Annals of Software Engineering 6 (1998) 343–363

Improving academic software engineering projects: A comparative study of academic and industry projects

Pierre N. Robillard [a] and Martin Robillard [b]

[a] *Laboratoire de Recherche en Génie Logiciel, Département de Génie Électrique et de Génie Informatique, École Polytechnique de Montréal, C.P. 6079, Suc. Centre-ville, Montréal, Qc., Canada H3C 3A7*
E-mail: robillard@rgl.polymtl.ca
[b] *Department of Computer Science, University of British Columbia, 2366 Main Mall, Vancouver, BC, Canada V6T 1Z4*
E-mail: mrobilla@cs.ubc.ca

A project course in software engineering is often part of the curriculum in computer engineering or computer science. This paper studies the relationship between academic and industrial projects in software engineering. The purpose is to compare the practices followed in a project-course approach with the practices of professional software engineers. The approach is to compare the measurements obtained from academic and industrial projects. The critical factors regarding the process, the people and the project are discussed. The structure of the software processes and the measurement tools are presented. The data analyses show that the academic projects are found to be strongly dominated by programming activities. Based on the data from the industrial projects, we formulate seven recommendations to improve the software engineering practices in academic projects. They are related to management, predevelopment, development, testing, reviews documentation and team activities. The concluding remarks discuss some of the actions that could be taken to improve academic projects.

1. Introduction

Many curricula in computer engineering or computer science include a project course in software engineering. These project-oriented courses are intended to initiate students into the practice of software engineering and to synthesize what has been learned in previous courses. These courses are usually based on teamwork.

The purpose of this paper is to compare the practices followed in a project-course approach with the practices of professional software engineers. The goal is to identify the relevant features of academic software projects and to formulate recommendations for the conduct and content of academic projects. The approach is to analyze the measurements obtained from both industrial and academic projects. Of course, such an approach must take into account the differences between the two environments and stress the similarities on which the analysis is based. Some actions based on the

recommendations that could be applied in most academic environments are discussed in the concluding remarks.

There are many difficulties with such studies, among them the differences in each of the three Ps: People, Project, and Process. The term 'People' refers, on the one hand, to professional software engineers (SE) who have at least two years of experience, and, on the other, students (ST) who have at most a summer of experience in software development or coding. Some of the impact of experience in software engineering will be measured in this study.

The two projects analyzed are different. The industrial project required 2400 hours to complete, while the academic project required 810 hours. However, the two projects have the following similarities: they have well-defined requirements, they require the design and implementation of computational algorithms and they have no input or output interface. All data are transmitted to and from files.

The process was defined and was the same for both projects. The instructor of the academic project ensured similarities in process since he was also the coordinator of the software engineering team for the industrial project. The projects were measured in the same way, by filling out a daily time log.

The academic and industrial projects used for this analysis were selected from among the eight academic projects and four industrial projects that had been conducted in the past three years and that were based on the same approach. The projects selected were the most successful projects in each group. Many characteristics of students' projects have already been published [Robillard 1995, 1996a].

The following section describes the major characteristics of each project. Then, the processes used by both teams are described; the measurement tools are presented; and data for each phase of the software life-cycle are presented, followed by recommendations regarding improvement of the academic project.

We are aware that the data presented in this paper are based on two specific projects and that a word of caution is required before generalizing the conclusions. The various numbers and ratios presented in this paper are very specific to the projects and may not be generalized. However, we believe that the recommendations derived from the data analysis are of general interest and can be useful.

2. Project description

The goal of the industrial project was to design and implement a simulator of Petri Net [Marsan 1990]. This simulator is integrated into a modeler, which is distributed worldwide. Requirements are well defined and the task is to design and implement the simulator. The project was developed in a Windows environment in C++. An object-oriented methodology was used. The project was successful, all requirements were implemented and tested, the documentation was appropriate and the schedule and the budget were respected.

The goal of the academic project was to develop a software tool for software quality measurement. The software quality model [Boloix and Robillard 1995] is

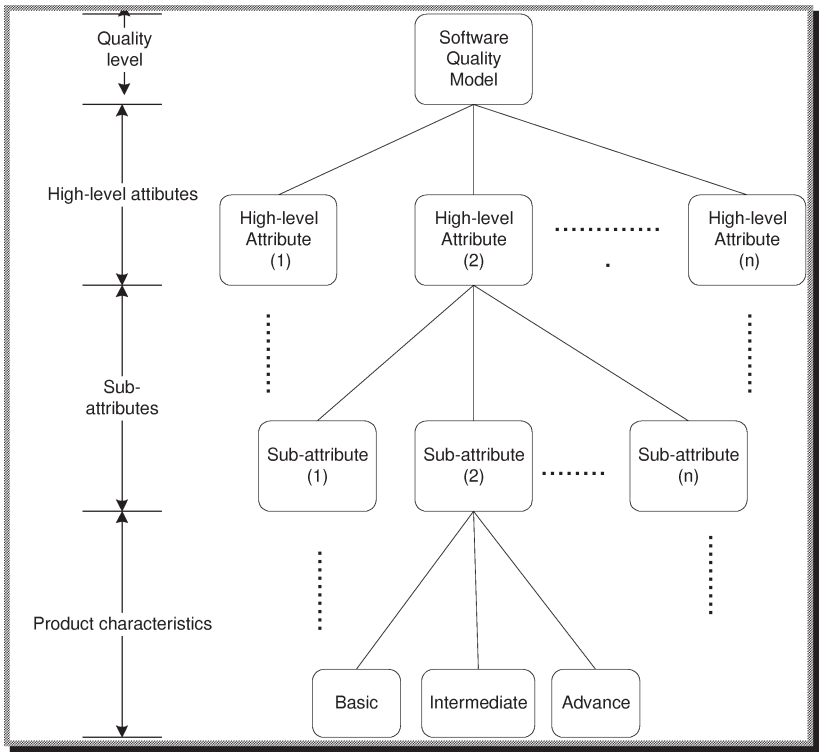

Figure 1. Architecture for the software quality model.

