Community-Based Software Development Under Time Pressure: New Jersey and San Diego in the 21st Century

Bob Neches[1]

Distributed Scalable Systems Division

University of Southern California Information Sciences Institute

4676 Admiralty Way

Marina del Rey, CA 90292

This note looks at the challenge problem of reducing the time required to develop and field significant applications from months or years down to days or weeks. It looks at what it takes to cobble together applications that have utility within days to weeks to months, and which then evolve over usage across years and years. The "new" concept of Extreme Programming represents the recognition that top-down program development does not work in practice. However, it represents more an abandonment of that approach than a clear statement of a principled alternative. Central to development of that principled alternative is the idea of tools that support hacking now, while keeping track of structure, requirement, and architectural issues in order to support cleaning up later. Such principled, high-speed, bottom-up development also requires a closer conceptual coupling between users and developers across much more of the software lifecycle than is currently generally conceived.

[1] *Without implying their endorsement of the positions I have taken, I would like to acknowledge my indebtedness to In-Young Ko, Pedro Szekely, Bob Lucas, Craig Milo Rogers, and Bob MacGregor for discussions which have greatly influenced the content of this note.*

Introduction

Some years back, a famous essay, *The New Jersey School of Programming*, contrasted the Unix culture with then-prevalent alternatives and argued that systems which “make 80% of what you want easy,” always win over more elegant systems that buy flexibility and coverage at the cost of complexity. What has happened since has perhaps not been within the letter of the predicted victory for Unix, but is largely consistent with the essay’s spirit. Unix is the system of choice for servers which require its reliability and the experts who run them. Microsoft Windows -- less functional but simpler (for its user community) -- spread among other users like drugs in the inner city. Palm leads Microsoft-based PDAs (again, simplicity with respect to intended use), although the latter may be gaining as the valuation of integration with Windows Office functionality grows.

The principle seems to hold -- as long as we remember that what constitutes an 80% solution is not fixed, but is relative to a user community and the tasks they perform.

Why is this important to a discussion of software design and productivity? Because we live in a world with a huge capital investment in software and a huge capital investment in people -- we can neither afford to replace the former nor retrain the latter. A large body of our future software is going to need to have certain critical properties of *adoptability* and *evolvability*. For a large body of software, *the critical need is to cobble together applications that begin to offer practical utility within days to weeks to months, but which can then evolve over usage across years and years*.

Not all software fits this mold, nor do these issues solve all design and productivity issues. There are separate issues of reliability and robustness, particularly in embedded software, which make it highly desirable to explore techniques for transforming centralized algorithms into distributed, parallel and (sometimes) redundant computations. These are separate research threads that need to be pursued in parallel. I do not argue that adoptability and evolvability are sufficient elements, only that they are necessary.

For what kinds of software are these critical issues? Virtually anything that is initially conceived as *data or data types for end-user applications* (because what starts out as data becomes objects that we want to have behavior and characteristics, e.g., we start with pictures, then want to be able to animate and retrieve them), anything written as *supporting services for those applications* (because, shortly, we will want it to handle multiple data types and requests from multiple sources), and the end-user applications themselves (because those front-ends soon will grow in functionality, and/or migrate to become back-ends of new front-end services). This covers pretty much anything intended to operate in a network-centric environment, be it commercial or military.

In the sections following, I will argue that enhancing software design and productivity in network-centric settings creates both an opportunity and a necessity to relate the New Jersey School of Programming to a concept from the West Coast: User-Centered Software Design (which came out of UC San Diego in the 80’s). In the process, both must be updated to reflect the realities of the Internet in the 21st century.

Software Under Time Pressure

Commercial software usually gets built under time-to-market pressure. Government software often has similar time pressures for different reasons[2]. Rather than lamenting lack of time to "design it right", let's consider why time pressure is inevitable:

1. **Better now is often better than perfect later.** One of our projects is concerned with reducing task overload in AV8-B Harrier military aircraft operations. The Harrier accident rate is six times the next most dangerous military aircraft (this is *after* the prior accident rate was nearly halved). *Delivering some improvement as soon as possible and incrementally adding more later is <u>far</u> better than waiting*.

