Quality Gates in Use-Case Driven Development

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
In use-case driven software development, a project is organized as sequence of use-case development workflows. Here, we describe a process where each workflow instance is a sequence of model transformation steps, whose outcome has to pass defined quality gates. This process was applied in two classes of an industry-oriented software engineering course. Collected size data confirm the positive influence of inspections on the quality of design gates. This process was applied in two classes of an industry- transformation steps, whose outcome has to pass defined quality factors. Inspection and audit data confirm the positive influence of inspections on the quality of design material, but show that improvements are needed for code and managerial artifacts.

Categories and Subject Descriptors
D.2.9 [Software Engineering]: Management – Software quality assurance, Software process models, Life cycle.

General Terms
Management, Measurement, Documentation, Design, Standardization, Verification.

Keywords
Software process, course project, use case, software quality.

1. INTRODUCTION
Iterative software development processes have become widespread in industrial practice. Some of these are use-case driven, that is, “the development proceeds through a series of workflows that derive from the use cases”, as defined in the Unified Process [1]. Examples include the RUP [2] and XP [3].

Use cases capture slices of system requirements [4]. Use-case driven iterative processes aim to implement a subset of the use cases of a system, by each iteration end. This eases getting users’ feedback, managing product size, and negotiating possible scope changes, or cost and schedule overruns.

In this paper, we discuss some issues related to quality in use-case driven development, bearing on our experience with the Praxis process [5]. This is a software development process designed for education and training in software engineering; its highlights are presented in [6], [7] and [8]. Praxis has also been tailored for use as a real-life industrial process, by our own organization and others.

The second section outlines a use-case development workflow, as specified by Praxis. The third section explains the quality gates that mark the milestones in those workflows. The fourth section presents a short experience report. The last section summarizes conclusions.

2. USE CASE DEVELOPMENT
Praxis has a strong focus on developer guidance, consistently with its prime objectives as an educational process. It has been evolving in the last years, drawing lessons mainly from its application in software engineering courses. We have found that most beginner developers need to be told not only which tools and techniques they should use to implement the use cases, but which steps they should follow for every use case. Moreover, Praxis is also a model-based process; it prescribes which model transformations the developers should perform, to go from requirements to validated final code.

2.1 Use case development states
Praxis models development of each use case workflow as a sequence of states. The name of each development state evokes how far the use case has advanced towards complete implementation and validation. Each state is associated with a set of resulting work products [8], which become inputs to the following steps.

Table 1 (next page) summarizes the standard set of use case development states, adopted in the current version of Praxis.

The first three states reach an increasingly more precise specification of the problem slice that the use case represents. From a brief, typically single-paragraph description (Identified), the use case evolves to a collection of event flows that express functional requirements (Detailed), and then to their realization in the conceptual problem domain (Analysis). These three states are problem-oriented. Their main artifact is the Analysis model, containing essential use cases, as defined in [9].

To reach the Designed state, the developer transforms the analysis use cases into design use cases. These describe user-system interaction in terms of user interface elements, abstracting the features the user sees and uses, while hiding underlying implementation details. The Specified state is reached when a test designer derives test specifications from the design use cases. To attain the Realized state, the developer must design and implement a collaboration of classes in the lower architecture layers, passing the specified functional tests. In the Implemented state, finally, the upper layer classes must pass both functional and presentation tests. These four states are solution-oriented.
The Verified state is achieved when the use case passes automated and manual testing by an independent test team. A user review and appraisal of the new product function and companion documentation leads to the Validated state. The Complete state is met when the whole material produced in this use case cycle passes a quality audit, and is deemed ready as project baseline. These three last states are quality-oriented. They are meant to assess final product quality. In course applications, they are repeated by the instructor; in real life, they should be repeated by the costumer, or by an independent validation team.

