Software process in the classroom: the Capstone project experience

Article *in* IEEE Software · October 2002

DOI: 10.1109/MS.2002.1032858 · Source: IEEE Xplore

3 authors, including:

David Umphress
Auburn University
66 PUBLICATIONS **737** CITATIONS

SEE PROFILE

Software Process in the Classroom: The Capstone Project Experience

David A. Umphress, T. Dean Hendrix, and James H. Cross, *Auburn University*

A process-oriented perspective on large student projects guides students in integrating end-to-end life-cycle skills and provides consistency of experience among projects. The authors discuss what they learned after conducting 49 capstone projects.

Imagine the traditional college project course: students bunch together in groups, are assigned a sizable software development task, and are told to "have at it." Instructors expect them to integrate the technical skills they've learned in previous courses, learn to work synergistically as a team, plan and track their work, satisfy their customer—and produce sound software. Yet, more often than not, projects so framed teach their participants yet another way not to develop software.

Some in the academic community might argue that such reality-based software development reflects industry practice. This might be an accurate generalization of the past, but recent trends suggest that a purposeful and disciplined approach to software development is more effective than an extemporaneous one.[1] Organizations (such as Raytheon[2] and General Dynamics[3]) are making conscious efforts to transform projects from initiation rites that yield *next-time-I'll-do-it-differently* lessons into being ventures that promote *this-worked-and-I'll-do-it-again* experiences. They see software development as more than a grab bag of technical undertakings—they envision it as the purposeful orchestration of technical and nontechnical activities. We can mirror this vision—and hopefully the successes—in college project courses if we are prepared to accept what industry is learning. The lesson is that software development requires discipline, and we can foster discipline using well-conceived software processes.

We outline our rationale for moving from a product-oriented approach to a process-oriented one, our experiences in determining suitable process weight, and lessons we learned in attempting to make reality-based software experiences less painful and more real.

Using processes in the software engineering classroom

Using processes to develop software in the college classroom has mirrored industry history (see the "A Brief Process History" sidebar on page 81), although on a much more attenuated and compressed time scale. While most projects adhere to the classical school of development, there is a growing recognition by instructors of the usefulness

0740-7459/02/$17.00 © 2002 IEEE

of process, however meager, for student projects in the academic world.

Although much can be said for the feelings of freedom and independence gained by letting students build software as they please, the simple fact is that most students have not been taught beginning-to-end project skills. Current computer science and software engineering curricular models ensure students have taken a variety of classes relating to programming. However, this does not assure that they have integrated the knowledge they gained in coursework into a comprehensive and functioning whole, nor does it guarantee that they have the social skills required for a successful project.

Using software processes in the classroom helps in three ways. First, the processes describe the tasks that students must accomplish to build software. Suitably detailed, a process model can describe the life-cycle activities: their sequence and starting and stopping conditions. It can also define the project's artifacts, outlining their appearance, content, and the method students should use to build them. In a sense, processes let students focus on the creative task of building software by defining activities instead of wasting time inventing them. Second, processes can give the instructor visibility into the project. Viewed this way, they describe the rules under which the project operates. The instructor can query students on their progress down to the granularity of the process description; students follow the rules outlined in the process to report status to the instructor. Third, processes can provide continuity and corporate memory across academic terms. Instructors can build solutions to general project problems into processes, thus improving the project's educational worth over time.

Capstone projects

We have concluded 49 capstone software development projects, 27 of which used various process-oriented approaches. Each project entailed a team of three to five students developing new software or making significant enhancements to existing software. All the projects involved a customer who used the product after delivery. We conducted the projects at the graduate and undergraduate levels. Graduate teams worked for a year, while undergraduate teams worked for a single academic term. On average, graduate students had five years of software development experience; undergraduates generally had no professional exposure. All students had taken programming-intensive classes and at least one design class before starting their project. Most students had completed a software process class. Projects ranged widely in domain, programming language, and hardware platform; examples included constructing software tools in Java for desktop computers, developing controllers in C for embedded hardware, and building project-tracking tools in Visual Basic for handheld computers.

