Software Engineering Education in the Era of Outsourcing, Distributed Development, and Open Source Software: Challenges and Opportunities

Matthew J. Hawthorne[1] and Dewayne E. Perry[2]

Empirical Software Engineering Lab (ESEL),
Dept. of Electrical and Computer Engineering,
The University of Texas at Austin,
Austin, Texas, USA
{hawthorn, perry}@ece.utexas.edu
[1]http://www.ece.utexas.edu/~hawthorn
[2]http://www.ece.utexas.edu/~perry

Abstract. As software development becomes increasingly globally distributed, and more software functions are delegated to common open source software (OSS) and commercial off-the-shelf (COTS) components, practicing software engineers face significant challenges for which current software engineering curricula may leave them inadequately prepared. A new multi-faceted distributed development model is emerging that effectively commoditizes many development activities once considered integral to software engineering, while simultaneously requiring practitioners to apply engineering principles in new and often unfamiliar contexts. We discuss the challenges that software engineers face as a direct result of outsourcing and other distributed development approaches that are increasingly being utilized by industry, and some of the key ways we need to evolve software engineering curricula to address these challenges.

1 Introduction

Driven by a critical combination of technological and economic forces brought on by ongoing developments in technology and economic pressures caused by globalization, software engineering is changing in fundamental ways. We must rethink many of the assumptions that have provided the basis for software engineering education in the past, and make fundamental changes to the way we educate software engineers in order 1) to prepare them to navigate the increasingly dynamic environment that software engineering has become, and 2) to equip them with the perspectives and skills they will need to thrive in the midst of the even greater challenges they will face throughout their professional lives. In the remainder of this section, we highlight three basic trends that are changing the way organizations develop software: third party components, integration platforms, and globalization. These trends are already changing the practice of software engineering, and we believe they will continue to impact the practice of software engineering for the foreseeable future.

Third-Party Components

The first trend is the increasing reliance on *third-party components* for core system functionality, including open source software (OSS) [13] and commercial off-the-shelf (COTS) components. The impact that delegating significant functionality to OSS or COTS components has on software development projects is similar in many respects to that of outsourcing: more system functionality is being developed by third parties, potentially reducing the amount of software that organizations develop internally, but at the same time, making integration technologies, architectures and frameworks, and software engineering competencies related to integration, of paramount importance to software development organizations. Project planning and engineering need to expand beyond traditional technical concerns to encompass methods for locating and evaluating candidate third-party components, and selecting the optimal set of components for a given project by evaluating and balancing trade-offs between such diverse concerns as functionality, licensing terms and fees, integration costs, and technical support, etc. This trend toward increasing use of third-party components is also bringing about basic changes in software development organizational structures and processes. Requirements engineering assumes a more significant driving role in developing software systems. Quality and evolution issues are complicated and exacerbated because of the lack of control over component problems and evolution. Integration is hindered by architectural mismatch, etc.

Integration Platforms

The second trend is the growth and maturation of integration platforms and architectural frameworks such as J2EE, .NET, web services, and service-oriented architectures (SOA), as well as problem domain model-oriented abstraction approaches to system design such as model-driven architecture (MDA) and the semantically richer intent-based system abstraction technologies of the near future. These approaches are providing new tools, architectural platforms and system abstractions to integrate the diverse set of components that inevitably results when development organizations build products that incorporate a large number of third-party components. Since current technologies and standards are likely to continue evolving rapidly, emphasizing a thorough understanding of the principles behind these approaches and their applicability for solving software engineering problems will be more useful over the long term than complete mastery of the minute details at a given point of time. However, using appropriate platforms and standards as teaching tools is an excellent way to illustrate and reinforce architectural styles, patterns, techniques and principles related to integration.

Globalization

The third trend that is changing software engineering is the *globalization* of software development, often referred to as "outsourcing". We make no distinction in this paper between literal outsourcing, in which one organization pays one or more other organizations or individuals to develop software components or systems, and "internal outsourcing", where an organization distributes development projects to one or more of its own development teams located in different countries. The overall impact on software engineering is similar in either case: organizations are transitioning from primarily local models of software development to globally distributed models. This transition is motivated by two basic issues: cost and time. In the first case, lower labor costs in other countries has had the same effect on the software industry as it has had in other industries – namely, significant parts of the work are done either by external contracts or by opening international facilities using local talent. In the second case, the possibility of *round the clock* development has the attraction of significantly shortening the lapse time required for software products.

This trend towards globalization brings with it its own set of problems and further complicates and exacerbates existing software development problems.

- *Cultural differences*. With different countries come different social interaction assumptions and rules. There are differences in expected and acceptable behaviors and interactions. For example, a simple request may be perceived as a significant social obligation. Moreover, different languages may result in radically different interpretations from what is expected. For example, the word "envy" has both positive meanings in English, Italian and French. However, there is no positive meaning of "envy" in German.
- *Legal differences*. One significant legal difference may be that involving the laws about working overtime. What is permitted in one country may be prohibited in another. Doing business in a specific country may require a certain percentage of the workers to be local to that country. And there may be different rules and regulations about the domain of the software systems involved. For example, there are significantly different rules and regulations about telecommunications system in different countries that must be accommodated by the systems.
- *Interaction differences*. A significant amount of problem solving while developing software systems is done informally at the coffee machine, over lunch, etc. With geographical and temporal separation, these informal modes of interaction are virtually impossible. Given that roughly 75 minutes per day in one project were spent in short (3 minutes or under) unplanned interactions informally solving problem [15], the removal of these informal channels of interaction seriously affects project interactions.

Challenges and Opportunities

The increasingly distributed nature of development and the ubiquitous use of third-party components, along with new integration platforms and tools, have created a

dynamic software development environment in which organizations continue to search for ways to reduce the cost of software development through outsourcing and other means. This process is compounded by the fact that 1) new OSS and COTS components, 2) development tools are continually being introduced and existing ones extended, and 3) there is lack of local control over improvements and problems fixed. Software engineers need to be prepared to deal with unprecedented levels of diversity and change, not only in the technologies and components they will be working with, but also in the nature of the development teams and organizations with whom they will need to interact on a regular basis, and even in the very roles and responsibilities they will need assume as development processes and organizations continue to evolve. The new integration platforms and technologies represent further movement away from the system development processes and architectural styles of the past, which tended to be much more monolithic, toward more loosely coupled integration models that will better enable systems to be composed of components from diverse sources.

The impact of these trends on software engineering projects, processes and practitioners means that the mix of competencies required to practice successfully software engineering now includes much more than just software development skills. In today's competitive global environment, short-term market opportunities and financial concerns increasingly impact software development projects, often causing project requirements and priorities to change on short notice, while simultaneously increasing the pressure to produce software at the lowest possible cost in the shortest possible time. At the same time, the increasing commoditization and distribution of hands-on software development via outsourcing and the use of third-party components is causing the demand for traditional software development skills in many areas to be eclipsed by new opportunities available to software engineers who possess certain kinds of enhanced software *engineering* expertise.

To enable engineers to take advantage of these opportunities, we need to augment traditional software engineering strengths in requirements engineering [11], software architecture [14], design, and development processes with new techniques that will enable software engineers to *apply engineering principles* in larger organizational and project management contexts. We need to produce software engineers who are just as comfortable practicing engineering in environments where the "tools" are distributed project teams and third-party components as they are designing and implementing complete systems themselves. We also need to equip engineers with architecture, design, and development approaches that facilitate third-party component integration, and the simultaneous evolution of product family architectures and external components. Given the right set of knowledge, skills and perspectives, we see this trend toward distributed and third-party development as an excellent opportunity to build on the unique strengths of software *engineering* as a distinct discipline, by equipping software *engineers* to play a central role in ensuring the success of development organizations as they explore these new development models.

In the next section we briefly discuss several new "core competencies" -- attitudes, perspectives, and skills that are not usually considered core aspects of software engineering education, but that we believe will serve new software engineers well as they navigate the increasingly dynamic development environment, and help equip them to flourish throughout their careers as they face whatever unforeseen challenges may

arise in our profession in the future. In the following several sections, we discuss several areas of competency that we believe will be among the most valuable for software engineers, and hence require additional curricula to support. These include *organization and process engineering*, *system and product family architectures*, *integration techniques and technologies*, *product and product line management*, and *distributed project management*. Finally, we briefly discuss several important aspects of ethics as they relate to software engineering education and practice.

2 Non-Technical "Core Competencies"

Non-technical skills and characteristics such as verbal and written communication, social sensitivity, adaptability and creativity have always been important factors contributing to the success of many of the most successful software engineers. The pervasive changes software engineering is undergoing are making it more critical than ever that we prepare future engineers to be excellent communicators (e.g., so they will be able to communicate effectively with diverse groups of stakeholders and developers who may be globally distributed). Further, engineers of the future must be prepared to be extremely adaptable, and to consider solutions that may be "outside of the box" of previous software engineering experience, including their own. And they must be sensitive to the cultural and linguistic contexts in which these projects take place. This section discusses several of these non-technical competencies and characteristics that we believe will be crucial to the success of future software engineers, and that software engineering programs will need to address if we are to prepare our students to leverage the new opportunities that change will inevitably bring to our profession. Since any specific tools, technologies or programming languages we teach student software engineers today are almost certain to be obsolete long before the end of their engineering careers, it is imperative that we also give them the tools and perspectives that will enable them to adapt to whatever the future holds for our profession.

Targeted Communication

Interpersonal communication has always been an integral part of software engineering, since most software projects involve some kind of team interaction, and even engineers working on single-person projects must still communicate with users, managers and other stakeholders at some point. But communication is increasingly critical to the success of software engineering projects, especially given the increased reliance on third-party components. Communications with distributed development team members, project stakeholders, managers and other members of the extended development team should be undertaken as deliberately as any other critical-path engineering task. Software engineers need to learn and practice purposeful or *targeted communication*, directed toward achieving specific results. This requires engineers to understand how software development projects and organizations work, and how the different roles that individuals may assume within projects and organizations

relate to their own projects. It also requires them to have a clear understanding of what and with whom they need to communicate to accomplish a given purpose, what specific effect they wish their communication to have, and how best to accomplish the desired effect.

While traditional technically oriented engineering communications such as requirements documents, UML diagrams, etc. will continue to be important skills, engineers also need to learn to be comfortable and confident communicating in other contexts such as marketing and business planning meetings, as well as meetings with customers and other non-technical stakeholders. And as software engineering becomes more distributed and project-oriented, the "target" or goal of targeted communication by engineers increasingly will include areas that were previously the domain of business managers, especially in the area of obtaining and justifying the resources necessary to complete projects successfully. Project planning and design activities increasingly will incorporate P&L (profit & loss) projections. Whether P&L models remain the province of business managers in a given organization or not, engineers will be under intense pressure to minimize costs by maximizing the use of third-party developers and components. In such an environment of intense competition for project resources, business advocacy and persuasion skills, or at least a basic understanding of business drivers combined with strong communication skills, increasingly will become important for many engineers if they hope to compete successfully for resources for their own projects, much less reengineer and optimize the processes and engineering organizations to which they belong.

Along with traditional and extended engineering skills, engineering training should emphasize communicating engineering models, evaluations, plans and other engineering results both to technical and non-technical stakeholders, especially business managers and user representatives. Engineers have traditionally excelled at presenting technical facts. Targeted communication goes far beyond imparting technical knowledge, and is geared toward communicating in a way that will achieve desired results. This means that the engineer must understand 1) what they want to say, 2) what result or results their project requires that the communication should achieve and 3) how to communicate what they want to say to the target audience so that the communication achieves those desired results. This is why we sometimes refer to targeted communication as goal- or result-oriented, or engineered, communication. What this really means is that engineers need to stop imagining that everybody who reads their documentation or listens to their presentations is also an engineer. To function effectively in modern development organizations, engineers need to understand how to communicate effectively with diverse members of the "extended engineering team", including business management, sales and marketing, users and user domain experts, customer support, and others. Finally, as engineers, we should point out that the ultimate "target" or desired result, of targeted communication is always to enhance the success of the engineering projects with which we are involved.

Professional Habits and Traits

The increasingly dynamic nature of software engineering means that to maximize their chances for success in this environment, future software engineers will need to practice professional behaviors and habits such as adaptability and creative problem-solving in addition to mastering the appropriate tools, technologies and processes. Although such personal habits or traits are widely held to be inborn traits that people "naturally" possess to a greater or lesser extent, when we talk about characteristics such as flexibility or inquisitiveness, we are really referring to related sets of behavioral traits that, if habitually practiced in the context of software engineering projects, will greatly contribute to their success over the long term. Since we are essentially talking about behavioral habits that can be learned, or at least enhanced with practice, for the purposes of software engineering education and practice, it doesn't really matter whether these characteristics are more the result of "nature" or "nurture". Merely adopting these behaviors is sufficient. For example, if we can convince software engineers who are "naturally" inflexible, or who are not particularly inquisitive by nature, of the benefits of making flexible or inquisitive behavior an integral part of their software engineering practice, their projects and careers will still reap the same benefits, regardless of whether they feel "naturally" inclined to exhibit those behaviors.

- *Adaptability*. Software engineering will continue to change, whether software engineers are prepared to deal with change or not, and the most successful software engineers will be those who are the most able to adapt. Indeed, for the foreseeable future at least, the rate of change appears to be accelerating, and we don't see anything on the horizon that is likely to change this significantly anytime soon. Adaptability is really an application of one of the engineering discipline's greatest strengths – the "filter" of practicality. Good engineers adopt the best tools available for a given task at the time, and engineering tools always change over time, even if they did so at a less dizzying pace in the past. Adaptable software engineers grow professionally primarily by abstracting essential principles from past experience, while staying open to new models, paradigms, technologies and methodologies. This allows them to benefit from past experience, but at the same time, never to allow that experience to limit their perspective.
- *Intellectual curiosity*. Along with adaptability, practicing intellectual curiosity is a valuable skill for software engineers. While adaptability is practicing openness and flexibility toward new ways of practicing software engineering, intellectual curious or inquisitive software engineers are always proactively looking for better ways to practice their profession, including new tools, technologies, architectures and processes. And when new models and paradigms are developed, inquisitive engineers will be the first to notice and adopt them, if indeed they are not the ones who developed the new models and paradigms in the first place.
- *Creative problem-solving*. Another important skill that will enhance the success of software engineers in a dynamically changing environment is creative problem-solving. By definition, ongoing and future change implies uncer-

tainty, since no matter how forward-looking we may be, the future we envision will inevitably include unexpected developments (indeed, the world would be a dull place if this were not the case!). While practicing adaptability and intellectual curiosity will help software engineers stay open to new things, greatly helping them to avoid ever "missing the boat" by not noticing or embracing new developments in software engineering, the creative problem-solvers will be the engineers who are driving future change in our profession. Since engineering always concerns solving some problem or set of problems, creativity in engineering is mainly the ability to conceive of solutions that are "outside the box" of normal theory and practice.

- *Knowledge assimilation and categorization*. Finally, to complement adaptability, intellectual curiosity and creative problem-solving, software engineers of the future will need to be very adept at rapidly assessing, assimilating and adopting new knowledge. This knowledge may be in the form of new development models, tools and technologies, or it may be involve new abstractions and paradigms for which we currently lack even the concepts and language to discuss. But the important thing is that whatever form it may take, there are always going to be new models, theories and technologies that a software engineer will need to evaluate to determine first of all, whether they are applicable to the set of problems they are trying to solve, and then, whether they are the best solution available. Since there is no way we can teach new software engineers all the technical knowledge they will need over the course of their careers, we must at least give them the tools and perspective they will need to assimilate new technologies rapidly.

3 Requirements Management and Engineering

In recent years, there has been a trend in many software development organizations for software engineers to focus almost exclusively on software development, with marketing representatives or other "domain experts" increasingly taking responsibility for defining, analyzing and managing system requirements, sometimes even including aspects of system design (e.g., human-computer interface (HCI) design, use case engineering, etc.). Although business-related drivers related to outsourcing have motivated much of this change, a tendency among software developers to avoid being limited to a single application domain or industry by focusing on implementation technologies like programming languages, tools and techniques, while deemphasizing problem domain analysis and expertise, has also contributed to this trend. While the need to involve domain experts actively during requirements gathering and prioritization is a well-established principle in software engineering, ongoing industry trends and upcoming developments in software engineering theory and practice make it more critical than ever that software engineers refocus on requirements analysis and engineering to counter the recent trend toward non-engineers "owning" or managing many aspects of system requirements.

One basic reason software engineers need to refocus on requirements is because requirements are arguably the most fundamental software engineering concern. The functional requirements of the system (often referred to as "goals"), along with the

non-functional requirements (often referred to as "constraints") together form the basis for the architecture and design of the system. While non-engineering domain specialists may have some idea about the overall problem domain functionality of the system, without the system, product line, and architectural perspectives of software engineering, they are unlikely to be able to produce the kind of coherent set of system requirements that can form a robust basis for the system architecture. Since current requirements engineering practices (RE) [11] include analyzing, refining and rationalizing the requirements, all of which involve changing the requirements because of *engineering* concerns, it is critical that software engineers fully understand the central role of requirements in the software development process, and learn the requirements elucidation, analysis and engineering skills that will enable them to play an active leadership role in managing the requirements for the systems they design and implement.

Requirements are becoming even more fundamental to designing system architectures as a result of new architectural design paradigms that are being developed, e.g., rationale and intent-based architectures [5] [4] [1] [2] [8]. Building on recent developments in RE, problem domain modeling [9], and decision-based architectural models [3] [10], under these new approaches, models of the functional intent of the system as expressed in the requirements form the basis for the system architecture. In many respects, under these new system abstractions, the requirements will *be* the system architecture, allowing models based on the functional requirements of the system to function essentially in much the same way as we currently use architectural designs. With requirements poised to become, if not the actual system architecture, at least the integrally connected wellspring and literal basis for the system design, it is all the more imperative that we prepare software engineers to take the lead in requirements analysis and engineering. They must also be willing to "dirty their hands" by embracing application domain expertise and communicating with actual system users and other stakeholders, so they will be able to elicit, analyze and engineer models of system requirements that will enable the system design to fulfill the desired functional intent of the system. Requirements are the ultimate wellspring of system architecture and design.

Also, whenever business unit proxies for actual system users are inserted into the development process as designated domain experts, they can serve to isolate the system architects and developers from meaningful contact with users. While this kind of arrangement may be useful from the standpoint of condensing divergent user requests into a coherent set of requirements and buffering the development staff from being "bothered" by users, unless the user proxy is exceptionally good at requirements elicitation and analysis, the quality of the system may suffer from inadequate requirements analysis. Furthermore, each additional indirection and communication link that requirements must traverse to reach the implementation team increases the potential that requirements may be misinterpreted or misunderstood somewhere along the way.

A final reason we need to reemphasize requirements is related to the "business-related drivers" to which we previously alluded. Transitioning requirements gathering and analysis from software engineering teams to business groups has contributed to the marginalization of internal software development organizations, effectively causing all software development, including in-house development, to be done "to

order", according to specifications developed by the business group. This kind of ubiquitous outsourcing model of product development has been particularly attractive to organizations that are actively exploring or practicing significant software development outsourcing. While it is beyond the scope of this paper to argue the relative benefits and drawbacks of outsourcing in general, attempting to outsource software development without retaining a software engineering perspective in the system requirements specification and architectural design loops is a sure recipe for disaster, including shortcomings in system architecture and design, as well as product line integration problems, among others.

4 Processes and Organizations

As software development becomes more globally distributed, we envision software engineers increasingly utilizing their expertise in architecture, design, and process engineering to take ownership of *engineering-in-the-large*, moving beyond implementation engineering to reengineer the very organizational processes and teams that are involved in conceiving, designing, developing and deploying software solutions. Organizational and process modeling are critical from multiple standpoints. During system design and development, business modeling and simulation techniques are important means of understanding target user organizations and how they will use a software system. Further, they are especially useful for developing, refining and validating system functional requirements. Business and process modeling also are becoming increasingly critical aspects of system architecture and high-level design, as system designs increasingly make use of problem-domain-based architectural abstractions.

In addition to utilizing organization and process models as an integral part of design and development, software engineers must also possess strong organizational and process modeling and optimization skills, techniques and perspectives if they are to reengineer and to optimize their own software development organizations and processes. Pervasive pressure to reduce software development costs, combined with increasing third-party development options, means that software development organizations and processes are continually being "reengineered", whether actual engineers are involved in the process or not. While it is not our intention here to focus on any potential career benefits that may result from being personally involved in managing organizational change, at least relative to the alternative (i.e., being affected by such processes without a seat at the table), as a practical matter, an engineering perspective is critical to ensure that cost-driven reengineering of software development projects, organizations and processes ultimately produces successful results. Organization and process engineering to optimize software development includes determining the optimal organization of distributed projects that may include a mixture of internal resources and outsourcing, as well as third-party (COTS and OSS) components. *Project allocation, estimation and control* present special challenges in this kind of mixed-mode distributed project, and depend on stringent *requirements specification*. Besides new approaches to software development itself, software engineers should understand the entire range of roles in the extended software engineering organization, including business management (e.g., *cost and. risk analysis*), sales and

marketing (e.g., *requirements gathering and prioritization*), customer support (e.g., *usability and supportability*), and others. An understanding of the *intellectual property ownership* and *security* issues involved with incorporating external project teams and components is also critical.