<table>
<thead>
<tr>
<th>State</th>
<th>Work products</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified</td>
<td>Use-case model (analysis model).</td>
</tr>
<tr>
<td>Detailed</td>
<td>Use-case description (analysis model); requirements inspection report.</td>
</tr>
<tr>
<td>Analyzed</td>
<td>Use-case realization; detailed class model (analysis model); analysis inspection report.</td>
</tr>
<tr>
<td>Designed</td>
<td>Design use-case description; user interfaces model (design model); external design inspection report.</td>
</tr>
<tr>
<td>Specified</td>
<td>Test model and specification (design model); test design inspection report.</td>
</tr>
<tr>
<td>Realized</td>
<td>Design realization (design model); source code and unit test scripts for entity and control classes; implementation and internal design inspection reports.</td>
</tr>
<tr>
<td>Implemented</td>
<td>Source code for boundary classes; executable code; implementation and internal design inspection reports.</td>
</tr>
<tr>
<td>Verified</td>
<td>Verified executable code; test reports; user documentation.</td>
</tr>
<tr>
<td>Validated</td>
<td>Validated executable code; user appraisal report.</td>
</tr>
<tr>
<td>Complete</td>
<td>Complete code and artifact baseline; quality audit report.</td>
</tr>
</tbody>
</table>

### 2.2 Quality assurance gates

Reaching a new state always means passing an objective quality assurance gate, that is, a verification procedure, performed either by independent reviewers or by automated scripts. Therefore, state transitions are not solely dependent on the developer’s personal appraisal.

Reviews and inspections follow the guidelines of the respective IEEE standard [10]. Since the Identified state is extremely simple, a management review is considered enough (this is true for the standard educational process, not necessarily for real-life tailored versions). For each of the following six states, the respective work products undergo inspections, conducted along process-defined checklists.

After the Specified state, executable code is written, and tests become possible. Testing proceeds from unit tests to integration tests, and then to system tests (as defined in [11]). The user appraisal includes acceptance tests and evaluation of the product documentation and usability.

The final quality check is typically performed by quality assurance personnel, following a quality audit, along the lines of the IEEE standard [10]. This audit also follows a process-provided checklist, and passing it is a requirement to qualify for an official configuration management baseline.

### 3. QUALITY ASSURANCE PROCEDURES

In this section, we discuss the most relevant procedures used in the quality gates. Each kind of gate is performed in standardized ways, using process-supplied standards, templates and examples. We also discuss the interaction of quality assurance and reuse practices, since these also contribute to reduce defect sources.

Spreadsheets are the main tools for most of these procedures. They are used for inspection checklists and reports, manual tests, user appraisal and quality audit logs, test specifications, and configuration management records. Also, they are used in the analysis and design models, as attachments that hold most non-UML information.

#### 3.1 Inspections

In Praxis, each major model view is verified by an inspection, performed by developers who were not authors of the inspected material, using process-standardized checklists, procedures, and report forms. The inspection procedures contain rules for handling several issues, such as generated code and reused parts, doubts and suggestions, and evolution of the checklists themselves.

A checklist typically contains some tens of items that should be checked in the given material. Those items cover compliance with standards; quality attributes (e.g. precision, completeness, internal consistency); traceability to predecessors (e.g. design and test elements to requirements, code to design); and suitability to successors (e.g. whether design elements are testable and may be implemented with the chosen technologies). Every checklist item has a defect code, based on the PSP classification [12]. We had found, in previous experiments, that most inspectors made frequent mistakes in defect classification.

Every defect found is recorded in an inspection report. Later, during the quality audits, these reports will be checked themselves; for instance, defects codes and counts must be consistent, between report and checklist. But, while a checklist item covers a defect category, a report defect item must describe the defect location and nature, precisely enough for defect removal. Each report defect item contains an action field, where the material author will later report the fixing of the defect, or explain why it was not fixed as required.

Our own experience confirms that one of the greatest barriers to adopting inspections is the mistaken perception that they are too expensive. Often, we have found project managers that are acquainted with literature data about inspection effectiveness, and expect them to spare much rework; nevertheless, they drop inspections, because they did not allocate resources for them, in the budget and schedule of the project. To collect the necessary data, the inspection procedures require that inspectors record their efforts, both for preparation and meetings, and every report defect item has a field to record fix time.

