Using Model-Driven Development in Time-Constrained Course Projects

Wilson Pádua
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

Educational software development processes, used in course projects, must exercise practices and artifacts comparable to similar industry-level processes, while achieving acceptable productivity and quality, and, at the same time, complying with constraints on available student time. Here, we discuss our experience with a specific model-driven development process, applied in a time-constrained software engineering course. The course projects are developed in iterations, each delivering a subset of the product functions. These, specified as use cases, undergo a sequence of model transformations, until they become tested code. Transformation steps are verified using standardized quality gates (inspections, tests, and audits), which serve three purposes: teaching verification, validation and quality assurance; helping to assess and grade projects; and providing feedback for process improvement. Size, effort and defect data is recorded in standardized reports. Collected data show that the quality gates proved effective to ensure compliance with the prescribed process, and that using a balanced reusable framework is necessary to achieve satisfactory productivity and quality.

1. Introduction

Software development processes try to guide software engineers towards delivery of high-quality, cost-effective software products. Educational software development processes help to teach or perform course projects, where cost-effective delivery of quality products remains a major success criterion. In the design of educational processes, there are trade-offs between software engineering goals, such as product quality, project productivity and risk management, and educational goals, such as range of covered material and effectively acquired proficiencies.

These trade-offs become harder if the educational process is to support time-constrained course projects. This happens often, both in Computer Science programs, where Software Engineering may be covered in one or two courses among many, and in industry-oriented programs, taken by professionals who must perform coursework in their spare time, usually at nights and weekends.

Model-driven processes are based on “the notion that we can construct a model of a system that we can then transform into the real thing” [1]. Although almost everybody uses at least a few informal models as analysis and design aids, model-driven processes assume lifecycles centered on model transformations. We discuss our experience with Praxis, a model-driven process designed for education and training in software engineering. Its main features and some results of its application have been presented elsewhere ([2], [3], [4]). Here, we focus on the aspects that are more relevant for time-constrained course projects.

In the second section, we review the highlights of those aspects of the Praxis process. The third section describes a case where the process was applied in time-constrained course
projects. Results are discussed in the fourth section. The last section presents conclusions and ongoing work.

2. Process highlights

2.1. Model transformations

With Praxis, developers work on two UML models: the problem-oriented analysis model, and the solution-oriented design model. Code is generated from the latter. Fig. 1 shows those models, with their views and transformations. These are represented by several kinds of UML dependency relationships: «refine», «realize» and «derive», as defined in UML 2.0 [5]. The analysis model describes the problem to be solved, in a requirements view and an analysis view. The design model describes the adopted architectural and technological solution, in an external design view, a test view, and an internal design view.

![Figure 1. Meta-model of Praxis views and transformations](image)

The requirements view expresses product functional requirements as essential (design-free) use cases, whose behavior is specified by low-formality text. Spreadsheet attachments record user interface and non-functional requirements. A low-fidelity prototype attachment helps to clarify interface and functional requirements.

The analysis view uses analysis classes and their attributes and operations, to represent problem domain concepts, user and system interfaces, and, if necessary, algorithms and rules. Collaborations of those classes realize (satisfy the behavior specified by) the use cases, in a conceptual and technology-free way, verifying whether the concepts represented by the classes suffice to implement the use cases. Each realization includes sequence diagrams,
illustrating scenarios that match use case behavior. The process supplies heuristics to count function points [6] from this model, which then becomes the base for scope sizing.

The external design view represents proposed solutions for product user interfaces and how users interact with them. Major user interface elements, such as screens, fields, and commands, are modeled by stereotyped classes, attributes and operations. UML associations among components represent parent-child, containment, and navigation relationships. Changes in the interface elements, such as habilitation and visibility, are described by state diagrams. User interactions are described by design use cases, where user actions and product reactions are modeled by messages in sequence diagrams. High fidelity prototypes supplement the design use cases, helping to visualize proposed behavior. The external view must provide for usability, testability and ease of implementation, while satisfying the requirements stated in the analysis model.

The test view represents the test design specifications for the product, as defined in [7]. Typically, each design use case is realized by a test, which assembles test procedures (sequences of actions for the execution of a test) and test cases (specifications of inputs, predicted results, and a set of execution conditions), besides descriptions of tested features, test approach and success-failure criteria.