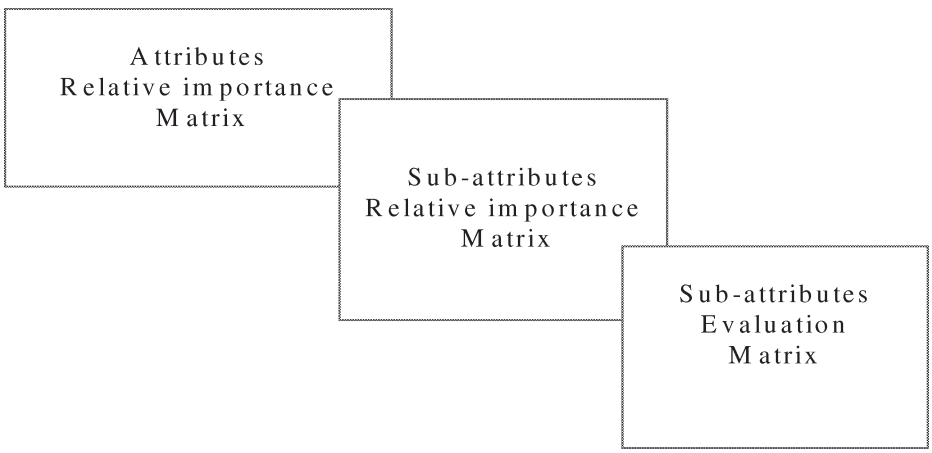

Figure 2. Matrices to be filled out and computed for the quality model.

based on two hierarchical levels where each level has a variable number of attributes. The user must specify the attributes for each level. Typical quality model architecture is illustrated in figure 1.

The task is to compute each matrix shown in figure 2 according to the Analytic Hierarchy Process model [Finnie *et al.* 1993]. Each matrix generates an eigenvector, which is used to classify the relative importance of the elements for each level. This eigenvalue is used as a weighting factor for each attribute of the system. The results are computed level by level to obtain a global evaluation of the software system.

The two projects are based on algorithms for mathematical computation and there is no interface development involved. Essentially, the design task was based on the processing of complex mathematical equations and on the managing of appropriate data structures.

3. Process and life-cycle definition

The two teams used essentially the same life cycle and software development approach. The main characteristics of software development processes are:

- A systematic approach to software development.
- The use of appropriates software engineering standards and CASE tools.
- The use of reviews for each step of the software development process [Freedman and Weinberg 1990].
- Preparation of the appropriate documentation for software development.

The software process used is inspired from the Capability Maturity Model (CMM) key practices and the documentation of the software life-cycle phases are based on the following IEEE standards [Software Engineering 1997]:

- IEEE std 829 Software Test Documentation.
- IEEE std 830 Software Requirement Specifications.
- IEEE std 1008 Standards for Software Unit Testing.
- IEEE std 1016 Recommended Practice for Software Design Description.
- IEEE std 1058 Software Project Management Plan.
- IEEE std 1074 Developing Software Life-Cycle Processes.

The framework for the development of the software is adapted from the software life cycle as defined in IEEE std 1074. The life cycle is based on five phases, which are management of the project, predevelopment, development, post-development and the integral phase. The academic project is based on the realization of a software project through only four phases of its life cycle. The post-development phase was not used in any of the projects.

Students are grouped into teams of six. Each team applies a defined process based on systematic reviews at each phase of the software life cycle. This process integrates various CASE tools and was centered on the tool SCHEMACODE [Schemacode 1998; Robillard *et al.* 1996]. This tool was used for detailed design, coding and

formal inspection. The students have an overview of technical reviewing, programming standards and quality assessment, and videos on formal inspections and scenes of software inspections are presented as part of the project.

3.1. Project management

The first phase of the software life cycle is to analyze the project characteristics. All aspects concerning project management are documented according to IEEE std 1058 and are described in a document called **S**oftware **M**anagement **P**lan (**SMP**). A major difficulty in this phase is to define the list of tasks and the schedule. This information is usually available after completion of the **S**oftware **R**equirement **S**pecification (**SRS**), which is described below.

According to IEEE std 1058 [Software Engineering 1997]:

- *A task is the smallest unit of work subject to management accountability.*

A task is a well-defined work assignment for one or more project members. The specification of work to be accomplished in completing a task is documented in a work package. Each task is prioritized and assigned to a team member in order to derive a schedule. The list of tasks and the schedule are not final and are likely to be modified as the project progresses. At the end of this phase, a team review validates these two documents. At various times during the project, these documents will be revised and updated.

3.2. Predevelopment

The predevelopment activities are defined in the Predevelopment section of IEEE std 1074. They are:

- Identify ideas or needs.
- Formulate potential approaches.
- Conduct feasibility studies.
- Refine and finalize the ideas or needs.

The documentation of these activities is based on the **SRS** as defined in IEEE std 830. Predevelopment is the phase that defines the **D**ocuments for the **A**rchitecture (**DA**) of the system and the **L**ists of **T**asks (**LT**). The **T**est **P**lan (**TP**) is also drawn up during this phase. Team members validate the **SRS** through a **R**eview of the **P**redevelopment documentation (**RP**).

3.3. Development

The development phase is used to define the details of the task and the functions that must be implemented. This process recommends the following activities, which are described in the Development section of IEEE std 1074:

- Analyze task implementation.
- Design.
- Code.

The documentation for the **D**etailed **D**esign (**DD**) is based on IEEE 1016.

According to this approach, a thorough understanding of the task is required. This activity is documented in an **A**nalysis of **T**ask (**AT**). There is an **AT** for each task. Once the **AT** has been completed, the software engineer can design the task. The design is created at the pseudocode level according to a defined notation called **S**chematic **P**seudocode (**SP**) [Robillard 1996b]. A CASE tool (SCHEMACODE) [Schemacode 1998] is used to assist with the design activity. A team **R**eview of the **D**esign (**RD**) concludes this step.