5 System Architectures

The increasing use of distributed and outsourced development, and the resulting need to incorporate third-party components within existing and newly developed systems, makes it more critical than ever that software engineers understand how to design and use system abstractions and architectural models that support the integration of components from diverse sources. Incorporating third-party components from diverse sources also compounds and complicates security and dependability concerns. Techniques and technologies that will enable engineers to design architectures that integrate diverse sets of components into systems that fulfill the functional and non-functional goals and constraints of the system include *product family architectures*, *integration styles and patterns*, *system abstractions* derived from functional requirements, and application *integration frameworks*.

- *Product family architectures*. Incorporating diverse sets of third-party components into existing or newly developed products will require that software engineers understand how to design, develop and extend *product family architectures* [16] that are robust and comprehensive enough to encompass all these disparate elements, and organize them into a consistent architectural view.
- *Integration styles and patterns*. To enable integration at a finer level of architectural design granularity, engineers will also need to be very familiar with *integration styles and patterns* such as *connectors*, *adapters* and *wrappers* so they can use them as building blocks to create architectures that effectively manage the complexity induced by components designed and developed in parallel by diverse teams. These and similar styles and patterns that encapsulate functionality and component interactions within consistent interfaces are also important techniques for simplifying the incorporation of security concerns into complex systems by allowing security functionality to be centralized within the design of the encapsulating framework entities (i.e., the connectors, adapters, wrappers, etc.).
- *System abstractions*. System abstractions that enable implementation architectures to be derived more directly from abstractions of the problem domain, such as *goal- or prescription-based architectures* [2] and *intent-based architectures* [5], will also be important techniques to reduce complexity and enhance architectural consistency.
- *Integration frameworks*. Industry application frameworks such as J2EE [7] and .NET [12], and platform-independent integration frameworks such as web services and service-oriented architectures (SOA), are extremely useful technical skills for software engineers.

In general, system abstractions, architectural styles and design processes that enhance the ability of software engineers to design flexible software architectures with enhanced support for architectural evolution to accommodate changing functional

specifications and integration requirements will become increasingly critical competencies for software engineers as system evolution and integration, along with related security concerns, increasingly come to dominate the set of architectural concerns that engineers must balance.

6 Product and Project Management

Software development is rapidly moving away from the traditional model [17], which primarily involved designing and building systems entirely within a given organization, increasingly utilizing a range of mixed development models incorporating varying levels of third-party (OSS and/or COTS) components and outsourcing, and increasingly including projects that are primarily, or even completely, outsourced. Mixing outsourced component development, where the paying organization retains at least some level of management input into the design and implementation of the contracted deliverables, with OSS and COTS components, where the paying organization may have little or no input into design and implementation, adds to the challenges faced by the software engineer/architect who must somehow integrate these diverse components into a working system.

Managing versus Contributing Organizations and Projects

One result of outsourced development in particular is that a dichotomy is emerging between the role of a software engineer in the *managing* organization for a project (i.e., the organization managing and ultimately paying for a given development project), and that of a software engineer or developer in a third-party *contributing* organization (i.e., an organization that is developing components that will be incorporated into the managing organization's product). The situation is further complicated by the fact that a given organization may be simultaneously managing some software projects and contributing to others. While the emerging trends we highlight in this paper can and do affect both kinds of projects, over the near term, contributing projects will continue to include more traditional hands-on software implementation, while managing projects will move more rapidly toward requiring the kinds of extended software engineering competencies we emphasize in this paper.

Product Management

Product management, or managing the direction and functionality of software products and product families, including many aspects of *requirements engineering (RE)*, is increasingly performed by marketing professionals. Software engineers may only become involved in software development projects at implementation time, after the domain and marketing experts have specified the requirements (and often, the resource budget and project schedule), without considering engineering concerns like reusability and product family architectures. We need to prepare software engineers

to be fully engaged in projects from the outset, when important decisions about project feasibility are made. As approaches like goal-driven prescriptive architectures and intent-driven implementations become more prevalent, requirements analysis and engineering will become the most critical part of the software engineer's job. Software engineers will need to become *problem domain experts* and experts in *requirements elicitation and analysis* to engineer the implementation domain. This will require engineers to work more closely with customers, business managers and other stakeholders, and to be flexible enough to reason about software design and usage from diverse points of view.

Project Management

Project management will also become an increasingly important concern of software engineering as development projects become more globally distributed. Software engineers will increasingly find themselves working on projects where their ability to deliver their product directly depends on third-party development teams delivering components on time and according to agreed-upon functional specifications. Managing software project resources and schedules is notoriously difficult even under the best of circumstances. Adding distributed development teams to the mix will require software engineers to become much better at *planning and estimation*, using advanced techniques for accurately estimating the time it will take the various internal and external teams to complete their parts of the project, including integration and quality assurance (QA). This will also require much better *interpersonal communication skills* than many software engineers currently possess to communicate project requirements effectively to distributed and outsourced team members, and verify that the requirements are satisfied. Software engineers also need a theoretical and practical background in *"agile" methods* such as XP. Agile methodologies challenge traditional design-centric models of software engineering by using test code, user representatives and other relatively lightweight methods to represent the functional intent of the system, as well as changing many aspects of project planning and estimation.

