Extending Software Engineering Research
Outside the Digital Box

Barry W. Boehm
University of Southern California
Center for Systems and Sw Engr.
941 W. 37th Place, SAL 328
Los Angeles, CA 90089-0781
+1-213-740-8163
boehm@usc.edu

ABSTRACT
Since software is developed to run on computers, there is a
tendency to focus computer science and software engineering on
how best to get software to run on computers. But, engineering is
different from science: the Webster definition of “engineering” is
“the application of science and mathematics by which the
properties of matter and the sources of energy in nature are made
useful to people.” Thus, it would follow that the responsibility of
software engineering and its research would include the utility to
people of the software and the software-reliant artifacts they use,eyond thinking within purely digital boxes. This position paper
addresses two perspectives on the future of software engineering
when viewed in this broader context.

Categories and Subject Descriptors
D.2 SOFTWARE ENGINEERING

General Terms
Measurement, Performance, Design, Economics, Reliability,
Experimentation, Human Factors, Languages, Theory,
Verification.

Keywords
Future, Research, Software Engineering, Digital Box

1. INFRASTRUCTURE, APPLICATIONS
   AND USER PROGRAMMING
Figure 1 shows a 1995 attempt to characterize future software
engineering practice and associated software cost estimation
needs as a step in scoping the COCOMO II software cost model
[1]. Subsequent research [2] indicated that the trends and
quantities were not too far off base, and were continuing to
proportionally increase.

With respect to the future of software engineering research, the
main concern with respect to the figure and trends is that all the
sectors are critical to how well software serves human needs, but
that most software engineering research is focused on the
Infrastructure sector. Research on Infrastructure software is
important, but is largely unrepresentative of Applications
software is several key ways:

- Its users are largely programmers, and its research tends
to produce programmer-friendly capabilities, whereas
Applications software largely needs to be
nonprogrammer-friendly.

- Much of infrastructure software can be developed using
an open-source approach. This is because its
developers are its users, unlike for applications
software. This has been a major boon to empirical
software engineering research, as it has created a large
corpus of easily-accessible software artifacts and
histories for empirical analysis. However, the degree to
which the resulting research is representative of
applications software is open to question. For example,
open source software deals largely with context-free,
dimensionless data, whereas many applications
interface problems come from data dimension
mismatches and domain assumption mismatches.

- Infrastructure software generally treats each byte,
packet, record, pixel, and transaction as equally
important, whereas in most software applications, 20% of
the transactions account for 80% of the application’s
value, and value-neutral capabilities tend not to be cost-
effective.

A key need for future software engineering research is to establish
a better balance between infrastructure and applications software
research. Also, user-programming research needs more emphasis;
analysis of spreadsheet applications generally show that about
half of them contain defects serious enough to cause corporate
problems if encountered. Some research and a series of ICSE
workshops focused on creating the equivalent of seat belts and air
bags for user programmers has been started, but more is needed.
2. INTEGRATING SOFTWARE, HARDWARE, HUMAN FACTORS, AND SYSTEMS ENGINEERING

Many applications-area projects involve the need to integrate software with hardware devices and human controls. Left to themselves to determine the system architecture, the hardware or human factors personnel will often make commitments that severely complicate the software engineering function. A good example is the choice of best-of-breed hardware components or user applications with incompatible COTS products or user interfaces. Research is needed on integrated software-hardware-human factors system definition and design that involves software engineers in both the research and the use of the resulting methods, processes, and tools.

A valuable perspective on the mismatches between traditional hardware-oriented systems engineering architectural structures and modern software architectural structures has been provided in [3]. First, traditional hardware-driven systems engineering methods functionally decompose the systems architecture into one-to-many “part-of” or “owned-by” relationships. This means that much of the software is fragmented into part-of children of numerous scattered hardware components, while modern software methods organize system capabilities as layers of many-to-many “part-of” or “owned-by” relationships. This makes for slow and cumbersome software adaptation to change, and difficulties in creating high-assurance systems.

Second, hardware interfaces tend to be static: sensor data flows down a wire, and the sensor-system interface can be represented by its message content, indicating the data’s form, format, units of measure, precision, frequency, etc. In a software-intensive, net-centric world, interfaces are much more dynamic: a sensor entering a network must go through a number of protocols to register its presence, perform security handshakes, publish and subscribe, etc. When these interface aspects are neglected (as they frequently are), many later integration problems will cause project overruns and operational shortfalls.