Software engineers complete the implementation by developing the schematic pseudocode to code level. Programming guides are available to reduce the so-called **V**ariations **I**n **P**ractices (**VIP**). Once the coding is completed, a **R**eport on **T**ask (**RT**) is produced to document the problems faced during implementation. The source code is formally inspected and **I**nspection **N**otes (**IN**) are documented in SCHEMACODE. A designated team member performs task acceptance. The purpose is to make sure that the task has been completed according to the process.

3.4. Integral

These activities are performed over many phases during the process of software development. They are restricted mainly to test and documentation activities. The first test activities are performed during the predevelopment phase. At this stage, the **T**ests **P**lanning (**TP**) is included in the **SRS** document. The second test activities are performed during the development phase. The third test activity takes place during the integration of the modules and the fourth test activity is the acceptance test for the system. The instructors to conclude the course complete this last stage. The set of **T**ests (**TT**) is documented according to IEEE std 1008. The four activities related to testing are:

1. **T**est **P**lanning (**TP**).
2. **T**esting the software **U**nits (**TU**).
3. **T**esting the **I**ntegration of the software units (**TI**).
4. **T**esting for **A**cceptance of the software system (**TA**).

All the documents and practices are derived from professional standards. The process proposes four types of documents, one for each phase of the software life cycle. The document for the management of the project, which is derived from the **SMP**, includes a description of the project, a **P**roject **S**chedule (**PS**) and a budget in terms of resources needed. The predevelopment document, which is composed of the specifications of the projects, includes the architectural design the list of task and the plan tests. The

Table 1
Required documents and the corresponding IEEE standards.

1.	*Project management*	
	SMP: Software Management Project	IEEE std 1058
	PS: Project Schedule	
2.	*Predevelopment*	
	SRS: Software Requirement Specification	IEEE std 830
	DA: Design of the software Architecture	
	LT: List of Tasks	
	TP: Tests Plan	
	RP: Review of the Predevelopment SRS	
3.	*Development*	
	Analysis and detailed Design	
	DD: Detailed Design	IEEE std 1016
	AT: Analysis of Tasks	
	SP: Schematic Pseudocode of the design	
	RD: Review of the Design	
	Implementation	
	*.**x**: *source files*	
	RT: Report on Task implementation	
	IN: Inspection Notes	
4.	*Integral*	
	TT: Test for the Total system	IEEE std 1008
	TU: Tests of software Units	
	TI: Test of Integration of software units	

development design includes the analysis and design document and the minutes of the review meeting, and, of course, the source codes. The document for the integral phase includes the test sets and the verifications. Table 1 shows the lists of documents required by the process, the phase in which they are written and the IEEE standards from which they are derived.

4. People

The industrial project lasted 19 weeks and required 4 full-time software engineers. Two had a degree in computer engineering and one a degree in computer science. The coordinator had a Master's degree in software engineering. Their professional experience ranged from 2 to 7 years. They used a democratic team approach, with one of the team members acting as the coordinator.

The academic project is conducted within a course called the Software Engineering Studio, which is given during the winter semester and lasts 13 weeks. Each week, the 3-hour laboratory session is preceded by a 1-hour lecture. The lecture time is used to introduce some process elements and to comment on the specifications of the product. The student lecture time is not part of the data measurements.

Table 2
Structure of the activity record.

TIME:ID:DURATION:NB:PHASE:ACTIVITY:TASK:COMMENTS

Table 3
Definition of the field of the records for the logbook.

Record	Definition
TIME:	Clock time. It is captured automatically.
ID:	Identification of the team member.
DURATION:	Time spent on the given activity (multiples of 15 minutes).
NB:	Number of team members involved in the given activity.
PHASE:	Name of the phase of the process.
ACTIVITY:	Name of the activity being measured.
TASK:	Name of the task as defined in the planning of the project.
COMMENTS:	Any comments that are relevant to the activity record.

Each team is composed of three pairs of students. The schedule for delivering the final product was fixed and firm. Students were free to take extra time if they wished. However, they were often reminded that this project is a regular 3-credit course and that they should budget their time accordingly. A formal acceptance test session is held on the last day of the course. The results of the competition count for only 20% of the student's total grade. The remaining 80% of the grade are based on successive evaluations of the documentation released during the project.

The teams were relatively autonomous and made their own decisions as to the workload of each team member. A full afternoon was reserved every week during the semester for team activities and the instructor was available at this time. Each student spent an average of 10 hours a week on the project.

Student backgrounds are homogeneous in that they are all completing the last semester of a 4-year computer-engineering program. They know each other before they begin this course. The Software Engineering Studio is an elective course for fourth-year students with a major in software engineering.

5. Definition of the measures

This section describes the approach used to gather information on the process activities. The time spent by an individual on each activity is captured by filling out an activity record, shown in table 2, once the activity has ended. All the records are stored in the same database (Access). An activity record is made up of 8 fields, which are defined in table 3.

Each individual was responsible for filling out his or her own logbook. One team member was responsible for collecting the completed logbooks of teammates every week and integrating them into the team database. All the data presented in this paper were obtained through analysis of the logbooks.

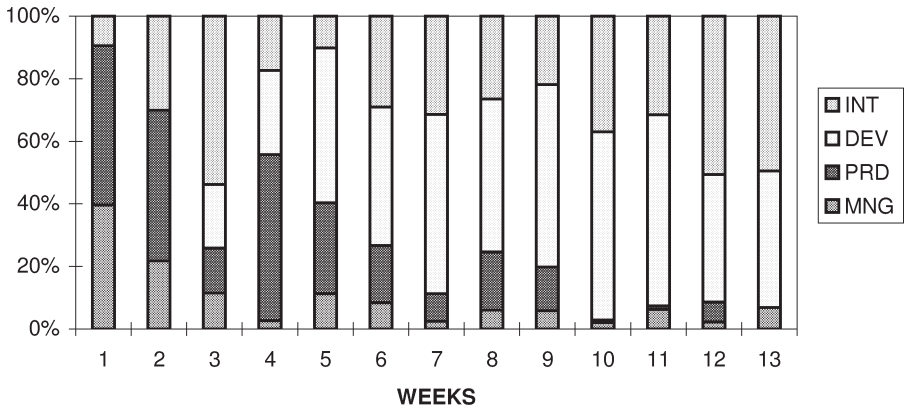

Figure 3. Weekly distribution of relative effort per phase for the academic project.

