Assessing Quantitatively a Programming Course

Maurizio Morisio, Marco Torchiano, Giovanni Argentieri
Dipartimento di Automatica e Informatica
Politecnico di Torino
Italy
{maurizio.morisio;marco.torchiano}@polito.it; giovanni.argentieri@fastwebnet.it

Abstract

The focus on assessment and measurement represents the main distinction between programming course and software engineering courses in computer curricula.

We introduced testing as an essential asset of a programming course. It allows precise measurement of the achievements of the students and allows an objective assessment of the teaching itself.

We measured the size and evolution of the programs developed by the students and correlated these metrics with the grades. We plan to collect progressively a large baseline.

We compared the productivity and defect density of the program developed by the students during the exam to industrial data and similar academic experiences.

We found that the productivity of our students is very high even compared to industrial settings. Our defect density (before rework) is higher than the industrial, which includes rework.

1. Introduction

Usually in computer science and software engineering curricula we can find a clear distinction between courses where programming skills are taught, and those where software assessment and measurement are taught. The former concentrate on language level programming features and apply measurement seldom. The latter teach measurement but rarely it is aimed at assessing programs written by the students themselves. Besides, in programming courses the grading of students is seldom done using quantitative assessment of the programs developed by the students.

The authors, being involved both in teaching programming courses and in software measurement research, wanted to address this gap, and were able to develop a sustainable and effective approach thanks to new tools and methods, such as the test first approach [4] developed within XP [3].

The computer science curriculum at Politecnico di Torino (three years, roughly corresponding to a B.S. in the American system) includes three programming courses, each roughly 50 hours of lessons: basic programming (using the C language), algorithms and data structures (using C language with pointers and dynamic memory), object oriented programming (using Java). The authors are involved in the latter course.

In these courses students attend lessons, and practice programming in the labs. Finally they sustain an exam, which consists in developing a program. The program is developed on paper in a traditional classroom (no PCs available) during a session lasting around 2 hours. After this session the students, using a carbon copy of the program handed to the teacher, develop the program on a PC, completing and debugging it as needed. Next, they have another individual session with the teacher, who grades the student considering several factors: the program developed in the classroom, the program developed on the PC, the differences between them in terms of functionality provided, design choices and defects.

This procedure for the exam has several cons. From the point of view of the students, developing the program on paper is frustrating. From the point of view of the teacher, but also of the students, grading the students in a consistent and fair way, especially when more than one teacher is involved, is hard. And the textual description of the program to be developed is often subject to ambiguities or misunderstandings.

In the last academic year the teachers of the Java course introduced the test first practice [4] and the related supporting tools (JUnit [13] within Eclipse [8]) with the goal of exposing the students to a promising technique and experimenting its results [9].

With the availability of these tools and techniques developing a new procedure for the exams was straightforward. At the exam session, the teacher gives to the students the specification of the program to develop under the form of Java/JUnit acceptance tests. Students, working in the lab, develop the program and validate it against the tests. The teacher grades the student in function of the number of acceptance tests passed, as shown in Figure 1.

Proceedings of the 10th International Symposium on Software Metrics (METRICS'04)
1530-1435/04 $ 20.00 IEEE

The byproduct of this procedure is the availability of process measures, namely defect density and productivity, on the programs developed by the student population. The programs are available and therefore all source code product measures can be computed too. This makes it possible to develop a baseline of measures to characterize the student population, to compare it with professional developers, and to use it for experiments with student subjects.

Not last, the measure baseline can be used to assess the course itself: effectiveness of teaching techniques and teaching materials, performance of teachers and students. Finally, students live a development process that is much more similar to an industrial one, with measurement built into it.

The PSP approach [10] is probably the first attempt to teach measurement in practice, and has been used in several universities [1, 15, 17, 18]. It has also been considered as an ideal context for empirical studies [19]. However, the PSP is too complex to be used by students in their initial programming courses.

