What Cognitive Activities Are Performed in Student Projects?

Éric Germain and Pierre N. Robillard
École Polytechnique de Montréal
{ eric.germain, pierre-n.robillard @ polymtl.ca }

Abstract

Software processes are being increasingly taught to software engineering students. Previous studies have however shown that actual activities performed in the course of student projects differ widely from what students had been taught. This study defines a new cognitive activity classification scheme that has been used to record effort spent by six student teams producing parallel implementations of a same software requirements specification. Three of the teams used a process based on the UPEDU, a teaching-oriented process derived from the Rational Unified Process. The other three teams used a process built around the principles of the Extreme Programming (XP) methodology. Results show that coding-related activities dominate the effort distribution for all the teams. Also, variations in the relative emphasis put on each activity between processes are low and limited to a small number of activities. The study provides lessons that may be useful when evaluating the importance of specific software processes.

1. Introduction

There seems to be an increasing interest about the teaching of software processes (see for instance [1-4]). But this interest does not translate into the acceptance of a common set of process principles. In particular, two main software development philosophies seem to emerge. The first one promotes the utilization of a very well defined process involving precise definition of roles to be played, activities to be performed and artifacts to be produced. Such an approach generally involves the production of artifacts which purpose is to support early decision making on requirements and design matters, effective communication, knowledge reuse and mutual work inspection. The main principle here is that efforts made in upfront planning activities and in artifact production will result in lower overall cost, timely product delivery and better software quality. The Rational Unified Process (RUP) [5] is an example of a process that fits this approach. The UPEDU [6-8] constitutes the adaptation of the RUP for teaching the software processes in software engineering and computer science programs.

The other philosophy, called "Agile Software Development" [9-10], promotes quick response to changes in requirements as well as extensive and ongoing collaboration between the development team and the customer. The approach specifically downplays the importance of formal processes and comprehensive documentation. It is based on the assumption that one cannot truly anticipate project requirements right at the beginning of a software development project, and that the proper way to deliver timely, quality software in a cost-effective manner is instead to build flexibility within the development activities. The "Manifesto for Agile Software Development [10]" provides the basic values of agile development in detail. Some methodologies derived from this approach include Adaptive Software Development, Scrum, the Crystal family, Feature-Driven Development, Dynamic System Development Method and Extreme Programming.

In addition to such methodological variety, it is reasonable to consider that a generic process will have to be adapted to each organization and project that requires one. There is no

Proceedings of the 16th Conference on Software Engineering Education and Training (CSEET'03)
1093-0175/03 $17.00 © 2003 **IEEE**

such thing as a universal process. For instance, users of the RUP are provided with tools that allow them to build their own subset of the proposed activities and artifacts.

Students enrolled in a software engineering program and who have received training on software processes are expected to be, at the end of the program, more sensitive to issues affecting software quality, cost and lifecycle. This does not mean however that those individuals will apply everything they learned as is. Previous studies [11-13] in the context of the "Software Engineering Studio", a project-oriented course for senior-level students, have shown a significant gap between theory as taught and practice. Those studies were using effort slips as an indicator of relative activity intensity. Analyses performed were however limited by the activity and artifact classification of the UPEDU-based process used, which was reflected in the effort tracking tool used. It was thus rather difficult to determine exactly which cognitive activities had been performed.

Using those studies as a foundation, we defined a set of cognitive activities that aims at accurately recording the various activity states of a software developer in the course of a project. The utilization of such a classification allows us to study the impact of software process notions learned on the cognitive activities actually performed by the students during a project course.

We do not expect our results to be ready for immediate generalization to industrial practices because of the academic nature of the setting and of the impact of the particular project, lifecycle and technology chosen. Meanwhile, repeating such an experiment in an industrial setting would be quite difficult because of the need to record individual cognitive activities at developer level. However, we think that the study presented in this paper provides clues that may be useful when evaluating the importance of a specific software engineering process.