The first observation concerns the relative steadiness of the workload for the student teams during the semester. Contrary to what is often found in project courses, there was no excessive workload in the last few days. We believe that the workload was evenly distributed over the semester partly as a consequence of the software process and of filling out of logbooks.

6. Presentation of the results

The purpose of this study is to find the relationship between an academic project and an industrial project. These projects are studied from two viewpoints: the effort expended on the various software life cycle phases and team effort.

6.1. Software life cycle phase activities

The four software life cycle phases grouped various activities under generic names such as management, predevelopment, development and integral. This software life cycle model implies that an activity of a given phase can occur any time in the software development process. For example, management activities can occur at any time during the project's evolution.

Figure 3 shows the relative weekly distribution of effort in each phase for the academic project. For each week, it shows the percentage of time spent in each of the phases of the software life cycle. We recall that the weekly effort was quite uniform for the project duration and amounted to an average of 10 hours per student or 60 hours per week of team effort. The first column shows that the team spent 24 hours (40% of 60 hours) on management activities (MNG), which means that the six students spent 4 hours each in meetings. The team spent 30 hours (50% of 60 hours) on predevelopment (PRD) activities and 6 hours (10% of 60 hours) on integral (INT) activities, which most likely involves documentation. No development (DEV) activity was carried out during the first week. Except for meeting activities, the students do

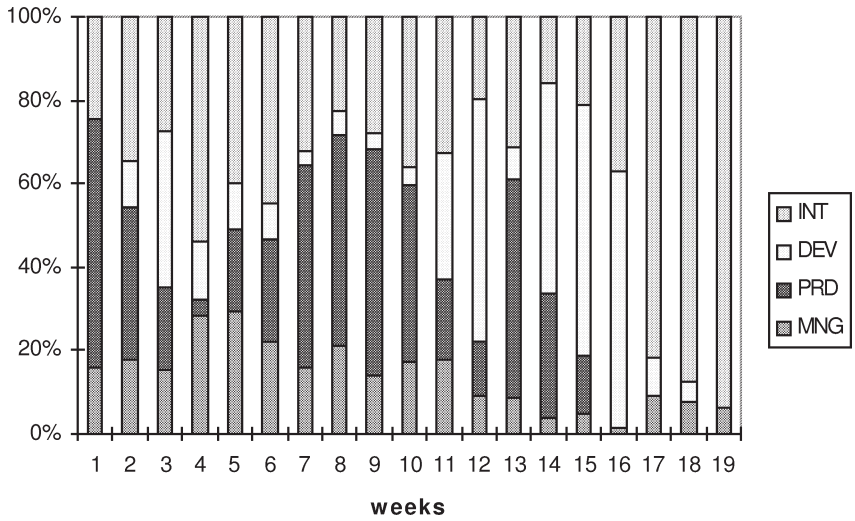

Figure 4. Weekly distribution of relative effort per phase for the industrial project.

not usually work together on the same activities. For example, the 6 hours of integral activity was mainly completed by a single individual.

We see that management activities are very important during the first three weeks, and then the basic weekly meeting schedule is resumed. The major integral activities during the third week involve writing the **SRS** that concluded the predevelopment activities of the first two weeks. Software development activities dominate for the remainder of the project duration. One can see in this data some tendency towards the waterfall approach, which is revealed here as major management activities followed by major predevelopment activities and then by major development activities.

Figure 4 shows the same data for the industrial project. Each column represents 150 hours of team effort (4 software engineers at 37.5 hours per week). This profile shows that there is sustained effort on management activities until the middle of the project. Predevelopment activities dominate for most of the project and development activities are carried out mostly in the few weeks near the end of the project. Integral activities are also important in week four, when the **SRS** was completed. It is interesting to observe that the project ends with an intensive integral phase, which is made up of testing and documentation activities.

This software life cycle is more of a spiral model where predevelopment activities are completed by a few development activities to build a prototype or to try out algorithm implementations. It can also be seen as a mixture of top-down and bottom-up approaches where a few modules are developed to test the understanding of a proposed solution. The main characteristic is that the major development effort is launched once the whole project has been understood and the specifications completed.

Figure 5 shows a comparative histogram of the relative effort in each project. The first column shows that the management activities account for 8% of the total effort for the academic project, while it accounts for 14% for the industrial project.

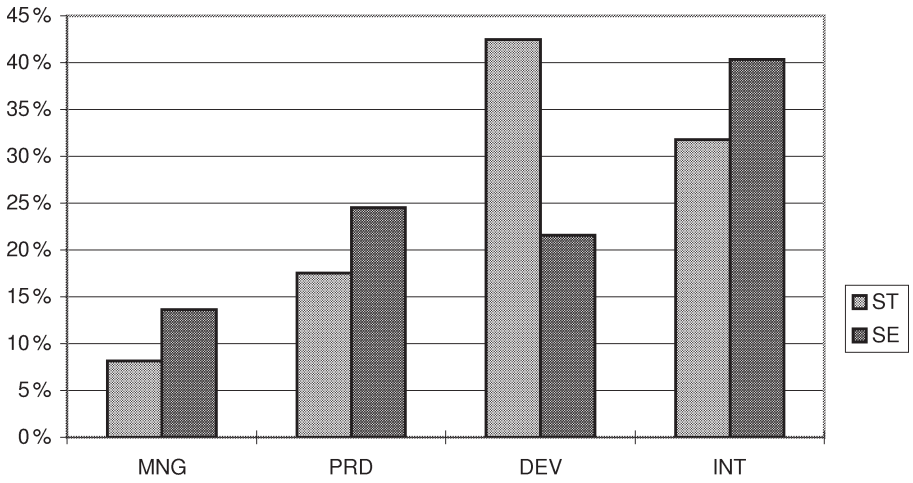

Figure 5. Relative activities per phases for the academic and industrial projects.

