Journal of Information Technology Education: Innovations in Practice **Volume 8, 2009**

Concurrent Software Engineering Project

Nenad Stankovic and Tammam Tillo
Xi'an Jiaotong-Liverpool University,
Suzhou, Jiangsu, PR China

Nenad.Stankovic@xjtlu.edu.cn; Tammam.Tillo@xjtlu.edu.cn

Executive Summary

Concurrent engineering or overlapping activities is a business strategy for schedule compression on large development projects. Design parameters and tasks from every aspect of a product's development process and their interdependencies are overlapped and worked on in parallel. Concurrent engineering suffers from negative effects such as excessive rework and increased social and communication complexity that negatively affect gains.

In the university environment, however, these difficulties and negative effects, if controlled, can help in promoting our educational goals such that they should be exploited rather than avoided. Although linear (i.e., waterfall) has been the most often used model in teaching, time constraints and an opportunity-driven learning process should make the concurrent model suitable for student projects. This paper elaborates on these ideas and reports on our students' experience.

Keywords: architecture, concurrent engineering, software engineering, teamwork.

Introduction

Most engineering problems are not tame but wicked (Rittel & Webber, 1973) or unbounded, meaning that there is no right or wrong solution such that quality becomes hard to assess before implementation. The design processes becomes complex due to new technologies, task interdependencies, and communication and coordination needs between people; so projects have to repeatedly search for satisfying solutions and deal with uncertainty. These present technical and nontechnical challenges that, in turn, require educated and experienced professionals who can produce quality products on schedule (Nikkei Business Publications, 2003).

A crucial factor in successful projects was ongoing client participation and commitment (Terry & Standing, 2004), as was mutual respect and synergy between team members. The impact of interpersonal conflict was negative, regardless of how it was managed or resolved (Barki & Hartwick, 2001). Of course, good domain knowledge and technical skills are important, too. Unfortunately, the type of training being provided to software engineering managers at the university level results in students knowing how to use the tools, but not necessarily knowing why they are important, or what their role is within the effort (Peters, 2003).

Process and project management and organizations are knowledge areas in the software engineering curriculum (e.g., The Joint Task Force, 2004), so similar problems and outcome can be expected there, as well as in other knowledge areas. For example, learning to program is essential for every engineer but gener-

Editor: Janice Whatley

ally considered hard, so programming courses often suffer high dropout rates. It might take up to ten years for a novice to become an expert programmer (Soloway & Spohrer, 1989).

Even on a much smaller time scale of days and weeks, acquisition of knowledge is not linear. It is an opportunity driven process different for each person and affected by social complexity (Figure 1.) (Conklin, 2006). Successful designers iterate frequently through various design stages rather than using a linear (i.e., waterfall) process in which a downstream activity follows the immediate predecessor only when it has become completed (Guindon, 1990). The frequency of the iterations depends on the person's familiarity with the problem and solution domain (Cross, 2004). The process of engineering design is not a totally formal affair, and drawings and specifications come into existence as a result of a social process (Ferguson, 1992). However, managers and engineers are trained to plan for one activity or task at a time instead of a set of concurrent activities, so they assume linear progress (Figure 1), and apply feed-forward project planning methods, such as PERT or Critical Path.

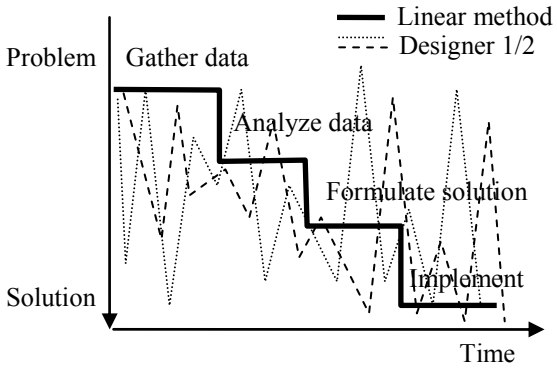

Figure 1. A wicked project with two designers

Linearization is also found in education where we simply proceed from one topic to another as appropriate. That is not negative by itself because some courses may not require much creative thinking or reflection, but it creates an expectation that all coursework is a simple transaction between the instructor and the student. In a transactional system, most students act in response to the extrinsic motivation by simply working on their assignments as they come and performing the best they can at that moment.

Courses that deal with complex problems and processes benefit from rework because this directly links together the phases and work products that have been worked on. Those courses should be accompanied by a transformational approach to coursework, and students should be encouraged to revisit and improve their past coursework and resubmit it. By looking back, students form a deeper understanding and learn that rework and constant improvements are important in real world projects.

Active learning, or learning by doing, has been used in academia and training for a long time. Learning becomes active when students employ their creative skills during the learning process. Schon (1987) suggests that engaging students in analyzing and solving complex problems promotes the habit of logical thinking and problem solving. A major report stated that engineering education would have to be redesigned to emphasize teamwork as well as individual effort, such that students become prepared to meet the demands of the workplace in a complex technological society (Pister, 1993). Teamwork requires alignment, communication, initiative, and group knowledge (Senge, 1990), and workplace implies quality, shared responsibility, and office automation tools.

