Benefits and Applications of Cross-Pollination

Tomoo Matsubara, Matsubara Consulting
Christof Ebert, Alcatel

Society in the Information Age heavily depends on reliable computing and communication technology. However, due to computer and software technology’s rapid evolution, the discipline itself cannot keep pace. Engineering foundations gradually build up, and software engineering increasingly provides a framework that educates and communicates independently of the specific application domains.

The speed of this evolution is both a challenge and a risk. It’s wonderful to see so many new and never imagined applications, but three decades after the discipline’s emergence, our society heavily depends on software technology. The public and the discipline’s practitioners dramatically underestimated a software product’s lifetime. For example, no one budgeted for Y2K corrections, and the complete cost—probably exceeding tens of billions of dollars—is yet to be calculated. We rely on software without knowing its impacts. Occasionally, we stumble across a blue screen when we do not expect it and recognize loopholes that weaken our security. If we drive a car or fly in a plane, we accept that there are certain risks that have been calculated before, allowing us to consciously decide. Software is embedded and hidden from daily life, and in most cases, nobody completely assessed its impacts.

Learning from Others

In the past, different disciplines emerged by learning from and influencing each other. Mechanics influenced medicine; statistics evolved quite rapidly in conjunction with social sciences. Although the first example is unidirectional learning, the second shows two disciplines that evolved because they influenced each other. In biology, which also shared parts of the learning curve with other disciplines including physics or chemistry, this kind of cross-influencing is called cross-pollination.

Webster’s Dictionary defines cross-pollination as “to transfer pollen from the anther of one flower to the stigma of another.” Engineering disciplines that exchange results and thus build on a common foundation of knowledge cross-pollinate each other. So the obvious question is, what should software engineering learn and pick up from other disciplines to facilitate and stimulate more cross-pollination?

Special Issue

The objective here is to share experiences...
and examples from disciplines that are cross-pollinating software engineering. We can identify three levels of cross-pollination in the theme articles:

- Influencing software engineering from other engineering disciplines, thus building up a solid stock of sound principles and methods that also work in other domains and disciplines. An example is experiments where other engineering and social disciplines heavily influenced software engineering.
- Guiding the evolution of software engineering as a discipline, such as with professional standards, education patterns, and the discussion of scientific methodologies. This happens in disciplines with converging needs that mature on a similar path, such as electrical engineering.
- Sharing common principles with another discipline so both emerge in parallel, thus exchanging results and growing. Examples are team management or architectural patterns.

We were surprised at the amount of feedback from engineers, scientists, and other professionals that we received after the first call for papers. Although we originally tried to narrow the call down to engineering disciplines, it soon became clear that there is a lot of overlap out in the trenches. Architecture design and business administration are just two areas from which we did not expect to receive serious submissions. Of course, other engineering disciplines were most frequently used in the submissions as the anther. Receiving articles from disciplines such as industrial or civil engineering also meant we had to look for reviewers beyond our focused software engineering perspective. This brought interesting contacts, such as architecture-software suppliers.

In a 1995 Computer roundtable discussion, a panelist questioned the necessity of software engineering as a discipline on its own, given that other disciplines influence most parts of computer science today. He pointed out that software engineering could become part of the different application domains' related disciplines. This special issue's viewpoint is less radical, instead questioning how to build on available experience and learn from other disciplines to qualitatively improve software.

We selected five articles out of the many good submissions to this special issue that represent the broad range of potential cross-pollination areas.

Shari Lawrence Pfleeger and Winifred Menezes provide a good overview on technology transfer from the business perspective in “Marketing Technology to Software Practitioners.” Marketing provides the underlying theory and supportive tools to understand the needs of technology adopters to successfully facilitate technology transfer.

Software engineering learned from design but should also follow the evolution without stagnation. Terri Maginnis points this out in “Engineers Don’t Build.” We too often mix the roles that are separated in a construction project. Software engineers still specify, design, construct, and even maintain the interior design. In construction projects, this approach was given up nearly a century ago. A clear split of roles with the individualized education help not only make responsibilities clear, but also better structure our software educational system.

The social sciences can teach us a lot about how implicit values and beliefs are held and propagated within our community. Unlike the more frequently quoted influences of social sciences on systems design, Helen Sharp, Hugh Robinson, and Mark Woodman, in “Software Engineering: Community and Culture,” detect that our software engineering culture is already based on shared beliefs, but we don’t yet try to adequately understand and communicate them.

In “Construction Industry Case Study: Identifying Software Productivity Improvement Approaches and Risks,” Peter Hantos and Mario Gisbert investigate what other engineering disciplines can add to software engineering to boost improvements in the way software is developed. The key to their success was a training video typically used in the construction industry. They translated its messages into practical software process improvements, because team work and dissemination of changes are not domain specific. Many project management aspects of the construction business mapped quite nicely to practical software project management. The observed advantage of a tangible subject such as building a house, however, is that domain-independent basic principles can be communicated easier.

In “A Model for Software Practices from
the Accounting Profession,” Bryan Kocher digs into a topic that has chilled our community for several years, 1 January 2000. Obviously, the article and this introduction will already have been challenged by reality by the time you read it. The analogy nevertheless holds valid thereafter, as similar incidents will surely follow—perhaps on a different scale. He compares 24 October 1929 (the US stock market crash) and the lessons the financial industry learned from this disaster with what we should learn from Saturday, 1 January 2000. With some key rules deduced from the many experiences gained out of the Y2K issue, we should be able to avoid similar wasted effort and resources in the future.

Where We Go from Here

Albert Einstein once stated that “the significant problems we face cannot be solved at the same level of thinking we were at when we created them.” Indeed, this is a lesson we can study because it questions some problems we’ve asked before. We must go beyond traditional boundaries and find answers in other disciplines. Even better, chances are good that the result will benefit both sides.

Technology transfer is the key to linking many engineering areas, across countries and disciplines. It means, of course, cross-pollinating each other. The adaptation of configuration and change management techniques from the aircraft industry is one example from the past. Today, it might be clear and stable interfaces between reusable software components. Requirements engineering might also learn from hermeneutics, which is concerned with broad interpretations of text. Software design might learn from architecture. Cross-pollination can work in the opposite direction as well. Cross-functional teams in many engineering areas can learn from each other and specifically from people-ware in software construction. Practical process improvement in other fields increasingly looks to experiences in software process improvement.

To facilitate cross-pollination, we must first be willing to learn from each other and sufficiently phrase problems to avoid technical misunderstandings. Finally, the exchange of results and experiences must work in both directions—after all, maturity growth and learning are continuous processes.

References