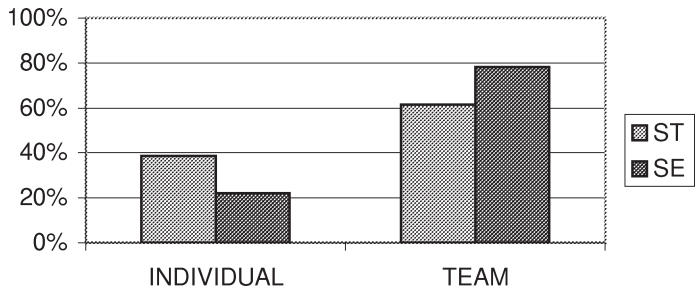

Figure 6. Management activities for the academic and industrial projects.

The software engineers (**SE**) spent relatively more time on all phases but development. It is just the other way around for the students (**ST**), who spent almost half (42%) their time on development (mostly coding), while software engineers spent a little more than 20% of their time on development. Students show a strongly code-oriented approach to software development. In the following, we analyze the dominant activities of each of the phases in order to formulate recommendations concerning the various practices.

6.2. Management activities

Management activities are more time consuming in the industrial project. Figure 6 shows that management activities can be either individual activities or team activities. The individual management activity time is mostly spent by the project coordinator on quality control activities.

The weekly mandatory team meetings recorded as management activities are the main component of the larger management activities of the industrial project. The students have a tendency to drop the mandatory team meeting once everyone knows

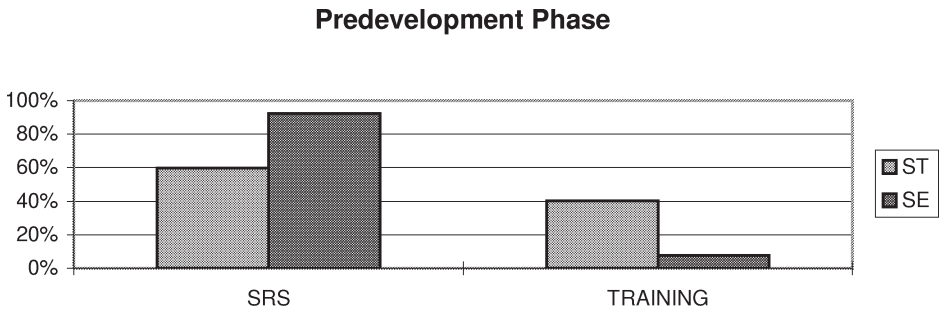

Figure 7. Documenting SRS and training activities in the Predevelopment Phase.

what to code. The same individual for the whole project usually performs the individual management activities, which include organization of the meeting, writing minutes and doing some scheduling.

The management activities in the academic project were not very well understood and a great deal of frustration arose from these team meetings. Natural leaders are rare and students do not have a strong incentive to deal with personnel problems. Their driving motivation is to get a good mark.

Recommendation 1. *Provide students with basic training in management and in project control.*

6.3. Predevelopment activities

Predevelopment consists of requirement analysis and training on the various aspects of the application domain. Figure 7 shows the relative importance of predevelopment activities for the students and the software engineers.

Software engineers spend most of their predevelopment time on requirement analysis and relatively little time in training to acquire knowledge of the application domain. We recall that for neither project anyone was familiar with the application domain, which was very specific in both cases, although all students entering the academic project course were familiar with the software tools used. In other words, individual experience has little to do with the training required to understand the application domain. By looking at the clock time spent on the learning activities, it is found that for the two projects each individual spent approximately 10 hours on training activities. Since the overall time spent on predevelopment activities is shorter for the academic project, the training activities occupy relatively much more time.

Recommendation 2. *In a project-oriented course in software engineering, defining a project in an application domain familiar to the student might minimize the training time. It has been found that almost 10% of the student's time are spent in acquiring the knowledge specific to the application domain.*

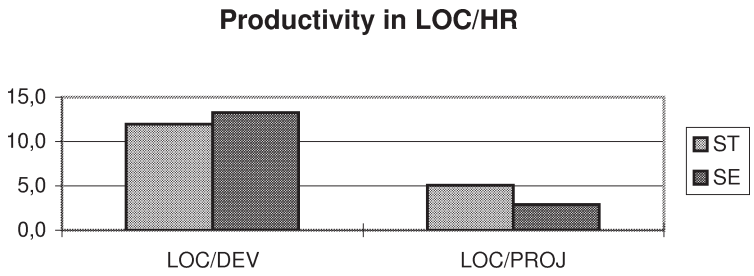

Figure 8. Productivity for the development phase (DEV) and for the project (PROJ).

6.4. Development activities

The development activities are dominated by the detailed design and coding activities. These two activities are so closely related that the participants were not able to fill out the logbooks time in such a way that enabled us to reliably distinguish between them. For both projects, the development activities were mainly concerned with pseudocode and code, and they both used the same structure editor Schemacode, the same development environment and the same programming language, C++.

Figure 8 shows that the productivity is nearly the same for the two projects. Productivity is measured in terms of lines of code (LOC) written per hour of development time. LOC refers to the number of Lines of Code documented.

The two columns of figure 8, furthest to the left show the ratio of number of lines of code to time spent in the development phase only. The two columns furthest to the right show the ratio of lines of code (LOC) to total time spent on the project (PROJ). It is found that experience has little effect on coding productivity. As has been observed in previous studies [Robillard 1996a], the more time students have for development, the more code they write. Figure 3 shows that students would code up to the last minute of the project. While software engineers seem to know when the project is finished, students always have something to add and it seems that they do not know when to finish a task. Two mechanisms may explain this situation. The first is inadequate predevelopment activities. The second, which may be related to the first, is the lack of feedback to signal an end to the development activities.