To address these ideas and findings, we organized a large student project in support of our software engineering course. This paper describes the approach and discusses the outcome from multiple perspectives.

Analysis

This paper describes the new approach to organizing a large student project with about 70 students and reports on our immediate experience. The laboratory project presents the fourth-year student with relevant technical, organizational, and managerial problems, all of which suit this environment. It adopts a flat organizational model and follows the principles of concurrent engineering. The project explores whether benefits that concurrent engineering brings to product development can be used in education. It aims at meeting the objectives as defined by the relevant academic initiatives and workplace needs.

This laboratory project takes one semester and covers the full software development lifecycle. The approach is based on our understanding of the university environment, which is very different from any real-world (i.e., professional) organization, and, therefore, it should serve our educational goals better. Likewise, this project does not require participation of a professional IT company because students may find that a distraction. Companies often adjust models and processes to their needs and make decisions at many levels. Some instructors, for example Alzamil (2005), who tried to organize a project sponsored by a company, found that due to the poor quality of students' work few companies would participate and their commitment was weak.

Organizational Model

Some instructors emphasize the process (e.g., Bernhart, Grechenig, Hetzl, & Zuser, 2006) and some use real world development methods, such as agile (Coppit, 2006), etc. Students, however, go to an instructional laboratory to learn something that practicing engineers are assumed to already know (Feisel & Rosa, 2005). Therefore, our goal is different and so is our approach. Our understanding is that the constraints imposed on student projects are not found in corporate environments. In the classroom, all students learn the same program, and they must satisfy the same pass-fail criteria, i.e., they do not specialize in one function or topic only. The differences and similarities between the university and the industry must be identified and resolved such that it becomes easier to achieve our goals and evaluate outcomes.

Universities are not professional companies with a multitude of roles, skills, and mechanisms of control. This is a decentralized, all-inclusive, and nonhierarchical environment, which makes expert power the only likely source of power and leadership among students. Other sources of power might become counterproductive and lead to conflicts. Even expert power, such as students with strong programming skills, is not necessarily an asset because those students may try to hijack the project. Therefore, our students are appraised as a team and individually. They share equal opportunity to engage, participate, and learn.

Most students have preferences regarding technology, such as databases, Web, or computer graphics. We assume that the average student neither understands the real world project roles, such as project manager, quality manager, software architect, etc., nor has the skills to fulfill those roles. If anything, they are only learning about the functions that those roles perform. Instead, students work in self-managed teams, plan together, and motivate and control each other. They gain insight into all project and lifecycle activities because that facilitates our educational objectives and their needs to learn. Large projects use multiple teams, so each team appoints a lead (i.e., a representative), and they form a coordinating team.

Process Model

Waterfall is the most often used style in teaching, wherein the topics are laid out in a sequence, as appropriate. We cannot explain all important concepts at first and in the same time completely cover the syllabus in a week or two. The software development lifecycle does not appear important if we do not understand the phases and activities, and those depend on the tools and paradigms. Although a known educational pattern states that important concepts should be taught early and often (Bergin, 2000), we find its merits relative. An iterative teaching model could be used to teach a programming course (e.g., Barnes & Kölling, 2006), but it is restrictive and controlled, which is counterproductive in systems development. How much knowledge and work is sufficient at any point in time to complete a task when the problem is unbounded? Waterfall courses can avoid such questions if they do not micromanage or linearize coursework.

On the other hand, can a process model (e.g., opportunistic) that is different from the lecture style (i.e., waterfall) be used in the laboratory? We expect that by improving, updating, and resubmitting their coursework, students will create an opportunistic, iterative, and interdependent software development environment. In our teaching plan we first cover the lifecycle phases (Figure 2) and briefly introduce project planning, because the managerial topics are taught last. Although students lack formal knowledge of software engineering processes and techniques, they should have programming and some design experience from their previous courses that dealt with the different areas of computing. This way, students can work on their project, and use the lecture layout as a guide to their long-term and ongoing planning.

In any case, the lack of knowledge to make qualified decisions and skills to execute is a constant threat, and it is understood that:

- Students cannot make far-reaching decisions that would make a positive outcome impossible.
- Students stay focused on the system.
- Students learn by immediately applying the new knowledge.
- Students learn to handle uncertainty and tolerate ambiguity.
- Students experience team inertia and decision making.
- Teams remain constantly engaged and communicate at different design levels.
- Instructors provide additional guidance when necessary.

The main challenge for the instructor is to ensure that the project is doable given the mentioned constraints and expectations. Any project that fails at an arbitrary point (e.g., a sink or swim project) will likely miss its educational objectives, but it does not have to be 100% complete either. Students do not build a fully functional system but gain exposure to each and every aspect of a full lifecycle system development project. Tradeoffs must be made between what is readily available at the start and what students must work on. In particular, architecture is concerned with the structure of a system and the relationships among its components. It is focused on the system as a whole, and it is a far-reaching decision for which, if the system is not trivial, undergraduate students lack knowledge. As explained below, this is not a simplification but an expectation and the usual approach on concurrent real world projects.