2. **Life's a moving target, always.** The software environment is perpetually in flux. (PERL/JAVA, SGML/XML/RDF/DAML+OIL, ...). Architectures, components and environments into which an effort must fit are often being defined in parallel with the definition of that effort, although they may have some longevity once defined. Users' requirements, expectations and skills change dramatically -- and quickly. Experience with other software is a factor; consider the impact of the introduction of the spreadsheet paradigm. Growth in sophistication is another; consider how videogame-/computer-literacy have impacted requirements. Simple familiarization with current versions is enough to create the problem. Users of the negotiation-based scheduling system we are prototyping for Harrier aircraft were grateful for our having reduced a six-hour process to five minutes, but are already starting to get restive with waiting five minute for schedules. *Waiting for quiescence before designing and building, risks never getting started.*

3. **We don't know what we want until we see what we've got.** The field has a technical name for this phenomena: "requirements creep." The state of the art is to either (a) fight it bitterly or (b) acquiesce and try to control and restrain it though techniques such as the spiral model of software development. Unfortunately, emergent requirements tend to be pesky; they arise at inconvenient times and refuse to wait for scheduled phases. Software designers don't all have technical terms for this, but project managers certainly do: "opportunities" and "showstoppers". For example, we did not -- could not -- know how speeding-up flight operations scheduling by 100x would change maintenance scheduling needs. *The forces behind requirements creep are real and powerful; resisting may seriously imperil a software project's quality -- or even the project itself.*

4. **When we don't have it, we don't know how badly we want it.** HTML and browsers were not designed to be the distributed hypertext system for the world. Rather, they were designed by physicists for physicists, to support exchanging papers. It grew from there. Computer scientists lament being out of the loop, claiming we knew how to do it better, but Doug Engelbart proposed the idea in the 60's and arguably we spent nearly 30 years overdesigning without delivering. For every WWW that maybe should have be designed better, there are probably dozens of systems that are throw-aways (or should be). *It's hard to predict what designing a system right means, much less predict whether taking the effort to do so will turn out to be worthwhile.*

[2] *A poster at NIST once described government as, "addressing 20-year problems through 10-year plans, implemented through 5-year programs, run by 3-year managers, with 1-year budgets."*

Implications of Time Pressure

The preceding observations argue towards two conclusions about software quality:

- **The problem with the "build it now and fix it later" mentality isn't how to overcome it, it's how to make it work.** Unless we can define a research program that will give mankind ubiquitous prescience, there may well be no viable alternative -- often we just don't know enough to design top-down.

- **The role of users in the software lifecycle starts earlier, runs longer, and intertwines more closely than we are currently recognizing and supporting**. Users know the requirements and quality priorities only a little bit better than we do, but they're in a much better position to discover them as things proceed. As the lifecycle plays out, the users change -- and so do the software developers[3].

We need to accept this, embrace it, and learn how to be much more responsive to it. We should set ourselves the challenge of reducing the time required to field substantial applications by an order of magnitude: from months to years down to days to weeks.

Current Approaches and Relevant Work to Build Upon

In taking on this challenge, we need not start from scratch. There are a range of developments in recent years that offer promise.

