A Software Process for Time-constrained Course Projects

Wilson P. Paula Filho
Synergia Systems and Software Engineering Laboratory
Computer Science Dept. - Federal University of Minas Gerais
Av. Antônio Carlos, 6627 - Belo Horizonte - MG – Brazil
wppf@ieee.org

ABSTRACT
Defined software engineering processes help to perform and guide software engineering course projects. However, several difficult issues are involved in designing a software process for this purpose. This design is even harder when it must suit time-constrained course projects. Here, we discuss several issues concerning such processes, focusing on an educational setting.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education – information systems education.

General Terms
Management, Measurement, Standardization.

Keywords
Software process, course project, lifecycle model, software engineering education.

1. INTRODUCTION
This contribution concerns software processes for time-constrained projects, in software engineering courses. The design of processes for course projects must satisfy various conflicting goals. When those projects must be performed under tight time constraints, the process trade-offs becomes even harder. Here, we discuss the development of one specific process, but we try to factor out its design issues, discussing their rationale, and comparing them with other approaches.

In the second section, we review the background for this work, and introduce the Praxis process. The third section discusses process design issues. In the fourth section, we briefly discuss process implementation, and process test is discussed in the fifth section, leading to the conclusions in the sixth section.

2. BACKGROUND
Usually, the main goal of a software development process is the delivery of a high-quality, cost-effective software product. In an educational software development process, used to teach or perform a course project, the main goal is teaching or learning software development; the cost-effective delivery of a quality product becomes a criterion to measure the acquired proficiency, rather than the primary goal.

This work reports our experience with Praxis, one educational software development process [1]. Praxis reuses and extends concepts from several widespread processes, such as PSP [2], TSP [3], MBase [4], the Unified Process [5], and XP [6]. Also, it draws from the IEEE software engineering standards [7], and the SEI SW-CMM maturity model [8]. It is aimed primarily at education and training in software engineering, of individuals, teams, and organizations. It has been applied in undergraduate and graduate courses, as well as industrial training programs. Praxis has also been tailored for use as a real industrial process, by our own organization and others.

3. PROCESS DESIGN
In this section, we discuss the most significant Praxis design issues: overall process architecture; time constraints; lifecycle model; project artifacts; verification and assessment; data collection; and reusability of a process framework.

3.1 Process Architecture
The Praxis process architecture broadly follows the Unified Process. Its elements are organized along three dimensions: artifacts (concrete work products), disciplines (knowledge areas), and iterations (minor temporal divisions). Disciplines are composed of partially ordered, conceptual workflow steps called activities. These have artifacts as inputs, and their outputs are artifact sections. To the Unified Process disciplines, Praxis adds project management, quality management, and process engineering, whose activities map to SW-CMM practices. Those disciplines are quite different from the disciplines of the Rational Unified Process (RUP) [9], also a concrete counterpart to the Unified Process; they are much simpler, as suits an educational process.

Iterations are minor formal divisions of project schedules; at their end, developers must deliver artifact baselines. Those baselines must pass quality criteria, such as reviews, tests, and audits. Iterations are ordered steps of phases (major temporal divisions). The standard phases are Inception, Elaboration, Construction and Transition, as in the Unified Process.

3.2 Time Constraints
So far, we have applied Praxis in a number of undergraduate and graduate software engineering courses. Among the latter, the hardest constraints appeared in an industry-oriented graduate program (a kind of lightweight M. Sc., called specialization in Brazil). This program recycles industry professionals in technologies and processes, with strong focus on practical aspects. These
professionals have previous programming experience, but many of them are not familiar with object-oriented practices.

Lectures are held at nights and weekends, since most of those students have daytime jobs. Therefore, time for homework is quite limited, and course effectiveness is especially important. This renders that program a challenging test bed for the alignment of education process and software process. The Praxis process itself has evolved, thanks to the feedback provided by its application in this and other programs. More detailed data about the process application is reported in [10, 11].

For this application, Praxis was required to fit projects of adequate size. The typical project develops a small information systems application, with size around 100 function points (as defined in [12]). This size proved to be large enough to exercise a meaningful set of important software engineering practices, in the available course time budget of 120 lecture hours. The available time frame for the completion of the course projects has averaged twenty elapsed weeks.

3.3 Lifecycle Model
The spiral lifecycle model exercises activities of all process disciplines in every phase and iteration. Therefore, it would require that students start the course projects having, at least, theoretical knowledge about all process disciplines. This requirement is a problem if the project course is the first or only software engineering course in a program, as often happens. For this reason, Praxis follows a hybrid model, going from waterfall to spiral.

In Praxis, the Inception iterations follow a waterfall style. They are heavily oriented towards requirements engineering; therefore, they may performed just after those concepts are taught in the first course lectures.

The Elaboration iterations emphasize analysis and architectural design. The students are required to fully implement the first few use cases. For real-life projects, these use cases should be chosen to address major architectural and technological risks. For students, our experience has confirmed that the first major risk is not properly learning the process itself. Therefore, the main purpose of the Elaboration phase for course projects is to help the students master the process.

Praxis Elaboration partially follows a spiral model; the only departure from full spiral is that analysis must be completed for all use cases. The Praxis standard for the Analysis model was designed to ease identification of the function point counting elements, such as elementary processes and logical files. Therefore, a complete Analysis model is necessary for precise function point counting. Function point sizes and measures of effort spent in the Elaboration use cases allow firmer estimates for Construction.

The Construction phase is performed within a full spiral model. Every iteration develops a set of the remaining use cases, to full implementation and validation. The process disciplines are repeatedly applied for every use case, reinforcing their practice.

Such a hybrid waterfall-spiral corresponds, to some extent, to the actual workload profile of the Unified Process and the RUP. According to our experience, it is preferred by many customers and managers, since it allows fixed-price contracts for Construction, which represents about 70% of the typical project effort, while a pure spiral lifecycle either requires time plus materials pricing, or variable scope with fixed price.

3.4 Project Artifacts
In the current Praxis version, work artifacts are both consumed and produced by process activities. In the course of a project, other artifacts, such as documents and components, are either consumed (for instance, standards or reusable libraries) or produced (for instance, specifications or code). Models are the main work artifacts, from which all other work artifacts derive. A single project uses just three UML models: the Process model, the Analysis model, and the Design model. Non-UML spreadsheets and text attachments complement those models.

While models are technical artifacts, databases and reports are management artifacts, used for project and requirements management. They are currently held in spreadsheets, since educational projects do not need high volumes of managerial data. For professional use, project management tools, relational database management systems, report generators and workflow tools should replace most spreadsheets.

Currently, Praxis requires three main project management artifacts. The requirements database records primary and derived requirements, maintains traceability among them, describes project scope, provides data for function point sizing, and tracks progress in meeting the requirements. The project planning workbook imports size data from the requirements database and historic effort data from historical process databases, to provide effort, cost and schedule forecasts. The project monitoring report imports progress data from the requirements database, actual effort data from effort logs, and defect data from the quality management artifacts. It tracks risks and issues, and compares planned and achieved project status. The remaining required reports are outcomes from quality management activities (reviews, tests, user appraisals and quality audits).

3.5 Criteria for Verification and Assessment
Verification and assessment of the output artifacts might be a heavy workload for the instructor and teaching assistants. We addressed that by using standardized inspections, reviews and tests, as verification and assessment criteria. Currently, the process includes inspections of requirements, analysis, user-interface design, test design, internal design, implementation, and test reports. The students, acting as inspectors, record their findings in standardized reports. For each defect found in an inspected artifact, its authors must either fix the defect or rebut the finding. The groups must be partitioned into subgroups; each inspects material authored by a different subgroup.

The students write automated test scripts, using a framework based on the JUnit tools [13]. The integration test cases must cover all significant use case scenarios, and significant samples of valid and invalid user-interface inputs. The process framework generates a test report, with test logs and database snapshots. By each iteration end, the use cases must have passed their entire unit and integration test suites. The students are instructed to perform three more quality assurance steps; teaching assistants will perform the same steps, during the grading process.
First, in acceptance testing, the test scripts and test reports must be inspected for coverage, correctness and conformance to the test specifications. Also, supplementary manual tests cover features that are best checked by humans, or try to break the product, as real users often do. A user appraisal evaluates product usability and user documentation quality. It is expected that the most important usability glitches be addressed in the early design prototypes, leaving just residual issues for this step. Finally, the process requires a quality audit, which re-inspects project artifacts or samples thereof, looking for issues of non-compliance with standards, or inconsistency within or among artifacts.

The project grade is determined by the amount and severity of the errors found by the teaching assistants, in their quality audits. Therefore, the final work grade becomes just one more quality metric, providing even tighter integration between process and course.

3.6 Data Collection
In Praxis, data collection is a central process concern, since monitoring the student workload is essential for trimming the projects to feasible sizes, and for balancing that workload among software disciplines. Applying the Goal-Question-Metrics paradigm [14], such goals raise the following questions: Given a maximum allowed workload, what is the maximum feasible project size? How is that workload distributed among disciplines?

To gather metrics that answer those questions, the students must collect effort data per artifact and discipline, in iteration effort logs, attached to the project monitoring report. Also, defect data are collected in every quality management report, and size data, such as counts of logical lines of code, classes and tests, are measured with appropriate tools.

Data quality is a quite difficult issue. Many students do not take data collection seriously enough, and data fudging is a concern. Therefore, we designed the effort logs so that we may detect, at least, the grossest inconsistencies. The students record fine-grained artifact-relative effort data; their aggregation is performed by automated spreadsheets. This and other automation resources reduce transcription errors. A few failed attempts by former students sometimes hard to build from scratch. Examples are also necessary, to provide guidance on artifact building. Templates and examples are kept in a support site, where new or updated material may be posted as needed.

Many industrial-grade software engineering tools are expensive. Praxis uses plain spreadsheets for most artifacts or attachments. Free software tools are used, when suitable, such as the JUnit tool. For UML modeling, either the instructor must use free tools, or she/he must enroll in one of the academic support programs offered by the industry. Currently, we use Rational Rose, supported by the IBM Academic Initiative.

5. PROCESS TEST
So far, we have applied the Praxis process in seven software engineering classes, in the kind of program described in Subsection 3.2. Here, we show the results for the seventh class, the first one to develop a complete application using the current Praxis version. Data for former classes is shown in [10, 11].

The proposed projects were information processing applications for small business organization. We supplied a grocery store information system, as a sample project. Table 1 shows the collected size data, with class average and standard deviation, as well as data for the supplied sample. The problem functional complexity was measured in non-adjusted function points (NAFP), since their counting is more objective than their adjusted counterparts. Logical lines of code (LOC) are bound to physical lines via the coding standards; these are counted by a tool that enforces a counting standard.

As Table 1 shows, the number of lines of code per class is similar to the sample; inspection and audit data show that this is partly enforced by the coding standards, and partly by the process design style. On the other hand, student data shows that they wrote fewer classes, and less code per function point, as compared to the sample. After the Conception phase, the students were allowed to trim their proposals to a size about a hundred function points. To meet course schedules, they chose simpler use cases, which allowed a higher degree of reuse than some more complex use cases shown in the sample.

Data for use cases and test cases are quite close to the sample data, showing that the process style for requirements and tests was closely followed. But the test code size was smaller than in the
sample, for the same reasons as the product code size. Most specified test cases were passed, especially at functional level (control layer); this indicates satisfactory compliance with functional requirements.

<table>
<thead>
<tr>
<th>Table 1. Size data for process application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metrics</td>
</tr>
<tr>
<td>Functional size</td>
</tr>
<tr>
<td>Code size</td>
</tr>
<tr>
<td>Code size</td>
</tr>
<tr>
<td>Class size</td>
</tr>
<tr>
<td>Relative code size</td>
</tr>
<tr>
<td>Use case size</td>
</tr>
<tr>
<td>Use case complexity</td>
</tr>
<tr>
<td>Specified test cases</td>
</tr>
<tr>
<td>Acceptance test cases</td>
</tr>
<tr>
<td>Control layer test cases</td>
</tr>
<tr>
<td>Entity layer test cases</td>
</tr>
<tr>
<td>Test code size</td>
</tr>
<tr>
<td>Test code size</td>
</tr>
<tr>
<td>Test code size</td>
</tr>
<tr>
<td>Relative test code size</td>
</tr>
</tbody>
</table>

The average of the total recorded effort was 9.583 person-months (PM), with a standard deviation of 3.247 PM. The recorded work hours were converted to PM, assuming that a normal development work load would be 120 hours per month, the rest being spent in meetings, training etc. This yields an average productivity of 11.7 NAFP/PM, compatible with what we have found in industry work.

The spent effort peaked in the Elaboration iterations; this was the phase where the students had to learn most of the process and framework. Considering effort per discipline, the implementation effort is the highest, but the proportions for requirements, analysis, design, and test are close to what should be expected in a model-based, test-driven process. For space reasons, we do not show here a complete effort breakdown per iteration and discipline, but it agrees with the process expectation. This seems to indicate that the process worked as it was taught to the students.

6. CONCLUSIONS
According to our experience, teaching a course project based on a defined software process is an effective way to teach relevant software engineering issues and practices, within limited resource and time budgets. We have found this especially useful for recycling industry-based students. However, careful process design and evolution is necessary, leading to many compromises among conflicting process goals.

So far, we have been able to accommodate reasonably sized projects within a schedule compatible with an industry-oriented graduate software engineering course. This was done using a process that shares many traits with widespread industrial processes, exercising test-driven, model-based, object-oriented development. For our next classes, we expect to tune this process, improving student work quality.

7. ACKNOWLEDGMENTS
We thank IBM Rational for supporting this work, within the IBM Academic Initiative.

8. REFERENCES