7 Ethics

While professional ethics may not appear to be related to outsourcing and the other developments in the software engineering profession we have been discussing, they bear mentioning here, both because the technical and business environment changes brought about by these developments are related to certain aspects of professional ethics, and also because professional ethics training in general appears to be lacking in many software engineering curricula. While we only discuss several aspects of professional ethics that we believe are most pertinent to navigating the current professional environment, more thorough treatments are available (e.g., in [6]).

The general ethical principle we should impart to our students is to practice *integrity* and *excellence* [18] in every aspect of their software engineering practice. While integrity and excellence are excellent guidelines for life in general, in the practice of software engineering this includes honesty and openness, good stewardship of time

and resources, and excellence in the application of software engineering principles. In the rest of this section, we discuss ways to apply these in software engineering education and practice.

Communication Integrity

Communication integrity includes openness and honesty. In software engineering, this includes being proactively honest and open when reporting project status, not only when the project is on schedule, but especially when problems are anticipated or realized. Software projects are almost never done in a vacuum; any technical, design, or schedule problems are very likely to impact other members of the development team, or even the success of the development project as a whole. In addition, project managers need to know about any problems as soon as possible to implement mitigation strategies, manage customer expectations, etc. That is why software engineers need to be trained to not only be honest (i.e., give an honest answer when asked), but also to be proactive and take initiative in communicating any information that is important to the success of their project.

Another area where communication integrity is important for software engineers is when they are called upon to provide an engineering evaluation or estimate, particularly when the honest answer is not what the listener (e.g., management) wants to hear. Examples include how long it will take to design or implement a given component or product, whether a proposed design or solution should be adopted, etc. More subtle, but equally common, are situations where someone asks the engineer "casually" or "off the record" whether something is possible, can be done within a certain time, etc. Since any statement, no matter how casual, may be repeated or otherwise used in unintended ways, software engineers should treat all project-related communication as an engineering communication for which integrity is imperative. It is important to emphasize to student engineers that communication integrity always leads to better results in the long term. For example, if as a software engineer they go along with a management schedule expectation that their own estimates show to be unrealistic, they will either end up working excessive hours under tremendous stress trying to meet the schedule expectation they failed to challenge, or else they will miss the target delivery date altogether and earn the wrath, however unjustified it may seem, of management.

Ethical Use of Time and Resources

Another ethical area that should be emphasized in software engineering education is good stewardship of time and resources. While the Internet is undoubtedly the most useful research and communication tool ever invented, it is also arguably the most efficient way ever devised to waste time. Software engineers need to be taught (or it needs to be reemphasized) that unless they are being paid as a contractor on a per-deliverable basis, any normal employer-employee relationship includes an implicit, if not explicit, contractual understanding that they will spend at least a certain number

of numbers per week performing the agreed-upon function (the actual number of hours varies by company and by country). For software engineers, this means that they have a professional ethical obligation to spend at least the designated amount of time actually practicing software engineering. Since software engineering schedules often include "slack time", e.g., times when an engineer is temporarily waiting for another part of the project to be finished, or when an engineer finishes his or her part of the project sooner than expected, integrity and discipline is necessary to avoid wasting time, and find something to do that actually contributes to the project. Engineers can also be creative; if they feel burned out or need to do something else for awhile besides staring at the computer, they can always spend some "slack" time helping or mentoring more junior engineers, helping other engineers brainstorm about the design of some aspect of the system, etc. Integrity does not need to be boring; it just needs to be practiced.

Personal Responsibility

Another ethical issue related to integrity is personal ownership and responsibility. Software engineering students should be taught always to take ownership of, and responsibility for, their own actions. This includes the results of their actions, including any projects they work on, software they design or implement, etc. Any blaming of others, or attempting to shift blame, is unethical. It is also counterproductive, because it poisons the software development team environment, and in the worst case, can degrade or destroy the ability of a team to function. The behavioral aspects of personal responsibility, taking responsibility for one's own actions, are included in communication integrity. What we are addressing here is really more of an ethical attitude of personal responsibility that will serve to motivate ethical behavior under any circumstance, than a set of behaviors *per se*.

Engineering Practices

While software engineering education does often emphasize engineering practices like good system design, the principle of integrity should be extended to cover all software engineering activities. The general principle here is to practice engineering at all times. For example, students should be taught never to make any unjustified (or undocumented) shortcuts, claims, promises, etc. All of these should be based on evidence. And valid evidence is always ultimately based on some kind of model, even if it is only a mental model based on experience. We need to equip students with the models they need now and the skills they will need to extend current software design and development models, and create new models, to meet unforeseen future software engineering needs. Technically, what we need to give engineering students is good software engineering domain meta-models, including useful entities, relationships and organizational principles they can use to create and adapt their own models in particular project and organizational contexts. Examples of software engineering domain models include application domain models, requirements analysis

and requirements engineering models, architectural models, design patterns and other design models, project planning and scheduling models, testing and verification models, and even models for more esoteric aspects of software engineering like team interaction and communication. Students should be taught to refine these core software engineering models continually as they gain experience and encounter new circumstances or challenges, and ultimately, to recognize new domains as they arise in the future, and create appropriate models for them.

Intellectual Property

Finally, no discussion of software engineering ethics would be complete without talking about intellectual property. For the purposes of this discussion, we consider intellectual property in the broader context as potentially including any information a software engineer gains while working for a company that may harm the company if it is disclosed. We do not include areas such as disclosure of illegal activity at the company to the proper authorities, where there may be an ethical or legal obligation to disclose information that does prove to be harmful to the company. But even leaving out such cases leaves a wide area where behaving ethically is not only a professional imperative, but may also save a software engineer from legal problems. For example, every software engineer knows, or ought to know, that it is illegal and unethical to steal or take source code from a company without permission.

Where software engineers have sometimes run into problems is cases like using proprietary knowledge (also known as trade secrets) they obtained while working for a company in their own business or while working at another company. Such proprietary knowledge can include details of system functionality and design, customer lists, knowledge of implementation plans and schedules, or any other private company information, whether it is explicitly protected by patent or copyright, or not.

In addition, the nature of software engineering is such that software engineers often have access to personal information, especially if they are involved with building or maintaining databases that contain personal information (e.g., human resources or financial data systems), or if for whatever reason they have root or administrator privileges on any system where personal information is stored. While it should be apparent, software engineers have an absolute ethical obligation not to use or divulge any personal information to which they may gain access. And while this also should be obvious, it is unethical for a software engineer to gain access deliberately to any database, directory, system, etc., in which they have no legitimate project-related business (i.e., all forms of “hacking” are unethical).

8 Discussion

To prepare software engineers to thrive in an increasingly distributed and outsourced environment, we recommend starting software engineering programs with an enhanced introduction to software engineering principles, including material that explicitly addresses outsourcing and distributed development, before any core technical

coursework is attempted. A basic understanding of *software as a service* will provide important conceptual, technical and practical perspective for applying the technical material in real-world situations. The goal is to give students both the tools and the perspective needed to become software *engineers* who design *solutions*, rather than programmers who write programs. Starting with engineering principles gives students a firm foundation in software engineering, avoiding the need to "retrofit" them with this knowledge later on, and also enables the technical course materials, assignments, and projects to reinforce and expand upon these principles. Industry practitioners and managers should also be brought in regularly to provide perspective on how software engineering principles are used in industry.

Software Development Process Curricula

Software development process curricula should emphasize *requirements engineering (RE)*, including *requirements management*, i.e., methods and techniques for managing the expectations of customers, managers and other stakeholders. While much of requirements gathering and prioritization is currently done by marketing professionals, software engineers should be prepared to take a leadership role in RE, taking market concerns into account, and working closely with market researchers and other stakeholders. While software as a service utilizes technical means such as service-oriented architectures (SOA), a complete understanding of the *service* aspects of modern software development and delivery also impacts all aspects of the software development process. Software engineering students need to be acutely aware that they serve a "higher purpose", in that their technical knowledge will only be useful to the extent that they learn to use it to contribute practically to providing a needed service. Curricula should also teach students when and how to adopt aspects of "agile" methodologies and other *iterative approaches* to software development projects. Process education should include *organization engineering*, i.e., methods and techniques engineers at all levels can use to reengineer their organizations to optimize the success of development projects.

Architecture and System Design Curricula

Architecture and system design curricula should emphasize *distributed system architectures*, and *component-connector architectures*, both of which make it easier to incorporate components developed by third parties, whether outsourced, OSS, or COTS. The trend toward providing and utilizing *software as a service* is impacting the way software-based systems are designed and deployed. Architecture curricula should include styles that support software as a service, particularly distributed service provider network architectures, such as *service-oriented architecture*. However, the service-oriented software approach also requires software architects and designers to approach basic system design in entirely new ways. Curricula should teach engineers to provide specifically needed services by developing service provider components, and architects to design applications and systems primarily by composing sets of these services, which may have been developed globally by disparate organiza-

tions. Architecture curricula should also include *product family architectures*, to give engineers tools to develop frameworks that bring order and consistency to systems developed by diverse distributed teams. Application frameworks such as J2EE, .NET, or similar platforms, should be used as practical training tools in their own right, as well as to illustrate, e.g., the difference between a common component model and a product family architecture. And finally, *prescription-based architectural approaches* [2] [5], in which the system architecture is directly derived from the requirements, will give software engineers important techniques to help ensure that systems they design fulfill the specified functional goals and constraints. Specific attention should also be given to *domain-specific* software engineering techniques, methods and tools.

Expanded Project Management Training

Project management education should be expanded to include *program management*, including such business-related concerns as cost/benefit analysis and market analysis, among others. Project management education should emphasize *cost and time planning and estimation for distributed projects*, including projects that utilize *internal and external outsourcing*. Software engineers need to think of themselves as manager-integrators who are comfortable engineering systems using outsourced teams and third-party components, while avoiding inefficient practices like unnecessary hands-on development. Outsourcing often includes international teams, so the effects of *cultural and linguistic issues* on team communication, expectations, etc., should also be addressed. And since the success of software projects largely depends on resource availability (time, engineers, equipment, tools, etc.), project management education would not be complete without addressing *advocacy and resource management* for development projects. Software engineers must be able to determine as early as possible whether the resources allocated to their project are sufficient to enable the project to be successful, so they can reengineer teams and processes, manage requirements and expectations, and do whatever else is necessary to ensure success.

