 1990s, Smalltalk had made a lot of inroads in the business world, and Java technology (although no one at Sun thought this way) followed the Smalltalk path, and then displaced it. I mean, Smalltalk was a server-side technology --
though the Smalltalk companies didn't like that -- and Sun initially thought of Java technology as client-side. Now Java software has its best successes on the server-side. And the customers are the same. The Java language rode on the coattails of C++ also, and didn't really replace C++, but certainly has taken a big chunk of its mental real estate.

And now, since we've been doing objects for about ten years with pretty good success, we don't look at functional languages, or logic-based languages, or declarative languages -- we don't look at lots of different things that we used to. I worked with the Jini™ network technology project early on. The world today contains many devices, and much of the software comes and goes, many devices come and go, and people upgrade the software. And, sometimes, they downgrade when newer apps on systems don't work well.

And the chaos will become more pronounced, more diverse. It'll begin to sound more like a biological system than a mathematical system. There will be fewer monolithic pieces of software. Most of any "application" will be mostly on one machine, tightly interwoven, but there will be parts that are kind of loose. It'll be very dynamic, with a lot of failures. Deutsch's seven fallacies will dominate. We need to prepare for this.

If you look at the narrative of computer science research, it's still focusing on object orientation. The Java language tried to pull together the best of the pure object model and static languages, so it's a little like Smalltalk and a little like Pascal and a little like Lisp even. I think we need to do more; we need to let other language ideas flourish, look at richer interactions, not just simple ones. We need more narratives at work to address the craziness we'll be facing.

Wasn't object-oriented software supposed to take the labor out of programming?

Yes, that was the promise, but it didn't exactly pan out. Unless it's been designed for reuse, you're going to have a really hard time taking software off the shelf. Research shows that, in order for software to be designed for reuse, it usually needs to be deployed in three different contexts.

Inheritance has made things more confusing, too. With inheritance, you write a class, and you write its methods, using traditional terminology; and then if you make a sub-class, you inherit the methods and maybe you augment them in various ways. But deep inheritance trees could cause confusion because of the amount of inheritance that you have to track and understand. So, you actually didn't get much benefit. Basically, because there's so much work involved in making things reusable, it doesn't usually get done. But the Java language is pretty successful, and folks write lots of apps in it.

With the Feyerabend Project, we're aiming at development efficiency, and thinking in terms of computers helping us grow the code rather than people typing it in, or taking it off the shelf. We have to open our minds to what's really happening, not what we hope is happening.

See Also

Richard Gabriel's Web Site
(http://www.dreamsongs.com/)