Reuse in software engineering

Rajendra K. Bera

One of the features of software engineering that strikes anybody who has migrated to it from some other branch of knowledge is its continuing state of uncertainty – not knowing where it stands and how it wishes to reach its goals. Considering that software engineering is at least 45 years old, this state-of-affairs is perplexing. The uncertainties manifest in many ways, one of them is the current emphasis on reuse because it seems that every software engineer has a strong tendency to reinvent the wheel time and again.

Barring software engineering, the author is not aware of any other branch of knowledge where reuse is such an issue. Indeed, our ability to invent is so meagre that our natural tendency is to reuse and reuse and reuse ... whatever little we know. It is an effort-saving means of extending our intellectual reach. We do it instinctively.

Thus in physics, we attempt to solve a problem by first invoking the laws of nature as we know them today. In mathematics, we first try to figure out the known datatypes, theorems and lemmas that we can use before proceeding further. In medicine we go through a standard diagnostic process before commencing treatment, in our culinary tastes we all love grandma's recipes, in social interactions we follow customs, and so on.

So it comes as a surprise that enforcing reuse in a software industry seems to require champions from senior management. Consequently, quality software is a rarity, not a rule. All this indicates that software engineering has yet to mature to a state where it can comfortably communicate with its scientific, engineering, and applications domains.

The world is mathematical

Mathematicians (the oldest practitioners of virtual reality) have created a whole world of abstractions and powerful means of dealing with them without miring themselves in alphabet soups. And physicists have never ceased to wonder why nature is so fantastically mathematical. The conceptual relationship between the abstractions created by the mathematicians (without their ever intending to explain nature) and the natural world is very deep indeed. And the physicists have quite eagerly exploited this by turning large parts of pure mathematics into applied mathematics!

Mathematicians and physicists do not introduce buzz words, nor do the practioners of other mature branches of knowledge (imagine people practising law or medicine with ever-changing buzzwords). When they introduce a new word or phrase or bring in new semantics, they do it after careful thought and consultation to prevent not only present, but also, future confusion. Such professionalism is lacking in software engineering. All branches of knowledge (not the arts) which aspire to reach greater levels of maturity eventually turn (or will turn) to mathematics (hence the current emphasis on mathematical modelling) to express themselves, or, in the least, adopt the rigorous discipline which mathematicians follow.

However, concepts and abstraction levels taken for granted by mathematicians and physicists appear novel (often intriguing) to software engineers. Mathematicians and physicists have taken structured knowledge to heights unmatched by any other branch of knowledge, and this knowledge is freely available. So, it is indeed surprising that software engineers have made very little effort to benefit from them. Instead, they have gone on a spree to rediscover long known concepts, have a particular fondness for alphabet soups, and constantly produce terminologies, which even their inventors do not fully understand or which contradict accepted usage of the terms.

Software is interesting because it encodes the know-how (or algorithms) of doing things that interest us. And it is done in mathematical terms! Yet, most programmers are oblivious of the fact that programming is a mathematical activity, and worse, many are afraid of viewing it in that light.

Symbolic systems are the key

The most interesting part of programming is, first, the selection of algorithms, then, their literal translation in a given programming language. A programming language can do this only if it follows the rules and semantics of mathematics. Unfortunately, all currently used programming languages have self-inflicted constraints – none of them match the symbolic and semantic richness that mathematicians take for granted the world over as a common heritage. It is said that the theory of relativity would not have progressed without the notations of tensor mathematics nor quantum mechanics without the notations of complex variable theory. More closer to home, we would have been at sea in our daily lives if we were to keep accounts in Roman numerals and not in Arabic. We encode our intellectual activities in symbolic systems, and the devising of these symbolic systems is one of the outstanding achievements of the human mind.

Computer scientists are, of course, well aware of this fact, indeed more, that symbols representing instructions are no different in kind from symbols representing numbers (recall the notion of Gödel-numbering). Yet they have failed to produce a powerful enough programming language and push compiler technology to the levels needed to support it. The technology for doing it has been available since the introduction of the first digital computer. Today isolated packages exist that do algebra, for example, Macsyma, but constructs that do symbol manipulation should have really been a part and parcel of a programming language by now. In its absence, even mathematicians have to go through mental and coding gymnastics to solve a problem on a computer. How much nicer it would have been if they could have used the symbology that is universally understood by their colleagues. Their symbolic system is so powerful that they move seamlessly from problem statement to problem solution, unlike a programming language which handles only implementation but not requirements, analysis, or design aspects of software development. Without appropriate symbology, theorem proving would have been impossible, and likewise without appropriate symbology, program proving will remain a dream. What software engineering desperately needs is a programming language that will naturally induce mathematical modelers to do their own programming using their own symbolic systems. That is the real key to software productivity.