Recommendation 3. *Do not allow development activities to proceed before predevelopment has been validated, and penalize code that is not derived from specific predevelopment specifications. Require team meetings to validate development activities.*

Figure 8 shows that productivity measured in terms of LOC per hour (LOC/HR) can be misleading. For example, when productivity is measured as a ratio of LOC to development time (LOC/DEV), productivity is nearly the same for the two projects, with a slight advantage to the software engineers. However, when productivity is

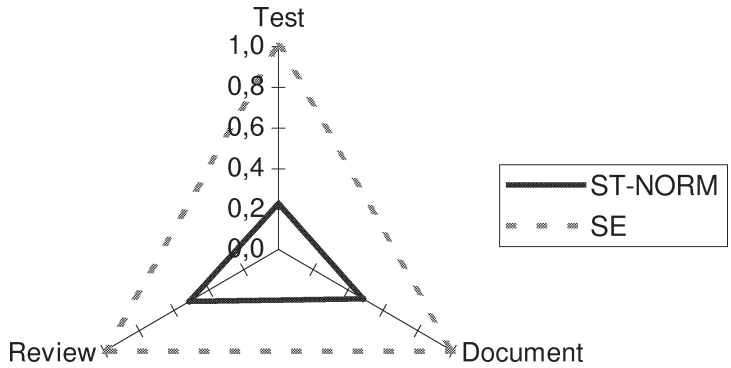

Figure 9. Normalized radar diagram of activities in the Integral phase.

measured as a ratio of LOC to total project time (LOC/PROJ), the students are almost twice as productive as the software engineers are. These measures do not take into account the quality of the source code written. However, in both cases code inspection was required.

6.5. Integral activities

The dominant activities of the integral phase are testing, documentation and technical revision meetings. These activities can best be evaluated in relation to the size of the product. The testing measure is defined as the testing time activity divided by the number of KLOC to be tested. The documentation measure is defined as the documenting time activity divided by the number of pages of documentation written. The review measure is defined as the number of pages of the reviewed document divided by the number hours of review activities. In order to compare these data, the ratios are normalized according to the software engineers' measurements. Figure 9 shows the radar diagram of the various normalized ratios. The software engineers' (SE) ratios are normalized to one and the student ratios (ST-NOM) are normalized with respect to SE. For example, the vertical axis label *Test* shows that the student test ratio is almost four times lower than the SE ratio. The students spend less than one quarter (0.22) of the time spent by the SE on testing activity.

Testing seems to be one of the weakest activities in the academic project. One reason might be that students stop coding too late in the project and do not have enough time to do the required testing, as shown in figure 3. Another reason might be that they are used to building throwaway software and are not aware of the testing time required or the nightmare of supporting bugged software.

The lack of strong testing activities is the major factor contributing to the apparent increase in student productivity at the project level. Students are motivated by the task of building a running program and are not really aware that the more code they write the more time it will take to test it.

Recommendation 4. *Emphasize validating running programs and require a minimum of testing time for every KLOC. Students are likely to spend more time on design in order to reduce LOC and corresponding testing time.*

According to the process, the same types of documentation were required for the two projects. Figure 9 shows that students wrote documentation twice as fast as software engineers. It took students only half the time to write the same number of pages. However, two differences in the way the process was conducted could explain the discrepancies between the two projects in terms of documentation time. The SE documentation was subject to extensive quality control based on peer reviews and walkthroughs. The documentation activities were part of the industrial project and were needed to support the spiral process. By contrast the students wrote documentation as a mandatory activity (as is often the case) and did not see the usefulness of it.

Recommendation 5. *Require and evaluate design documentation before allowing coding activities.*

According to the process, technical reviews were required before the delivery of any documents. The review axis of figure 9 shows that students review documents at a rate that is half that of software engineers. Students are twice as much slower in their reviewing process. Students' lack of experience in technical reviewing and the quality of the document being reviewed (see previous section) could account for their poor reviewing performance.

Recommendation 6.

- *Provide students with formal training on technical review meetings before beginning the project.*
- *Show a video of typical successful technical review meetings.*
- *Invite an experienced software engineer to act as moderator, at least for the first technical review meeting of the project.*

7. Team activities

Software projects involve team activities. Team activities are needed to share information, to organize tasks and to participate in peers' reviews. Coincidentally, the team activities represent 16% of the activities in the two projects. However, the distribution of these activities is not the same in the two projects.

Figure 10 shows the weekly relative distribution of team activities for the industrial project. For example, the first week of the industrial project was composed of 20% team activities (TEAM) and 80% individual activities (IND). The activities are carried out on a fairly sustained basis, except for Week 16, which is the coding

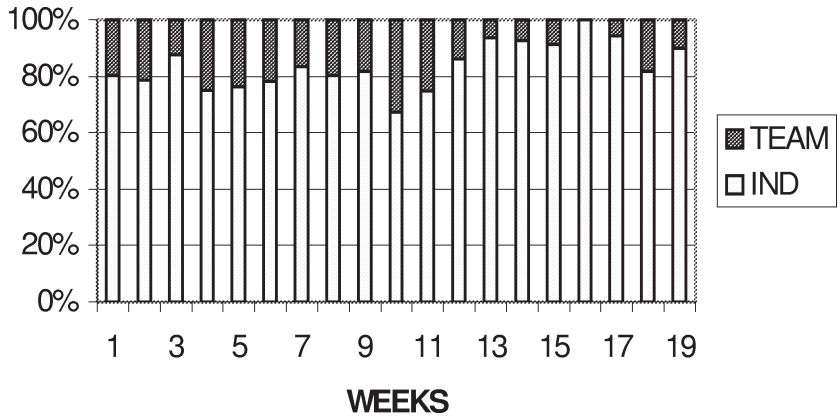

Figure 10. Weekly distribution of team activities for the industrial project.

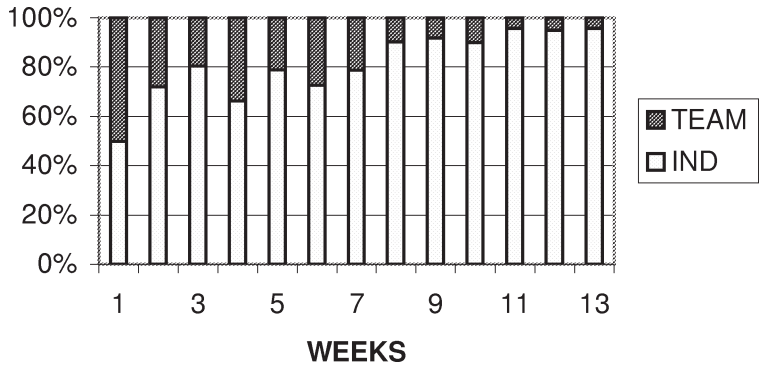

Figure 11. Weekly distribution of team activities for the academic project.

week (see figure 4). The weeks with important coding activities involved minor team activities.