Preparation

Industrial product development employs many people and teams, and successful companies must be consistent in organizational structure, technical skills, problem solving, culture, and strategy (Clark & Fujimoto, 1991). Nowadays, concurrent engineering is a key trend in product development in many industries because it enables schedule compression in order to reduce cycle time.

Toyota Motor Corporation has been at the forefront of this trend. At the same time, Toyota has been very innovative in its approach to concurrent engineering and organizational setup among others (Sobek, Liker, & Ward, 1998).

Concurrent engineering is a team-based process that allows overlapping downstream and upstream activities where tasks are based on the system (i.e., product) architecture. Tasks can also be overlapped within a stage, e.g., parallel design of multiple components. Another key feature of concurrent engineering is collocation of the team members, even though Toyota tends not to do this. Concurrent software engineering has not been explored and used much. It also involves multidisciplinary teams that overlap activities and iterate while converging on the product. The team consists of users, designers, programmers, testers, and other personnel. Figure 2 illustrates the flow of the activities within the software development lifecycle and how they become overlapped in time and iterated across. It also suggests that multiple if not all subsystems are being developed at the same time.

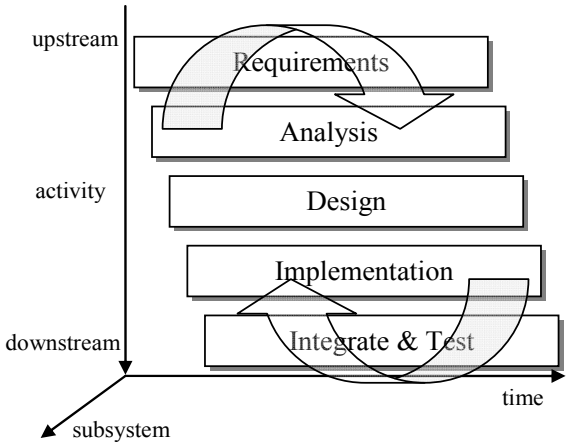

Figure 2. Overlapping activities

Because the activities in concurrent engineering projects are overlapped they all evolve at the same time in small steps, which requires frequent exchange of information in small batches supported by concise and standardized documents, tools, and communication technologies. On the downside, rework due to problem-solving oscillations can consume up to 50% of the engineering capacity (Clark & Fujimoto, 1991), which can cause our time-boxed project to fail early. Thus, we mitigate its impact via our architectural and organizational decisions. But for those less experienced, rework reveals dependencies between activities and work products and makes them relevant. Concurrent development assumes many loops, and project size reflects negatively when implementing a change because it gets more difficult to communicate and coordinate. Efficient communication and coordination mechanisms flexibility, prototyping, and understanding are all important. A distributed system, such as the one described below, is even more challenging because it is not easy to surmise the whole (Mihm & Loch, 2006).

System Architecture

The importance of system architecture (Figure 3) manifests itself in many ways, such as integration and modularity of function, project communication and organization needs, task definition and interdependence, technology, etc. Paulish (2002) states that projects should allocate 40% of their development time for design work, which may take up to three months for high level design alone, and the same percentage of project time is required for testing. Given these, our timeframe of one semester, and students' skills, this project is impossible. Therefore, the architecture must be defined before the start of the project, which facilitates our educational goals because students

learn from it, and it becomes instrumental during integration since each student must contribute code. To that end, students are taught how to build (and stabilize) a system incrementally.

A *Toyota example* may be helpful. It is known what components make up a vehicle and the regulatory requirements. The interfaces between those components are also known, such as how to mount a wheel on a chassis. Projects that design and build new vehicles can be organized and started without finalizing the details, such as styling, exact dimensions and weight, and type of engine. Those will be determined in due course depending on consumer feedback, knowledge, manufacturing capabilities, resources, time, etc., as well as dates, the process model both externally and internally, and tasks. The customer is not only a user but also another team on the project, and they can all affect requirements at any point in time. These principles hold irrespective of what vehicle is being designed.

Simple and well-defined modules and tasks facilitate communication, and the process model must be appropriate for the development environment. Real world projects benefit from weakly coupled or decoupled tasks and separation of concerns. Here, we prefer coupled tasks because they are worked on in parallel and, thus, promote collaboration, reuse, and shared responsibility. Software systems are much more homogeneous than vehicles, and they are easier to manipulate. For example, a chassis and engine have nothing in common, except for input parameters interdependencies in design. In software, it is possible to engineer a system, its components, and consequently development tasks to our needs, as we have done in this student project.

Approach

A goal of tasks definition is to minimize interactions between them due to unresolved or new design parameters and issues. Independent tasks are easy to schedule and work on in parallel to compress the schedule. In student projects, though, this is not desirable because a standalone module may require teamwork, but that team needs to know nothing about other parts of that system or other teams. Figure 2 reveals that lifecycle activities are sequentially dependent, which means that an upstream activity supplies information and the immediate downstream activity consumes it. Often, there are problems that must be resolved across tasks or activities which makes them coupled and subject to loops, as shown in Figure 1. To manage and converge quickly on a solution or through an iteration, those concerned must understand the problem and how to resolve it in a coordinated manner.