#### 3.2 Tests

Praxis adopts the style of test-driven development used in XP [14]. Unit tests are written together with product code. Passing the unit tests is a requisite for code approval, and they are part of the
quality gates for the Realized state. However, there are some architectural aspects that must be considered.

Praxis mandates the use of a layered architecture. All product code must be classified in one of three layers, from the bottom upwards: entity (domain concepts), control (validation and flow sequencing), and boundary (presentation). The classes of the first two layers, that collaborate to realize a use case, must be coded, inspected and unit-tested, in order to reach the Realized state.

Unit tests for entity classes are defined by the internal design of the collaboration, but the unit tests for the control classes closely follow the test specifications, output from the Specified state. However, there are some architectural aspects that must be considered.

In the current Praxis version, all unit tests are based on a test framework which in turn inherits from the JUnit framework [13].

Current XP practice suggests that the use cases (user stories, in XP parlance) “be translated into acceptance tests... A user story is not considered complete until it has passed its acceptance tests.” [14] In practice, there is a long way to go from informal, text-defined, essential-level use cases to automated test scripts. These scripts must refer to actual user interface elements, and they must follow good test design practices, such as the balance of desired test coverage versus available project resources. The first condition is met by the Defined state, the second by the Specified state.

In Praxis, the control layer unit tests perform double duty, as functional integration tests. On the other hand, to test presentation requirements and design, it is necessary to code and test the boundary layer. This is very dependent on the chosen user interface technology; in many cases, it may be necessary to use a robot-style tool, simulating user action events. Alternatively, the presentation layer might be exercised by manual tests only, leaving the highest level of automated testing at the control layer.

Since our current educational projects are Java applications with Swing GUIs, it is possible to test the boundary classes with the same JUnit-based framework used for the other layers. This is not a perfect solution, since there are occasional glitches in the Swing-JUnit interaction, but it proved useful enough. Therefore, such tests are used in quality gates for the Implemented state, playing the role of integration tests that cover both functional and presentation issues.

Praxis prescribes acceptance tests as part of the quality gates for the Verified and Validated states. In the former, they are performed by an independent test team, focusing on test coverage and product robustness. In the latter, they are performed by end users, checking again product robustness, plus product usability and user documentation quality, and doing a final check of requirements compliance. These quality gates include both manual robustness tests and inspection of the reports generated by the automated test scripts. These are required to leave an audit trail that shows which features were actually exercised.

For professional development, Praxis recommends that the independent test team write its own system test scripts, using a professional test tool. For educational applications, IBM Rational Robot was considered, but not adopted, since its script language (a dialect of Basic) does not lend itself to object-orientation and reuse. We expect to evaluate soon the adequacy of the IBM Rational Functional Tester, a Java scripting tool. Meanwhile, the JUnit-based scripts written by the product code developers are simply inspected for compliance with the test specifications, and rerun by the test team, perhaps in different environments.

3.3 Quality audits

Quality audits are essential to ensure closure of the suite of quality gates. According to [10], “The purpose of a software audit is to provide an independent evaluation of conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures”. To meet this purpose, the Praxis quality audit checks compliance with the process standards, double-checking the remaining quality gates.

The audit rechecks selected items of every inspected work product, evaluating thoroughness of the inspections themselves. For those which do not undergo formal inspections, such as databases, reports and plans, the audit provides a sanity check. Also, the audit checks for traceability and consistency between related pairs of artifacts, such design model versus analysis model, or project reports versus project plans.

The audit also checks the materials and reports of inspections, acceptance tests and user appraisals, for completeness, consistency and compliance with the applicable standards. The configuration management records are also checked. The auditors’ findings are recorded in a defect list, which contains action items for the project personnel. Removal of these defects is required to reach the Complete state.

In educational applications of the process, the quality audit checklist is used also as the grading checklist. For the project appraisers (in our case, teaching assistants), this provides for consistent grading of every project. For the students, their own quality audit should work as a grading preview.