In this paper we present the approach that we have developed, stressing the two main goals that we are pursuing and that can be replicated by the research and teaching community:

- improving the way of teaching both programming and software measurement,
- developing a baseline of measures for empirical software engineering research.

The paper is organized as follows. Section 2 defines in detail the goals of our activity and the measures collected, section 3 describes the new procedure used during exams, section 4 presents an analysis of the initial baseline, section 5 discusses the results obtained and finally conclusions and future work are presented in section 6.

2. Goals

There are two high level motivations for this work. First, we want to improve the quality of teaching in a programming course, and second, at the same time, we want to collect measures to develop a baseline for research purposes.

We can detail further the *teaching goals* (TG) as follows:

- *TG1:* Grade students with a quantitative, repeatable process based on process and product measures.
- *TG2:* Estimate the teaching effectiveness.
- *TG3:* Expose students to testing and coding activities similar to those used in industrial processes.
- *TG4:* Assess quantitatively the difficulty and length of programs given to students during exams in order to provide a reasonably constant level.
- *TG5:* Reduce ambiguities and misunderstandings in the specification of the program to be developed by the students.

In addition we had in mind two *research goals* (RG):

- *RG1:* Collect basic process measures (productivity, defects and defect density) on a large population of students to characterize it.
- *RG2:* Compare student population with industrial population, from the point of view of external validity of experiments performed with students.

2.1. Measures

Table 1. Definition of measures.

Name	Definition	Unit	Collection procedure
Effort	Time spent by student to develop program for exam	Min.	End - Start Start time is equal for all students. End time collected on file time stamp
Defect	Defect in a program as evidences by failure of an acceptance test		Number of failed acceptance tests performed with JUnit
Size	LOC: Line of Code: total lines of code, only counts non-blank and non-comment lines inside method bodies	LOC	Used JavaNCSS[1], a simple command line utility which measures standard source code metrics for the Java programming language like "non commented source statements" (NCSS); packages, classes and methods number and other ones.
	NOC: Number of classes	NOC	
	NOM: Number of methods	NOM	
	AMC: Average methods per class	NOM / NOC	
	diffLOC: Number of changed/added lines of code	LOC	We run the GNU diff utility[2] and took the maximum among the lines added and removed comparing V1 to V2
Productivity	Size/effort (size as LOC)	LOC / Hour	(indirect measure)
Defect density	Defect/size (size as LOC)	Defect/ KLOC	(indirect measure)

[1] Available at: http://www.kclee.com/clemens/java/javancss/.
[2] Available at: http://www.gnu.org/software/diffutils/diffutils.html.

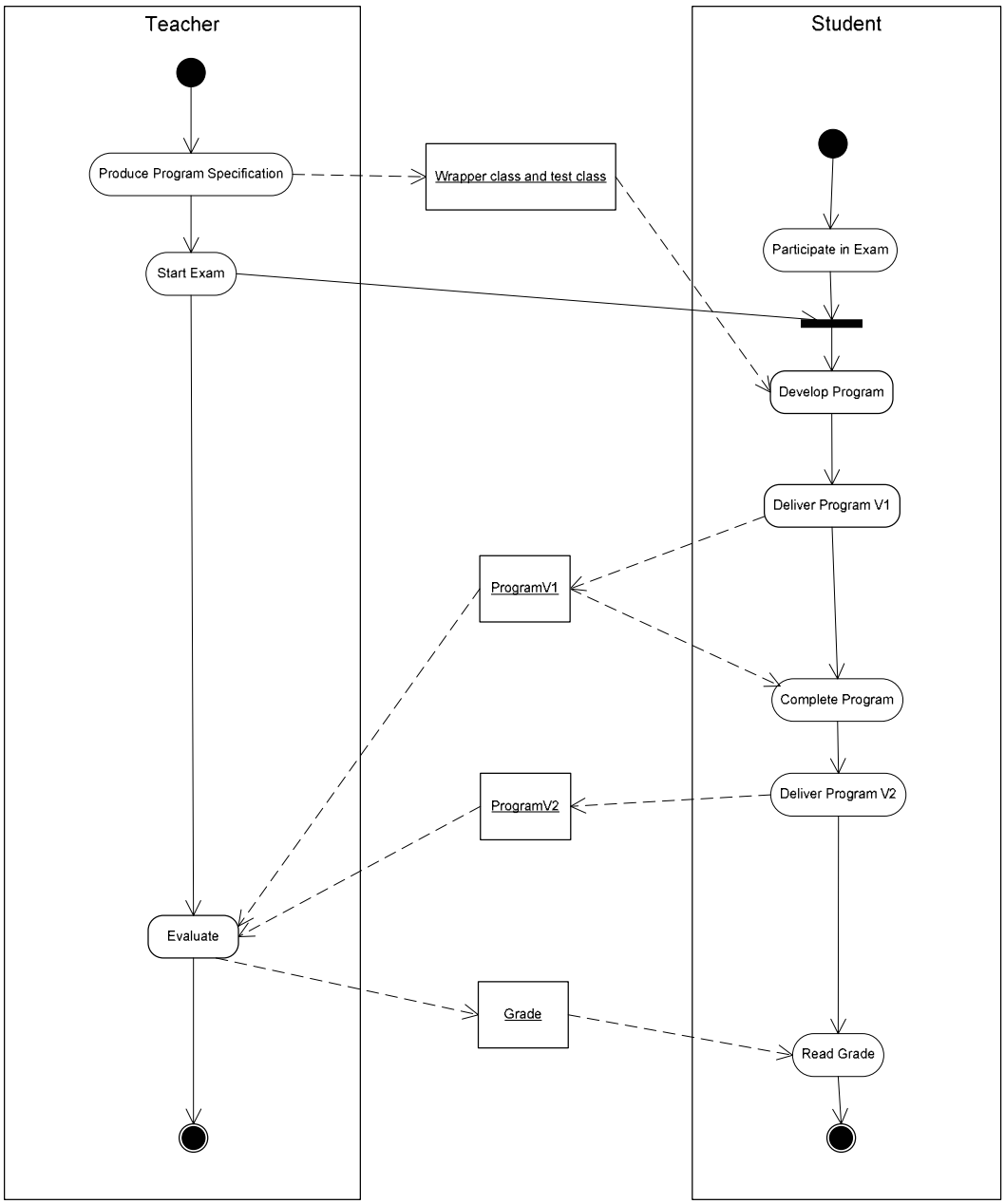

Figure 1. Examination process.

We concentrate on basic process and product measures, namely effort spent and defects. Table 1 defines in more detail the measures and how we collect them.

3. The examination procedure

The course teaches Java with a test first approach; students are used to the tools (JUnit, Eclipse) and the test first approach.

The examination procedure (Figure 1 contains the UML activity diagram describing it) starts with the teacher defining the program to be developed. The program is described textually, and also by a number of test cases, written in Java/JUnit, that define the functions requested. Figure 2 contains an excerpt from an exam. HealthSystem is the wrapper class with all requested functions defined, such as getPerson(). Exam is the test class that contains test methods, such as testPerson(). testPerson tests if the program is capable of adding a person, finding a person that has been defined or raising an exception if the person has not been defined. The wrapper pattern (e.g. class HealthSystem) is used to allow the students using the internal design they prefer.

Proceedings of the 10th International Symposium on Software Metrics (METRICS'04)
1530-1435/04 $ 20.00 IEEE

In case of conflict between the textual description and the Java test cases, the test cases prevail.

The students, working in the Lab, access the exam specification as a .java file shared in the LAN of the Lab (see Figure 3) and they can start programming directly from the file. Each position in the Lab is equipped with Java, Eclipse and JUnit.

```
public class Exam extends TestCase {
public void testPerson(){
   HealthSystem hs=new HealthSystem();
   hs.addPerson(
      "John", // name
      "Smith", // surname
      "123-45-6789"); // SSN
   try{
      Person p = hs.getPerson(
                        "123-45-6789");
      assertEquals("John",p.getName());
      assertEquals("Smith",
                     p.getSurname());
   }catch(ErrPersonNotExist e){
      fail("Smith should exist.");
   }

   try{
      Person p2 = hs.getPerson(
                          "111-22-3333");
      fail("SSN should not exist ");
   }catch(ErrPersonNotExist e){
      assertTrue(true); /* OK */
   }
}
```

Figure 2. Excerpt from a program specification.

Students have two hours to develop the program, then they have to hand it in. The means to do it is a web page with file upload facility. All files are then stored, under the student id, in the Exam db (see Figure 3).

We call this version of the program and related test cases V1.

Right after the end of the exam the teacher publishes on the course web site the complete set of acceptance test cases for the program. It should be noted that the specification of the program given to the students at the beginning of the exam is a subset of all acceptance tests.

The next step for a student is to complete the program and/or remove defects, running it against the complete suite of acceptance test cases. This work is done at home or wherever the student likes. Within a defined period (usually some days) the student uploads version V2 of his/her program, with the same procedure used for V1. A student can decide to retire from the exam, in this case he just does not upload V2.

V2 provided by the student must contain

- Program V2
- The result of running acceptance tests on V1
- For each failed acceptance test, a short explanation of the related defect, its location and its gravity according to the student.

Now the teacher can grade the programs. The grade depends on the number of acceptance tests passed, and on the severity of defects. Typically, clerical errors that require minor changes lower the grade very little. In principle, the grading procedure could be automated by assigning a weight to each acceptance test passed and/or a weight to the gravity of each defect. However, at this initial stage, the teacher still uses a manual procedure and judgment to define the grades.

We must clarify that we do not ask the student to improve very much V2. The grading is mainly done on the basis of V1; the purpose of V2 (and its difference from V1) is to spot and assess the gravity of the errors made by the student during the exam.

A side effect of this new procedure is the availability of all measures defined in Table 1 and of source code for V1 and V2 of all programs. Only the classification of defects would require a manual intervention. This procedure will be applied to all exam sessions of the course, for around 200 students per year.

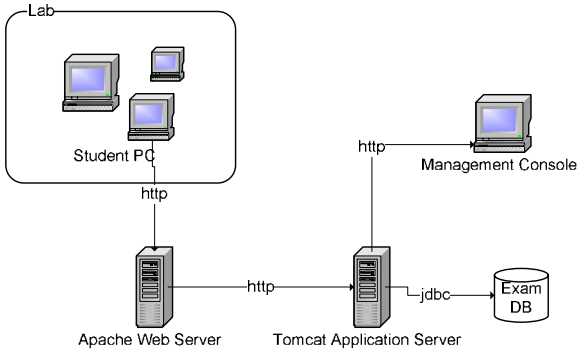

Figure 3. Lab configuration.

4. Data analysis

We present here the analysis of data we conducted on the first exam session conducted using the presented approach.

The analyses considered only the actual classes developed by the students, thus they excluded:

- JUnit test classes,
- exception classes.

The former were excluded because they were provided by the teacher. The latter are very simple classes containing one method and a few lines of code, and given the small number of overall classes therefore they could unbalance excessively the statistics.

In the session we analyzed, 35 students delivered V1 of their program. Among those, 24 (68%) delivered also version V2. Eventually 20 students passed the exam.

It must be noted that the students are allowed to deliver the V1 as many times per year as they like. But they are considered to actually attend the exam only if they deliver V2. In the following analysis only students who have delivered both versions will be considered.

4.1. Size

We examine the size of the programs developed by the students under two different perspectives. First of all we look at the evolution between the two versions, the one developed in the laboratory during the exam session and the one corrected and enhanced at home.

Second, we compare the programs with the solution provided by the teacher, assuming that this is the 'optimal' one. We compute the distance of the programs developed by the students from the teacher's one in term of size (both LOC, classes and methods). We assume that the lower the distance, the better the students have understood the concepts presented during the course.

The first size metrics we consider is the *number of classes* (NOC), which is summarized in Table 2.

Table 2. NOC summary.

	V1	V2	Teacher
Min	3.0	3.0	
Max	8.0	8.0	
Mean	4.5	4.6	4
Median	4.5	4.0	
StDev	1.3	1.3	

The distributions are not normal, as expected they are positively skewed (skewness 1.07 and 1.20). Figure 4 contains the boxplots of NOC for V1 and V2. The thick line in the plot represents the NOC for the teacher's solution.

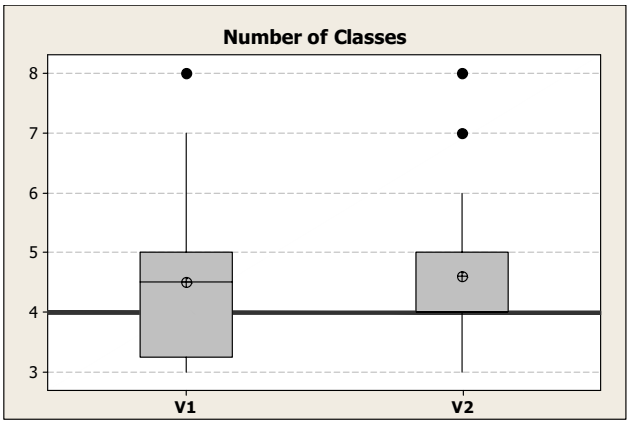

Figure 4. Box plot of NOC.

We can observe that there are very small changes between V1 and V2 and a reduction of the spread. In addition the median for V2 equals the NOC for the solution developed by the teachers.

The second size metrics we analyze is the *average number of methods per class* (AMC). The measurements are summarized in Table 3.

Table 3. AMC summary.

	V1	V2	Teacher
Min	2.00	2.00	
Max	7.30	7.30	
Mean	4.10	4.70	5
Median	4.25	4.80	
StDev	1.50	1.60	

Both versions' metrics are close to a normal distribution (Anderson-Darling p = 0.571 and 0.493 respectively), they are little skewed (skewness 0.20 and -0.12). Figure 5 shows the boxplots of the AMC for V1 and V2. The thick line in the plot represents the value for the teacher's solution.

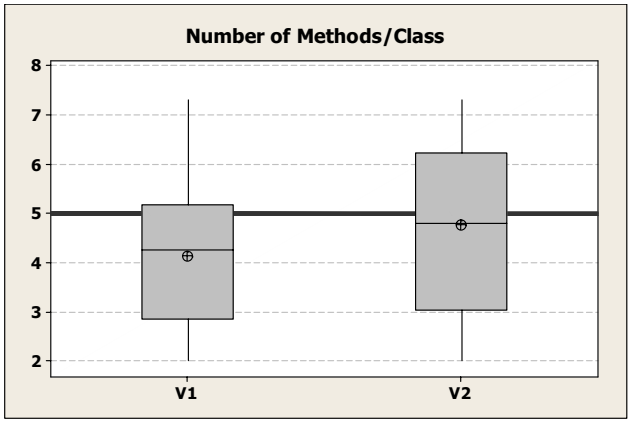

Figure 5. Box plot of AMC.

As in the case of NOC, we can observe a convergence of the two versions towards the value of the teacher's solution.

The third measure of size is the number of lines of code (LOC). Table 4 shows a summary of the LOC metrics.

Table 4. LOC summary.

	V1	V2	Teacher
Min	72.5	110.0	
Max	354.0	354.0	
Mean	147.5	181.1	181
Median	131,5	171.5	
StDev	65.8	59.4	

The distributions of metrics of the two versions are positively skewed (skewness 4.31 and 2.56); they follow a log-normal distribution. Figure 6 shows the boxplots of the LOC for V1 and V2. The thick line in the plot

represents the value for the ideal solution.

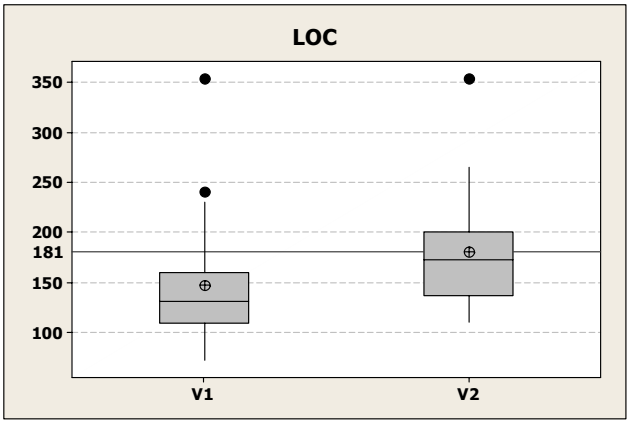

Figure 6. Box plot of LOC.

In this case too, we notice a convergence towards the value of LOC in the solution provided by the teacher.

The convergence of all three size metrics towards the teacher's solution suggests a convergence of the design towards the teacher's design. This was confirmed by looking at the designs produced by the students. The students' designs were extracted from the code using the open source tool ESS-Model[3].

Our interpretation is that the design concepts taught in the course were learned correctly. On the other hand it is true that in the exam the students dealt with a very simple problem with no big design choices to make.

For a complete assessment we must take into account that, even if the number of LOCs does not change, there could be substantial differences between V1 and V2. A more accurate measure of how programs evolve between V1 and V2 is given by the diffLOC, which measures the number of lines added or changed. The distribution is close to normal (skewness = 0.37, kurtosis = -0.28) as we can see in Figure 7. The mean of diffLOC is 94.1, median 93.0 and standard deviation 61.6.

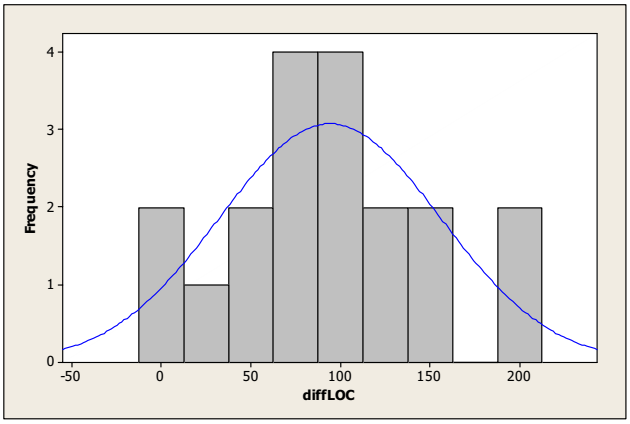

Figure 7. Histogram of diffLOC.

[3] Available at: http://essmodel.sourceforge.net/.

The diffLOC could also help us achieving a constant level of difficulty for programs given at exams. If the average diffLoc for a program grows too much, this could be an indicator that the program was too difficult.

4.2. Grading

We investigated the relationship between the metrics and the grade that was assigned independently by the teacher (i.e. without looking at the metrics). The variables with higher correlation coefficient with the grade are defect density and diffLOC. The regressions are presented in Figure 8 and Figure 9.

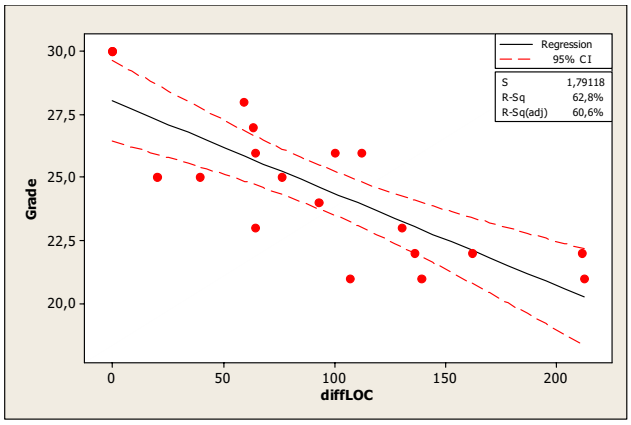

Figure 8. diffLOC vs. grade.

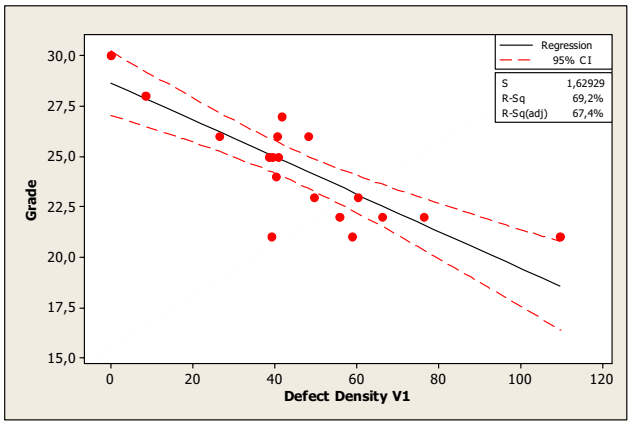

Figure 9. Defect density vs. grade.

The regression of grade versus the two combined variables has an R^2 = 77% (74.2% adj.). The variables are significant at the standard α-level of 0.05.

Coefficients and relative P-values of the joint regression are shown in Table 5.

Table 5. Productivity statistics.

Predictor	Coefficient	P
Constant	28.972	< 0.001
Defect Density	-0.059	0.006
diffLOC	-0.018	0.033

The regression equation constitutes a good starting point for the definition of a tool to support the grading procedure. However, the regression does not consider other factors, such as the gravity of the defects and the design, that are currently evaluated by the teacher, and that can remarkably influence the final grade.

What we aim at is not a fully automated grading system, but a tool that provides an initial estimation of the grade, which can be refined by the teacher.

4.3. Productivity

The second category of metrics we considered is the productivity; measured in terms of LOC per hour, LOCH. We measure productivity only with reference to V1, since we have timings only for the exam. The main statistics are summarized in Table 6.

Table 6. Productivity statistics.

	V1
Min	32.00
Max	157.00
Mean	65.56
Median	58.45
StDev	29.20

The distribution of LOCH is heavily skewed (skewness is 1.8) and presents a relevant peak (kurtosis = 4.3); it appears to follow a log-normal distribution as shown in Figure 10. It has a large standard deviation.

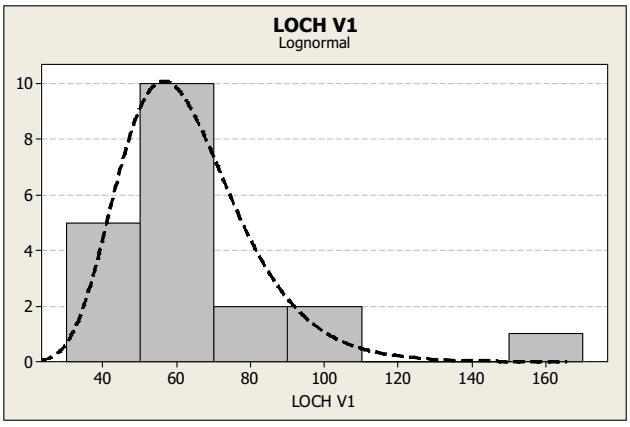

Figure 10. Histogram of LOCH.

We can compare the productivity we measured with other measures found in the literature. They are summarized in Table 7.

Table 7. Productivity comparison.