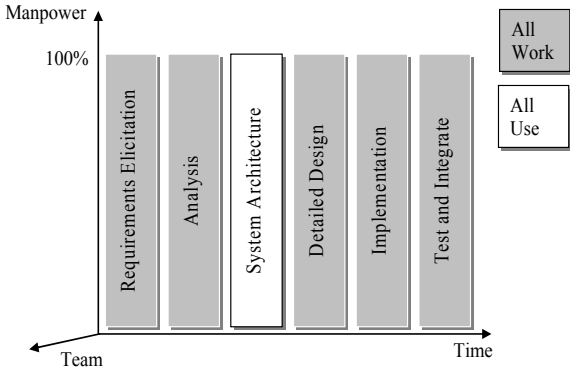

Figure 3. Lifecycle activities and loading

Figure 3 presents the software lifecycle and its realization in terms of resources and division of work. The 100% manpower loading means that all the students work on all the activities. The architecture, though, must be only refined and documented. In our experience, when students work in functional teams or roles they engage accordingly. Here, teams are formed per subsystem and each student shapes the product and outcome. They all elicit requirements, design, code, inte-

grate, etc., as per their work package. They share the laboratory for two sessions a week, but they can also meet outside those hours.

When students cannot communicate as often as necessary, it is important that they are familiar with the system as a whole and their subsystem, so that they clearly understand how their individual contributions fit in. For the same reason, it is beneficial that their work packages are similar and based on repetitive tasks. The architecture must be defined such that they can discover patterns. For example, if there are ten database tables to display, it takes ten queries, ten database interfaces, ten GUI forms, and two collaborating teams (e.g., DB and FC in Figure 4) of ten students each. These tables store different content, but the mechanism remains the same across a sequence. The database interacts with all subsystems via the same interface, and there is no need for iterations that are subsystem specific, which economizes the solution domain. The benefit of these is profound because it makes development and integration easier, and facilitates group knowledge.

Project

The project is called Sky Highways and is an exercise for students to build an air traffic control simulator. This simulator comprises seven subsystems (Figure 4) the students must specify, design, and implement to simulate the interaction between the Flight Control and airplanes that fly along their flight paths or taxi at the airports. The laboratory project is supported by a manual that has ten A4 pages. The manual explains the main characteristics of the project, including the problem statement and the concurrent engineering model.

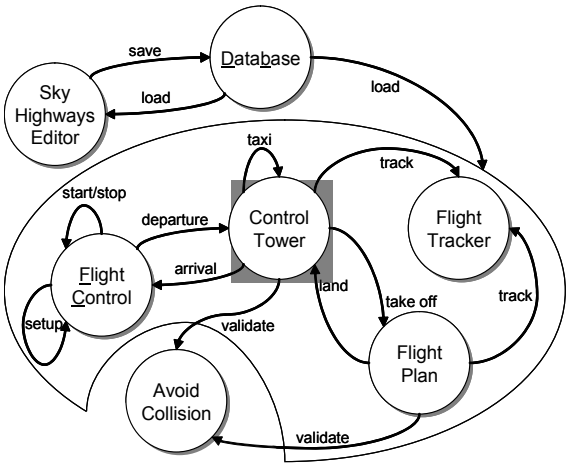

Figure 4. Sky Highways system

The problem statement can be summarized as follows: An airplane is guided from its departure airport to its destination via a series of radar handoffs. The relay starts with the control tower at the departure airport and continues through one or more en route radar control centers that define flight paths. The relay ends with the tower at the destination guiding the plane in for a landing. Up on sky highways each airplane follows its flight plan, and a safe distance between airplanes must be maintained both horizontally and vertically, as well as collision avoidance. Airplanes may be rerouted horizontally or vertically to avoid collision but they should resume their flight paths as soon as possible. The icon of an airplane changes its size as a function of altitude on which the airplane flies. These are the basic ideas the students work on and refine or augment based on the architecture with seven distributed subsystems.

Each subsystem is assigned to only one team, and the actual size of each time is presented in **Error! Reference source not found.** (Team axis). Students join a team according to their prefer-

ences and stay with that team until the end. Each team has a lead who is responsible for liaising with other teams and for weekly progress reports. This is a hands-on role, meaning that team leads do the same work as other team members but the work package is smaller to compensate for the additional responsibilities. Large teams can split into subteams that share similar work packages, or work with other teams. Project management is performed by students and the laboratory supervisors only provide assistance and guidance upon request or when deemed appropriate.

