Winning and Losing in Large-Scale Software Development: A Multi-Decade Perspective

Walt Scacchi, University of California, Irvine

What have we learned more than fifty years into the life history of software development? This article highlights some lessons about winners and losers, what has worked and what hasn't. A short history of large-scale software development transitions into a set of topics that range from software seen as intellectual property, software requirements, software components, the organization of software production, and software maintenance and evolution. It concludes with observations about the direction of large-scale software development.

A socio-technical history of large-scale software development

Modern, industrial scale software development has roots in the System Development Corporation.[1] SDC was conceived in the 1950's to develop software for the Semi-Automated Ground Environment (SAGE) air defense system. SAGE was the first large-scale software system. It was programmed by a staff of 500 people. Developers underwent five weeks of concentrated programming training. These software developers were generally not scientists, engineers or mathematicians. But SAGE proved the concept of applying computers to complicated technical activity like multi-site radar monitoring.

By the early 1960's hundreds of computers were deployed to support applications in military defense, space exploration, scientific research, and business data processing, especially in finance and inventory control. High-level programming languages such asAlgol, Fortran, Cobol and many others were invented to simplify algorithmic computation and other programming challenges. However, large-scale software development also involved coordinating the work of teams of programmers, engineers, and analysts, sometimes spanning multiple contractors.

Teams increasingly faced challenges of organizing their collective efforts to specify functional software and operational system requirements. Early efforts often produced incomplete requirement specifications that necessitated ongoing modifications and program updates. Often, software development teams lacked management process definitions and software production controls. This now seems unremarkable, given the prevalence of these in modern software development.

Large-scale software development was an international military-industrial imperative by the late 1960's. The NATO Software Engineering Conferences in Garmisch, Germany and Brussells, Belgium are often considered seminal in this regard. Software development was unfamiliar and difficult. Companies and R&D groups often openly shared their source code. Software development was costly, usually done only by contractors or consultancies within managed development projects that produced contract milestone documents describing software requirements, design, test cases, user manuals and more. These contract fulfillment and document

production milestone stages became a prescription for software development life cycle models. It matter less whether anyone read the documents.

Research funding by agencies such as the Defense Advanced Research Projects Agency, and the hosting of software engineering conferences and journals by societies like the ACM and IEEE, brought academics to the field of software R&D in the 1970's and early 1980's. Major U.S. software research projects were launched, focused on knowledge-based software engineering and the development of new programming languages and support environments (e.g., Ada). There was hope that programming could be automated and software development streamlined, while producing higher quality code. But these hopes were not realized.

Much academic research attention focused on ways and means for formalizing the precise description or representation of software life cycle document content. Debates centered around choice of mathematical basis, representational notation, or computational language constructs employed. All focused on software as a product. Software development as organizational processes articulated by people with ordinary technical skill and diverse interests generally escaped attention. Unsurprisingly, attention to management and cost accounting practices applied to industrial software development often backfired, as with the Mythical Man-Month, discovering that adding programmers to a late project furthered slowed development by adding communications overhead.

Poor quality, lack of process management standards, and cost problems continued to plague software projects, causing software development to be recognized as a challenge involving technical and social components. Of special concern was the organization of people, processes, and resources to enable productive work.

There was a need to focus attention on improving software development practices through defined and managed processes, codified and transferred to military and industrial software contractors. The Software Engineering Institute (SEI) was formed to work on this. Technology changed, too: networked computer workstations and personal computers made the distribution of software development tools possible on a personal basis. Networked distribution of software development resources supported Computer-Aided Software Engineering (CASE).

The World-Wide Web enabled new socio-technical project organization. Nevertheless, challenges remained. Rather than wait for industrial or governmental solutions, software development students, hobbyists, and programmers who just wanted to code began sharing open source software (OSS). OSS was used to develop and host large-scale software system applications.

More recently, software has taken on the form of services, apps, cloud repositories, mobile user interfaces, social media, computer games and more. Nearly all software now lives on wide-area networks. Few software developers make personal software just for themselves, or produce documents no one reads. OSS has itself challenged the traditions of large-scale software development by adding shared ownership and control of software, development processes, and meritocratic decision-making.

Software as intellectual property: closed, free, and open sources

Software is an information artifact. It is complex and costly to produce and maintain. It has become a form of intellectual property (IP). Much like literature, poetry, music, and artwork, software is protected from unauthorized or infringing use by copyright expressed in license agreements.[2] But software is a peculiar kind of IP. It is not similar to IP that is collected, studied and treasured. Great software systems are not open to historical study or displayed in museums. Although software expresses procedural or non-procedural narratives of computation and interaction control rendered with programming languages, software might be disposable IP. While programming languages are non-spoken cultural modalities, they express formulae, calculations, control flow logic, data communications and procedural computations. Nonetheless, software often entails organizational implementations of data processing procedures.

Software development needs functional specifications, system design architectures, reusable software libraries, user interface layouts, and control protocols. These are also IP. They can be more valuable in the market, and often more reusable than the code that implements them. Software business models are increasingly codified within software IP licenses. No other kind of IP has these features.

Much late 20th Century industrial and commercial software development created and disseminated products with IP licenses restricting unauthorized sharing or reuse. In response, the modern OSS movement emerged. Routine sharing of free OSS and "copyleft" IP licenses ensured digital civil rights to access, study, modify, and redistribute OSS code, while allowing limits on access to other valuable and informative software artifacts. This was seen in OSS-based computer games, where sharable, open, and extensible code was accompanied by required purchase of a game to access proprietary IP assets. This became a common business model.

The 21st Century has brought widespread adoption of free, open, and proprietary source code. This code can be accessed from shared, cloud-based software repositories such as SourceForge and GitHub. Such code is capable of operating across the Internet/Web, and led to the emergence of software ecosystems, also called platforms or systems of systems. These are ever-more complex socio-technical configurations that bring together communities of like-minded users. Co-joined social and IP challenges now arise in managing open, extensible software ecosystems that span loosely-coupled organizations and development teams. These challenges are as daunting as those facing software development half century ago.

Protecting software, whether free, open or proprietary, entails cybersecurity obligations and controls that are seldom clearly specified in IP licenses. Ongoing software updates pushed to users to insure security and data protection are frequently incomprehensible to software users who increasingly must act as IP and cybersecurity administrators. In this way, software remains a challenge even with the emergence of OSS and new tools for IP and cybersecurity control.

Software requirements: what, when, why and for whom

Envisioned software is supposed to solve problems, raising the question of software requirements.[3] In closed, well understood domains like Arithmetic, external references describe acceptable/correct solutions. Yet, even this can be a challenge. Computer arithmetic is finite,

while traditional arithmetic is not. Computer representations of integers versus real numbers, numerical rounding error propagations, and hardware limits on numerical resolution, can interfere with computerized calculation for interplanetary flight trajectories, financial budget estimates, global climate models, and other problems without careful attention to the mundane constraints of finite arithmetic. Other idiosyncratic and circumstantial features can cause the simplistic, closed domain requirements to fail. Only unseasoned software developers seek requirements predicated on the assumptions of closed, internally consistent, and well understood domains.

Software requirement specification falls on whoever has requirements that can be specified. This has historically been the customers who pay for software. They are expected to provide functional and operational requirements for the software they seek. But who knows what is required for software destined for new or poorly understood problem domains. Requirements are often obscure for defense against incoming missiles or urban terrorists, creation of a global Web platform, making a decentralized cryptocurrency, building an immersive game-based virtual world, creating a repository for open software sharing, or integrating personal mobile devices into proprietary enterprise systems. Who gets to say what the requirements for such things are?

Requirements specification is inherently socio-technical work. Each effort entails negotiations among parties, some with conflicting interests. Software business managers focused on controlling software development costs and delivery schedules might have conflicting interests with software developers. Developers may want to contribute to real-world systems by using powerful tools while having fun and avoiding poisonous team members. These interests might differ from enterprise customers who want software that supports business processes at affordable cost, with ease of change to new software providers when needed. And these interests differ from those of end-users who get stuck using software that someone envisions will make user work more productive and controllable but does not, and instead mainly makes work easier for supervisors to evaluate. Legacy business systems often fall into this category, as managerial oversight accretes around the routine system usage no matter how problematic.

Challenges with software requirements have proved difficult to solve. Collaborative development techniques such as “participatory design” that advocate stakeholder involvement in requirements specification are easier said than done. Software developers are not routinely trained to do it, and stakeholder inputs may be excluded or limited to make development easier. Stakeholders seldom know what is possible or effective until after the software is deployed, making their value to early specification low. In OSS development, those who contribute code or sustain OSS development resources are the participating stakeholders assumed to have the requirements.

Software development contracts/deliverables often determine functionality, operation, ownership, usage rights and obligations, establishing requirements for software IP before and during development, and sometimes for maintenance and field support as well. Who will win situated conflicts over the definition of problem spaces, who is expected to have solutions, and who controls resources affecting software development becomes the main focus of requirements work.

It is possible that software requirements should be determined *after* the system becomes available. Despite the illogic of having a coded solution before the problem is specified, the practice is more common than it appears. Users of mobile phones or tablet regularly acquire software apps for Web search, Web browsing, GPS navigation, integrated email and calendars,

multi-player games, or mobile eCommerce after they get the phone or tablet. Most users operate within software ecosystems where they are willing to accept new products and unfamiliar brands of software needed to address problems that were unknown until seeing the apps. Reverse engineering requirement specifications from deployed code is an intractable problem, and people do not do it. Instead, they define their requirements by what's provided, available to install and use, as part of a larger online community.

Software requirements has long been one of the most difficult challenges in large-scale software development, and looks to remain a difficult challenge. More attention needs to be directed at understanding software "provisionments," or discovering what's provided in available software, rather than expecting some determination of what's needed prior to software being ready to use.

Component-based software development

Building software systems from reusable components was discussed at the NATO Software Engineering Conference in 1968. However, software developers found source code components are not inherently reusable. Reusability, like sustainability, reliability and quality, entail costs and trade-offs. Components can be well-designed and carefully crafted collections of related software functions or modules. Alternatively, software parts can be bricolage assemblies, mashups of components, software ecosystem stacks, or turn-key applications from diverse and unknown software providers. Or they can be something in-between.[4] They can be domain-specific software packages, application-independent software libraries, or networked services. Too often they entail obscure build-time or run-time dependencies that are not part of a component's package.

Well-managed component interconnections and interfaces can be clean, thin, transparent and tractable, then configured into services, product lines and platform ecosystems. Nevertheless, it can be difficult to create smaller software components as products that justify license fees or other economic returns. It is also difficult to create software products that enable remote maintenance and offer one-stop acquisition, but sustain software vendor lock-in business models. Embedded interconnected component dependencies can be difficult to spot without extraordinary time-consuming effort. This often creates a kind of dependency hell where opaque, interconnected software interfaces span proprietary and OSS code bases in multi-component software system configurations.

In the half century since the NATO Conference, turn-key software apps are now routinely available in online stores/repositories. Yet obscure/opaque interconnection dependencies sometimes allow cybersecurity attack mechanisms entry into a system or platform. Consequently, recurring challenges remain: conceptually mapping the functionality available and interconnections within the component package, and identifying the security vulnerabilities when the app is interconnected within a larger system configuration.

The challenges of components are exacerbated by the rush for public engagement with coding. Current training in coding usually ignores components. Components require specialized coding techniques ranging from small, language-specific routines, to speciality software encapsulation methods and interconnection mechanisms, to large software application platforms coded in multiple languages across decentralized software stacks. Component size usually translates into

complexity, and greater complexity brings difficulty in comprehension and diagnosis of operational system problems. Complexity requires a diversity of computational capabilities, while multi-language, multi-component implementations give rise to the kinds of procrustean, ad hoc exception case handlers seen when translating from one programming language to another.

The vision of reusable software components assumes a world where large-scale software artifacts can be assembled out of standardized component parts, put together with customization code developed for such purpose. In fact, a half-century of effort has shown this vision to be much more difficult to achieve than to describe. Comprehensive component-based software development remains elusive.

The socio-economic organization of software production

Large-scale software development entails planning, staffing, budgeting, scheduling, resource allocation and control.[5] Projects compete for the best qualified staff. Software is typically domain-specific, so staffing often requires arcane domain knowledge and skill with related domains. Experience with new software tools, techniques, and applications further stratify the market for skilled developers. Improving software development productivity, managing cost and schedule drivers, and understanding teamwork structure effects on software development practices and system architecture all require non-functional expertise in software process and project management. Learning to code does not provide this expertise.

Most computer science degree programs nor business schools typically provide training in project management for software production. Capabilities critical to productive and efficient software development work are left to ad hoc formulations. Ironic constructs such as the Mythical Man-Month (or what 42 software developers do in the morning before lunchtime), provide modest insights but not deep understanding. Software developers might become project managers to escape coding hell or get away from death march projects staffed with poisonous developers. The appeal of OSS development is in part the self-selection of participants who seek work with OSS developers who enjoy software development challenges, collaborative learning opportunities, and who just want to code. Though OSS can enable alternative roles and career trajectories for developers, it can also erect socio-technical barriers against those who do not want "read the source" first.

Software coders must comprehend organizational regimes before they can determine how, and for how long, a software development job will advance their career as a software professional. It was once common for software developers to stay in positions in single large software enterprises or contractors for a decade or more. They could master domain-specific knowledge for new, complex applications and pursue long-term career contingencies. A career strategy for long-term employment might mean maintaining a "mission-critical" software system. Software development people remain critical social resources affecting software production. Yet technological and business exigencies enable, disrupt, or frustrate software development work.

Commercial software development is a business activity, and developers can learn how business concerns drive project management choices. For example, projects with development cost liabilities can be turned into maintenance efforts that generate revenue, and this accounting trick can enable development projects over-budget or over-schedule to stay in business, borrow money against assets, and sustain developer jobs.

The advent of Internet/Web over the past decades has broadened and strengthened software infrastructures. Networked desktop services have enabled geographic distribution of software development work across timezones, national boundaries and cultures. There has been an economic rush to produce, capture, and harvest a new generation of software services. This has accompanied distribution of software development project opportunities, as software developers reframe their career opportunities around workplace mobility and global software development facilitated by multi-national firms or self-interested OSS developers.

Developers learned to work collaboratively with others near and far using online information artifacts, tools, and repositories. Such information capabilities record why and how development actions and decisions are made, with what consequences, for whom, and to what ends. Informal software descriptions such as multi-party text/chat streams developers routinely use outperform formal life cycle documents in software production work.

The socio-economic organization of software production is, in a way, where the software engineering field started. It remains a major challenge for the field. Those trained and experienced in software development are often not trained in the kinds of socio-economic organizational skills needed for running large-scale software development. The socio-technical and socio-economic requirements for software production work remain understudied and elusive. This applies to the development and maintenance of proprietary software, and to OSS.

Software evolution and continuous software development

Software evolves. Understanding this is one of the oldest empirical study areas in software development.[6] The amount of deployed software now reaches hundreds of billions of lines of source code in different programming languages, worth more than a trillion dollars. Software code is valuable and often proprietary, which would suggest that it would be maintained. In fact, much code is not maintained. The number of professional programmers worldwide is often estimated at 10-20 million, so in principle there is substantial amount of code for programmers to maintain. Yet attention and training in “coding” stresses programmers writing new code, not maintaining or evolving legacy code. This is analogous to learning only to write sentences or paragraphs, ignoring interrelated encyclopedic life stories/narratives. Why continue to create software that no one wants to sustain?

This problem is worse than it appears. Software begets more software. Developers create software to address new problems or unprecedented requirements that accompany emerging technologies or conditions. This software creates the need for more software. As discussed above source code is usually not reusable. If it is not maintained it might be neither usable nor reusable. The simplest explanation for the source of this problem is that software is most interesting when it is being developed, not when being maintained. The best programmers often want to work only on software development, rather than maintaining someone else's code. Yet software systems must be maintained in revisions (alternatively versions, releases, or upgrades) to sustain their practical utility and serve as conduits for innovative and incremental enhancements. These evolutionary enhancements embed, extend, and tend to organically lock-in business relationships between software producers and customers.

Software evolution differs by proprietary or OSS. Much of the Internet/Web is based on OSS

code. Many commercial software development firms contribute to OSS when it can help lock-in customers. Maintaining OSS applications and services can help level the playing field among software producers, but maintaining OSS code means maintaining a loosely-coupled community of OSS enthusiasts who are mutually interested in maintaining the code. Some large, complex OSS software artifacts are not maintained as might be proffered in software engineering textbooks, as indicated by the difficulty of finding explicit requirements documents (or any formal life cycle documents) for OSS operating systems, Web browsers, Web servers, etc. Similarly, the multitude of variants and clones of popular OSS application systems, utilities, and components also reflects the freedom enjoyed by OSS developers to choose what software to produce and sustain, as well as whom to work with in producing useful and usable software. OSS development practices shape OSS evolution, and OSS evolution shapes new OSS development work, work products and business models.

Useful software becomes legacy software. Whether proprietary or OSS, bugs/faults appear, needs/opportunities for improved performance become clear; platforms evolve; and new requirements emerge. Useful software is subject to continuous update, thus software is meant to be changed. Software can serve as an engine of innovation because it can be readily evolved, albeit sometimes with great effort. But legacy software regularly dies from lack of sustaining effort and resources. Other software is abandoned when budgets are reduced, when new platforms are employed, or when "faster, better, cheaper" software becomes available.

Software systems have different end-of-life trajectories. Time-sharing systems and batch application programs stored on card decks or magnetic tapes went away. Enterprises dread depending on software written 20 or more years ago, but often they do. Some old software runs on machine emulators when the base hardware is no longer available. Yet some legacy software systems are so deeply embedded in organizations that their source code may be lost, and there are no more developers familiar with their antiquated code. In these situations, the cost of their organizational extraction and replacement may be monumental.

Old software is seldom recycled, parted out, revitalized or memorialized. Again, where are the great works of software for study? The software development life cycle has an interesting blind spot regarding the dying and death processes of software.

Software evolves because the circumstances surrounding it change and software can be changed to meet new needs. However as time goes on, the bias in favor of new development and against maintenance results in increasing distance between the condition of the software, what capabilities it provides, and the needs to which it is applied. Eventually useful software becomes legacy software, sometimes maintained for its application, sometimes running under emulation and seldom examined even by software maintenance specialists, and sometimes abandoned, forgotten or killed off.

Conclusions

Large-scale software systems are a major enterprise resource and a key expression of IP. Software is an essential socio-technical resource that drives the success of organizations, be they business, government agencies, or non-profit institutions. Subsequently, operational software capabilities matter more than formal documents.

Proprietary and OSS software components, whether applications, services, or mashup compositions are now considered necessary, despite the fact that software requirements are less commonly derived from customers and end-users. Software developers produce useful applications useful to those who employ them until interest fades or dies. Free/open software apps with in-app purchases and cloud-based software subscription point to greater insinuation of software into society as a web of socio-technical resources, infrastructures, ecosystems, and socio-economic capabilities.

Software development is an engine of innovation fueled by new platforms and opportunities. Common reliance on software applications, and development work that enables such technologies, reinforces global socio-economic transformation. This engine and transformation increasingly relies on continuous software development processes and practices, as well as how software life cycles end.

References

1. C. Baum (1981). *The System Builders: The Story of SDC*, System Development Corporation, Santa Monica, CA.
2. B.A. Galler (1995). *Software and Intellectual Property Protection: Copyright and Patent Issues for Computer and Legal Professional*, Quorum Books, Westport, CT.
3. S. Robertson and J. Robertson (2013). *Mastering the Requirements Process: Getting Requirements Right* (3rd Edition). Pearson Education, New York.
4. M. Rigby (2016) *Component-Based Software Engineering: Software Architecture*. CreateSpace Independent Publishing Platform, New York.
5. R. Fairley (2009). *Managing and Leading Software Projects*. Wiley, New York.
6. N. Madhavji, J.F. Ramil and D. Perry (eds.), *Software Evolution and Feedback: Theory and Practice*, John Wiley and Sons Inc, New York, 2006.

Walt Scacchi is senior research scientist and research faculty emeritus at the Institute for Software Research, University of California, Irvine. Contact him at wscacchi@ics.uci.edu