2. The Software Engineering Studio

The Software Engineering Studio is an optional project-oriented course offered to senior-year students in computer engineering at École Polytechnique de Montréal. Its purpose is to allow students to get a practical experience of software development by participating in a small-scale, complete software development project. Teams of students must develop a complete implementation based on software requirements specifications provided by the instructors. They also must use a well-defined software engineering process. Participants thus get an early experience in building an operational software project from A to Z through design, implementation, testing and management activities. This project course teaches them the realities of teamwork and of project completion within schedule. As a secondary objective, students get more familiar with a specific application domain or set of technologies. An earlier version of the Studio has been presented in [14]. The Studio has also served as a testbed for the study of development effort and artifact quality. Some individual studies performed using data generated in the course of a Studio edition have been documented in [2-4].

The Winter 2002 edition of the Studio featured the development of a Web-based meeting management system aimed at organizers of meetings where the number and geographic dispersion of participants make scheduling difficult. The software system to be developed would allow meeting coordinators to send availability requests to a set of individuals so that each one can specify their personal availability periods. The set of availability periods would then be graphically represented using a special calendar tool that would allow a coordinator to visualize the relevant information at a glance, making the scheduling decision easier to take.

The decision would then be transmitted electronically to all participants. The software system would be responsible, among other things, for ensuring proper data storage, update and communication between all participants. All communications would be performed using standard e-mail. Figure 1 shows a screenshot from one of the software products delivered.

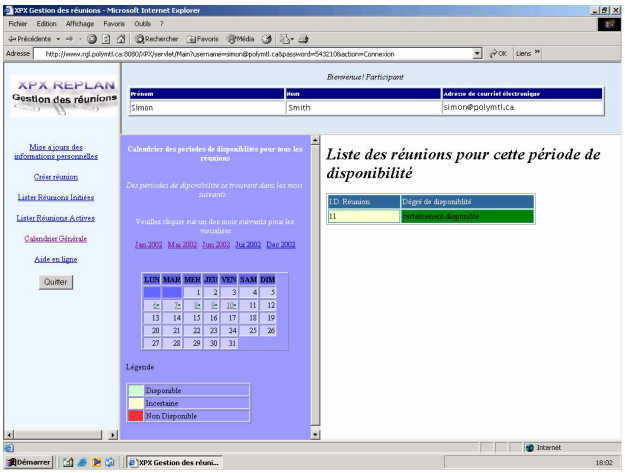

Figure 1. Screenshot from one completed product

The main feature of the Winter 2002 edition was the use of two different software engineering processes. One of the goals of the study was to determine the influence of the software process on the participants' behaviour. The instructors therefore chose to assign each half of the class to one of the two software processes selected. Thus, three of the teams were assigned a process based on the Unified Process for EDUcation (UPEDU) [8], which is derived from the Rational Unified Process [5]. The other three teams were assigned a process built around the Extreme Programming methodology (XP) [15]. Figure 2 illustrates the prescribed software lifecycle calendar. The diagram shows the iterations prescribed for each process. Iterations with the form "XP*" relate to the XP-based process, while those with the form "UP*" relate to the UPEDU-based process.

A common release-level framework was used to define the lifecycle for both processes. Thus, for all the teams, an initial specification was provided at the beginning of the semester. A complete implementation of that specification was due after 45 days. Thereafter, a second specification was issued that requested a moderate architectural change to the system. Implementation of that change was due after an additional 15-day period. Iterations XP1 through XP5 and UP1 through UP3 belong to the initial development cycle, while iterations XPM and UPM belong to the end-of-semester maintenance phase.

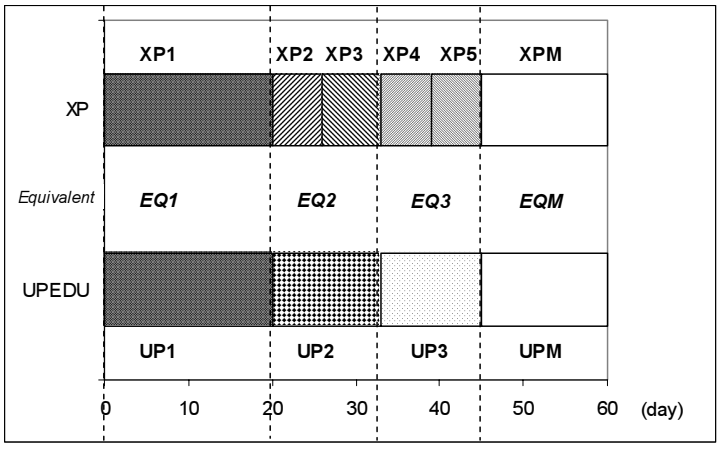

Figure 2. Software lifecycle calendar (in working days)

At the iteration level, the lifecycle was customized for each of the process used. Since iterations in an XP project are usually shorter than in the typical UPEDU project, the lifecycle for the XP process included a greater number of iterations covering the same time frame. The iterations targeted by this alteration are those at the middle and at the end of the development cycle. It was however not obvious that the first iteration should be shorter for the XP teams, since this initial iteration is crucial for laying out the skeleton of the system.

Also, the instructors wanted to leave enough time for every participant to get used to the development environment provided and to get minimal comfort with the language and technologies, to which most students had not been exposed previously. The initial iteration has therefore been kept identical in length for both processes. Also, the maintenance cycle has been limited to a single iteration due to general agreement by the students that this would be sufficient considering the limited scope of the changes requested. The remaining iterations have been set out so as to get a ratio of two XP iterations to one UPEDU iteration. The correspondence of UPEDU iteration end dates to XP iteration end-dates was required for the purpose of facilitating the analysis of the resulting data on effort spent. Figure 2 illustrates the equivalent iterations EQ1, EQ2, EQ3, EQM, that have been defined for that matter.

The team and individual evaluation grid for the course is shown at Table 1. 75 points out of 100 were attributed to each team as a whole. 25 points were allowed on an individual basis.

3. Cognitive activity classification

In previous editions of the Studio, students were asked to record effort spent under each process activity. This approach has the benefit of allowing a direct measurement of the process itself. However such a classification can only be used under the assumption that the list of activities defined in the process covers every possible work situation without bringing excessive overlap. Such an assumption has not been confirmed. Indeed, an analysis performed using data from the 2001 edition of the Studio showed possible presence of ambiguity and confusion among participants in relation with the process activities as defined by the instructors [11]. Another problem with the approach was that the presence of two separate software processes prevented the utilization of a single process-based scheme that would allow comparison of effort spent for all the teams. An alternative approach was to use a process-independent classification that lead to implicit assignation of effort to the proper activity. A classification based on the evaluation of explicit, mutually exclusive cognitive activities constituted an interesting path to this target.

Table 1. Evaluation grid

Scope	*Criterion*	*Weight*
Team-level evaluation	Product quality	25 %
	Artifact quality and timeliness	25 %
	Effort slip quality, completeness and timeliness	25 %
Individual evaluation	Contribution to the team	25 %

Table 2 illustrates the classification that was used for the purpose of the study. The classification includes 14 activities that are grouped into four categories. Participants were presented all 14 activities without the category framework, which has been defined strictly for analysis purposes.

Table 2. Cognitive activity classification

Preparation	*Implementation*
Think Read Browse / Search Draw Write Discuss	Code Code & Test Test Integrate & Test
Control	***Support***
Inspect / Review	Tech. Administration Training Other

Although most activity names are self-explanatory, we provide below a short description of some of them. Category "Preparation" encompasses cognitive activities that are related to activities that may be considered as prerequisites for coding. Activity "Think" refers to the process of self-reflection and thus encompasses every effort

spent by a stand-alone participant for which no input nor output was present. Activity "Read" refers to the action of reading a specific document such as a textbook or an article for the purpose of assimilating a well-defined block of information, while activity "Browse / Search" was aimed at the action of reading documents or web pages in a non-specific order, as when searching for documents that will eventually be read. Activities "Draw" and "Write" refer to the respective production of diagrams and text of all kinds. Activity "Discuss" refers to every discussion taking place between a team member and one or more persons that may or not be team members.