Change Management

Although we have mentioned the need for software engineers to manage change in different software engineering contexts throughout this discussion, we believe this is an important enough topic to merit special mention here. All core software engineering curricula should emphasize the dynamic nature of software engineering. For example, in the real world, projects must often be completed under different organizing principles, using a different set of development team members than the ones who were available when the project began. And requirements often change many times before the project is finished, sometimes invalidating the fundamental assumptions underlying the original architecture and design. This intrinsic need for system evolution is further complicated by the increasing reliance on components developed and separately maintained by distributed teams from different organizations or open source communities, since these disparate components usually also continue to evolve

in parallel. In general, software engineering curricula needs to stop acting as though software development occurs in some kind of a vacuum where a developer or team can use a given set of requirements to derive an architectural design and implementation, without the need to accommodate changes from various sources at inopportune times throughout the process. Some ideas for incorporating change management training into software engineering curricula include changing the requirements for design and implementation projects at different project phases, changing the individual roles and membership of project teams during class projects, and making change management concerns an integral part of all software engineering process and design curricula.

Practical Ethics Education for Engineers

Software engineering curricula should include a course in software engineering ethics, and all core software engineering curricula should encourage students to practice software engineering in an ethical manner by highlighting the practical ethical concerns related to utilizing the technologies and methodologies being taught. However, curricula should avoid presenting software engineering ethics as merely a set of rules, which students might be tempted to dismiss as irrelevant. All software engineering ethical principles and guidelines have very practical benefits, which may include preventing interpersonal or legal problems, or enhancing other aspects of software engineering practice. Preventative benefits of ethical behavior range from preventing various practical or team-related problems, to in extreme cases, perhaps protecting software engineers from lawsuits or other legal penalties. Most of the ethical software engineering behavior we have discussed also yields practical benefits that either directly contribute to the success of software engineering projects (e.g., communication integrity and engineering practice integrity), or at least help prevent practical problems later on. So software engineering core curricula should include the appropriate professional ethical context, and in turn, the ethical discussion should be grounded within a framework that includes practical benefits.

9 Conclusions

For our profession to remain relevant, we need to prepare software engineers to assume leadership roles in engineering system requirements, building solutions using internal and third-party components and distributed development resources, and integrating software elements from these diverse sources into coherent product families using optimal component and connector architectures. To develop systems successfully using distributed resources, engineers will need to learn better product and project management, organization and process engineering, and interpersonal and cross-cultural communication skills. We see these challenges as an excellent opportunity for properly prepared software engineers to play a central role in ensuring the success of software projects and product families, by applying enhanced engineering princi-

ples to manage the architectural and process complexity induced by the current drive toward distributed and third-party development models.

References

[1] Bosch, J.: Software Architecture: The Next Step. In Proceedings of the First European Workshop on Software Architecture (EWSA 2004) (2004) 194-199

[2] Brandozzi, M., Perry, D.: Architectural Prescriptions for Dependable Systems. ICSE 2002 Workshop on Architecting Dependable Systems (WADS 2002) (2002)

[3] Dueñas, J., Capilla, R.: The Decision View of Software Architecture. In Proceedings of the 2nd European Workshop on Software Architecture (EWSA 2005) (2005) 222-230

[4] Grunbacher, P., Egyed, A., Medvidovic, N.: Reconciling Software Requirements and Architectures with Intermediate Models. Software and Systems Modeling, Vol. 3 No. 3 (2004) 235-253

[5] Hawthorne, M., Perry, D.: Exploiting Architectural Prescriptions for Self-Managing, Self-Adaptive Systems: A Position Paper. ACM SIGSOFT 2004 Workshop on Self-Managed Systems (WOSS '04) (2004)

[6] IEEE Computer Society/ACM Joint Task Force on Computing Curricula: Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, http://sites.computer.org/ccse/SE2004Volume.pdf (2004)

[7] Sun Developer Network Products and Technologies: Java Platform, Enterprise Edition (Java EE), http://java.sun.com/javaee/index.jsp (2006)

[8] Divya Jani, Damien Vanderveken and Dewayne E Perry: "Deriving Architectural Specifications from KAOS Specifications: A Research Case Study", European Workshop on Software Architecture 2005, Pisa Italy, June 2005

[9] Hall, J., Jackson, M., Laney, R., Nuseibeh, B., Rapanotti, L.: Relating Software Requirements and Architectures Using Problem Frames. In Proceedings of the IEEE Joint International Requirements Engineering Conference (RE'02) (2002) 9-13

[10] Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005 (2005)

[11] van Lamsweerde, A.: Requirements Engineering in the Year 00: A Research Perspective. In Proceedings of the 22nd International Conference on Software Engineering (ICSE'00), Invited Paper, ACM Press (2000) 5-19

[12] Microsoft Developers Network (MSDN): Microsoft .NET Framework Developer Center, http://msdn.microsoft.com/netframework/ (2006)

[13] The Open Source Initiative (OSI), http://www.opensource.org (2006)

[14] Perry, D., Wolf, A.: Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes, 17(4):40 (1992)

[15] Perry, D., Staudenmayer, N., Votta, L.: People, Organizations, and Process Improvement. IEEE Software (1994)

[16] Perry, D.: Generic Descriptions for Product Line Architectures. ARES II Product Line Architecture Workshop (1998)

[17] SWEBOK: Guide to the Software Engineering Body of Knowledge. www.swebok.org

[18] The Tau Beta Pi Engineering Honor Society, http://www.tbp.org (2006)