Figure 11 shows that for the academic project the number of team activities decreases regularly as the project proceeds. We attribute the emphasis on teamwork in the first weeks of the academic project to the fact that students form a democratic team. The various individual roles within the team are not well defined. It takes some time before they realize that the work must be coordinated and that someone needs to take the lead. We believe that assigning a team leader or project coordinator and defining the role that each individual must play within the team could normalize this behavior. The weak team behavior at the end of the project corresponds to intensive coding activities and is similar to the behavior observed in the industrial project.

Recommendation 7. *Assign a team leader and define the role of teammates before the beginning of the project.*

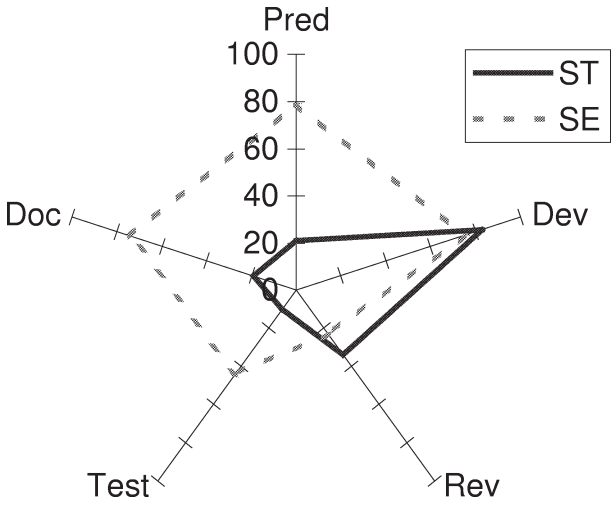

Figure 12. Radar diagram of hours spent per KLOC in each phase for the two projects.

8. Project characteristics

This section illustrates two characteristics of a project: its software process and its size. The radar diagram of figure 12 illustrates the characteristics of the software process. It shows the ratio of hours to KLOC for the two projects. For example, the vertical axis, labeled Pred, shows that the software engineers (SE) spent 79 hours on predevelopment activities for every KLOC of source code delivered, while the students (ST) spent 21 hours on the same activities. The development (Dev) axis shows, as stated before, that the time to write one KLOC is almost the same for the two projects. The review time per KLOC (Rev) is lower for the students as discussed before. Figure 12 shows that predevelopment, test and documentation activities are kept to a minimum in the academic project. The academic project is strongly oriented towards the production of code, even if a software process was used.

Figure 13 shows a surface graph of the activities for the two projects. Revision and testing activities have been merged into the quality control activities (QC). The figure shows that industrial projects spend almost the same number of hours per KLOC (within 10%) for each of these activities. Students emphasize the development phases, with a gradual decrease in effort per KLOC for all the other phases. Since resources are limited, the number of hours available for the project is fixed, which means that the area under the curve is constant for a given project. The ratio of activities in other phases of the software process can be increase by applying the recommendation 3 (validate the predevelopment activities), recommendation 4 (normalize testing time) and recommendation 5 (evaluate design documentation). If these recommendations are not enough then one should consider reducing the size of the project.

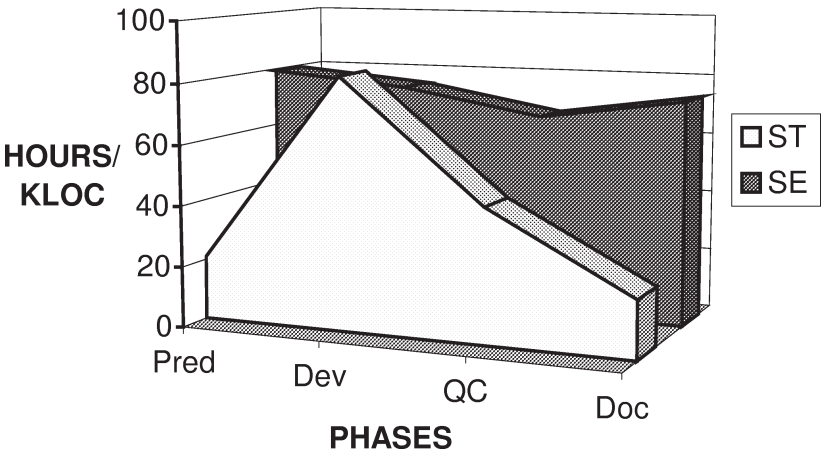

Figure 13. Surface graph of activities in HOURS per KLOC for the two projects.

9. Concluding remarks

The goal of this paper is to illustrate the difference between software engineering projects in the academic and the industrial environments. The purpose is to determine what academic projects provide to students in terms of learning professional practices.

The purpose of the academic project is to initiate students into software engineering. The general conclusion from the data presented in this paper is that an academic project is a good introduction to industrial software engineering since students are initiated into the major phases of a software project. However, students are still placing too much emphasis on the development phase. It seems that the main objective of students is still to produce code. The following concluding remarks suggest how the recommendations presented in this paper could be used to improve an academic software-engineering project.

Table 4 summarizes the seven recommendations that are likely to bring academic projects closer to industrial projects. The direct consequence of implementing these recommendations will be to reduce the size of the academic project and improve the student's understanding of software engineering practices.

The following illustrates the type of actions based on these recommendations that could be taken to improve academic software engineering projects. Some of them have been successfully implemented at Ecole Polytechnique.

The students in the post-mortem analysis of the project required the basic training in management activities referred to in the first recommendation. This recommendation results in the introduction of *software management* topics in the course on project management for all engineering students.