3.4 Reuse

Reuse of quality artifacts improves quality proactively, reducing not only development efforts, but also opportunities for defect injection. Praxis has evolved to make reuse an essential part of its strategy for quality. In the current Praxis version, reusable material includes templates for every artifact, a design model framework and a code framework. Currently, this framework has some 6,000 lines of Java code in 83 classes; about one third of it is in the test layers.

The code framework includes components for every architectural layer, plus additional persistence and system (general utilities) layers. Those components are used via inheritance and composition. The framework also includes test layers, which allow table-driven implementation of most test cases; those tables may be directly generated from the test case specifications, while the base test procedures are coded in the framework test scripts. Test cases and test procedures are defined as in [15].

The design framework provides for modeling counterparts of the code framework components, including the test parts. Also, it includes base abstractions and components for the user interface elements, and abstract design use cases that represent the most common user interaction patterns. The process includes rules for use case inheritance, which UML leaves mostly open ([4], [16]). For most concrete design use cases, this means defining some scenarios that are left abstract (undefined) in the base use case, such as editing specific fields, or looking up specific data.
Generally, user interface framework components and base use cases have counterparts in the test and implementation framework sections. Thus, if a developer reuses a user interaction pattern available in the framework, it is likely that significant reuse of the test and implementation assets will follow.

4. APPLICATION RESULTS

In recent years, we have applied Praxis in many software engineering courses, as a required process for course projects. Most of the courses we taught belonged to an industry-oriented graduate program (a kind of lightweight M.Sc., called specialization in the author’s country). This program recycles industry professional-technical knowledge and processes, with a strong focus on practical aspects of software engineering.

Lectures are held at nights and weekends, since most of the students have daytime jobs. Therefore, time for homework is limited, and the process used in course projects must be quite effective: it must exercise the most important disciplines and practices, while requiring delivery of products with acceptable quality levels, and allowing projects of non-trivial size, within the available course schedules.

The course projects were information processing applications for small business organizations, such as a school or a video rental store; their size was limited to about a hundred non-adjusted function points (17). The students were required to use Java, with Swing GUIs and relational databases, managed by the persistence layer embedded in the process framework.

In the author’s own organization, a tailored version of Praxis was used for real-life applications. It was applied mostly in government information systems projects, ranging from several hundred to a few thousand function points. Those were also written in Java, but used Web technology and professional databases and persistence layer. We have gathered much useful qualitative information from these projects, but, so far, the tailored process has not reached enough stability to provide for normalized data. We are aware of the use of Praxis by other courses and development organizations. So far, however, we did not perform a systematic survey of that usage, and this paper is solely based on our own experience.

In the following subsections, we discuss some data collected in process applications. The term Sample refers to the reference sample project, provided with the process. This is a small grocery information system, with ten analysis use cases. These range from very simple (e.g. user management), to somewhat complex (e.g. sales operations and purchase management). It contains examples of all process artifacts, and it is part of the downloadable process package available in [5]. Although its artifacts are written in Portuguese, it may give a general idea of the kinds of artifacts prescribed by Praxis, since space does not allows us to provide specific examples here.

Class 7 and Class 8 refer to the two classes that used the current process version. Former course classes used previous process versions; some of their data is reviewed in [7]. The data shown are the averages of their project groups. Standard deviations are shown in parentheses. We discuss reuse and test in more detail, since we have better data about those.

4.1 Reuse data

For comparison purposes, solution sizes must be normalized in relation to problem sizes. For the former, we use logical lines of code (LLC); the counting standards are the same as in PSP. For the latter, we use non-adjusted function points (NAFP), since they focus on functional sizing and are standardized by ISO/IEC 20926.

The Sample implements 165 NAFP with relative code sizes of 125.2 LLC/class and 38.7 LLC/NAFP. Class 7 implemented 103.5 (4.9) NAFP, in 119.2 (12.9) LLC/class and 32.1 (5.4) LLC/NAFP. Class 8 implemented 111.7 (8.4) NAFP, in 120.7 (8.9) LLC/class and 29.5 (0.5) LLC/NAFP. It is seen that, for both course classes, the average Java class size (in LLC) is quite similar to the Sample data. This is partly induced by the design style supported by the framework, and partly by the coding standard.