Domain experts are crucial

To succeed in any complex software project, the first and foremost task should be the identification of the domain experts, and then the superstar programmers. Success rates will be high because they will seek to reduce the complexity level of the project by being knowledgeable, experienced, and above all, by working at conceptual levels. By training and experience, they will bring into play a wealth of organizing principles, and therefore a very high level of reusability in their work. They will be a much smaller team thereby tremendously reducing the interpersonnel overheads that otherwise occur with larger teams.

Unfortunately, the absence of domain experts and superstar programmers is all too common in software projects and hence the perennial problems of missed schedules, unpredictable quality, and uncontrollable costs.

Why reuse is lacking in software?

As already noted, in most human activities and spheres of knowledge reuse is practised routinely in the form of tradition, custom, concepts, theorems, lemmas, recipes, and so on. Even home grown software engineers in the scientific community just go about their business of creating math and other libraries without using the word *reuse*. That is because traditionally libraries have been an epitome of reusability in propagating knowledge since the past few thousand years.

On the other hand, people who professionally call themselves as software engineers have developed a style of functioning which is often viewed by others as unscientific and unstructured; is sometimes illogical; often makes them wary of mathematics; and often makes them reluctant to consult domain experts.

These are among the primary reasons why reusability in software engineering is so poor. Software is being developed without a firm tradition of professional excellence, and rarely by the domain experts themselves.

But what is truly needed to set things right is that domain experts come down from their ivory towers, learn the art of programming, and develop software in their domains of knowledge. The practice of reuse, above all, requires people with experience and great sensitivity to pattern matching at abstract (conceptual) levels, people who can spot often-needed patterns. It requires insight, not common sense. The most elegant and optimal form of reuse is at the level of basic concepts. It therefore comes from the mind, not from programming fashions. However, the dominant mindset in software engineering, at present, is to focus on code reuse, which is really the lowest and least interesting form of reuse because they are tightly bound to a programming language and implementation details. But even in code reuse there are disincentives! Programmer productivity is too often measured by the number of lines of code he/she produces thereby discouraging code reuse, specially if reuse means transferring from direct coding activities to thinking activities.

Reuse is a means of economizing on intellectual effort, time, expense, life-cycle costs, code size, and test requirements, as well as to improve on clarity, and quality. Serious reusability requires deep knowledge of the domain and across domains. Consequently, all complex projects (even small projects can be complex, for example, a safety-critical code segment) require domain experts for consultation and, if possible, for software development (analysis, design, coding, testing).

The scientific community has generally followed this principle. It develops its software through its own scientists. The so-called software crisis as a phenomenon is neither faced by them nor created by them. The codes they developed decades ago (when programming languages were primitive, and without software tools) for weather forecasting, nuclear weapons design, engineering structural design, drug design using quantum chemistry, math libraries, graphics libraries, etc. are still used with great confidence. Even the WEB was designed by a physicist!

While many scientists do their own coding, business managers rarely do so (they would have if they had good training in maths). The engineers usually fall inbetween. Sometimes they do, sometimes they don't, depending on how much they liked their math courses. The software crisis exists because business and, to a lesser extent, engineering software is being developed by people who are not domain experts.

Conclusions

Until computer scientists provide a powerful enough programming language that will match the richness of the symbolic systems used by mathematicians, and till domain experts learn the art of computer programming and actively participate in software development, software engineering will remain in a state of limbo.

At the same time, programmer training also needs an overhaul. The overwhelming emphasis has been on teaching programming language syntax, creating and memorizing jargon (and alphabet soups), learning to use (unstable) tools, and, generally, the mechanics of doing fairly mundane things. Not much, and sometimes no, attention is paid to teaching about algorithms, approaches to problem solving, etc. They learn nothing about what makes a programming language great, how concepts finally get translated into elegant code, how to frame intelligent test cases. They learn very little of memory and cache management, limitations of finite state machines, and the performance and accuracy sensitivities of algorithms to them. And they learn practically nothing of great software designs and designers, software architecture and its relationship to maths, nor are they exposed to an anthology of great codes written by the masters. In short, the software engineering community has failed to set up an environment whereby people can be nurtured into a professional culture through osmosis, mentoring, critical appreciation, discipline, etc. Indeed, some might say, such a culture is yet to take roots.

Rajendra K. Bera is in the Systems Group at IBM Global Services India Pvt. Ltd, Bangalore 560 017, India.