Software process trends in student teams

Our outlook on process follows the history of industry trends; that is, we started with no process, injected too much process, and then backed off to what we felt was a satisfactory balance of discipline and chaos. Indeed, we found that injecting process into the academic environment requires the same technology transition skills and tools used in industry.

Our traditional capstone projects followed the classical school of software development. Although each team developed software for widely differing customers and domains, the projects were similar in nature: They captured requirements and designs, but seldom validated or verified them. Teams ignored designs in the heat of final deadlines. The bulk of the project work came at the end of the project, and was normally done by a small subset of the team working exhaustive, long hours. Project documentation was pretty, but generally vacuous. It was often relegated to the weaker members of the team as a way of providing them a task that got them out of the way of the software construction. Requirements were scaled down from the initial promises to the customer, usually near the project's end. The product delivered by the team could not fully pass customer-designated acceptance tests. Delivered software consisted of graphical user interfaces with little functionality underneath. Customers could seldom install the software without significant assistance.

Three themes emerged from the postmortem analyses we conducted across the projects. First, teams had difficulty balanc-

We started with no process, injected too much process, and then backed off to what we felt was a satisfactory balance of discipline and chaos.

Skills that students learned in their course work did not scale well to multiple-person large projects.

ing the workload throughout the project. They conducted elaborate requirements statements and designs, but then discovered that they had underestimated the technical prowess necessary to implement the product within the project time frame. In most cases, their designs relied on mental models of how they thought software components—either prebuilt or organic—worked, only to discover later that those models were incomplete, too simple, or erroneous. Because they typically followed a waterfall-like life-cycle, they discarded their carefully crafted soft upstream project artifacts when building hard deliverables. Even when teams built software in iterative cycles, they tended to first concentrate on features they knew how to write and delayed working on technically difficult or complex features. Projects thus consisted of large amounts of unproductive work punctuated with short bursts of intense development.

Second, skills that students learned in their course work did not scale well to multiple-person large projects. Configuration management was the most obvious stumbling block. Many teams did not have a mechanism for controlling their source code, claiming that such measures added unwanted overhead. Instead, they placed their code in a central repository, copied needed files to their local workspaces, made changes, and then replaced the files without regard to whether the original contents had changed in the interim. They depended on email and word of mouth to control inadvertent file destruction. Not only did these informal communication methods break down with erratic work schedules, but the overall scheme of continually evolving the software prevented falling back to known baselines in times of trouble. Additionally, teams reported that they did not track defects, especially in the project's final stages, and often did not know if bug fixes were ever incorporated into the delivered product.

Finally, team members' responsibilities were not clear. Although all project participants had some technical background, few had training in team dynamics. At the project onset, teams invariably organized around an egoless model without a formal leader and few defined roles. They rationalized that this model advocated the most polite form of team government: students generally felt uncomfortable about presuming a self-appointed leadership role and waited for a natural leader to emerge. They felt similarly about other team responsibilities, assuming that team members would naturally gravitate to whatever tasks felt most comfortable to them. Although some teams contained the right personalities for this to work, most teams quickly became rudderless committees with few clear lines of accountability for project activities. Team members with weak skills or shy personalities eluded productive work by shielding themselves behind ambiguous role expectations. Members with strong personalities, not wishing to receive a poor grade, compensated for weaker members by taking on additional work, thus paving the way for ill feelings. Once the personality of the team finally emerged, it was often one of cliques, contention, and miscommunication.

Interestingly, during project postmortems, technical programming skills surfaced as only a minor concern. The team participants felt they were adequately equipped to write code. What they lacked were skills to deal with integrated project issues. They needed guidance—at some level of abstraction—on how to develop software. In other words, they needed a process.

Process 1: MIL-STD-498

We chose to adapt MIL-STD-498[4] as the first process model, abandoning an initial attempt to cobble together a software development process from scratch as too contrived. We chose this standard because it was actively used at the time, described information requirements of numerous software life-cycle activities, and was free.

Working with the customer and the project supervisor, each team spent the project's first week editing the MIL-STD-498 document specifications to meet project particulars. This period's goal was to have each team define how it would conduct project business. In reality, this did not happen. Just as traditional teams spent precious upfront time putting together intricate but unrealistic designs, these new teams developed elaborate process descriptions that they soon jettisoned as too complex. Projects devolved to ad hoc development, but with standardized documentation.