On the other hand, the number of lines of code per function point is about 30 for both course classes; this is about 25% lower than for the Sample. The explanation is found in the respective analysis and design models. Since the students were free to define their requirements, they modeled them so as to take maximum profit of the reusable framework. The Sample, on the other hand, implements more realistic business rules, and some use cases require more ad-hoc realization.

On the positive side, the students were able to make good use of the framework, although this does not contain such facilities as visual design tools, wizards or tutorials. Also, they achieved better productivity and product robustness than previous classes, which used smaller framework versions.

On the negative side, the students did not exercise some difficult design issues, which arise when reuse must be combined with ad-hoc design and code, to address requirements not easily matched by the framework capabilities. To enforce this, the project statements should be enhanced with some additional functional constraints.

4.2 Test data

Table 2 (next page) shows comparative test size data. The test case rows show numbers of test cases per non-adjusted function points (NAFP). Specified test cases are those listed in the test specifications; all other test case counts are the numbers of test methods in the respective test classes, according to JUnit conventions (JUnit nomenclature is not used, since it does not comply with the IEEE standard).

The number of automated boundary layer test cases (that is, those which exercise the boundary layer) is generally slightly lower that the specified test cases, because some of these may be merged in the scripts. When there is a more significant difference, as it happened to Class 7, it means that some required functional features were not exercise by automated testing; there is a grading penalty for this. The coverage of the tests was good in every case, since further manual testing uncovered very few defects, and these were generally minor presentation issues.

The counts for control layer test cases are somewhat higher than for the boundary layer, because sometimes, for robustness reasons, the test framework supports testing some inputs that are not allowed by the boundary layer. For instance, it supports test-
It is seen that the entity layer test cases are about half the control layer test cases. This happens because the control layer is responsible for validating most constraints, which are application-specific, but not domain-generic. Therefore, the control layer tends to have many more test cases with invalid input.

An interesting point is that, for both course classes, the test cases counts are about 25% higher than for the Sample. This behavior seems to be the opposite of the effect on product code size, but is also caused by framework reuse. Within the Sample itself, use case realizations that follow the framework patterns tend to have more screens, input fields, and test cases. Test code size has about the same normalized size for the Sample and the course projects, apparently because those effects cancel each other.

Considering all, we conclude that test-driven implementation was a very significant factor in improving quality and productivity. This is confirmed by the application of Praxis in real-life projects, in the author’s organization (the Synergia laboratory). There, for much larger projects, with more complex architecture and technology, test-driven implementation was considered the single most important factor for recent improvements in product quality.

4.3 Inspection and audit data

Our test data show that most functional and presentation defects are avoided by test-driven implementation. Therefore, the main purpose of inspections is no longer to find that kind of defects, but to ensure consistency, traceability and compliance with standards, all of which promote maintainability.

Also, we expect that inspections improve conformity to requirements and design quality; these kinds of defects adversely impact downstream tasks, causing rework and productivity losses. A positive contribution to maintainability is a long-range effect, which we can only confirm by long-term collection and analysis of maintenance data. As short-term assessments, quality audits provide a measure of how many defects escape inspections, and might remain in the delivered product material.

Table 2. Test data

<table>
<thead>
<tr>
<th>Metric</th>
<th>Class 7</th>
<th>Class 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specified test cases</td>
<td>Avg. 0.70</td>
<td>Std. Dev. 0.92</td>
</tr>
<tr>
<td>Boundary layer test cases</td>
<td>Avg. 0.70</td>
<td>Std. Dev. 0.84</td>
</tr>
<tr>
<td>Control layer test cases</td>
<td>Avg. 0.73</td>
<td>Std. Dev. 0.91</td>
</tr>
<tr>
<td>Entity layer test cases</td>
<td>Avg. 0.38</td>
<td>Std. Dev. 0.42</td>
</tr>
<tr>
<td>Test code size</td>
<td>Avg. 26.9</td>
<td>Std. Dev. 28.7</td>
</tr>
</tbody>
</table>