In the internal design view, the design use cases are realized by collaborations of internal design classes. The process mandates a layered architecture; every internal design class must reside in one of three layers, from the bottom upwards: entity (domain concepts), control (validation and flow sequencing), and boundary (presentation). These design layers also refine the analysis layers, reorganizing and detailing them, as determined by non-functional requirements and adherence to the chosen implementation technologies.

Praxis includes a reusable components framework, which helps to write all design views, as well as product and test code. For the external design view, the framework provides for base abstractions and components for user interface elements, and abstract design use cases that represent common user interaction patterns. The framework test view contains specifications for abstract test patterns. Generally, these mirror the abstract user interaction patterns, containing common test procedures and parameterized test cases. Finally, the framework internal design view includes base classes and reusable components for every application layer, plus additional persistence and system (general utilities) layers. The external design patterns have counterparts in test and internal design.

2.2. Coding and testing

Code is divided in product code, which actually implements product functions, and test code, which ensures their conformity with requirements. The product code is generated from the internal design model, using the Java plug-in of the modeling tool, whose reverse engineering capability is used to synchronize design and code. Most product classes inherit from base framework classes, or compose framework components.

Praxis adopts test-driven implementation practices: unit tests are written together with product code, and passing them is required for code approval. The developer must write three test suites, exercising, respectively, the entity, control and boundary layers. The entity layer tests derive from internal design, the control layer tests from the functional aspects of the test view, and the boundary layer tests from both functional and presentation aspects of the test view. All test suites reuse the Praxis test code framework, itself based on JUnit tool [8].
2.3. Use-case driven development

Praxis views development as a sequence of model transformation cycles, going from requirements to code, for every use case. After key transformations, a use case reaches a new development state, if it passes a set of verification procedures, called quality gates. Each use case workflow is a development micro-cycle, bound to the project management macro-cycles by mapping use cases to iterations (minor divisions) and phases (major divisions).

### Table 1. Use-case states, work products and exit quality gates

<table>
<thead>
<tr>
<th>State</th>
<th>Work products</th>
<th>Quality gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified</td>
<td>Requirements view (use cases)</td>
<td>Management review</td>
</tr>
<tr>
<td>Detailed</td>
<td>Requirements view (use case flows)</td>
<td>Requirements inspection</td>
</tr>
<tr>
<td>Analyzed</td>
<td>Analysis view</td>
<td>Analysis inspection</td>
</tr>
<tr>
<td>Designed</td>
<td>External design view</td>
<td>External design inspection</td>
</tr>
<tr>
<td>Specified</td>
<td>Test view</td>
<td>Test design inspection</td>
</tr>
<tr>
<td>Realized</td>
<td>Internal design view; code and test scripts for entity and control layers</td>
<td>Implementation and internal design inspection; unit tests</td>
</tr>
<tr>
<td>Implemented</td>
<td>Code and test scripts for boundary layer; executable code</td>
<td>Implementation and internal design inspection; integration tests</td>
</tr>
<tr>
<td>Verified</td>
<td>Verified executable code; test reports; user documentation</td>
<td>System tests</td>
</tr>
<tr>
<td>Validated</td>
<td>Validated executable code; user appraisal report</td>
<td>User appraisal</td>
</tr>
<tr>
<td>Complete</td>
<td>Complete code and artifact baseline; quality audit report</td>
<td>Quality audit</td>
</tr>
</tbody>
</table>

Table 1 summarizes the standard set of development states and quality gates, discussed in more detail in [3] and [4]. For the first seven states, the gates include reviews and inspections that follow the respective IEEE standard [9]. After the Specified state, product code is written, and tests become possible. Testing proceeds from unit tests to integration tests, and then to system tests. The user appraisal includes acceptance tests and evaluation of the product documentation and usability. The final quality check is a quality audit, following the same IEEE standard. A release plan defines which uses cases to develop, in each iteration.

3. Process application

3.1. Application environment

So far, we have applied Praxis in many undergraduate and graduate courses. The hardest constraints appeared in an industry-oriented graduate program, which recycles industry professionals in technologies and processes, with a strong focus on practical aspects. These professionals have previous programming experience, but many of them are not familiar with object-oriented or model-driven 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. Its typical course project develops a small information systems application, with a minimum size of 100 function points. This proved large enough to exercise a meaningful set of software engineering practices, taught in the available course time budget of 120 lecture hours, in about twenty elapsed weeks. Students usually work in self-selected teams of five to seven.