Category "Implementation" was aimed at those activities that are central to the coding process. This category was especially important from an experimental point of view since coding-related activities constitute the vast majority of the effort spent under strict implementation of the Extreme Programming methodology. The classification had to reflect the fact that, under XP, coding, integrating and testing often occur as intertwined activities. Activities "Code", "Test" and "Code & Test" have therefore been defined in order to take account of the possible combinations of coding and testing. Activity "Integrate & Test" reflects the fact that, presumably, integration is a short duration activity that leads immediately to testing.

Category "Control" was aimed at the quality assurance actions that were likely to take place after every preparation or implementation step. It encompasses one single activity called "Inspect / Review" which refers to the technical review activities that may be performed after the initial production of any artifact. Category "Support" included other activities which occurrence would be interpreted as merely accidental and weakly linked to fundamental behavioural characteristics of the participants.

4. Analysis of cognitive activities performed

Figure 3 illustrates effort spent on each cognitive activity as a percentage of total effort spent in each of the three following grouping: XP-based projects, UPEDU-based projects, total (sum of the six projects). The three most important contributors to effort are the same for all groupings: "Code", "Code & Test" and "Write". Those activities, along with "Draw", are the most output-oriented of the activity classification. They amount to 57% of total effort spent under the total grouping. Coding-related activities alone amount to nearly half (47%) of total effort under that same grouping. This shows clearly that, beyond central analysis, design and testing skills that are, rightly, promoted within the software engineering community, this discipline remains a coding-intensive one, even when performed by students aware of the importance of software process activities. This finding might provide a part of the answer to the question raised by McConnell: "How important is software construction?" [16] Software construction is indeed a very important matter, at least in terms of its intensity relatively to other disciplines.

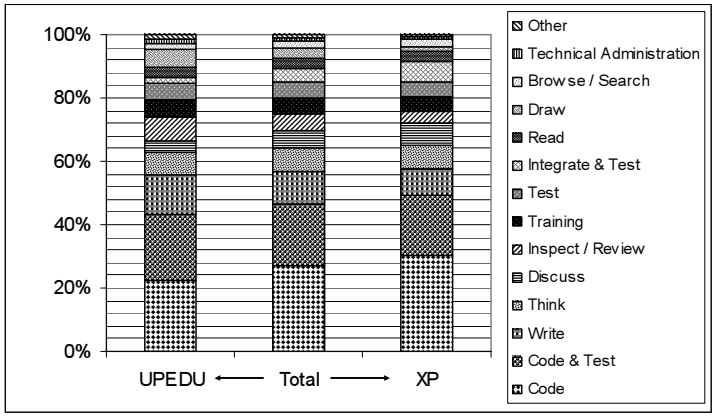

Figure 3. Effort distribution by activity (%)

A Pareto analysis [17] of figure 3 shows that half of

the activities (7 / 14) cover 80% of the total effort spent under the general grouping. Meanwhile, a thorough look at the center of the Pareto distribution shows that 7 activities gather between 3% and 5% each. Support activities encompass only 7% of the total effort, most of it being spent in training.

Figure 4 illustrates the same distribution, but modified to help analysis of the central activities. Coding activities and less relevant support activities have thus been removed from this activity distribution analysis. Activity distribution within this partial set does not follow a typical Pareto distribution. The first 2 activities (starting from the bottom) account for 40% of the effort, instead of an expected 80% using the Pareto principle. It is necessary to add up effort spent on the first six activities to reach that 80% level. This is an interesting result since it shows that our classification fills out its purpose of acting as a powerful discriminating criterion for activity classification. Some activity merging would however have to be performed so to help provide a clearer picture of which activities predominate among those performed by the participants.

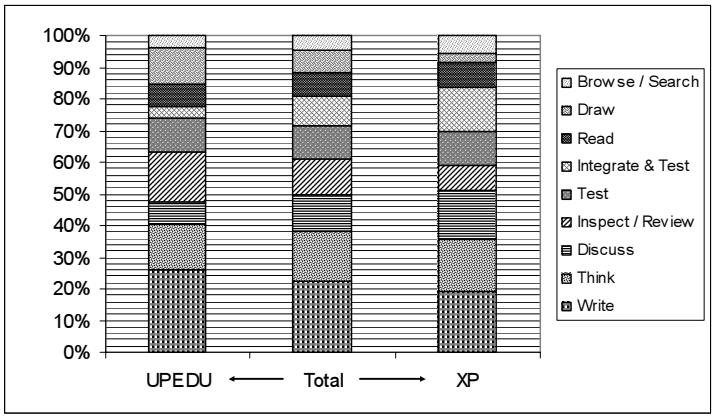

Figure 4. Effort distribution by activity, partial set (%)

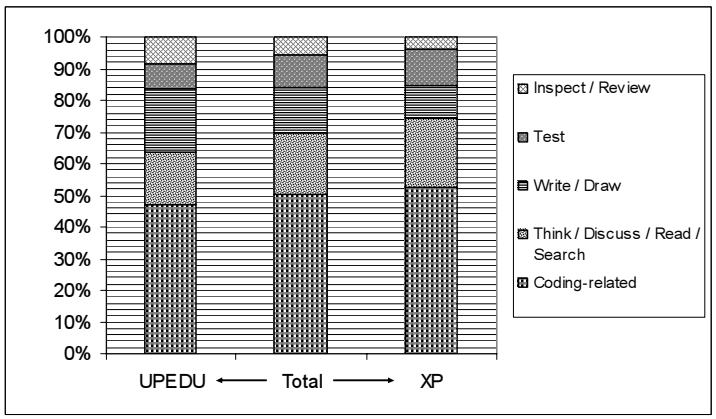

Figure 5. Effort distribution by composite activity, partial set (%)

First, output-producing activities "Write" and "Draw" differ only by the type of output generated. We chose to merge them into a single "Write / Draw" composite activity. Also, activities "Integrate & Test" and "Test" cover essentially the same kind of work. We chose to merge them into one "Test" composite activity, encompassing only testing made outside of a coding task. Finally, activities "Think", "Discuss", "Read" and "Browse / Search" all cover tasks that are performed as preliminary steps to the output of any artifact, while not producing artifacts themselves. We therefore chose to merge them into a single composite activity.

Figure 5 illustrates the results of the merging operation. Under this reclassification, the output-less composite activity becomes the second most effort-intensive one, right after the coding-related activities. It is interesting to note that this particular set of activities amounts to 39% of the total non-coding and non-support effort. Also, coding-related and output-less activities amount to 70% of all effort spent.

5. Concluding remarks

Developing software is an open problem. There can be as many solutions as there are individuals or teams. In this study, all the teams provided acceptable software products in relation to the requirements specification issued. All were also constrained by the common lifecycle and met all deadlines.

Even though the software process used seemed to have an impact on the importance of some cognitive or composite activities, we did not observe any significant relation between the process used and the overall effort magnitude. Effort spent by XP teams as a whole indeed exceeded effort spent by UPEDU teams by a whopping 29%. However, external factors that may have affected this figure are numerous and thus make it highly questionable. The only three-participant team was a UPEDU team and showed the smallest total effort of the six teams. We may interpret this as the expression of the fact that those students had to be more productive than the other teams to reach their objectives, or that they may have benefited from their size in terms of reduced required interactions.

The project required quick learning of the Java Servlet technology by the participants. Since the XP teams had to start coding almost immediately, they faced technological difficulties earlier than the UPEDU teams. We observed significant technology-related knowledge transfers from the XP teams to the UPEDU teams at the time when the latter started producing code. It must be noted that other kinds of knowledge transfers, for instance ones related to architectural decisions, seem not to have occurred on a large scale. Traces of such transfers have not been found in the resulting artifacts, except in the form of common reuse of a few key external components. Total absence of knowledge transfer would have been very difficult to achieve in practice. However, the use of a project definition that is less challenging from a technological point of view than the one actually implemented would possibly have downplayed the importance of this particular factor.

This study illustrates a basic observation of team software development based on two different software engineering processes. In spite of the limited scope of the study, a few general conclusions can be drawn. These conclusions need more experimentation in order to be validated.

The effort spent on core activities within each development project are more or less independent of the software engineering process used. The process will just bring more emphasis on one type of activity rather than another. This shifted emphasis does not have a spectacular effect on the overall distribution of the cognitive activities performed. One possible interpretation is that some core activities will require a minimal effort investment regardless of the software process used.

We observe that a well defined software process such as the UPEDU will put more emphasis on the engineering aspects of the software implementation by stressing the pre-coding activities while the XP-based process will put more emphasis on testing and ad hoc communications. While these observations are totally in line with the definition of the processes involved, what is most interesting is that these differences between processes are simply not as great as one may have expected and do not impact the effort-intensive coding activity family.

6. Acknowledgements

We are grateful to Mihaela Dulipovici who participated in the preparation of the semester, acted as teaching assistant for the course and was deeply involved in artifact and effort slip quality evaluation. Also, this project would not have been possible without the participation of all the students enrolled in the "Software Engineering Studio" course during the Winter 2002 semester. We would also like to thank Alexandre Moïse and Martin Robillard for their insightful comments while we were building the requirements specification for the semester project.

This work was partly supported by the National Sciences and Engineering Research Council of Canada (NSERC) under grant A0141.

7. References

[1] M. Halling, W. Zuser, M. Köhle, and S. Biffl, "Teaching the Unified Process to Undergraduate Students", Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET'02), IEEE Computer Society, 2002, pp. 148-159.

[2] D. Umphress and J.A. Hamilton, Jr., "Software Process as a Foundation for Teaching, Learning, and Accrediting", Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET'02), IEEE Computer Society, 2002, pp. 160-169.

[3] M. Höst, "Introducing Empirical Software Engineering Methods in Education", Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET'02), IEEE Computer Society, 2002, pp. 170-179.

[4] El Emam, K., "Software Engineering Process", SWEBOK – A Project of the Software Engineering Coordinating Committee (trial version 1.00), IEEE, Los Alamitos, CA, 2001, pp. 9-1 – 9-18

[5] Kruchten P., "The Rational Unified Process: An Introduction", Reading, MA, Addison-Wesley, 2000.

[6] Robillard, P.N., and P. Kruchten, "Software Processes with the Unified Process for Education (UP/EDU)", Addison Wesley, Boston, MA, 2002

[7] École Polytechnique de Montréal, « UPEDU », http://www.upedu.org

[8] P.N. Robillard, P. Kruchten, and P. d'Astous, "YOOPEEDOO (UPEDU): A Process for Teaching Software Process", Proceedings of the 14th Conference on Software Engineering Education and Training (CSEET '01), IEEE Computer Society, 2001, pp. 18-26.

[9] Cockburn, A., "Agile Software Development", Addison Wesley, 2002

[10] Agile Alliance web site, http://www.agilealliance.org.

[11] É. Germain, M. Dulipovici, and P.N. Robillard, "Measuring Software Process Activities in Student Settings", Proceedings of the 2nd ASERC Workshop on Quantitative and Soft Computing Based Software Engineering (QSSE 2002), Banff, AB, Canada, 2002, pp. 44-49.

[12] É. Germain, P.N. Robillard, and M. Dulipovici, "Process Activities in a Project Based Course in Software Engineering", Process Activities in a Project Based Course in Software Engineering, IEEE, 2002, pp.S3G-7 – S3G-12.

[13] P.N. Robillard, "Measuring Team Activities in a Process-Oriented Software Engineering Course", Proceedings of the 11th Conference on Software Engineering Education and Training (CSEET '98), IEEE Computer Society, 1998, pp. 90-101.

[14] P.N. Robillard, "Teaching Software Engineering through a Project-Oriented Course", Proceedings of the 9th Conference on Software Engineering Education (CSEE), IEEE Computer Society, 1996, pp. 85-94.

[15] K. Beck, "Embracing Change with Extreme Programming", Computer, 10/1999, pp. 70-77.

[16] S. McConnell, "I Know What I Know", IEEE Software, 05-06/2002, pp. 5-7.

[17] Juran J.M., F.M. Gryna, Jr., and F.M. Bingham, "Quality Control Handbook. Third edition", McGraw Hill, New York, 1979; cited in Fenton, N. and Ohlsson, N., "Quantitative Analysis of Faults and Failures in a Complex Software System", IEEE Transactions on Software Engineering, 08/2000, pp. 797-814.