Seventy one undergraduate students enrolled and participated. They all must know Java and the Java 2 Platform Standard Edition, the relational model, and computer graphics, all of which they have learnt before in their studies. They use Microsoft Office, Visual Paradigm for UML (www.visual-paradigm.com), and Eclipse. The subsystems in Figure 4 can be described as follows:

- The Editor subsystem is used to define interactively a map with airports, flight paths, etc. This is a 2-D graphical editor.
- The Flight Tracker subsystem displays a map with dynamic updates. Notifications come from the Control Tower subsystem and the Flight Plan subsystem.
- The Database subsystem is used to store and retrieve maps, flight paths, and other persistent information to a relational database. It has a front end for uniform communication across the whole system.
- The Flight Plan subsystem guides an airplane along its flight path.
- Each airport has a Control Tower that manages airplanes on the ground. An airplane that took off is handed over to the Flight Plan subsystem. An airplane that landed is handed back to the Control Tower, and, when it reaches the terminal, Flight Control stops sending update requests for the airplane.
- The Flight Control subsystem is responsible for starting and stopping experiments. It provides the interface to configure experiments, such as the heartbeat, visual appearance, number of passengers, etc.
- The Avoid Collision subsystem is responsible for collision detection and avoidance by rerouting or delaying the airplanes.

To build this system requires resolving multiple interdependencies between the subsystems. We can start by first building a Sky Editor because it defines the database model on which other subsystems depend. Once a sky highway map can be stored to the database, all other teams can start their work, either immediately or in some order, which is also captured in Figure 4. This strategy is not applicable here because:

- The timeframe of fifteen weeks would leave the remaining five teams with almost no opportunity to do their work.
- Mistakes and rework would cause more delays, and other teams may have requirements and dependencies that have not been considered.

Teams must work concurrently by making assumptions in those areas that are not well understood or cannot be precisely defined, which means that their requirements, design, and implementation will constantly change. For example, the Flight Plan team assumes that a flight path is a sequence of straight lines that can morph into circles to avoid collision, which is handled by the Collision team. The Control Tower team may assume that each airport is a collection of lines that represent taxi strips and runways, and boxes that represent terminals and hangars. An airplane is a box that moves along its path, and these algorithms do not depend on database design. A layer is needed to

decouple the internal model from the database model. Such techniques and patterns are well known but in textbooks they are often described at a high level and many students get confused. Here, they find their application and their importance becomes obvious because each change request triggers a rework iteration that gets potentially harder to implement as more progress is made.

On the other hand, there is no need to force subsystems to be decoupled without a tradeoff analysis. The Tracker team works closely with the Editor team because these two subsystems share many elements. Decoupling tasks on these two subsystems is not productive. The same is true for the Plan and Tower subsystems that can share the algorithms. The students stay engaged and communicate throughout the semester. They learn not to over-commit far in advance or converge quickly on a solution, to recognize similar work packages, and to share the same understanding of the system and mechanisms. They learn to prioritize features and tasks and to stabilize components before adding new features. The teams will act and evolve in response to a problem rather than to programmed role expectations (Bennis, 1993).

The project was supported with various communication mechanisms. It used mutual adjustment as the integrative social process, such as formal face-to-face meetings, standardized documents and reports, and online forums. They are designed to get enough information and knowledge in time in order that procedural formalities serve their purpose instead of stifling progress. Key points and information must be recorded and published in the forum for those concerned. We refrain from direct supervision and management because we could only do so in the laboratory, while the self-managed teams proved effective in doing so, as expected (e.g., Barker, 1993). As noted above, integrative leadership and control should come primarily from them all and the liaisons since they coordinate and communicate concerns between teams.

Instead, by setting up checkpoints we control the progress, and by allowing resubmissions of work products the quality is improved. During a checkpoint, each student briefly explains his/her work and answers questions, gets feedback, and submits the work products. The first two checkpoints were generous, being four weeks apart, in that the lectures provided enough knowledge to get them going and to try to overlap the activities as per lectures. The first checkpoint was to confirm that the system requirements phase and the analysis phase had been worked on. But, most of all, they needed time to learn how to work together and how to use the tools. The second checkpoint should prove that the design and some coding as a proof of the concept also became overlapped and iterated over. The third and the forth checkpoints shortened the interval to three and two weeks respectively, such that the outstanding issues got reiterated and integrated faster.

Towards the end of the semester students asked for permission to come to the laboratory on weekends. The project was completed and fully integrated. However, the collision avoidance algorithm had problems that could not be rectified due to the lack of time. With many active airplanes, some may collide when a control tower takes over the navigation. Overall, the system met the requirements and (almost) worked as intended. All these provided a sense of achievement, and the project has been considered a success.

Findings

Evaluating student learning and the effectiveness of this laboratory is not straightforward. Our past experience with the toy projects did not reveal any opportunism (i.e., opportunity-driven process) or loops in students' coursework even though they were graded only once, at the end of the semester. They were selective in their work, as to test a concept or technique, instead of defining and designing a system. This project confronted them with the system from the start and, therefore, their attention was focused on building that system by producing a comprehensive set of relevant work products.

