Debate: votes (some hanging chads)

<table>
<thead>
<tr>
<th></th>
<th>Pro</th>
<th>Con</th>
<th>Both/Abstain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crisis</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Empirical</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Key open questions: from you

- When manufacturing physical products, industrial processes require constant monitoring and evaluation for quality control. A challenge to doing this in a large-scale software development process project because of expense, difficulty, or lack of availability. Are there methods to instrument, characterize, and audit the processes and tools used to provide suggestive, relevant, and timely feedback? Would freely distributing quality tools like pre-fast/pre-fix (a la MS), improve quality control? Can a framework for integrating suites of quality control tools be built/distributed and mandated for acceptance?

- In the physical world, there are limitations of physical and materials that limit how large a structure can be. Is there a fundamental limit to the size of a software project (code or people)? Consider the "skyscrapers" of the software world — the 50-M loc required for Windows Vista and the 30-M loc found in the Red hat 7.1 linux distro (2001). Though built with substantially different processes (open source vs proprietary), they both reached the same order of magnitude in size. Why and how was this possible? Do they achieve those results because of some common techniques/processes, or are there fundamentally different issues that allow software to reach that scale?

- Thus far our discussion and papers on software engineering has cited individual, large, monolithic projects as examples of software engineering marvels/disasters. This seems analogous to physical engineering marvels such as massive bridges, skyscrapers, and massive tunnels. Is software engineering research efforts focused on massive but distributed code bases such as extremely heterogeneous distributed situations (I'm thinking think of swarms of cell phones interacting with rids, motex, and other devices through pico network). Similarly, are there software engineering research efforts focused on massive Google-like cloud computing infrastructures?

- Education, accreditations and certifications are suggested as mechanisms to improve the practices of the software engineering discipline. This is true for many professions such as civil and mechanical engineers (PE), medical doctors (MD), lawyers (JDs), and accountants (CPAs).

- Interestingly, computer hardware engineering, an example cited during debate as an industry with robust engineering practices, is an industry that less software engineering does not require practitioners to have special certifications. I posit (from experience working in that industry) that this is because the financial and liability risks involved with failure cause them to use conservative design techniques.

- Is employing legal and liability incentives a mechanism that will force software quality to improve? Would less innovating but conservative and robust designs be capable of surviving the marketplace?

- What are the commonalities of successful software engineering methodologies? Of these, which ones are actually productive traits vs. common bugs?

- When working toward process improvement, what metrics should a team actually focus on? Is there a higher-level theory of how to improve a process?

- What is the proper place of analytical approaches toward software verification? How can analytical approaches best augment or improve test-based verification? Is it possible to fuse the two approaches in a "most effective" manner?

- New SWEng methodologies often practice "test-driven development." Is this leading us down a wrong path? Would "verification-driven development" make any sense? Does test-driven development improperly deemphasize verification?

- Most software engineers by trade were computer science majors in undergraduate years. Software engineering as a major (undergrad or masters) is often looked down upon by those with CS degrees. Is this helping or harming the quality of graduates sent into industry? Does a CS degree best qualify someone to develop software? Do practices which go into CS curricula actually hinder the development of software engineering skills in students?
• What makes software so hard and costly?
• Is software / software engineering in a crisis?
• Is it possible for software engineering to become a standardized engineering discipline?
• Can software eventually achieve maturity? What are the maturity criteria?
• How can we distinguish good software engineering research from bad software engineering research? Are there effective evaluation methods?

• One thing that both teams touched on was the large body of old code that's still in service. How does the development of new standards and approaches relate to modifying and extending that code? What is there between the extremes of "Make it conform to new standards or rewrite it" and "ignore it"?
• The kinds of software being developed do change quickly, and sometimes in qualitative ways (such as the relatively new need for efficient use of multi-core processors). Are there higher-level software engineering research results that can apply across such changes (for example, mechanisms for eliciting design requirements)? In what categories "could" such results exist - management? Testing?
• For areas where high-level approaches aren't sufficient, such as SWEBOK, what's a good model for rapidly modifying those approaches? How does the need for rapid updating interact with the pace of the industry in adopting new approaches?
• Is it really accurate to say that software development is necessarily more of a craft than an engineering discipline? If so, does it follow that it doesn't make sense to try to impose further order on the process?

• Development of effective tools to help Software Engineers cope with changing requirements and better analyze, predict, and control different properties of software systems
• Development of models for defining evolution of Software Engineering to cope with changing requirements; i.e. Empirical Software Engineering Research, etc
• Software engineering management methods to help better predict and control quality, schedule, cost, cycle time and productivity
• Theoretical efforts in defining software engineering processes. For example, empirical software engineering research, etc.

• One interesting problem in software engineering today is that of processes used to produce software. Specifically, I mean the question of how to arrange people to produce good software. Recently, systems like agile development and its descendants have been pushed as a good way to quickly produce working software. However, such development methodologies have also been criticized for their lack of strong planning and problems with applying them to the creation of larger software systems, especially in comparison to older, heavier weight methods like the spiral model. What is the best model is varies from project to project, but the creation of new models for software development that can strike a balance between extremes is an open question.
• Another interesting question is that of methods for guaranteeing the correctness of software. The De Millo et al. paper (Social Processes and Proofs of Theorems and Programs) argues that formal mechanisms for this are not worth pursuing. However, in the thirty years since the paper was written, vast improvements have been made in the field. All sorts of static and dynamic analyses have been introduced and studied. Many of these have resulted in tools that are used to catch bugs and check the validity of existing programs. However, many of these focus on simple, local properties. Although there exists some work in guaranteeing program-wide properties, this is a place where there is much opportunity for SE research.
• Another open question in software engineering is how to better apply empirical techniques to computer science research. While slightly off the Tichy et al. paper discusses the paucity of experimental evaluations in the software world. Such evaluations are at the heart of any hard science. It would be useful to be able to better apply the lessons of science to computer science. However, to a large extent, the culture of computer science is not set up to produce such evaluations. How and when do we integrate them into computer science research is an open research question.
Having software verification as the research area, one important question I always ask is: Shall we write a program as general as possible in order to cut the development cost? What causes the development cost in the first place? What are the factors that programmers do not fully understand? Do we need more formal theory? How easy is it to develop our programs? What is the best way to test a software? Should we concentrate on testing common cases, or boundary and rare cases? How to popularize ideas such as graphical programming and automatic programming? How much are formal methods used in evaluating software engineering research? How effective are they when used? How well do these formal methods reflect the real world?

Another concern is that processes like CMM looks at the software from a very high point of view. However, decisions about programs may require a deep look at the development strategy of a company, as Microsoft does. This would be a good way to constraint programmers about strict rules of the program’s development. For example, how to assert that a program is ready to ship. Does following CMM standards like coding conventions (which is actually forces readability and comprehensibility)?

Following the discussion about empirical evaluation, there is clearly a need to extend the evaluation to other levels to allow for example project managers to see in what maturity or how current status of a development project is.

Another concern is that processes like CMM looks at the software from a very high point of view. However, decisions about programs may require a deep look at the development strategy of a company, as Microsoft does. This would be a good way to constraint programmers about strict rules of the program’s development. For example, how to assert that a program is ready to ship. Does following CMM standards like coding conventions (which is actually forces readability and comprehensibility)?

Following the discussion about empirical evaluation, there is clearly a need to extend the evaluation to other levels to allow for example project managers to see in what maturity or how current status of a development project is.

Following the discussion about empirical evaluation, there is clearly a need to extend the evaluation to other levels to allow for example project managers to see in what maturity or how current status of a development project is.
• How much of the success of building and shipping products is attributable to tools & process rather than to personality, group dynamics, or organizational structure? Do tools & process matter?
• If we were to design a new tool or process, what tool, and what part of the process will have the most impact? What area is currently most underserved in the development cycle? How and is it possible to measure these things?
• What social factors - personality, experience, group dynamics etc are most important in what sort of projects? How can you measure these things?
• What is the right mix of dev/test/PM/architect skills for a given project?
• Is there a time, and if so when is the best time in a software lifecycle to scrap the current code base & start fresh? That is to say, if only I knew then what I know now! When is it too late?
• Is there an indicator test at the individual or group or corporate level (GRE, past experience ) that predicts successful project completion, bugs, maintenance, budget, etc?
• Law firms have a useful informal label for the roles attorneys assume as they progress in their careers- Grinders, Finders & Minders. Is there a similar progression for software engineers, is it useful?

• Can software engineering help exploit multi-core architectures?
• Can better techniques be developed for time/budget estimation? This currently seems to be a guessing game we are bad at.
• How can we measure good programmers, if a body of knowledge cannot be formulated? Determining who is skilled at a craft is generally done by reputation. The scale of the software industry makes this difficult.

Software Design
• What is the first amazon.com hit found, in books, by searching for "software design"?
• Is this book in the top 10, 100, 1000, 10,000, 100,000 or 500,000 on the Amazon sales rank list (as of 1/25/08)?

Desirable characteristics
• Correctness
• Feasibility
• Extensibility
• Robustness
• Reliability
• Safety
• Fault-tolerance
• Security
• Maintainability
• Understandability
• Compatibility
• Modularity
• Reuse
• Testability
Generic but important reminders

- Risk reduction is often a major influence on design decisions
- Conceptual integrity has core value
- Just as Perlis said, “One man's constant is another man's variable,” – “One person’s implementation is another person’s design.”

Perlis: “If a listener nods his head when you're explaining your program, wake him up."

- So goes design
- Making good design decisions requires clarity of thought, which in turn benefits from clarity of notation
- This is not a plea for UML or any other specific design notation (and there are tons)
- Rather, it’s what it says it is – a plea for clarity
  – Let’s look back at some of the design images from earlier
- *nolo bene*: there are surely times where sketchy ideas are of value – and it is surely important to distinguish if and when this is important
Clarity

• What’s a box?
• What’s an arrow?
• What’s a module?
• What’s a layer? Or a level? Or a tier?
• What does it mean to perform an external operation (such as “turn off furnace” or “launch missile”)?
• What do correctness, feasibility, extensibility, robustness … and so on mean in a given context?
• …
• … and more, more, more!