The second recommendation is aimed at reducing the relative effort required to acquire the knowledge related to the application domain. The objective of the project is to enable students to apply their understanding of software engineering to the implementation of computer programs and not to learning how to solve problems. We

Table 4
List of recommendations for each measured aspect.

R	Aspect	Recommendations
1	Management	Provide training in management and projects control
2	Predevelopment	Minimize problem-solving
3	Development	Require detailed design traceable to code
4	Testing	Require a minimum testing time per KLOC
5	Documentation	Evaluate design documentation before coding
6	Review	Provide training on technical review meetings
7	Team	Assign roles

believe that enrolling students with homogenous background and relying on experienced instructors are factors that could minimize problem-solving activities.

The third recommendation is aimed at reducing the urge to code early. We find that once some lines of code have been written most of the students' thinking is based on these lines of code. LOC should be the result of a carefully designed process and not the design process itself. LOC are an end product. This recommendation requires students to show that the thinking process is finished before code writing can begin. The design could be approved by the instructor or by a peer review. Reduced marks could also be given when source code cannot be traced back to the detailed design.

The fourth recommendation concerns the testing activities. We find that testing is not really planned as such by the students, even if it has been strongly recommended. Students have very limited experience with testing activities. Testing should not be confused with debugging or desk checking. By requiring a minimum of documented testing time for every line of code, students could more likely realize the cost of LOC and be encouraged to improve their designs to avoid any unnecessary coding.

The fifth recommendation stresses the importance and usefulness of appropriate documentation. We implement this recommendation by dividing the project into three parts and asking each sub-team of two students to design and document a module that will be coded by another sub-team.

The sixth recommendation is related to the practices involved in the technical review meeting. We show students a video of a typical review meeting. These videos are taken from the industrial projects. We believe that experienced software engineers should act as moderators, at least for one of the first review meetings.

The seventh recommendation is related to the teammate roles. It is more democratic to let the students find by themselves the role they want to play in the team. However, it could be time consuming and it is rarely done in industrial project. People do the job they have been hired for. Based on our experience, it seems that assigning student's role right at the beginning of the project is more efficient than letting the team member find in a democratic way their respective roles.

We find that the programming language is almost irrelevant. In previous years, we have worked on projects in FORTRAN, Pascal, C and C++ (this year). We feel that it is important that all the teams use the same programming language, however. This

greatly eases the job of the instructor and favors team exchanges. We also find that the methodology used (OO or procedural) has little impact on the effort distribution over the various process phases.

Students should be well prepared to take such a course. A previous course in software engineering is a prerequisite. There is no time to learn a new programming language or software tools.

It is difficult to find a suitable academic project. This is because our main concern is that students follow a defined process, and not solve a problem. Problem-solving is very time-consuming and requires creativity, and would be inappropriate for such a course [Robillard 1999]. However, the problem should not be trivial either, since the students must recognize the need to use a software process to implement the software product. Also, it is important not to underestimate the complexity of the project. If the project is too complex, the students might see the process as the main barrier to their success. They might also skip the reviews or some of the testing, as sometimes occurs in the real world.

The instructor should be an experienced software engineer. Students need coaching. They easily get lost in fruitless meetings. They are not used to playing with official standards and they want everything to be perfect. Standards are extensive, and are generic guides, which should fit almost any case. Students have to learn to adapt or interpret the standards to the needs of their projects. They cannot always differentiate the accessories from the mandatory components. For most of them, this process is a great learning experience.

Acknowledgements

This approach is the result of many years of experience and has been developed in collaboration with the Laboratoire de Recherche en Génie Logiciel de l'École Polytechnique de Montréal and various industrial partners. Special thanks to DMR Consulting Group Inc. for providing the industrial project. We are grateful to Benoit Lefebvre, Luc Lalande, Carl Paquet for their collaboration and to Martin Truchon who acted as coordinator for the industrial project and as instructor for the academic project. We are grateful to the students for their enthusiasm and excellent collaboration. This work was supported by National Sciences and Engineering Research Council of Canada under grant A0141.

References

Boloix, G. and P.N. Robillard (1995), "A Software System Evaluation Framework," *IEEE Computer*, December, pp. 17–26.

Finnie, G.R., G.E. Wittig, and D.I. Petkov (1993), "Prioritizing Software Development Productivity Factors Using the Analytic Hierarchy Process," *Journal of Systems and Software 22*, 129–139.

Freedman, D.P. and G.M. Weinberg (1990), *Handbook of Walkthroughs, Inspections, and Technical Reviews*, 3rd Edition, Little, Brown Computer Systems Series, Boston, MA.

Marsan, A. (1990), "Stochastic Petri Nets: An Elementary Introduction," In *Advances in Petri Nets*, G. Rozenberg, Ed., Springer-Verlag, pp. 1–29.

Robillard, P.N. (1995), "Experience in Teaching Team Software Design," In *Proceedings of the 6th World Conference on Computers in Education*, Chapman and Hall, Birmingham, UK, pp. 441–453.

Robillard, P.N. (1996a), "Teaching Software Engineering Through a Project-Oriented Course," In *Proceedings of the 9th Conference on Software Engineering Education*, Springer-Verlag, Daytona Beach, FL, pp. 85–94.

Robillard, P.N., M. Simoneau, J. Mayrand, and D. Coupal (1996), "Experimental Data on the Usefulness of a Structured Editor," In *Structured-Based Editors and Environments*, G. Szwillus and L. Neal, Eds., Academic Press, Amsterdam, The Netherlands, Chapter 6, pp. 193–206.

Robillard, P.N. (1996b), "Programming Environment for Eliciting Knowledge Types," In *Proceedings of 8th Annual Workshop*, Psychology of Programming Interest Group, Gent, Belgium, pp. A1–A9.

Robillard, P.N. (1999), "The Role of Knowledge in Software," *Communications of the ACM 42*, 1, 87–92.

Schemacode (1998), *Schemacode International Inc.*, www.rgl.polymtl.ca/schema/schema.htm.

Software Engineering (1997), *IEEE Standards Collection*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY.