Measuring complexity, effectiveness and efficiency in software course projects

Wilson Pádua
Federal University of Minas Gerais
Rua dos Oitis, 288 – Morro do Chapéu
34000-000 Nova Lima – MG – Brazil
+55-31-3547-4172
E-mail: wppf@ieee.org

ABSTRACT

This paper discusses results achieved in measuring complexity, effectiveness and efficiency, in a series of related software course projects, spanning a period of seven years. We focus on how the complexity of those projects was measured, and how the success of the students in effectively and efficiently taming that complexity was assessed. This required defining, collecting, validating and analyzing several indicators of size, effort and quality; their rationales, advantages and limitations are discussed. The resulting findings helped to improve the process itself.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**: Metrics – *Complexity measures.*

General Terms
Management, Measurement, Documentation, Design, Economics, Experimentation, Standardization, Languages, Verification.

Keywords
Software complexity.

1. INTRODUCTION

As often stated, Software Engineering is about complexity: its techniques are applicable and necessary when the solution of a problem demands software that is complex enough. A software engineer needs to be able to deal, at least, with the following kinds of complexity.

1. **Problem complexity**, embodied in the requirements (both functional and non-functional) that must be met.
2. **Solution complexity**, related to the structural complexity of the code that must be built, and the technical complexity of its platforms and components.
3. **Process complexity**, related to the complexity of the techniques that must be understood and applied, and the complexity of managing the solution development.

ICSE '10, May 2-8, 2010, Cape Town, South Africa
Copyright © 2010 ACM 978-1-60558-719-6/10/05 ... $10.00.

Course projects should give students a taste of real life work in a small scale, within a controlled environment, a limited effort budget and a bounded time schedule. They must be complex enough to give students a taste of the complexity they will meet in their future work assignments; but that complexity must be trimmed enough, to fit the course constraints.

Taming complexity means providing an **effective** solution; that is, the solution must conform to problem requirements and other applicable constraints, and such conformity must be verifiable by a set of acceptance procedures. Besides, the solution must be **efficient**, that is, reachable within given cost and time constraints.

To assess effectiveness and efficiency it is necessary to measure work attributes, such as effort, cost, time, and quality levels. Of course, problem complexity tends to increase effort, cost, and time necessary to reach a proposed quality level; therefore, it is also necessary to measure it, in order to normalize effectiveness and efficiency indicators. Also, problem complexity measures are usually required as input to the estimation of work attributes.

These considerations are valid for both real-life and course projects. For the latter, those measures serve several purposes: teaching students to measure, estimate and control software projects; keeping projects within course budgets for time and effort; quantifying project success; and assessing adequacy of the employed processes and technologies.

This paper deals with measuring complexity, effectiveness and efficiency in course projects. It is based on course projects developed by thirteen classes, in an industry-oriented graduate program, from 2002 to 2008. Its students were industry professionals, with variable degrees of programming experience, but usually not familiar with the state of the art in software engineering, except for some implementation languages and technologies. Most of them had daytime jobs, attending lectures during some nights, and doing homework at nights and weekends.

Our typical course projects develop small information systems applications, with a functional size about 100 function points. Each class had two to four self-selected project teams, ranging from five to eight students. The prescribed process [1] required that application requirements and design should be modeled in UML; then its code should be implemented in Java, using test-driven techniques. Models, code and test scripts and results should be verified by process-defined appraisal procedures.

Projects usually took about six months, divided in use-case driven iterations. The students started with simple CRUD use cases, later developing progressively more complex functions. Some measurements collected from those projects were discussed in previous papers [2, 3, 4]. Here, we compare detailed data from the first four classes, held from 2002 to 2005, totaling 13 teams. For uniformity, we do not include three more classes held from 2006 to 2008, totaling eight teams, since they had a somewhat shorter schedule, and a few different grading rules, but they did not show any qualitatively significant difference.

The students received a model and code framework, which supplied resources such as user interface skeletons and persistence services. A sample project, complete with all required artifact, artifact templates and a detailed process descriptions also helped them to use the process; also, they received from teaching assistants some on-line training with the process artifacts and procedures, and the use of the related tools.

Our department also has a software engineering laboratory that develops real-life software, mostly for government customers [5]. It uses a variation of the process adopted for the course projects; some interesting comparisons may be made between the results achieved in the course projects, and the findings resulting from application of similar practices in real-life projects.

The remaining sections in this paper summarize what those data show about project complexity, effectiveness and efficiency. Section 2 discusses measurement of problem and solution complexity, emphasizing measurement of functional and physical size. Section 3 deals with measuring effectiveness of the solution, focusing on achieved functional quality and conformity with requirements and standards. Section 4 considers measuring effort, as a basis for assessing productivity and efficiency. Finally, conclusions are drawn in Section 5.

2. COMPLEXITY

2.1 Sizing and Complexity

In our course projects, we have been using sizes as proxies for complexity measures, both in the problem side and in the solution side. A problem size of 100 function points translates into solutions with a few thousand lines of Java code; this is too small to have any significant structural complexity, as indicated by measures such as cyclomatic complexity or fan-out. Also, functional size is easily countable, and physical size is amenable to automated counting, as discussed in the following subsections.

2.2 Problem Sizing

Problem sizing was based on IFPUG function point counting practices ([6], [7]), which have the following advantages:

- Function points are widely accepted as problem complexity measure by our local market, where customers usually agree to be charged a fixed price per function point (so that the customer pays for recognizable product size, but not for possible low productivity of the developer).
- They may be counted directly from a problem statement, such as a requirements specification or an analysis model. Solution elements, such as design models and code, need not be available; indeed, no solution element should be counted.
- Although hardly amenable to automated counting, they may be easily deduced from an analysis model, if this follows a modeling style that represents data and transaction functions as recognizable model elements. Such style is enforced by the modeling style guide adopted in our process. This avoids a common pitfall of use case points, often cited as function point competitors: use cases may be stated at quite different levels of abstraction, and these lead to different counts.
- There is usually good correlation between size in function points and development effort, which allows the use of function points as input to estimation procedures such as COCOMO [8].

In the negative side, expert function point counting requires considerable training, such as required by the IFPUG certification, which would not fit in our course time budget. However, quite good approximate counts may be done by non-specialists, following predefined model mapping rules. According to McConnell [9], properly taught inexperienced counters are able to agree within a 25% range, as opposed to the 10% range achieved by certified counters. This is confirmed by the experience of our laboratory, since we use certified counters for real-life projects. Moreover, our process counting forms require justification for each proposed count item, allowing instructors to check whether counting was, at least, reasonably justified, during the grading audit, described in Subsection 3.4.

After initially using adjusted function points, we fell back on non-adjusted function points. This was done for the following reasons:

- Counting adjustment factors requires looking up a quite large set of IFPUG guidelines, usually open to divergent subjective interpretation, especially for non-experienced counters.
- Adjustment factors change the non-adjusted count by a maximum variation of ± 35%, whereas non-functional requirements, which they supposedly reflect, may easily have a much larger influence on development effort, or no influence at all, depending on circumstances that those factors do not reflect (for instance, available hardware).
- Non-functional requirements are usually not significant in small-sized educational projects, anyway.
- Only non-adjusted function points comply with ISO/IEEE guidance for functional sizing [10] (reportedly, adjusted function points were excluded from them due to the two first reasons above).

Every course project was required to keep a minimum complexity of 100 function points. The students were required to develop a preliminary specification with that functional size. Requirements changes were allowed, and even advised in some cases, when the instructor perceived that some proposed function might be especially difficult to implement, given the available resources. But every change was required to keep functional size around the 100 function point mark.

Our real-life projects usually follow a similar policy of keeping functional size while allowing requirement changes demanded by the customer. This policy usually has good acceptance among customers, since, under fixed prices per function point, such policy translates into keeping a fixed cost budget, while trading requirements according to their current priority.

2.3 Solution Sizing

Meaningful code sizing requires following at least two standards:

- A **coding standard** for each of the chosen languages, such as the standard Java conventions [11], ensures that logical lines of code are mapped to physical lines in a standardized way. Our projects were required to follow those Java conventions, enhanced by widely accepted good coding practices [12, 13]. The use of the coding standard should be verified by code inspections; but a tool such as the Eclipse formatter for Java code may enforce most rules. This particular tool allows for very flexible, process-specific tailoring of the rules.
- A **counting standard**, which determines what kinds of lines of code should be counted, and how. Our course projects used the very simple counting standard adopted by PSP [14]; our real-life projects employ the SEI counting framework [15]. Both are easily embodied in a counting tool.

In our projects, the counting tool was also used to count classes; it might easily count other object-oriented metrics, such as number of methods or hierarchy depths, but we did not consider those as useful for our purposes. It was also used to count test scripts, providing data for assessing test-driven development. Also, since the design standards mandated a specific folder structure for different kinds of classes (entity, control, boundary, unit tests and system tests), folder-stratified size data provided also information on application design and test design. Table 1 defines those kinds of classes; the test code layers are as defined in the IEEE Glossary of Software Engineering Terminology [16], and the product code layers are as proposed by Jacobson [17, 18].

Table 1. Product and test code layers

Kind	Layer	Definition
Product code	Boundary	Code that handles communication with the environment of the product, such as user interfaces and interfaces with other systems.
	Control	Code that handles the event flows, validations and algorithms that are not specific to some entity.
	Entity	Code that models concepts of the problem domain, such as physical or informational entities.
Test code	System	Scripts for testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.
	Unit	Scripts for testing of individual software units or groups of related units.

In these projects, it was sufficient to count Java code, since they were J2SE applications; other kinds of files, such as properties files used for configuration and test data, had negligible size. On the other hand, in our real-life projects, which are predominantly Web-based, other kinds of code, such as JSP and XML, become significant, and must also be counted.

Given fixed coding and counting standards, the dominant factor for determining code size becomes the amount of achieved reuse. Therefore, the count of lines of code (LOC) per function point may be used as a reuse indicator. This count is also a fundamental parameter for effort estimation. The COCOMO equations relate LOC counts to estimated effort, cost, schedule and staff size, and the COCOMO user is expected to provide this parameter, as measured for the kind of projects that are being estimated.

Counts of lines of code and classes are logical sizes, from a conceptual viewpoint. However, we call them also physical sizes, because, in both cases, there is a one-to-one mapping. In the case of lines, as stated before, this is enforced by the coding standard. In the case of classes, the compiler imposes a mapping between classes and Java source files, except for inner classes, which, in our kind of projects, do not significantly add to complexity (and are mostly contained in the framework classes).

Test code may be sized using the same tools and standards used for application sizing. Our process requires heavy use of automated test scripts. So far, JUnit scripts were used for all kinds of tests. In the next courses we are switching to the IBM Rational Functional Tester environment for system tests; these also use Java scripts. Our real-life project switched to the Functional Tester sometime ago; previously, they used Rational Robot, which used a Basic-like script language.

Test case counts provide other interesting insights. Currently, the automated test case data for course projects must coded as text in property files, so that specified test cases may be counted from those files. Implemented and passed test cases may be counted from test logs, which students are required to deliver.

3. EFFECTIVENESS

3.1 Assessing Effectiveness

We consider that a project was effectively conducted, if its resulting product passes established verification and validation procedures, "*determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements"* [16].

Verification and validation provide qualitative checks of the conformity with requirements and standards. Of course, most of those procedures are regular parts of the development process, and defects are not a problem per se, provided they are removed before delivery. Thus, evaluating effectiveness is partially done by assessing how many defects remain in the delivered material. Additional indicators are provided by some indirect measures, based on the previously described direct size measures.

3.2 Kinds of Appraisals

The process adopted in our course projects mandates several kinds of appraisals for verification and validation:

- **Peer reviews**, performed on the problem model, solution model and test code. Formal inspections, as defined by the IEEE standard for reviews [19], are recommended, but less formal peer reviews are accepted, provided their reports conform to a prescribed standard, and the reviewers do not include the authors of each material. Along the project iterations, author and reviewer roles should be rotated.

- **Tests**, such as unit tests, performed by code implementers and system tests, performed by testers who did not write the code under test. Implementer and tester roles should also be rotated.
- **User evaluations**, important in real-life projects, are just simulated in course projects, since they do not have real users.
- **Quality audits**, discussed in detail in [4], appraise the results of the other appraisals, as well as other artifacts, such as plans and reports, that are not object of peer reviews. They should be the last iteration activity, providing a last check on the material delivered for grading. The grading process essentially repeats the quality audit.

3.3 Ensuring Data Quality

Manual appraisals, such as reviews, evaluation and audits are always performed using checklists. For each kind of appraisal, a corresponding checklist codifies the kinds of checks required by the process standards, for the respective material. For each checklist item, the appraiser must record the number of defects found, in a checklist log. Also, each defect found must be described by a record in a defect list in the appraisal report. Each record provides a detailed description of the defect nature, referring to the respective checklist item, and transcribing a defect classification from that item.

The appraisal reports list the name of the appraisers, and the date and time of preparation and meeting sessions, providing detection effort data for every appraisal. Every material author receives this defect list and, for each defect, they must record the applied fixes, or justify why they were not fixed. Besides, they must record the time spent for each fix, which allows measuring fix efforts. Both checklist logs and appraisals reports must be included in the delivered materials, and checked by the quality audits.

For reviews, the defect classification is currently based on the **Orthogonal Defect Classification** [20]. Those data might eventually be used to study defect causes, but this was not done, so far; their current purpose is, mostly, to teach about defect classification. In earlier versions of the appraisal system, when the classification was not embedded in the checklist, but left to the student, they usually found it very difficult. Also, classification is used in consistency checks between checklist and report.

This system evolved after some data quality problems were detected when analyzing the data provided by earlier classes. Currently, it allows the quality audits to perform the following checks on the remaining appraisals:

- Every defect description is accurate, self-consistent, and actually reflects the corresponding checklist item; it is not merely a matter of opinion. Moreover, this sometimes uncovers cases where the rule in the checklist was not properly understood by the reviewers, suggesting possible improvements in the checklist itself.
- Total defects of every kind match in the checklist log and in the respective report. Inconsistencies usually indicate recording or transcription errors (sometimes, faked data).
- When fixes are not performed, this decision is well justified.
- Fix times are consistent with the nature of the defects.
- Total detection and fix efforts per appraisal were correctly recorded.

For every course offering, the checklists are kept constant, for each kind of appraised material. They do slowly evolve from offering to offering, partly reflecting findings about the effectiveness of the previous generation of checklists. In the long term, manual checks tend to decrease, since new releases of the tools usually improve automated checks. For instance, the recently adopted *Checkstyle* tool drastically reduces the checklists for code inspections, which used to be the heaviest kind of review.

In real life projects, schedule pressures tend to hurry reviews. Therefore, for those projects there has been substantial investment in enhancing the tools validation resources with home-made scripts. In some cases, OCL constraints have proved useful, but, for course projects, this approach has yet to be tried, since it requires additional investment in retrofitting tools.

For both manual and automated tests, the students must generate test logs. Test code contained in the reuse framework helps to generate automated test logs. After performing the tests, checklists are used to verify test scripts and logs, especially for coverage. The logs must include records of expected and actual values, and selected database snapshots, which help to uncover false positives and false negatives in the automated test assertions. Test reports keep track of test sessions, defects found and respective fixes, and detection and fix efforts.

Since our real-life projects require higher proportion of manual tests than the course projects, we have found useful to the IBM Rational Manual Tester, which records step-by-step execution of manual scripts. This tool has also proved useful to record the execution of review checklists. So far, however, it was not tried in the course projects, since introducing new tools requires a slice of the course time budget, to teach their use, and its likely benefits must be carefully weighed against this cost.

3.4 Audits and Grading

Grading audits are quality audits performed for grading, usually by a teaching assistant. Their checklist is the same used for the student-performed audits. They check whether the requirements are completely covered by the design, code and tests, and whether all specified tests were correctly implemented and passed. They check also whether the delivery includes all the required artifacts, complete with every required section and attachment, and conformant with the applicable standards. In the case of models, they check whether these pass automated tool checks and conform to the modeling style standards. For logs and reports, the audit emphasizes checking for data quality.

The quality audits also include some manual tests that are intended to check for combinations of inputs that were not covered by the automated tests. Defects of that kind have been seldom found, showing that, for the problem complexity of the projects, automated tests usually provide enough coverage.

In our real-life projects, which use a similar hierarchy of appraisals, it is not generally possible to provide 100% automated testing, and some defects are found during operation and handled by maintenance. However, the resulting levels of defects found in operation have been quite low, and clients have a good perception of this [5], confirming the effectiveness of that hierarchy.

Grades are based on a severity-weighted count of the defects found in the grading audits. Projects are approved it their final grade is 60% or more; these seldom miss artifacts and specified tests, and seldom show non-passed tests, faulty models or bad data, since those are considered as heavily weighted critical defects. Therefore, this grade indicates conformance with both product requirements and with required standards.

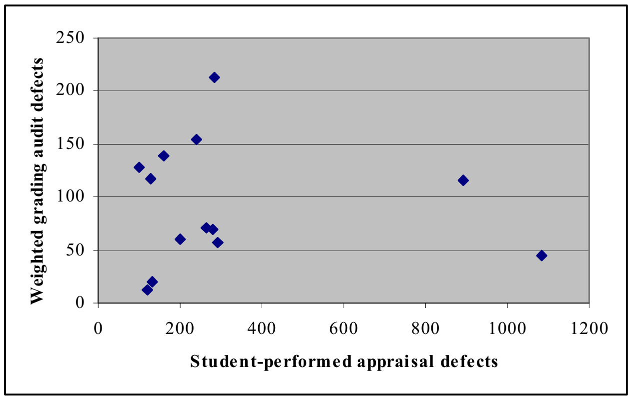

Figure 1. Grading audit defects versus appraisal defects

We did not find a correlation between the weighted defect counts found in grading audits and the defects found in student-performed appraisals, as shown in Figure 1; their correlation coefficient was -0.07. In this case, review defects of every kind are simply summed; audit defects are classified as minor, major or critical, with respective weights of 1, 3 and 15 (critical defects usually mean bad data, missing artifacts or non-passed tests).

We concluded that the final quality level, as measured by the grading audits, did not depend on whether the teams did a more careful work before reviews, leaving few defects to be found in them, or left to the reviews the burden of finding defects, fixing those that were found. Comparing with real-life projects, we found a different situation; in this case, schedule pressure often prevented fixing all defects found in the appraisals (both reviews and tests). To keep product quality, the end of the projects was often burdened with delayed fixes. In this case, it was found that incentives for removing a larger proportion of defects before the appraisals did result in improvements in final quality levels.

3.5 Effectiveness Indicators

Reuse has a quite significant impact on quality and productivity. Increasing reuse is desirable, not only for every application code layer, but also for test code, since this, as confirmed by size counting, has a significant volume, and, therefore, is a major contributor to effort and defects. For a course project, reuse opportunities are determined by the fitting between problem statements and available functionality of the reuse framework.

Since the LOC count per function point is a reuse indicator, as discussed previously, its measurement might indicate whether the projects had a satisfactory degree of reuse. High LOC/FP indicates poor reuse, and too low counts may suggest that problem statements are too easy, and do not provide sufficient exercise. LOC counts per class may be used as an indicator for goodness of design. In a balanced design, they should be neither too large nor too small, and should vary across different kinds of code in a way that is consistent with the design practices recommended by the adopted process.

As discussed in [3] and shown in Figure 2[1], LOC counts found in our course projects were nearly constant, for product and test code, per class and per function point. This suggested a good level of compliance with design and code standards, which was confirmed by inspections and audits. Both kinds of code ranged around 30 LOC/FP, lower than the often quoted industrial counts for Java (53 LOC/FP is the COCOMO default value).

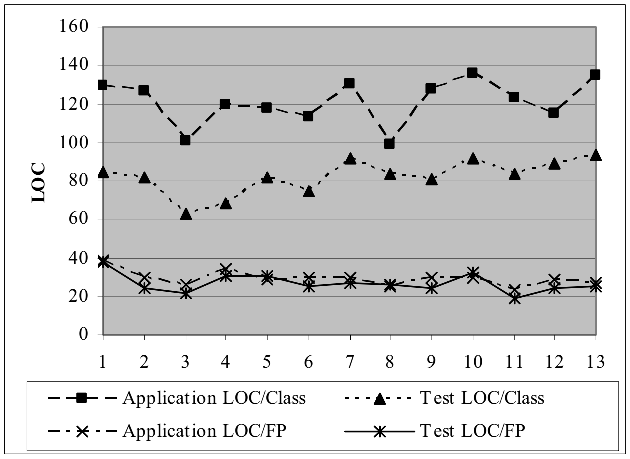

Figure 2. Application and test LOC per class and FP

Our indicators suggest extensive reuse of the design, test and code framework that was provided for the projects. Indeed, it was lower than the indicator for the sample project provided by the process (around 40 LOC/FP), which prompted us to introduce additional requirements for the use cases implemented in the last iterations of each project, in order to make reuse a bit more difficult, and require more design exercise.

Our real-life projects have shown test counts above 70 LOC/FP. Partly, this reflects the presence of significant amounts of non-Java code, required by the adopted J2EE technologies; but it also shows how reuse is more difficult to achieve in real-life projects, where use cases come in a much larger variety of flavors, and the technologies themselves are less stable. Currently, for instance, those projects are in transition from Struts to JSF. Well designed frameworks may alleviate the impact of technology changes on reuse, but it is not generally possible to wholly shield applications from those changes. Our own framework for course projects has been recently ported from an ad-hoc persistence layer to a Hibernate-based one; a façade kept most of the changes hidden from the layers above, but, even then, some rework was needed.

The application code ranged about 120 LOC/class, and the test code about 80 LOC/class, reflecting its simpler logic and structure. However, the number of test classes is larger, so that the LOC/FP count is about the same (the two lowest lines in Figure 2 almost superpose). That larger number of classes reflects a division of labor that is suggested by the test design guidelines,

[1] Throughout this paper, when abscissas are not explicitly labeled, they simply indicate project numbers.

and enforced by the test reuse framework itself: test classes are specialized as managers (implement the test procedures), inspectors (send test data to the application, and collect results from them), and checkers (compare expected and actual results).

For the test classes, size is also somewhat dependent on the style imposed by the underlying test framework. For instance, JUnit typically requires one test method per test case, and each test case method is expected to ensure its own preconditions, since it is hard to guarantee a precise order of test case execution. The Functional Tester does not require these specific constraints, and, therefore, it is expected that its use will bring a significant reduction in test code size, also indicated by some preliminary experiments. On the other hand, it is a much heavier environment than JUnit, and this will continue being used for unit tests.

In the past projects, where all test scripts were JUnit-based, tests for the boundary layers played the role of system tests. Their count for implemented and passed test cases matched closely the count for specified test cases, which was to be expected, since this was a major criterion for use case acceptance.

Under the prescribed design guidelines, the control layer handles validation of user inputs. Therefore, the test case count for this layer should be somewhat higher than for the boundary layer, since, for robustness sake, the control layer must check some inputs which are prevented by the user interface design, but might be generated by a defect in the boundary layer. For similar reasons, the test case count for the entity layer should be much smaller than for the control layer, since, normally, it does not have to check for abnormal inputs. The collected data confirmed that those design guidelines were generally followed.

4. EFFICIENCY

4.1 Measuring Efforts

Efficient projects have good productivity, that is, effort per problem size. Therefore, assessing efficiency requires measuring efforts spent in the project. But ensuring data quality for efforts is even harder than ensuring it for defect counts. Effort spent in every project task must be faithfully recorded, so that no actually performed task is omitted. Moreover, it should be recorded just after a task session is complete; otherwise, details are forgotten, and records may become less accurate. Worst of all, our experience with project effort data collection has shown that many students do not appreciate the importance of accurate effort measurement, and outright faking may happen.

Also, some degree of effort classification is necessary. Effort distribution per process discipline provides yet another indicator of process compliance, as discussed later. However, students who are learning software processes found task classification difficult, in most cases. Some tasks are in the borderline between disciplines, and even different processes classify them in a different and somewhat arbitrary way. Moreover, it is useful to separate primary development efforts from quality-related efforts, such as detection and fix effort.

So far, our effort data recording sheets have gone through four generations, each building upon the deficiencies found in the previous ones. The current generation provides the student with a process-defined task list, which represents a typical schedule for one iteration. The **EPF** *(Eclipse Process Framework*) open source tool, currently used to provide the process description, easily generates that task list. An average task requires a few hours of effort, and may be performed in one or more sessions.

For every session, the students must provide the performers' names and the session date, start time and end time, which allow some consistency checking. Also, a free-format detail field provides more detail on the actual work done, allowing checking whether it was recorded as the correct task. Process tasks have a predefined relationship with process disciplines, allowing automated effort totaling per discipline.

A design issue is whether or not appraisal-related tasks, already recorded in the appraisal reports, should also appear in the effort logs. In previous log generations, they should, and this provided additional consistency checks in the quality audits. These, in some cases, helped to uncover data faking. However, it also meant increased process overhead, and more bona fide transcription errors. Currently, we suppressed those quality-related tasks from the effort logs, relying instead on checking the consistency of the recorded date and time of the appraisal sessions with the dates and time of the tasks where their respective material was built. Moreover, we found that the grossest cases of data faking usually lead to quality-related efforts that are either too low or too high, when compared with the corresponding development efforts.

4.2 Assessing Productivity

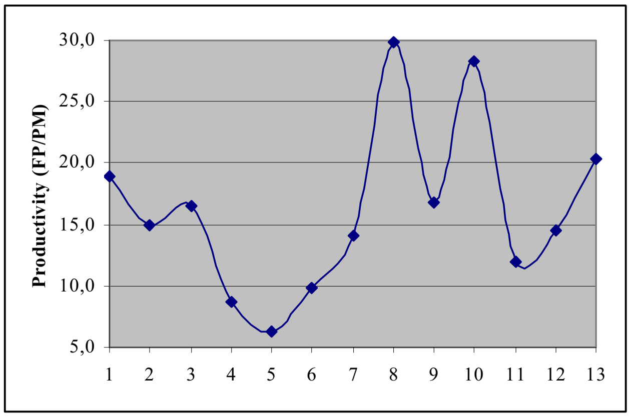

Figure 3. Productivity (function points per person-month)

Figure 3 shows now productivity varied across the analyzed course projects. It is seen that there is a wide variation, ranging from near 5 FP/PM to near 30 FP/PM. Also, there does not seem to be a very definite trend, and there are also wide variations within each class (the fist group of four teams is one class, and the remaining groups of three are different classes). Analyzing the causes of such variations might yield some useful information.

Productivity is, of course, heavily influenced by the: degrees of reuse. Also, process might play a role: either improving it, when it reduces rework, or lowering it, when its overhead offsets the gains in rework reduction. Finally, predictive models such as the COCOMO [8] suggest that, even for similar applications and environments, some factors related to the team profile may heavily affect the productivity outcome.

Figure 2 and the discussion in Subsection 3.5 suggest that there was very little variation in degree of reuse, across the analyzed projects. In the following subsections, we discuss the remaining hypotheses.

4.3 Effort Distribution per Discipline

Table 1 shows the effort distribution per process discipline. These include specification disciplines: requirements (RQ) and analysis (AN); solution disciplines: design (DS), implementation (IM) and test (TS); and management disciplines: project management (PM) and quality management (QM). The columns show the effort fraction per discipline, as minimum and maximum value, average (μ), standard deviation (σ) and coefficient of variation (σ/μ).

This table shows that the process usage did not differ significantly among teams, especially if we consider the much larger differences in their productivity. The largest variation is found in the requirements efforts, probably because some teams had more requirements changes, and in the management disciplines, which have a smaller effort fraction. We did not investigate the causes for the variation in test effort. The implementation effort, which had the largest effort fraction, has the lowest variation.

Table 1. Effort distribution per discipline

	MIN	MAX	μ	σ	σ / μ
RQ	4,9%	25,4%	13,3%	6,6%	0,50
AN	6,5%	14,7%	10,4%	2,7%	0,26
DS	13,6%	25,5%	20,5%	3,5%	0,17
TS	6,6%	22,3%	13,4%	5,6%	0,42
IM	22,3%	35,6%	29,8%	4,4%	0,15
PM	1,3%	16,3%	7,2%	4,0%	0,55
QM	1,8%	8,8%	4,9%	2,2%	0,44

Also, it confirmed our expectation about process effort distribution: around 25% for specification, 10% for management, and the remaining for solution. The uniformity of distribution becomes even more apparent if averaged per class, as was done in a previous work [3]. That paper also shows that the effort distribution across project iterations varied in approximately the same way along the course classes, and also conformed to the process expectations. From this analysis we concluded that the process was followed in a reasonably faithful way, reinforcing the conclusions of Subsection 3.5.

4.4 Appraisals and Rework Effort

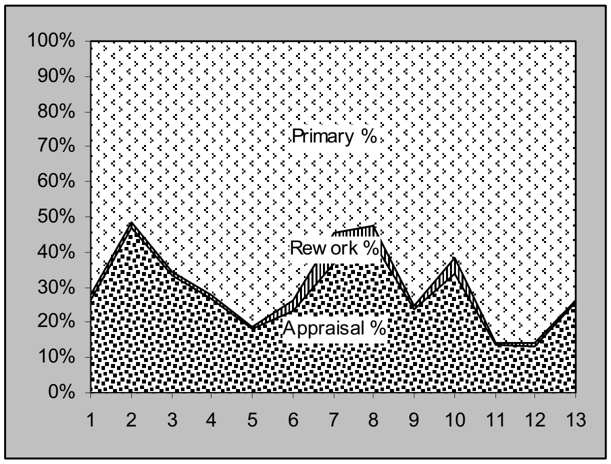

Figure 4. Effort distribution: appraisal, rework and primary

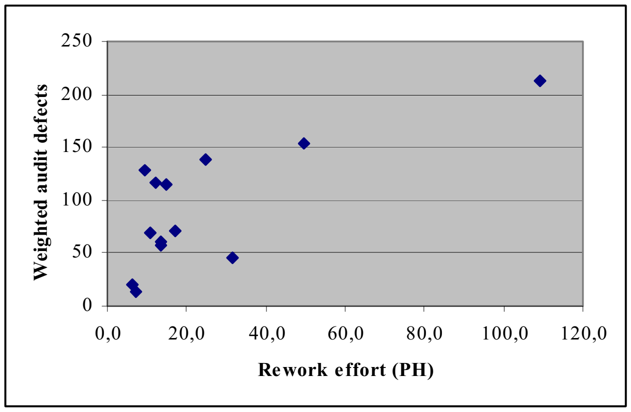

Figure 5. Weighted audit defects versus rework effort

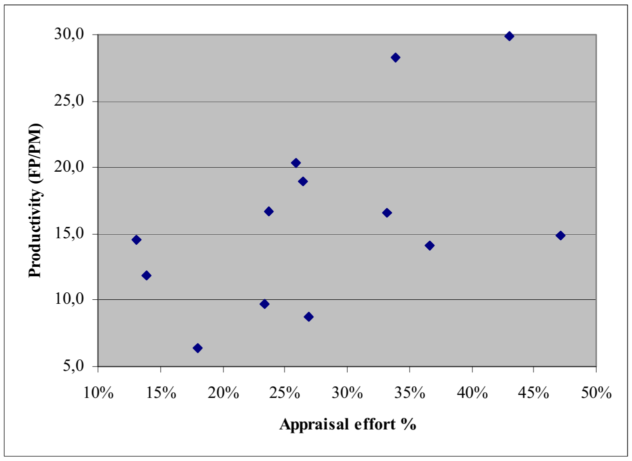

Figure 6. Productivity versus appraisal effort fraction

The distribution of effort among disciplines does not show the distribution of effort among appraisal, rework and primary development tasks. This may be done with the detection and correction effort data, collected from the appraisal reports (although, in this case, fixing defects found in appraisals is the only kind of rework that is measured). The respective area chart is shown in Figure 4.

The rework fraction is very low, indicating that most defects found in the appraisals are easy to fix. Indeed, we found a reasonably good correlation between weighted audit defects and rework effort (person-hours), shown in Figure 5 (correlation coefficient = 0.74). Keeping such a low rework effort seems a harder proposition in larger, real-life projects; for those, we have usually considered a 20% rework fraction as reasonably good.

Also, the data show a positive correlation (correlation coefficient = 0.51) between productivity and the appraisal fraction (Figure 6). This suggests that appraisals have a positive influence on productivity, even if they do not find many defects. Here again, such high appraisal effort fractions, spent by some of the more productive teams, are seldom found in real life projects. It is possible that those figures appear here because of the small size of the project teams; primary efforts tend to be individually spent, while appraisals are usually done by groups of at least two people.

4.5 COCOMO Analysis

To investigate other possible influences on project productivity, we analyzed which sets of COCOMO factors would forecast values of productivity compatible with the found range, using standard COCOMO calibration and the same parameters used in the previous analyses (30 lines of code per function point, 152 work hours per person month). Those values were assigned according to their interpretation provided in the COCOMO manual. We assumed that the factors shown in Table 2 were common to all teams, since they used the same technology and tools, the problem size was the same, and the problem statements were controlled to avoid excessive non-functional complexity.

Table 2. Assumed common COCOMO factors

Category	Factor	Meaning	Level
Product	RELY	Required Reliability	Low
	DATA	Data Base Size	Low
	DOCU	Documentation Needs	Nom.
	CPLX	Product Complexity	Nom.
	RUSE	Developed for Reusability	Low
Platform	TIME	Execution Time Constraint	Nom.
	STOR	Main Storage Constraint	Nom.
	PVOL	Platform Volatility	Low
Personnel	PCON	Personnel Continuity	High
Project	TOOL	Use of Software Tools	High
	SCED	Required Develop. Schedule	Nom.
Scale	PREC	Precedentedness	Nom.
	FLEX	Development Flexibility	Low
	RESL	Architecture/Risk Resolution	Nom.
	PMAT	Process Maturity	High

Table 3. Assumed team-specific COCOMO factors

Category	Factor	Meaning	Best	Worst
Personnel	ACAP	Analyst Capability	Nom.	Very low
	APEX	Applications Experience	High	Low
	PLEX	Platform Experience	High	Low
	PCAP	Programmer Capability	High	Low
	LTEX	Language and Tool Experience	Nom.	Very low
Project	SITE	Multisite Development	Very high	High
Scale	TEAM	Team Cohesion	Nom.	Low

Table 4. COCOMO estimated productivities

Productivity	Best profile	Worst profile
Optimist	38,04	10,30
Most Likely	30,43	8,24
Pessimist	24,35	6,59

We separated another set of factors, which were assumed to vary according to the team composition. Table 3 shows the sets of values which yielded estimated productivities consistent with the observed range. Those are shown in Table 4, in function points per person-month, as the COCOMO optimistic, most likely and pessimistic estimates for the best and worst assumed profiles. The profiles described in Table 3 are also consistent with our direct observation of the teams, as well as with the range of experiences reported in their applications to the course.

This analysis shows that the observed productivity variation might well result from pre-existing variations in the team profiles. Also, it suggests an alternate interpretation for the correlations found in the previous subsection. Instead of causing higher productivity per se, higher investment in appraisals (with subsequent lower rework) might be a trait of the best team profiles. Deciding this issue would require more data; in the next section, we discuss a possible solution.

5. CONCLUSIONS

5.1 On General Issues in Measurement

We analyzed a set of thirteen course projects with tightly controlled problem complexity, performed during a period of a few years, within a quite uniform environment, which included a software development process oriented towards educational projects. This process required filling reports that provided a set of basic size, defect and effort measurements. Some indicators, derived from those measurements, were used to study the achieved levels of effectiveness and efficiency.

The students were expected to fill those reports in simple ways, without excessive overhead. A degree of automation and self-check should be built within the reports, to decrease recording and transcription errors, and the grading criteria should reward good data collection. The instructor should be able to easily retrieve the relevant data, detect bad data and derive those indicators that were not contained in the reports themselves.

Achieving those goals required several redesigns of the process and its artifacts, building upon deficiencies found in previous generations. The projects analyzed here represent the last of the past process versions; this experience was embodied in a new process version, where the artifacts underwent a major revision. This revision is to be tried for the first time in the near future.

5.2 On Specific Issues in Measurement

5.2.1 Problem Size

Function points were adopted as problem size metric. They are well accepted in real life projects, and their counting is well defined and standardized. Generally, they are not easy to count, but our course projects usually contain just simple cases, and the adopted problem modeling standard eases the identification of the countable functions. Students must provide written justification for their counting decisions, so that those may be checked. Only non-adjusted function points are considered. A project rule states that requirements changes in mid-project are allowed, but a minimum count of 100 function points must be kept.

5.2.2 Solution Size

Physical size, measured as the number of lines of code and classes per software layer, was adopted as solution complexity metric; other complexity metrics were not considered useful, due to the relatively small size of the project code material. The use of both coding standards and counting standards was mandated, helping to ensure correspondence between logical and physical size. The use of the coding standard is checked by code inspections, currently helped by a compiler-embedded formatter. A counting tool that enforced the counting standard was supplied, so that code counts might be easily repeated and checked.

Unit and system test scripts were separately counted. Test cases were counted from their respective property files; the project audits included checking their traceability to requirements, to assess their coverage. Also, those audits checked whether the system test logs showed evidence that all of the specified test cases had been implemented and passed, providing a reasonable guarantee that the product was free of major functional defects, and actually implemented the required functions.

5.2.3 Achieved Quality

The process used in our projects required performing, for every project iteration, a series of appraisals that included peer reviews, tests, user evaluations and a quality audit. The grading procedure repeated the quality audit, computing a weighted count of the remaining defects, where weights depended on defect severity.

The weights were such that it is almost impossible to receive a passing grade if the material did not implement and pass all specified tests, contained all required artifacts and attachments, and its models at least passed automated tool checks. The grading audit checked whether appraisal checklists and reports, test logs and effort logs had been properly filled. Therefore, they were a major contributor to data quality.

Since critical defects usually prevented getting a passing grade, the grade assigned to projects that reached approval basically indicated the amount and severity on non-critical defects, usually resulting from lack of full compliance with standards. We found no correlation between those counts and the counts of defects found by the students in the appraisals, suggesting that, concerning the final quality level achieved, it did not matter whether most defects were fixed before or after the appraisals. A different situation was found in larger, real-life projects that use a similar process.

5.2.4 Size Ratios as Indicators

Some ratios involving solution and problem sizes also reflect useful information about solution effectiveness. The ratios of lines of code to function points, for instance, was lower than the ratios reported for average Java code, for both application and test code, suggesting a high degree of reuse of the provided design, test and code framework. This was confirmed by the grading audits. Also, they were quite lower than those found for real-life projects, reflecting the fact that reuse is more difficult in those projects.

Ratios of lines of code per class were computed, for application and test code. They were found consistent with what should be expected from following their respective design guidelines.

5.2.5 Effort Data

Effort data are one of the hardest kinds of data to collect. Some technology-intensive environments may partly automate their collection, but, for most usual environments, they must be recorded by the students themselves, in appropriate log forms. These should be designed to provide some degree of automated task classification and totaling, and to allow some basic checks.

Appraisal reports provide an appropriate place for recording quality-related efforts, such as detection and fix tasks. These do not need to be recorded again in the effort logs, provided some kinds of cross-checking between them are possible.

5.2.6 Productivity

Computed productivity showed large variation across project teams. Different degrees of reuse might have affected it, but, as shown by the discussion of size ratios, this degree did not vary much among teams. Effort distribution across process disciplines was also found to vary little across teams, and to be consistent with process expectations. So, it was also discarded as a cause for productivity variation. Productivity did show to correlate positively with higher appraisal effort and lower rework effort.

Simulated COCOMO analyses confirmed that variations in some COCOMO cost factors, due to differences in team profiles, would also suffice to explain the found productivity variation.

5.3 On Process Improvement

Data collection issues often drive improvements in a process and its artifacts. As an example, consider the following case. In the past process version, the set of artifacts delivered by the end of each iteration included some cumulative artifacts, such as models and code, where each iteration included the material of previous iterations as a subset. Defects in those artifacts, once found in the quality audit of that iteration, had to be fixed by the next quality audit; otherwise, they would be found and penalized again.

Other artifacts, such as effort logs and appraisal reports, were iteration-specific, being replaced by artifacts of the same kind, in the next iteration. Therefore, if they had bad data or even symptoms of faking, those were found and considered as critical errors. The new iteration provided a set of new data, usually with better quality, since the students learned that bad data are usually detected; but the older data were not fixed, except in some special cases. For effort logs, for instance, it was required to deliver corrected versions of previous logs. It was somewhat difficult to design report formats that would be both cumulative and easy to fill, but this was our design goal for the current process version.

The new process version requires recording the self-assessed COCOMO factors for each team, to perform COCOMO estimates, instead of cursory estimation using assumed productivities. Moreover, teams will be required to remake the estimates in mid-project, if they show discrepancies with the actual values; doing this requires improving the profile self- assessment. This should help to study how they affect project performance, and how effective the course might be in counteracting profile weaknesses.

We consider rework measurement to be an important and still open issue. Clearly, just summing the recorded fix efforts does not account for the entire actual rework; other kinds of rework should be marked as such during effort data collection. However, any criteria for that must distinguish between avoidable, undesirable rework, which standards, tools and reuse should help to avoid, and rework which is a healthy part of the process, such as refactoring; and this distinction should be easily performed.

5.4 On Lessons Learned

Finally, we summarize as a sequence of steps what we think was learned about measurement in course projects:

1. Choose a set of basic size, effort and quality metrics that are easily collected by the students themselves, in their normal project work, and recorded in required project artifacts.
2. Automate collection, transcription and totaling, whenever possible.
3. For manual recording, design the forms so as to include means for self-checks and cross-checks. If data faking is considered possible, do not publicize all of the checks.
4. For data involving subjective judgment, require written justification.
5. For sanity checking, compare with industry, literature or tool-provided data or estimates, whenever possible.
6. Include in the process a consistent set of appraisals, requiring that defects and detection and fix efforts be recorded in ways that allow their easy checking.
7. Grade based on audits that check according to publicized rules, using criteria that reward meeting requirements, complying with required standards and reporting good data.
8. Look for simple correlations and trends (or absence thereof), leaving more sophisticated analyses for issues where they are better suited.
9. Leverage on perceived strengths and weaknesses of the metrics to improve both the process as a whole, and the metrics definition, collection and analysis.
10. Apply the improved process, collect new data and iterate.

6. ACKNOWLEDGMENTS
We thank IBM Rational for supporting this work, within the IBM Academic Initiative. Also, we thank Vítor A. Batista, who was teaching assistant for part of the projects described here, for his comments and suggestions.

7. REFERENCES
[1] W. Pádua. A Software Process for Time-constrained Course Projects. In Proceedings of the 28th international Conference on Software Engineering (Shanghai, China, May 20 - 28, 2006). ICSE '06, 707-710. DOI= http://doi.acm.org/10.1145/1134285.1134397

[2] W. Pádua. Quality Gates in Use-Case Driven Development. Quality gates in use-case driven development. In Proceedings of the 2006 international Workshop on Software Quality (Shanghai, China, May 21 - 21, 2006). WoSQ '06, 33-38. DOI= http://doi.acm.org/10.1145/1137702.1137710

[3] W. Pádua. Using Model-Driven Development in Time-Constrained Course Projects. Using Model-Driven Development in Time-Constrained Course Projects. In Proceedings of the 20th Conference on Software Engineering Education & Training (July 03 - 05, 2007). CSEET, 133-140. DOI= http://dx.doi.org/10.1109/CSEET.2007.55

[4] W. Pádua. Using Quality Audits to Assess Software Course Projects. In Proceedings of the 2009 22nd Conference on Software Engineering Education and Training - Volume 00 (February 17 - 20, 2009). CSEET, 162-165. DOI= http://dx.doi.org/10.1109/CSEET.2009.12

[5] B. Pimentel, W. Pádua, C. Pádua, and F. T. Machado. Synergia: a software engineering laboratory to bridge the gap between university and industry. In Proceedings of the 2006 international Workshop on Summit on Software Engineering Education (Shanghai, China, May 20 - 20, 2006). SSEE '06, 21-24. DOI= http://doi.acm.org/10.1145/1137842.1137850

[6] IFPUG. Function Point Counting Practices Manual. Release 4.2.1., Jan. 2005.

[7] D. Garmus and D. Herron. Function Point Analysis: Measurement Practices for Successful Software Projects. Addison-Wesley, 2000.

[8] Barry W. Boehm, Chris Abts, A. Winsor Brown and Sunita Chulani. Software Cost Estimation with Cocomo II. Addison-Wesley, 2000.

[9] Steve McConnell. Software Estimation: Demystifying the Black Art. Microsoft Press, 2006.

[10] IEEE, IEEE Std 14143.1-2000, Implementation Note for IEEE Adoption of ISO/IEC 14143-1:1998 Information Technology— Software Measurement— Functional Size Measurement— Part 1: Definition of Concepts, IEEE, New York, NY, 2003.

[11] Sun Developer Network. Code Conventions for the Java Programming Language. Sep. 1997. Available at http://java.sun.com/docs/codeconv/.

[12] Steve McConnell. Code Complete, 2nd Edition. Microsoft Press, 2004.

[13] Alan Vermeulen, Scott W. Ambler, Greg Bumgardner, Eldon Metz, T. Misfeldt, J. Shur and P. Thompson. The Elements of Java Style. Cambridge University Press, 2000.

[14] W. S. Humphrey. A Discipline for Software Engineering, Addison-Wesley, 1995.

[15] R. E. Park. Software size measurement: A framework for counting source statements. Tech. rept. CMU/SEI-92-TR-020. CMU/SEI. 1992. Available at http://www.sei.cmu.edu/pub/documents/92.reports/pdf/tr20.92.pdf.

[16] IEEE. IEEE Std. 610.12-1990 (R2002): IEEE Standard Glossary of Software Engineering Terminology, in IEEE Standards Collection – Software Engineering. IEEE, New York, NY, 2003.

[17] Ivar Jacobson. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1994.

[18] Ivar Jacobson, James Rumbaugh and Grady Booch. The Unified Software Development Process. Addison-Wesley, 1999.

[19] IEEE. IEEE Std. 1028-1997 (R2002): IEEE Standard for Software Reviews, in IEEE Standards Collection – Software Engineering, IEEE, New York, NY, 2003.

[20] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray; Man-Yuen Wong. Orthogonal Defect Classification – A Concept for In-Process Measurements. *IEEE Transactions on Software Engineering* 18(11), 1992.