In retrospect, choosing MIL-STD-498

A Brief Process History

Two disparate schools of thought regarding software development have emerged from the software industry over the past 25 years. The classical school views building software primarily as a *product-driven* activity, one that treats software as a "thing" with little regard to how it was developed. It professes that anyone with suitable programming skills can write software and that activities leading to a software product are free form and chaotic by nature. This school obtained and retains its outlook from programming's early days, a time in which software development was a cottage industry that depended on expert artisans to craft software. Its hallmarks are reliance on gurus and heroes, homegrown methods and techniques, and a body of knowledge that is largely anecdotal.

A contemporary school of thought appeared in the 1970s. This school drew its ideals heavily from the manufacturing sector and, as such, emphasized that the way software is built determines, to a large extent, the end product's quality and the developers' quality of life. This view of software as a *process-driven* activity recognizes the value of technical prowess central to the classical school but suggests more. It proposes that software construction is not a free-for-all but rather is best done under circumstances in which relationships between key development activities are defined explicitly. These relationships, also known as *processes*, identify the tasks needed to produce a software product of known quality.

Most projects in industry today fall on a continuum somewhere between the classical and contemporary camps. Of particular interest is what portion of the continuum the industry emphasizes at any point in time.

The industry primarily focused on the classical school in programming's early days. Methodological processes describing the technical steps for building software appeared in the 1970s. These processes, combined with the industry's interest in software life-cycles, pushed the philosophical focus away from pure programming classicism. This shift became more pronounced in the 1980s with the US government's move to contain costs in embedded system software. This was manifested with the Ada movement and its attempts to marshal frameworks for common methods and tools. DOD-STD-2167 and subsequent military and industry efforts to standardize high-level life-cycle activities fueled the momentum toward the contemporary school during this time. The 1990s saw possibly the height of the process dialectic with the application of the ISO 9000 series to software development and the Capability Maturity Model for Software. Numerous process models appeared at this time, including MIL-STD-498 and IEEE 1074 (for a comprehensive history, see Yingxu Wang's and Graham King's *Software Engineering Processes: Principles and Applications*[1]).

Focus on the contemporary philosophy took an interesting turn of events in the mid-1990s. Until this point, the working assumption was a more-is-better attitude in defining the way in which software is built. Standardized and de facto process models touched on virtually every development aspect. Empirical evidence showed that process discipline improved projects' ability to contain cost, schedule, and defects;[2] but this discipline came at a cost. Operating an organization at CMM Level 5, fully compliant IEEE 1074, or other so-called heavyweight process meant overhead in the process's care and nurturing. Many developers felt encumbered by unnecessary bureaucracy in strongly process-driven efforts.

This led to a grass roots backlash in the late 1990s and early 2000s against heavyweight processes to process models that were not as confining. Lightweight processes—now known as *agile* processes—appeared to establish equilibrium between the classical and contemporary extremes, consequently shifting the development focus to a more nonpartisan part of the philosophical continuum.[3] Processes and process patterns, including Extreme Programming, Scrum, Adaptive Software Development, and Crystal, are currently embraced by the industry as ways of providing some of the contemporary school's engineering discipline, while leaving room for the classical school's raw creative horsepower.[4]

References

1. Y. Wang and G. King, *Software Engineering Processes: Principles and Applications*, CRC Press, Boca Raton, Fla., 2000.
2. S. McConnell, "The Business Case for Better Software Practices 2002 Keynote," 2002, www.construx.com/BusinessCaseForSoftwarePractices-Keynote.pdf.
3. M. Fowler and J. Highsmith, "The Agile Manifesto," *Software Development*, vol. 9, no. 8, Aug. 2001, pp. 28–32.
4. M. Fowler, "Put Your Process on a Diet," *Software Development*, vol. 8, no. 12, Dec. 2000, pp. 32–37.

was naïve. We thought team participants would be able to define how to conduct a project if they knew what content they needed to produce for project documents. Taking the view that documentation is a side effect of the software development effort, we thus thought our students would use the documentation templates as guides as to what activities they should perform.

For seasoned software developers, this is not an unrealistic expectation. However, the student teams perceived the main focus of their work to be documentation, not software. The standard outlined the content of artifacts resulting from each life-cycle stage, but it did little to help them identify and organize development tasks that would produce the artifacts. Project postmortems showed that the generic MIL-STD-498 was too abstract to achieve reasonable consis-

Students appreciated the high-level discipline that the TSP imposed, but they balked at the details.

tency among teams without a significant amount of added detail. Although teams that used MIL-STD-498 for their project refined the standard slightly for subsequent teams, it retained a document-centric aura that crippled its acceptance by the students.

Process 2: IEEE 1074

We tailored the documentation templates of MIL-STD-498 to the bare minimum and then turned to IEEE 1074[5] to add an activity-centric flavor to the software development process. We chose this because it enumerated a development project's activities and described the interchange of project artifacts among those activities. Thus, IEEE 1074 described project tasks and MIL-STD-498 described document formats.

Teams began their projects by defining their own procedures for conducting major technical development activities; that is, they defined their own Software Life Cycle Model Process and Development Process portions of IEEE 1074. We provided them with procedures for performing activities with which they had little experience, such as configuration management, product installation, project monitoring, and so on. In short, we gave them instructions on how they should carry out the standard's Project Management, Pre-Development, Post-Development, and Integral processes.

Here again, the teams were overwhelmed with the process's weight. The portion of the process we provided was a hefty 41 pages, to which we expected the project teams to add their individual process descriptions (usually amounting to approximately 10 pages). Our document shrank to 20 pages over two years, but never seemed to be distilled to a kernel that could capture the students' imagination and dispel their suspicions of unnecessary bureaucracy.

Like the MIL-STD-498 experience, teams using IEEE 1074 developed procedures that were well intentioned but unrealistic. Team participants had been writing code in an ad hoc fashion throughout their college education; consequently, defining technical tasks that team members could carry out uniformly was unfamiliar territory. Team processes typically polarized: they were either so vague that they were of little guidance, or so legalistic that they were impractical to implement and monitor.

The intended effect of using IEEE 1074 was to show the teams that they could apply discipline to software development. However, because they had not experienced processes independent of structured classroom assignments, at the project's outset, the teams focused their attention more on satisfying the process than on building a product. On average, several weeks passed before they adapted and adopted the process. Because of process overhead, IEEE 1074 teams produced the same amount of software in an entire project as the ad hoc development teams produced in several weeks, thus raising skepticism among the students about a process-oriented approach's value.

Process 3: Team Software Process

Realizing that the process was encumbering the product development, we moved next in the direction of lightening the process weight by adopting the Team Software Process.[6] Our students had more success with this because the process defined explicit project scripts and team roles. The TSP was out-of-the-box ready—students did not have to add their own process descriptions.

Project teams understood the entire process within the project's first days and could adjust it when needed. The time they spent on the project was no longer concentrated at product delivery time but was distributed more evenly across the project. The cyclic activities that the TSP promotes let the students build, test, and deliver software in manageable increments. Moreover, the end-of-phase reviews that it prescribes gave convenient points for the instructor and the customer to provide feedback to the team.

The downside of the process was in the bookkeeping. Students overwhelmingly rejected the myriad forms that the the TSP requires, even when those forms were available electronically. They felt imposed on to follow the personal processes that the TSP prescribes—so much so that we came to doubt the veracity of process data we collected by the project's end. Candid post mortems on these projects showed that the students appreciated the high-level discipline that the TSP imposed, but they balked at the details.

Process 4: Extreme Programming

We turned to Extreme Programming[7] in an effort to break out of the TSP's heavy reliance on staff work. XP had the TSP-like advantage of being simple and understandable. It also advocated an overall development discipline in which project teams had to produce demonstrable results at regular intervals. Unlike the TSP, in which scripts describe explicit instructions on conducting various process activities, XP gives general guidelines. XP philosophy reduces software development processes to the bare essentials. It stresses a working product over elaborate documentation and measurements. It was also on the cusp of the classical school of thought; only its reliance on established, predefined commonsense activities prevented it from degenerating into chaos.

XP's emphasis on getting software working early in the project and its relaxed approach to requirements and design appealed to our project teams. It provided a suitably welcoming bridge from the extemporaneous software development of the classroom to a more disciplined mindset required for the project. Like the TSP, XP's incremental cycles smoothed the students' effort across the project. That we described the XP process using guidelines rather than prescriptive rules was not an obstacle. The guidelines were sufficiently detailed and intuitive enough to let the teams know what they should be doing, yet abstract enough to let the teams choose how best to carry them out. Finally, project success—which we measured by how the finished software met customers' expectations—surpassed that of previous projects, whether they had used a process or not.

Conducting the projects using XP was not without disadvantages. First, the project teams collected little statistical process data. XP's project velocity let teams measure progress, but the metric was not detailed enough to provide any insight into process improvement. The project velocity was simple to measure and difficult to conceal, thus providing an accurate glimpse into project progress, but we had to augment it with other metrics. Second, student class schedules made XP's pair programming activity unrealistic. Several project teams experimented with programming in virtual pairs using online collaborative tools, but they stopped when their frustration with the tools outweighed the perceived advantages of pair programming. Third, our students underestimated XP. They mistook its informality as an invitation to seat-of-the-pants design and coding. We took special efforts to point out that although XP's published descriptions seem casual, its mechanics of system metaphors, design refactoring, delayed optimization, and configuration management require careful attention.

Lessons learned

Table 1 summarizes our experiences with software processes, showing that the academic environment is not so different from the industrial one. Indeed, it mirrors in miniature industrial trends and attitudes toward software development—software developed using the classical school of thought was not meeting our customers' needs. Our developers could not produce a consistently high-quality product by carrying out project activities extemporaneously. For us, heavyweight processes—that is, processes that were highly detailed and prescriptive—reduced project ad-libbing; however, they introduced a level of bureaucracy that interfered with product development. Ironically, like the classical school of thought, the end result was software that did not meet our customers' needs. We had to find a suitable process weight that balanced what was being built with how it was being built. This then gave us sufficient control over projects to maintain a relatively consistent level of software quality and a relatively consistent educational experience for the students.

So how can educators infuse process into projects? We have five suggestions.

Develop a process culture

Introducing processes into the classroom environment is not easier than injecting them into the workplace. It requires careful work with all project stakeholders (students, instructors, and customers) to educate them on the processes' purpose, obtain commitments to abide by processes, identify process boundaries, and so on.

Seek agility

Students typically enjoy coding, but not analyzing, designing, testing, and communicating. Agile processes, such as XP, are easiest

Introducing processes into the classroom environment is not easier than injecting them into the workplace.

Table 1

Process evolution over 49 capstone projects, with significant lessons learned

	Ad hoc	MIL-STD-498	IEEE 1074	Team Software Process	Extreme Programming
Experience level	22 projects over 10 years	5 projects over 2 years	10 projects over 4 years	4 projects over 1 year	8 projects over 1 year
Advantage	Rite of passage	Insight into artifact content	Insight into network of project activities	Understandable process	Emphasis of commonsense approach
Disadvantage	Product seldom met customer expectations	Documentation-centric outlook	Complexity	Paperwork	Perceived informality
Feature adopted and carried forward	Projects with actual customers	Documentation templates	Process framework	Cyclic development, team roles	All except pair programming
Biggest lesson learned	Little control over project variability	Knowing documentation content does not necessarily lead to knowing project activities	Overwhelming unless tailored	Process measurement accuracy decreases as number of measures increases	Refactoring, configuration management, and design simplicity are deceptively difficult

for students to accept and perform because the processes focus on working software as the main artifact. In contrast, a heavyweight process gives the impression of ponderous bureaucracy. Matching the process weight to the students' abilities, expectations, and tolerance is vital to a project's success. None of the process models we constructed are bad; in fact, we used bits and pieces from each process model in the subsequent one. We found that, in our environment, lighter process models more closely matched our students' culture. Specifically, XP was a suitable process because it was malleable enough to fit the diversity of our development efforts and the variety of our students' skills.

Develop a process infrastructure

Giving the students a written process description is not enough. Instructors must also provide them with process orientation and resources they can use when they have a question. Most importantly, students must have the tools that support process activities. This includes adequate hardware and software for development, configuration management, testing, method support, and so on. We caution others attempting this that the infrastructure must support the process, not subvert it. Tools can influence the way in which students carry out process activities. An ill-conceived collection of tools can encourage actions the process does not allow, resulting in a tool-process impedance mismatch; on the other hand, wisely selected tools can actually promote process discipline.

Use processes to focus learning

Few students are skilled in all the technical and nontechnical activities that project work requires; consequently, instructors should develop processes that focus students' efforts on project learning objectives. For instance, instructors might construct a project process to give only light guidance to tasks familiar to students. It might give detailed, prescriptive guidance to unfamiliar tasks that, if students invented it, would detract from their learning objectives. It might even describe a "meta" process for having the students define their own tasks for a particular segment of the project.

Seek realism

Like its counterpart in industry, process enactment in the classroom requires enforcement and adjustment. Instructors should expect students to know the process and adhere to its guidelines. In-project process audits are effective tools for assessing process use and perception. When a process-related problem arises, a mechanism should be in place to change the process so that the problem is not repeated in subsequent projects. Students should not be denied success because of a poor process; similarly, they should not blame the process for their lack of success.

We came to the hard realization that we were faced with the challenge of infusing a new technology—software processes—into the classroom. Contrary to the conventional wisdom that we could dictate to our students how we wanted them to write software, we had to carefully couple the software process to the students' abilities, expectations, and cul-

About the Authors

David A. Umphress is an associate professor of computer science and software engineering at Auburn University. His research interests are software processes, requirements engineering, and software engineering education. He received his PhD from Texas A&M University. Contact him at the Department of Computer Science and Software Engineering, 107 Dunstan Hall, Auburn University, AL 36849; umphress@eng.auburn.edu; www.eng.auburn.edu/~umphress.

T. Dean Hendrix is an associate professor of computer science and software engineering at Auburn University. His research areas are software engineering, software visualization, and reverse engineering. He received his PhD from Auburn University. Contact him at the Department of Computer Science and Software Engineering, 107 Dunstan Hall, Auburn University, AL 36849; hendrix@eng.auburn.edu; www.eng.auburn.edu/~hendrix.

James H. Cross is professor and chair of the Department of Computer Science and Software Engineering at Auburn University. His research interests include software engineering environments, software visualization, and object-oriented methodology. He received his PhD from Texas A&M University. Contact him at the Department of Computer Science and Software Engineering, 107 Dunstan Hall, Auburn University, AL 36849; cross@eng.auburn.edu; www.eng.auburn.edu/~cross.

ture. None of the processes we used were inherently poor; we had to find the particular process that matched our culture and let us achieve the educational objectives of the project course.

In the end, we found that introducing processes into our capstone project courses benefited our instructors, students, and customers. Instructors have increased visibility into projects, students have guidance on how to conduct themselves, and customers have a better engineered product.

References

1. S. McConnell, "The Business Case for Better Software Practices 2002 Keynote," 2002, www.construx.com/BusinessCaseForSoftwarePractices-Keynote.pdf.
2. P. Bowers, "Raytheon Stands Firm on Benefits of Process Improvement," *CrossTalk*, vol. 14, no. 3, Mar. 2001, pp. 9–12.
3. M. Diaz and J. King, "How CMM Impacts Quality, Productivity, Rework, and the Bottom Line," *CrossTalk*, vol. 15, no. 3, Mar. 2002, pp. 9–14.
4. *Defense Department MIL-STD-498, Software Development and Documentation*, Washington, D.C., 1994.
5. *IEEE Std. 1074-1997, IEEE Standard for Developing Software Life Cycle Processes*, IEEE Press, Piscataway, N.J., 1998.
6. W. Humphrey, *Introduction to the Team Software Process*, Addison-Wesley, Boston, 2000.
7. K. Beck, *Extreme Programming Explained*, Addison-Wesley, Boston, 2000.