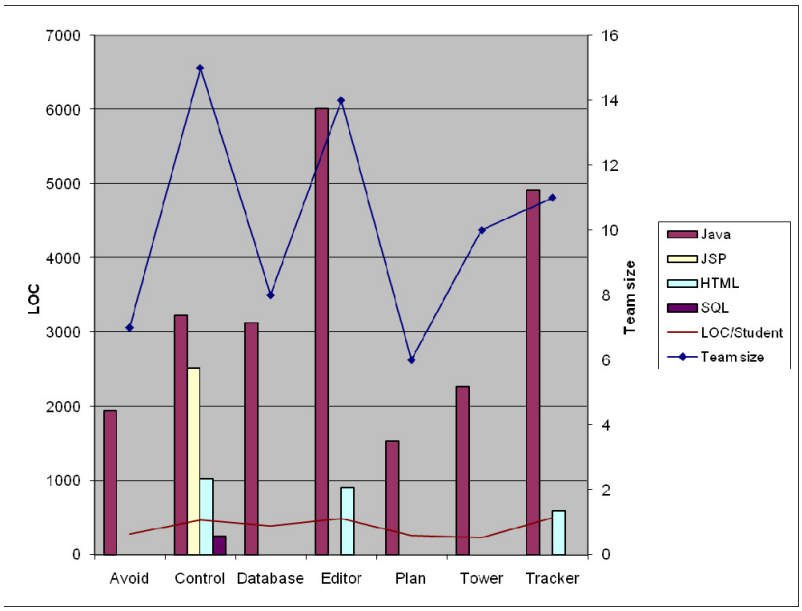

Figure 5. LOC per subsystem (bar chart) and per student (bottom line chart) and team size

The bar chart in **Error! Reference source not found.** presents the productivity of each team in terms of lines of code (LOC). The productivity varies from team to team, with the Flight Control (Control on chart) and Editor teams clearly standing out, followed by the Database team. To correctly understand these data one must understand the underlying characteristics of each subsystem, which may be very different. The former two subsystems could be easily extended with more use cases and features, which is not the case with the latter. Also, user interface and computer graphics programming often produces lots of code compared to algorithms or similar items because they have to implement both the boundary objects and their functionality and take care of event processing. These tasks are mechanical and repetitive and can be facilitated by an interactive drag-and-drop editor. They do not require much analysis and object design, except for the user interface layout. On the other hand, the development of the Collision Avoidance and the Flight Plan required deriving and testing rather sophisticated algorithms. The average for the whole project is 380 lines of code per student.

Our decision to allow resubmission of work products for reevaluation was important as it made rework rewarding and this approach possible. The students produced about 80 megabytes worth of artifacts, or about 1100 files worth of code, design, requirements, meeting minutes, status reports, etc. We defined the templates for each of those. The data in Figure 6 are for requirements and design, because they are of interest for the resubmission. They are based on the number of files marked as a different version and do not include the multiple diagrams, scenarios, and use cases inserted in those documents.

Also, many diagrams have been submitted or resubmitted and used as standalone documents to rectify the problem or provide additional information upon request. However, some students were not very consistent in creating documents based on the templates, opting instead for the screen dumps of their UML diagrams. Nevertheless, the total number of files demonstrates their interest in the different UML diagrams, and the project in general. It may be worth mentioning that the Flight Control team produced a detailed and colorful deployment guide for the whole system. It has 40 pages and 46 screen dumps.

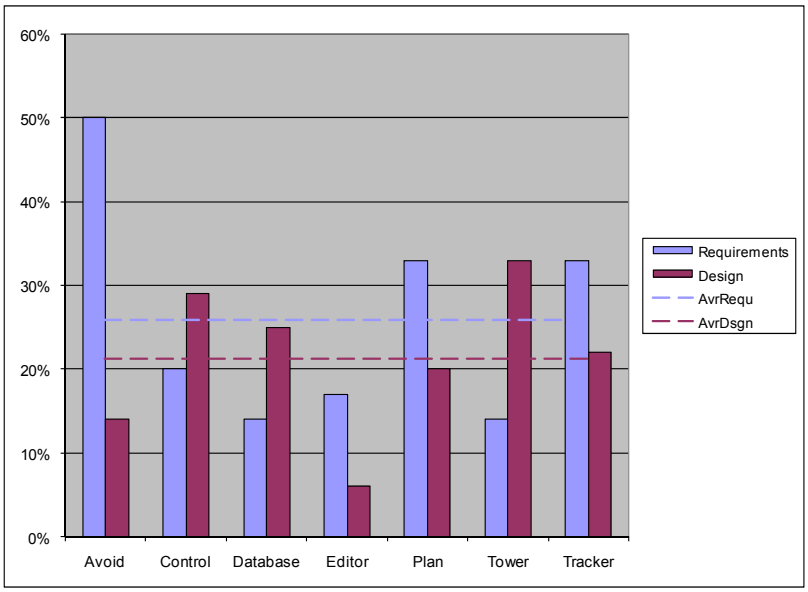

Figure 6. Ratio of originally submitted documents in the total output

Our Findings

In the past, a laboratory of similar number of students working on toy projects in teams of six or eight students would produce up to about 200 artifacts that illustrated their work, but it was often very difficult to relate the design with the system. In the past, students assumed that having a working system was more important and they demonstrated most appreciation towards programming. The teams never interacted because their systems had little if anything in common. They used a variety of languages and tools, which made any interaction even more questionable.

In contrast, the documentation produced in this project was generally detailed and could be easily linked to the code and the system. We noticed that only the diligent and experienced students demonstrated an opportunistic approach to problem solving by intention and enjoyed the freedom to explore back and forth. Most students are used to working on simple problems and in small teams, and working on a large project and resubmitting work products were completely new for them. They follow lectures, making sure that what they do is formally correct. Yet, in this project, because they had to interact and because they could resubmit their work products, they spontaneously developed opportunism, i.e., started overlapping the activities and investigating deeper. Their different styles became initially a source of frustration as they were trying to synchronize.

Overall, our findings on this project can be summarized as follows:

- Although many students simply followed the lecture plan, problem solving typically occurred at different levels of abstraction. For example, many use case diagrams turned out very complex and detailed enough to implement, like a functional decomposition.
- In concurrent engineering it is known that converging on a solution without examining alternative solutions may lead to selecting and pursuing an infeasible solution or an unnecessarily complex one. Students' knowledge is insufficient for generating alternative technical ideas easily, and their experience is not broad. For example, they understand the client-server model but do not realize that one component can implement both roles.
- Sometimes, students lack practice or motivation to search beyond the obvious, assuming that course material provides all answers. For example, they understand what a thread is

and how to create one, but how do subsystem boundaries and multithreading affect each other in a distributed application? How is that facilitated by the middleware?

- Overall, we were positively surprised by many aspects of students' work, such as their capability to operate autonomously, generally resolve issues on their own, and engage in discussion of difficult topics that required coordination and interaction of multiple teams. Although experiencing difficulties at times, because they had to learn to rely on and share with their teammates and other teams, they distributed responsibilities and work appropriately and learned to rectify mistakes as making progress.
- It was easy to form teams, and the storming phase was short. They settled their differences in motivation in about two sessions. Each team managed to create a core that was excited about their work and determined to successfully complete the project. Those less motivated or slower then followed. Soon, they all came to understand that every member is responsible for the progress and success.
- Some team leads were very active and demonstrated good organizational skills. They immediately exchanged mobile phone numbers with their teammates and other team leads and created a Yahoo Group as their forum.
- Some team leads were very good students who attracted and motivated their teammates with their knowledge and confidence even when they made mistakes. Yet, on occasions, they had to remind their team to remain engaged and focused.
- Two teams (i.e., Control Tower and Flight Control) followed a prototyping approach, and they shared their findings with other teams. The Tracker team knew how to build the subsystem, which likely explains their low resubmission rates. They quickly turned their ideas into code.
- The Editor team and the Tracker team did not share together as expected because the latter decided to use 3-D graphics. The Tower team and the Plan team were more successful in pursuing their tasks together by sharing their design and code. The Collision team and the Flight Control team collaborated on interfaces. The Database team decided to assign a table or two per student who were then responsible throughout.

Student Feedback

At the end of the laboratory project, a discussion was held with the students about their experience and thoughts on this approach. For most students this was a completely new and positive experience. This was certainly a reflection of the excitement in the laboratory after the final system demonstration when many became jubilant. Despite the challenges and the lack of prior similar experience, the above presented output data clearly stand in support of the effort, excitement, and interest that were provoked by the project.

The project provided them with an opportunity to learn new tools, understand course material, and reuse their previous knowledge and findings. The students enjoyed working together, and the apparent lack of interest by a few students was not a factor in the end. The use of new tools was also perceived negatively because they had to learn things that were not in the course material (e.g., JSP). One student stated that he would prefer doing something simple as in other courses. In contrast, some students took their freedom to experiment and do things they could not practice in the past, and asked for more references (e.g., advanced computer graphics).

In addition, students mentioned a number of lessons they took away from the course. They experienced the importance of accountability and time management and learned that effective project communication and learning also depends on documentation that must be correct, meaning-

ful, and up to date. They pointed out that they could not rely only on their programming skills to produce a system and that immediately it became evident that constant commitment was necessary.

Many students appreciated the opportunity to work in a new context, where the problems were generally more complex and did not always match the structure of the examples covered through lectures and tutorials. The opportunity to apply those examples and theory to solve useful problems made them feel more confident about how important knowledge, methods, and technologies could be for their future careers as software engineers. The project allowed them to experiment and become significantly more competent in the technical domain and more confident in their skills and understanding of the theory covered in this and other courses.

Conclusion

Concurrent engineering is a powerful industrial process model that many manufacturing companies now use in product development. Yet, it is difficult to implement because it requires organizational and cultural changes within an enterprise. On the other hand, we find it interesting for educational purposes because a more natural model of learning and working environment can be put in place.