Component based software. The ability to assemble software from components is growing significantly. As of this writing Sun's JavaBeans web site alone lists 725 available components in 28 different categories. Initiatives for languages and protocols for registering components, accessing them, composing them, exchanging data between them are taking shape, e.g.:

- Resource Description Format (RDF) for the Semantic Web(http://www.w3.org/RDF/)
- Web Services Description Language (WSDL), an XML format for describing network services (http://msdn.microsoft.com/library/?url=/library/enus/dnwebsrv/html/wsdl.asp?frame=true)
- Universal Description, Discovery, and Integration (UDDI) a registry service for Web services (http://www.uddi.org/index.html)
- Simple Object Access Protocol (SOAP), a protocol for exchange of information in XML (http://www.w3.org/TR/SOAP/)
- Web Services Flow Language (WSFL), an XML language for the description of Web Service compositions (http://xml.coverpages.org/wsfl.html, http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf)

Extreme Programming. The notion of very small teams rapidly producing software has some history. Studies at IBM showed that teams of 3, 10, and 100 programmers, given the same problem, achieved completion in order of their size. The upsurge of interest in "Extreme Programming," indicates a growing sense that top-down design and large teams don't produce needed speed and quality.

Faster computers, the power of brute force, and the opportunities of complexity analysis Where we once practiced structured programming and, working top-down, wrote stubs for modules with the intent of coming back later to program them efficiently, today it is often possible to write temporary versions that run "efficiently enough" for a set of cases. Things that had to be "done right" just to get adequate performance can often be done later. Functionality can grow over time across a range of inputs, as more difficult cases get handled. Furthermore,

[3] *To illustrate, three quick questions for workshop members: How many use the web? How many are from CERN or NCSA? How many use Mosaic as their browser and send gripes to NCSA?*

computational complexity analyses are becoming increasingly useful. Work on analyzing and predicting phase transitions, for example, in the DARPA Autonomous Negotiating Teamware Computation Complexity program is being applied to anticipate difficulties for resource allocation systems, enabling systems to switch methods in midstream for the most efficient. Thus, efficiency concerns can be addressed through heterogeneous, case-based approaches rather than through design of uniform optimal algorithms.

Gauges, probes and runtime monitoring from DARPA's DASADA program. Another example of rapid programming enabled by embracement of heterogeneity lies in DARPA's Dynamic Assembly for System Adaptability, Dependability, and Assurance (DASADA) program. This effort is creating architectures that support probes that monitor run-time performance and behavior of software components and report back to systems that allow dynamic switching to alternative components or repair methods in the event of problems. We are in the process of using these techniques in a very compute-intensive system, GeoTopics[4]. Processes which have to strip and analyze every word in the HTML source of every article in nearly a dozen on-line news sites are being monitored for time-outs and replaced or repeated as needed.

New insights into APIs. There is a great deal of knowledge about good programming practices to support rapid development among small teams. These techniques, although familiar to practitioners, have not necessarily been studied closely, nor do tools exist to support them. For example, our group at ISI has in recent years produced a number of substantive systems on a rapid development cycle in a component-based approach. Among the key software engineering practices which have enabled their rapid development are recognizing that, "good programming practice with respect to achieving robustness and avoiding brittleness focuses on structuring the code so that invalid or meaningless inputs cannot even be represented in the system. Such tight definition of inputs and outputs establish a contract between a module that provides a capability and the modules that use that capability... [thus avoiding] time-wasting misunderstandings [between programmers] about what constitute valid inputs." (Pedro Szekely, 6/13/01). Another related technique focuses upon utilizing the structure of object-oriented languages by preventing errors in which programmers apply borrowed code inappropriately through access to functions other than through method calls within objects. In conjunction with the first principle, this allows programmers to borrow and reuse each others' code very quickly with relative confidence that it will operate as needed for the use they expect to make of it.

New insights into tools for user-centered software development. Development of quality software under time pressure requires dramatically reducing the time delay between users' expressions of requirements and/or feedback on software releases. It also entails eliminating the sequential cycles of extracting requirements, prototyping, evaluating, and repeating. Instead, all of these activities need to go on in parallel, all the time. Synchronous collaboration tools such as NetMeeting, PCAnywhere, WebEx, and many others, facilitate this somewhat. Depending upon same-time availability of developers and users is neither reliable nor manpower-efficient, however. E-mail improves the situation somewhat, but it is difficult to organize and relate the content to a complex body of software (or even, often, to figure out exactly what the user was doing and what problem they're talking about). We have taken to providing web-based tools that allow users and developers to share mock-ups of user interfaces and annotate them with comments. The mock-ups and the comments are represented in DAML+OIL, a semantic mark-up language under joint development between DARPA and the World-Wide Web Consortium. This allows us both to identify relationships of user topics to software components, and to route

[4] GeoTopics (http://www.isi.edu/geoworlds/geotopics) is a service that daily examines news reportage from the websites of a large and growing number of major worldwide news operations to help identify the "hot topics," and the most frequently referenced places, found that day in this very large collection of reports. The service ranks topics and places by the number of references found to them that day. It automatically compares these to the previous days' results, flags new topics that emerged that day, and tells you whether ongoing topics seem to be getting more or less attention than the day before. GeoTopics is but one example of many systems that very compute-intensive and involves many parallel threads.

user comments appropriately. We have also developed a number of end user programming tools that allow end users to participate in extending the software. These range from UIs which guide users through creating rules, to tools which help them find software components and organize them into parameterized data flows. (The latter can be put into libraries for other end users to borrow, modify, and re-publish.) Underlying these tools is what we have called a, "three-tier" model of a community: developers try to provide features that help power users help users.

Research Issues

Although some pieces exist, many holes need to be filled to realize a vision of quality production of community-based software under time pressure. Among them:

1. **Extraction / reverse engineering / reengineering tools.** A big barrier to redoing rapidly produced code today, is the difficulty of finding and modifying all the pieces of code and variable references that depend on the old approach to ensure that a systematic conversion to the new is effected. Work such as Kestrel Institute's on the Y2K problem has shown the feasibility of recognizing code managing time and date references in arbitrary "messy" source code. We need to apply those concepts to reconstructing models and idioms for software that was generated without making these explicit or without full consistency. We need environments for post-hoc model generation and conceptualization of a piece of code. We need to tools for mapping models to others (e.g., converting code for producing a data set to a generator of data elements). Building these tool will facilitate reconceptualizing code once it is well-understood, and semi-automatically reimplementing to a new model.
2. **Wrapper generation.** Tools for making components behave and interface with others.
3. **Run-time monitoring and repair kits.** The DASADA notion of probes and gauges is a start in this direction, but needs to be augmented with additional work on automating insertion of gauges, defining triggers on gauges to automatically trigger repair processes and rapidly generating component selection/swapping mechanisms to effect the repairs.
4. **Structure modeling and mapping**. The conceptual structure of rapidly generated code is emergent, and comes through the understanding of the application that develops over time through dialogue among the developers and between developers and users. Capturing that knowledge and formalizing it as models of as-is and to-be code, we need to explore relationships between semistructured and hyper-text-y representations and formal model languages, in order to provide a framework for capturing discussions, relating it to code, and gradually converting the emerging ideas into formal models.
5. **Collaborative requirements refinement.** In the same vein, similar capabilities for capturing informal discussion and converting it to formal content need to be applied to facilitating and managing the dialogue between users and developers. This idea is not entirely new. Similar ideas appeared in the plans (but not in the actual research work that was done) for the DARPA EDCS (Environments for Design of Complex Software) program around 1995. However, the concept still has not been realized, while the growth of tools on the World-Wide Web in the time that has passed offers enabling technology which makes it much more feasible to realize today. Extending that concept, we need an environment in which the lines between developers and end-users are blurred, as advanced end-users are helped to create extensions and macros that go back into the collaborative environment and become available to other users.

An initiative which supports these capabilities would go a long way toward supporting the way master programmers actually work to produce large bodies of reliable, working code -- and to enable other programmers to do the same. That mode of working, "despite what Brinch Hansen says, is not top-down structured, nor is it really bottom-up [extreme], either. It is really more like assembling a jigsaw puzzle -- you create pieces and lay them out with some blind faith that eventually you'll see how to fit them together." (Craig Milo Rogers, 11/5/01).