

Introduction

There has never been a silver bullet for solving problems. At the very least, one has to express the problem in some form. The fact that the translation of high-level statement of a problem can be automated does not alleviate the need to state the problem. For instance, the automation of solving certain algebraic equations does not suppress the need to map a real-world problem to a set of equations.

Our focus in this article is at application software. The majority of software products by far are applications. Even a small progress in making better applications is a giant step. In what follows, we will attempt to identify the factors that directly influence quality of software. Briefly, we need software engineering curriculums with focus on specialization. This in turn requires an abstraction for modeling in software engineering. ***The impeding factor is the lack of a properly designed, monotonically extensible language.***

Marketing strategies of larger companies have created strong whirlpools of confusion sinking activities not profitable to them. Nevertheless, they all are saying exactly one thing regardless of their choice of XML, JAVA, C# or anything else for that matter. **The design of Z++ is based on research for the future of software engineering, rather than being influenced by short-term marketing interests.**

Recognition of Expertise

We do not expect an electric engineer to build a bridge arguing that the mathematics required for all areas of engineering is essentially the same. Nevertheless, a software engineer is expected to write a parser one day, and the next day a financial product. An engineer's work is viewed anywhere from being same as writing the code to print "Hello World", to at most equivalent to performing a handful of arithmetic operations.

At dawn, there is still not enough light to see well. Like the dawn of engineering and medicine, a programmer had no area of specialization. **The pioneer programmers were domain experts who taught themselves how to program.** However, it is now possible to produce engineers with specialized expertise in many areas. Anymore, an engineer specializing in insurance should not be made to engage in a health related project *based on her programming skills in the language selected for the project.*

Having a domain expert guide a team of software engineers is only a good temporary solution to phase out the current shortage. Without domain experts that themselves are also software engineers the confusion and chaos will be compounded beyond "Web Services", et al. There are, however, several obstacles in producing expert engineers.

1. The underlying environments change too frequently and by much.
2. Lack of software engineering curriculum.
3. Lack of an abstract, extensible language.

Let us go over each of these factors in more detail.

Environment

In order to produce qualified expert software engineers, we must break the link between *application* software and the environment on which applications will run. The tool to break this link is an abstraction to serve a purpose similar to mathematics in other areas of engineering.

The vendor of the underlying environment must be solely responsible for repairing it without requiring certification to cope with its problems. Certification as a proof of familiarity with some vendor's *application* products is too narrow, and if the product is an operating system, then it must be stable enough so one can learn about it in college.

Certification hurts rather than helping the training of qualified software engineers with expertise. Many may think that being certified is equivalent to good software engineer. In reality all it means is that, the certified person is aware of certain bugs and other complications, and then for a short period before the next release of the product. Nevertheless, those with certification will fill the positions and continue to plague the end users with ever-increasing bugs. *Perhaps users should require proper engineering college degrees from a vendor, rather than the vendor requiring certification from his partners.*

Curriculum

Separating Computer Science and Software Engineering curriculums would help understand the engineering component with less interdependence. Whether or not a scientist should be an excellent programmer is a decision the scientist herself will have to make. The greatest physicists have also known a great deal of mathematics.

There is no similarity between a set of linear equations and the physical world. Yet, a domain problem is manipulated so one can map it to those equations and find some answers. **Thus, an engineer has a target and has learned how to manipulate the domain so it can be mapped to that target.** For software engineers the target is a programming language. While the process of modeling will remain under research indefinitely, computer scientists have already contributed enough towards a decent engineering curriculum. **The only missing component seems to be the target.**

There should be a core software engineering followed by various areas of specialization. Departments should offer specialized areas as needed by the industry. Over the years universities will find out how to streamline their offerings. **But that cannot even start until software engineering is recognized as a new discipline in its own right.**

A software engineering curriculum should have something to offer for those who enjoy the challenges of programming. Some of them will be the elite hackers of future. The C++ programming language is sufficient to deal with any operating system, device driver or any system program. Linux seems to be a valuable operating system growing

monotonically. That is, **you do not need to discard your previous knowledge with next release**, nor you need to renew your *certification as a proof that you are keeping up with an ever-growing bug list*.

Language

A programming language to a software engineer is effectively what mathematics is to an engineer. A software engineer needs some form of abstraction to use as part of his thinking while transforming a domain problem, just as an engineer uses equations as part of her thinking. To that end, the target programming language must be sufficiently expressive in dealing with abstractions.

Over the years engineers build the necessary intuition for manipulating problems in a domain and mapping them to solutions. This intuition will be chaotic unless an engineer continues to think in same one language.

Many engineers list half a dozen languages in their resume. Even without environment specific complications, this is chaos. An engineer must understand abstractions and possess the skills to express them in a programming language. **The mapping of abstractions takes years of experience and is not immediately transferable to a newly learned language.**

A programming language must possess certain desirable properties.

1. The language must provide the entire means to express a solution in its abstract form without having to rely on environment specific system calls. This is the closure property.
2. The language must be monotonically extensible.
3. Extensions must not create unusual irrelevant details to remember for avoiding inconspicuous bugs.
4. Language must support a high degree of semantic automation.

Extending a programming language requires more caution than extending mathematics. Many algebraic structures were simply discarded without disastrous consequences. The characteristics mentioned here are discussed in detail in article on Abstraction and Virtual Machine.

Closing Notes

What kind of problem is solved, by extending COBOL to support object-oriented paradigm? Even more amazing is a researcher extending PROLOG for the same reason. I suppose a term interacts with a statement by sending an instantiation message to it. We have already entered the era of object-oriented paradigm for the construction of large-scale distributed applications. However, extending any language just because someone could do that for C is not the solution. Nor is an object-oriented interface language making several unrelated languages interoperable. Confusing diversity with chaos

resulting from marketing strategies of larger companies is a major reason for most of the falls of smaller businesses.

Major vendors are tangled up in their past and biased towards their existing products. Consequently, their solutions for a universal language will lack the abstraction and generality to support monotonic extensibility. Of necessity, their solutions will have too many hooks to their past. **Application builders will have to take the liberating step to the freedom of virtual world, where their investments *will be available even for the future better operating systems.***

The Z++ abstract language is larger than C++, but not any harder to learn. Various components of the language will grow in a manner orthogonal to other components. For instance, extensions to distributed computing aspect will not affect query processing. **Orthogonal extension is a necessary condition for monotonic growth**, without which one has to keep track of the versions of some software running on certain versions of the virtual machine.

Dr. Z. August 1, 2002.