Software Research and Education: Meet the SIMS!

Susan Gerhart gerharts@pr.erau.edu and Paul Hriljac hriljap@pr.erau.edu
Embry-Riddle Aeronautical University, Prescott AZ

Abstract: The quality and productivity of software of the future will be influenced by both the new software visions proposed for this workshop and the educational infrastructure that inculcates these visions in budding software engineers. We raise basic questions about current education delivery: are books sufficient, and well matched with current learning styles? Why isn't computer science more laboratory-oriented? Does "dynamic dissonance" describe a retarding factor in learning? Could simulation games and algorithm animation enhance content delivery? Are the standard course-time units too relaxed for modern technologies? If computer science curricula were compressed and accelerated, what topics might be added to improve the delivery of software research visions? Can software address different modes of learning and instruction?

Dynamic Dissonance

Examine a typical high-quality computer science textbook: $80-$125 (- resale); 500-1000 pages packed with facts and explanations + review questions + exercises; some web links and literature citations; an instructor website or CD with more of the same plus online code -- nothing very dynamic or interactive. Now let's examine a kid's PC game, say SimCity (http://www.ea.com)or Theme Park 2: $39-$49 (+PC), selling millions of copies; 1000s of graphics and sounds; driven by scenarios of goals, events, and strategies; with an underlying real-world model combining economics, computing, and physical science -- highly interactive, empirical, and contextual.

Is the world of books, power point slides, and white board drawings sufficient in this clash of experience? Does the simulation game mode of learning offer a different learning style, one that some students always needed and the current generation now expects? If not books, then how is it possible to get across the basic facts, terminology, and explanations? Is the static nature of text +graphics on pages too limiting, too difficult to bring to life in the classroom? Are there better ways, using technology, for improving and accelerating learning about basic computer science subjects?

We dub this situation as *"dynamic dissonance"*, i.e. educational materials could be more dynamic and interactive, also better matched with student's experience. Seeing what Sims accomplish, can computer scientists and educators capture some of this experience of joint discovery, question and answer, and sense of surprise? How can we fully exploit this option for learning?

Is this statement an assumption, a requirement, or irrelevant? New visions of software design require changes in not only content but also style of classroom and self-learning presentation?

Where are the barriers to more interactive and empirical modes of education?

We recently started an NSF grant (http://nsfsecurity.pr.erau.edu) to increase security expertise in Embry-Riddle computing and global studies students, faculty, staff, and eventually the aviation industry. Our goal is produce, evaluate, and disseminate security-relevant modules suitable for technical computer science and software engineering courses. Not believing that one more tech report, monograph, or power point slide would really make much difference, we started from the premise that modules should be (1) interactive, contemporary multimedia, well-packaged within the existing body of knowledge (texts, papers, CERT advisories), and (2) more challenging, measurable for increased expertise.

We focused on the obvious technique of algorithm animation because the essence of any security exploit is a sequence of events and data transmissions that creates a surprising state within a system. That is, any

static view of the underlying phenomena of a security exploit is underwhelming compared with the potential of a dynamic simulation of the exploit occurring. We want to capture that sense of surprise of seeing interacting system features and the resulting damage, to leave a lasting memory and deeper appreciation of the security phenomena. The end product should be as stimulating as an interactive simulation, as captivating as a good well-linked website, and as enduring as a well-written textbook or article.

To sum up what we've learned so far: Algorithm animation is almost a wasteland of abandoned ideas. Pockets of activity exist at Georgia Tech (http://www.cc.gatech.edu/gvu/softviz), Duke University (http://www.cs.duke.edu/~rodger/tools/tools.html), Digital Research (http://www.research.digitial.com/SRC/zeus/home.htm), and collections such as CCAA (http://www.cs.hope.edu/~alganim/ccaa) and Schneiderman's *Readings in Information Visualization*. These indicate potential but not yet wide use, nor systematic analysis. Have any of these (mostly applets) demos been an unqualified success? where have they failed? Is the whole idea of animation a failure or just the individual packaging or choice of examples? If text books began integrating animations, would the outlook change?

Yet, the seeds of effective technology are there. For example, Duke's JAWAA (an animation generator, http://www.cs.duke.edu/csed/jawaa/JAWAA.html) is at the very least a prototyping tool and potentially a full-blown educational technology and methodology (the underpinning of a CS SIMS series). The outline of a DES animation was quickly doable (by a freshman student) and suggested a range of templates and representations for scenario-based cryptography and other modules. Another student translated buffer overflow techniques presented in hacker literature and security tutorials into a DOS demonstration of stack-smashing. This has challenged us to ask: if technology isn't a barrier, then how far could we take dynamic education? how can we measure its effectiveness?

Implications for New Visions of Software

How does "dynamic dissonance" relate to future visions of software research and applications? Where do alternative learning and teaching styles fit into SDP and HCSS?

1. *More alive, interactive, and contextual.*

CS curricula are well established, although highly variable within courses depending upon local success factors. For example, data structures is a semester, discrete math a semester, operating systems another semester, etc. Knuth wrote the stone tablet of data structures in 1973 (?) and it takes about a semester to plow through those data structure variations, get bitten by pointers and coding practice, come to appreciate big-O, and see some applications. We ask: suppose it didn't take a full semester to learn the standard set of data structures, what would be changed with respect to a future vision of software? Indeed, what is the least time required to actually reach an expected CS level of expertise in data structures? Is it a week? a month? a semester? Are there appropriate accelerating technologies such as animation races (speed=big-O), visualization (trees forming and changing), abstraction (nodes vs addresses), correctness (displaying invariant preservation and loss), robustness (effects of errors, memory leaks)? Can the larger concerns of high confidence software be forcefully introduced into data structures through rigorous testing and inspection techniques, extracting components from and reengineering legacy systems, and reuse (e.g. incorporating STL (Software Template Library)). Could a component oriented approach become a standard for learning and applying data structure knowledge? Can new educational tools be developed which encourage a student's understanding so that using the knowledge of data structures in various software engineering tasks comes naturally?

2. More empirical training.

Most CS courses don't have laboratories or lab technique emphasis. Suppose the CS curricula had a large collection of interactive algorithm animations (see CCAA, queuing for OS, garbage collection for programming languages, data structures, searching in AI, etc.). Could true laboratories be built around these interaction opportunities? An example lab might start with an introduction to the subject through traditional textbook description and instructor explanation. A lab would have a focused phenomenon or hypothesis, e.g.

when an algorithm's run-time becomes intolerable on certain architectures, that a buffer overflow could cause change of permission structures, that a code could be reverse engineered to determine its function, that a certain flaw such as buffer overrun could be detected with a static analyzer, etc. Each lab would have a defined procedure for the student to follow, including how to take measurements and what to observer during the experiment. The student would be responsible for summarizing the sequence of events, recording the measurements, describing phenomena observed. Finally the student writes up the conclusions or problematic indicators. This type of live experimentation with software is not possible without some interactive and dynamic apparatus (program), a well-written procedural description of some engineering practice, and a determined effort to increase students' analytic and observational powers. This could encourages other modes of instruction, such as a return to a Socratic form of education. Or a form analogous to the Harvard Business schools case studies.

3. *Flexible enough to embrace new visions of software.*
Consider just one theme of the SDP-workshop planning and panel statements: software artifacts. While the type of interactive animations discussed above may not apply, any thought of educating students to appreciate and learn from artifacts requires hands-on experience. That experience would certainly be structural, dissecting artifacts to see how the pieces fit together, but also benefit from some element of dynamics, including processing speed and data transformations. Especially with domain content, e.g. embedded aviation control systems, simulators would be useful.

In summary, new visions of software and software research must take into account current limits of educational methodology and technology. Books alone don't cut it anymore, students expect to *see things* and *watch events happen*. Learning styles change even in computer science. Once that change takes hold in the students, faculty and text authors must follow or lose ground. We know what the fundamentals are, but is their presentation stagnant, losing out on the interactive world, and too unchanging to accommodate even the best accepted new visions. New visions of software and software research must overcome not only old practices of industrial software engineering but also old practices of teaching computer science and software engineering. We have asked why one specific approach, algorithm animation, has seemingly failed to take hold while advancing technology became less of a barrier. If that straightforward an idea cannot be analyzed for its success and failure factors, then how can any more novel approaches to software be evaluated in either practice or learning?

Submitted to: http://www.isis.vanderbilt.edu/sdp/
Workshop on New Visions for Software Design and Productivity: Research and Applications
Dec. 13-14,2001 Nashville, Tennessee.