For concurrent projects to complete successfully, it is important that feedback from the teams and dependencies to previous tasks and components gets accounted for in real time. Concurrent model demands flexibility in defining, designing, and scheduling tasks that traditional feed-forward project planning methods do not. As students work, they learn how their ideas become interdependent, transformed, and shaped into products. We find these important for software engineering education, and the presented laboratory project a step forward in our teaching practices.

By working on complex problems students become better prepared for the workplace. There are two dimensions here: one about teamwork and sharing responsibility and the other about the unbounded nature of engineering problems. Human learning and projects are not linear processes, and (for the inexperienced) they require rework and exploration. Given that this project followed the recommendations and findings mentioned above, we hope this experience is a valid and lasting one for the students.

References

Alzamil, Z. (2005). Towards an effective software engineering course project. *ICSE 2005*, pp. 631-632.

Barker, J. R. (1993). Tightening the iron cage: Concertive control in self-managing teams. *Administrative Science Quarterly, 38*(3), 408-437.

Barki, H., & Hartwick, J. (2001). Interpersonal conflict and its management in IS development. *MIS Quarterly*, *25*(2), 195-228.

Barnes, D. J., & Kölling, M. (2006). *Objects first with Java* (3rd ed.). Upper Saddle River, NJ: Prentice Hall / Pearson Education.

Bennis, W. (1993). *An invented life: Reflections on leadership and change*. New York, NY: Addison-Wesley.

Bergin, J. (2000). Fourteen pedagogical patterns. *Proceedings of the 5th European Conference on Pattern Languages of Programs*.

Bernhart, M., Grechenig, T., Hetzl, J., & Zuser, W. (2006). Dimensions of software engineering course design. *ICSE 2006*, pp. 667-672.

Clark, K. B., & Fujimoto, T. (1991). *Product development performance*. Boston, MA: Harvard Business School Press.

Conklin, J. (2006). *Dialogue mapping.* West Sussex: John Wiley & Sons.

Coppit, D. (2006). Implementing large projects in software engineering courses. *Computer Science Education*, *16*(1), 53-73.

Cross, N. (2004). Expertise in design: An overview. *Design Studies*, *25*(5), 427-441.

Feisel, L. D., & Rosa, A. J. (2005). The role of the laboratory in undergraduate engineering education. *Journal of Engineering Education*, 94(1), 121-130.

Ferguson, E. (1992). Engineering and the minds eye. Cambridge, MA: MIT Press.

Guindon, R. (1990). Designing the design process: Exploiting opportunistic thoughts. *Human-Computer Interaction*, *5*, 305-344.

The Joint Task Force on Computing Curricula. (2004). *Software engineering*. 2004. IEEE/ACM.

Mihm, J., & Loch, C. (2006). Spiraling out of control: Problem-solving dynamics in complex distributed engineering projects. In D. Braha, A. Minai, & Y. Bar-Yam (Eds.), *Complex engineering systems* (pp. 141-157). New York, NY: Perseus Books.

Nikkei Business Publications. (2003). IT field survey 2003. *Nikkei Computer*, *587*, 50-70.

Paulish, D. J. (2002). *Architecture-centric software project management: A practical guide*. Boston, MA: Pearson Education.

Peters, L. (2003). Educating software engineering managers. *CSEET'03,* 78-85.

Pister, K. (1993). *Major issues in engineering education: A working paper of the Board on Engineering Education*. Commission on Engineering and Technical Systems, Office of Scientific and Technical Personnel, National Research Council, Washington, DC.

Rittel, H., & Webber, M. (1973). Dilemmas in a general theory of planning. *Policy Sciences*, *4*, 155-169.

Schon, D. (1987). *Educating the reflective practitioner*. San Francisco, CA: Jossey-Bass.

Senge, P. M. (1990). *The fifth discipline: The art and practice of the learning organization*. New York, NY: Currency and Doubleday.

Sobek, D. K., Liker, J. K., & Ward, A. C. (1998). Another look at how Toyota integrates product development. *Harvard Business Review*, *76*(4), 36-49.

Soloway, E. I., & Spohrer, J. C. (1989). *Studying the novice programmer.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Terry, J., & Standing, C. (2004). The value of user participation in e-commerce systems development. *Informing Science: The International Journal of an Emerging Transdiscipline, 7*, 31-46. Retrieved from http://inform.nu/Articles/Vol7/v7p031-045-216.pdf

Biographies

Nenad Stankovic received a B.S.E.E from the University of Zagreb, Croatia, and a Ph.D. in Computer Science from Macquarie University, Australia. He has been working both in academia and industry. He teaches and conducts research in the areas of engineering education, organizational learning, and software engineering.

Tammam Tillo received the degree in Electrical Engineering from the University of Damascus, Syria, in 1994, and the Ph.D. in Electronics and Communication Engineering from Politecnico di Torino, Italy, in 2005. In 2004 he was visiting researcher at the EPFL (Lausanne, Switzerland). From 2005 to 2008, he was a Post-Doctoral researcher at Politecnico di Torino. From January to March 2008 he was invited research professor at SungKyunKwan University, Suwon, S.Korea. Currently, he is with Xi'an Jiaotong-Liverpool University, Suzhou, China.