In this program, the students receive four software engineering grades, one for each course unit. Project grades represent 60% of course grades; they are based on the same quality audit
procedure that was performed by the students in every project iteration, using the same checklists. They reflect product quality, as might be perceived by users, but also the quality attributes of project artifacts, such as compliance with process standards, internal consistency, and traceability. Penalties vary with defect criticality, the heaviest penalties being levied on non-passed acceptance tests, and on defects that might distort project metrics.

3.2. Data gathering

The grading procedure itself yields the overall quality metrics, for every project delivery. The project reports also provide defect data for every quality gate; the defects themselves are considered as normal process outputs, and do not count for grading, but those reports must pass several validity checks.

Collected size and scope counts include function, implemented classes and lines of code, and specified and passed tests. Counting them is straightforward and partly automated, so that instructors may easily repeat it during the quality audits, to verify student data. Currently, we use non-adjusted function points, since these reflect purely functional complexity, complying with ISO/IEEE guidance for functional sizing [10]. Functions points are used to delimit the scope of project and iterations, and to normalize other data. For lines of code, counting follows the PSP standard [11], and the code itself must follow a rigorous format standard, based on usual Java practices [12].

Collecting reliable effort data is not easy, since it is necessary to uncover mistakes in data recording, or even data faking, used to cover-up laziness in data recording and transcription. So far, the process underwent three generations of effort log templates; currently, the log entries record task nature, start and end date and time, and task participants. Those entries are checked for consistency with data in quality assurance and configuration management records. Automated spreadsheets extract useful indicators, such as productivity, or effort distribution across process iterations and disciplines.

4. Results discussion

In this section, we discuss some indicators that yield information about process performance. For every chart, the indicators shows the numbers assigned to the course classes whose data were collected (class 9 was a canceled offering). Those classes used essentially the same Praxis version; only minor corrections and improvements in the templates and checklists were introduced during this period. One exception was the last change in the effort log template, introduced starting from class 10. In Figures 2 and 3, every data point represents the average for one course class; class 7 had four project teams, and the others had three.

Project management requires stable processes, since stable data are required for project estimation and forecasting. Stability is also a requisite for process improvement, necessary to check whether the actual, practiced process complies with the official, prescribed process, and to assess improvement in process performance.

In Figure 2, size indicators, averaged per course class, are used to check for stability of design, test and implementation practices. The left chart shows counts of function points (FP) and lines of code (LOC) for product and test code; LOC are normalized per Java class and FP. The LOC counts are nearly constant, for product and test code, per class and per function point. This ensues from the required compliance with design and code standards. Moreover, both kinds of code show about 30 LOC/FP, much lower than often quoted industrial counts (around 50-60 LOC/FP), and even lower than what we have found in many real-life projects. This is a consequence of reuse of design, test and code. Indeed, we have introduced additional requirements for each course class, to make reuse a bit more difficult; the slight increase in
FP count is probably a consequence. Interestingly, the students found ways to design around the additional requirements, trying to keep a high level of framework reuse.

The right chart shows test case (TC) counts: specified TC, as well as implemented (and passed) TC for every architecture layer. Boundary layer tests are used for acceptance, so that a product where this TC count matches specified count is probably conformant to the specified requirements; this matching improved in the last classes, where the penalty for not passing every TC was raised. Indeed, manual tests performed during grading could find just minor defects, in the worst cases. Also, control layer TC counts are somewhat higher than boundary layer counts, since they check for abnormal cases which, for the boundary layer, are ruled out by GUI design; entity TC counts are much smaller, since most abnormal cases are handled by the control layer. All these findings confirm the respective process guidance.

**Figure 2. Size: classes, lines of code and tests**

**Figure 3. Effort distribution per iteration and discipline**

Figure 3 shows effort distribution per iteration (left) and discipline (right). Each area segment is labeled by the corresponding iteration or discipline. The Y axis shows percentages of total project effort. Both charts show that the profile of effort distribution was quite stable, despite the wider variations in total effort, due to productivity differences among teams. Those profiles are also consistent with process expectations. The effort in the first Inception iteration (I1) is very low, since it consists in little more than writing a statement of work; I2 is heavier, since it includes most of the requirements work. The Elaboration iterations require more effort, since the teams implement complete use cases, and learn most of the framework and of the design, test and implementation practices. In the first Construction iteration (C1),
better performance in the technical disciplines is partially offset by the effort to learn more complex project management artifacts. In C2 the teams repeat what they have learned, spending less effort.

The distribution per discipline is also consistent with process expectation: requirements (RQ) and analysis (AN) require about one fourth of total effort; implementation (IM) share is much smaller than in traditional code-and-fix, with significant effort being transferred to design (DS) and test (TS); project management (PM) and quality management (QM) efforts are much lower, confirming that management overhead is within reasonable limits.

Concerning quality and productivity, these metrics showed a much higher variation within and across course classes, probably reflecting proficiency and commitment differences among teams. Productivity ranged from 5.0 to 23.6 FP per person-month; even the worst case might be considered acceptable [13], and the best teams had a very good performance. Since these are data for people who are yet learning modeling and processes, they seem to show that the process did not adversely affect productivity. Regarding quality, deliveries are considered acceptable if they show less than 40 defects, this count being adjusted for defect criticality. The teams may improve their grades by performing an additional iteration and producing a corrected version of the artifacts. In such cases, they are usually able to perform this in a few additional days, confirming the low severity of most defects.

5. Conclusions and future work

The process analyzed here was used in course projects, in four classes of an industry-oriented graduate program, where student teams developed small (circa 100 function points) Java applications. This might seem a small sample, indicating a study limitation. Actually, Figs. 2 and 3 summarize 104 data points, representing 8 different indicators, measured for the 13 teams that took part in those four classes. Of those points, only 36 fell outside the $\mu \pm \sigma$ limits (average plus or minus one standard deviation), and none at all outside the $\mu \pm 2\sigma$ limits. Although we did not undertake a formal statistical process analysis, this seems to show considerable stability of those indicators.

The collected data showed that the students actually used the prescribed process, complying with it to a satisfactory degree. This was a major goal of the course, since the developed application was simple, and the main point of the course project was to how to use models and model transformations in its development. The process quality gates, comprising mainly inspections, automated tests and quality audits proved instrumental in achieving that goal. The final quality audit for each use case was especially important, since it allowed the instructor to recheck the inspections and tests, to verify traceability, from requirements to test results, and to grade project in a manner consistent with their overall achieved quality.

The size data also show, in both product code and test scripts, a large degree of reuse of the process-supplied framework. A second conclusion is that a balanced reusable framework is essential to render the projects feasible within their time and effort constraints. A weak framework would have the students reinventing wheels many times, while introducing defects in those extra model and code pieces; that is precisely what happened with the project of the first course classes (numbered 1 to 6), when the framework was not mature enough. If the framework is too strong, as it happens in rapid application development environments, then there is not much to learn, especially about modeling. Even using a balanced framework, we found that it is necessary that the first project use cases be very straightforward, in order to teach the framework itself, but, as the course project advances, the students must be required to implement use cases which extend the framework in a non-trivial way.

In our university laboratory, a tailored and enhanced version of the Praxis process, called Praxis-Synergia, has been used to develop larger real-life applications (typically more than
1000 FP), by teams of tens of developers [14]. Such projects not only change scale, but use more complex, Web-based technologies, interact with real-life customers and users, and are subject to high personnel turnover. Therefore, this industrial process version has never been stable enough to undergo a detailed quantitative analysis. But Praxis-Synergia uses test-driven practices, quite similar to those of the educational version; those practices have driven product quality to a very good level (0.018 field defects per function point per year [15]). On the other hand, the sequence of model transformations and quality gates described here has been only partially adopted in Praxis-Synergia; this has reflected in lower reuse, as shown by its higher LOC/FP indicators, and in its difficulty to drive productivity above a ceiling of about 12 FP per programmer-month. Praxis-Synergia proves that the educational Praxis model sequence is scalable to real-life applications and technologies; but it also shows how hard it is to scale the quality gates and reusable framework, due to non-technical constraints, such as personnel turnover, organization culture and business opportunities.

Current process improvement work leverages on collected course data to produce a new major Praxis version, based on Eclipse, UML 2.0 and other recent advancements.

5. Acknowledgments

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

6. References