Third, hardware relations are assumed to be static and subject to static functional-physical allocation: if the engines on one wing fail, an engine cannot migrate from the other wing to rebalance the propulsion. But in software, modules frequently migrate from one processor to another to compensate for processor failures or processing overloads.

Thus, a hardware-first approach to system architecting is likely to cause significant problems. Table 1 below provides perspectives on why software-first or human-factors-first approaches are similarly unlikely to succeed, and why concurrent hardware-software-human-factors approaches and bridging personnel capabilities are needed. It summarizes some of the key differences between the phenomena, economics and mental models involved in hardware, software, and human factors engineering.

The major sources of life cycle cost in most hardware-intensive systems are during development and manufacturing, particularly for systems having large production runs. For software-intensive systems, manufacturing costs are essentially zero, and except for short-life software applications, about 70% of the life cycle cost goes into post-development maintenance and upgrades [4]. For human-intensive systems, the major costs are in staffing and training, particularly for safety-critical systems requiring continuous 24/7 operators.

As indicated in rows 2 and 3 of Table 1, particularly for widely-dispersed hardware such as ships, automobiles or medical equipment, making hardware changes can be extremely time-consuming and expensive. As a result, many hardware deficiencies are handled via software or human workarounds that save money overall but shift the life-cycle costs toward the software and human parts of the system.

As can be seen when buying hardware such as cars or TVs, there is some choice of options, but they are generally limited. It is much easier to tailor software or human procedures to different classes of people or purposes. It is also much easier to incrementally deliver useful subsets of most software and human systems, while core hardware capabilities tend to be indivisible: delivering a car without braking or steering capabilities is infeasible.

The science underlying most of hardware engineering involves physics, chemistry, and continuous mathematics. This often leads to implicit assumptions about continuity, repeatability, and conservation of properties (mass, energy, momentum) that may be true for hardware but not true for software or human counterparts. An example is in testing. A hardware test engineer can generally count on covering a parameter space by sampling, under the assumption that the responses will be a continuous function of the input parameters. A software test engineer will have many discrete inputs, for which a successful test run provides no assurance that the neighboring test run will succeed. And for humans, usage testing needs to be done by the users and not test engineers.
The main point here is that for the future of software engineering research, there needs to be a balance between pure-software research and research which involves software engineering researchers with their hardware, human factors, and systems engineering counterparts in creating ways to requirements-engineer, architect, and develop the increasingly complex software-hardware-human-intensive systems and systems of systems of the future. Some steps in this direction and case studies are provided in the NRC study, Human-System Integration in the System Development Process [5], and in Fredrick Brooks' recent book, The Design of Design [6]. However, if research priorities cause software engineering researchers, and the students they teach, to deal only with discrete mathematics and linguistics, the software engineering field will be poorly prepared for such interdisciplinary research.

Several additional references [7, 8, 9, 10, 11] provide further perspectives for extending software engineering research outside the digital box.

3. REFERENCES


Table 1. Differences Among Hardware, Software, and Human Factors Phenomena

<table>
<thead>
<tr>
<th>Difference Area</th>
<th>Hardware</th>
<th>Software</th>
<th>Human Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major Life-cycle Cost Source</td>
<td>Development, manufacturing</td>
<td>Life-cycle evolution</td>
<td>Training and operations labor</td>
</tr>
<tr>
<td>Ease of Changes</td>
<td>Generally difficult</td>
<td>Good within architectural framework</td>
<td>Very good, but people-dependent</td>
</tr>
<tr>
<td>Nature of Changes</td>
<td>Manual, labor-intensive, expensive</td>
<td>Electronic, inexpensive</td>
<td>Need personnel retraining, can be expensive</td>
</tr>
<tr>
<td>User-tailorability</td>
<td>Generally difficult, limited options</td>
<td>Technically easy; mission-driven</td>
<td>Technically easy; mission-driven</td>
</tr>
<tr>
<td>Indivisibility</td>
<td>Inflexible lower limit</td>
<td>Flexible lower limit</td>
<td>Smaller increments easier to introduce</td>
</tr>
<tr>
<td>Underlying Science</td>
<td>Physics, chemistry, continuous mathematics</td>
<td>Discrete mathematics, linguistics</td>
<td>Behavioral sciences</td>
</tr>
<tr>
<td>Testing</td>
<td>By test organization; much analytic continuity</td>
<td>By test organization; little analytic continuity</td>
<td>By users</td>
</tr>
</tbody>
</table>