Table 3. Inspection and audit data

<table>
<thead>
<tr>
<th>Class 7</th>
<th>Class 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Found by students</td>
<td>Avg. 12.5</td>
</tr>
<tr>
<td>Code inspections</td>
<td>Avg. 27.0</td>
</tr>
<tr>
<td>Total inspections</td>
<td>Avg. 39.5</td>
</tr>
<tr>
<td>Quality audit</td>
<td>Avg. 0.0</td>
</tr>
<tr>
<td>Found by instructor</td>
<td>Avg. 2.8</td>
</tr>
<tr>
<td>Code audit</td>
<td>Avg. 20.0</td>
</tr>
<tr>
<td>Design + code audit</td>
<td>Avg. 22.8</td>
</tr>
<tr>
<td>Other - audit</td>
<td>Avg. 31.8</td>
</tr>
<tr>
<td>Total - audit</td>
<td>Avg. 54.5</td>
</tr>
</tbody>
</table>

Table 3 compares several kinds of defects in the two course classes whose data were collected. Design inspection data are aggregates for the different kinds of design inspections, cited in Table 1. The data shown here refer to the last iteration in each project, since this is the point in the course where the students achieve maximum process proficiency. The standard deviations are relatively large, unlike the test data in Table 2, showing that it is far more difficult to standardize inspections than tests.

We focused here on design and code inspections, because each class performed them four times. Requirements and analysis inspections were performed once, in the first Elaboration iteration, on relatively simple materials, since the projects deal with quite small applications. For real-life projects in the author’s organization, requirements inspections have been routinely performed for years, for much larger products, showing a very positive impact on requirements quality.

Students found much more defects in code than in design, as expected, since code material is both larger and easier to inspect. On the other hand, the instructor’s audit showed that the design actually had fewer defects than code, and that most of the design defects were minor documentation problems, without impact on the implementation. Thus, the design artifacts might be considered satisfactory, which is probably a consequence of the large amount of reuse. Indeed, some students of Class 7 later assembled professional-style documentation packages for their projects, as part of a capstone project; they were able to reach a high level of compliance with the process documentation standards, with just a few days work.

Tests capture most functional code defects, and code analysis confirmed that most of the remaining defects were violations of the coding rules. Clearly, in these experiments, code inspections alone were not sufficient to ensure compliance with standards. Other quality resources might help, such as pair programming and automated checks; but we found that students in this course often program in pairs, albeit informally, and that several of those de-
fects were marked by the programming environment, but these markings went unnoticed by the students.

Also, Class 8 found few defects in their own audits, and Class 7 none at all. Since the instructor audits found many re-qualities found are quite low, and those data seem to be quite reliable from rework, caused by defects that escaped the quality gates.

5. CONCLUSIONS
In the Praxis process, the concept of use case development states is employed for fine control of use-case driven software development projects. Transitions between those states entail transformations of artifacts, which must pass quality gates. The Praxis gates combine several kinds of inspections, tests and audits. We discussed how the process uses them to ensure objectively appraisal state transitions, and how they interact with reuse practices.

We presented some data collected in the application of Praxis to course projects. Code size data seem to indicate that design and code process standards were indeed applied, with significant reuse. Also, test size data show that the students correctly applied the quality gates based on automated testing, and manual tests confirmed that they indeed led to quality improvement. On the other hand, inspection and audit data show that our current practices seem to suffice for design material, but need to be improved for code and managerial artifacts.

The weakest point of this study is that it uses data provided by just two student classes, which worked with the current process version. However, these data agree with the trends found in previous classes [7]. Also, for code and test size data, the vari-abilities found are quite low, and those data seem to be quite reliable and consistent. For inspections, we intend to collect more data from future classes, and to compare them with real-life applications of this process; also, we have recently introduced some improvements in the effort log and inspection report forms, in order to improve ease of collection, self-consistency, and reliability of the data.

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

7. REFERENCES