	Experiment type	Productivity (LOCH)
1	Test based	65.56
2	Test based with rework	42.09
3	PSP from [15]	35.63
4	Non PSP from [15]	30.00
5	Industry from [12]	7.60
6	Industry from [14]	5.50
7	PSP from [1]	20.50
8	XP (first release) from [2]	23.00
9	XP (second release) from [2]	31.00

The first row represents the productivity we measured during the exam.

The second row presents a proxy of the productivity considering also the rework effort required to produce version V2. This productivity includes the work performed both during the exam and at home.

For this estimation we assumed that the students worked at home with a productivity that is a fraction η of the productivity that we measured during the exam. Based on this assumption we estimated the time the students used to produce V2 with the following formula:

$$T_{V2} = \frac{diffLOC_{V2-V1}}{\eta \cdot LOCH_{V1}}$$

being $diffLOC_{V2-V1}$ the number of lines either added or changed from V1 to V2.

As a first approximation we assumed $\eta = 0.5$. Actually the rework involves finding the defects and correcting them, being the first activity very time consuming. In addition the stress condition existing during the exam does not hold anymore at home.

As a result we can compute the overall productivity including the rework as:

$$LOCH_{V2} = \frac{LOC_{V2}}{T_{V1} + T_{V2}} = \frac{LOC_{V2}}{T_{V1} + \frac{diffLOC_{V2-V1}}{\eta \cdot LOCH_{V1}}} = \frac{LOC_{V2}}{\frac{\eta \cdot LOC_{V1} + diffLOC_{V2-V1}}{\eta \cdot LOCH_{V1}}} = LOCH_{V1} \cdot \frac{LOC_{V2}}{LOC_{V1} + \frac{diffLOC_{V2-V1}}{\eta}}$$

Note that if $\eta = 1$ (i.e. the productivity at home is the same as during exam) and no lines are modified from V1

to V2 but only some lines are added (i.e. $diffLOC_{V2\text{-}V1}$ = LOC_{V1} - LOC_{V1}) then $LOCH_{V1}$ = $LOCH_{V2}$, as expected. Using this formula, we obtain a mean productivity of 42.09 LOCH.

Rows 3 and 4 report information from [15], which reports both PSP and non PSP projects. Considering only the Java PSP projects the average productivity is 35.6 LOC/hour, for these projects the average number of acceptance tests is 2.1 and the averages size is 384 LOC. For non PSP Java projects the average productivity is 30 LOCH/Hour, for these projects the average number of acceptance tests is 2.2 and the average size is 327 LOC. The project sizes and the number of acceptance tests make these projects comparable with our project (our average LOC is 181 and our number of acceptance test is 10).

Row 5 reports data from [12], which includes metrics also about analysis, design, unit and acceptance tests. For the coding phase it mentions a range from 3 to 50 functions-points (FPs) per staff month and a weighted average of 15 FPs per staff per month. These figures include also testing, initial design, and refactoring. We assume for coding only activities a figure of 20.

Since data are expressed in FPs we ought to translate them into LOCs. For this purpose we adopted the mean among the conversion factors proposed by Jones himself [12] (1 FP = 53 Java LOC) and QSM [16] (1 FP = 62 Java LOC), i.e 1 FP = 57.5 LOC. Using this factor we can compute an industrial individual productivity of 1150 LOC per month. Assuming a typical work month to be of 152 hours [5], we can assume a productivity of 7.6 LOC/Hour.

Row 6 reports data from 32 projects in HP and Agilent. The mean productivity is 26.4 LOC per person-day. Considering an average working day of 8 hours, where around 60% is devoted to programming, we can compute 5.5 LOC/Hour.

Row 7 reports metrics from [1]; they used a process similar to [15] but they dealt with a project five times bigger (1821 LOC) than the previous case. This explains a lower productivity.

Rows 8 and 9 report data from two case studies adopting agile methodologies, XP in particular. Thus they used the test first approach, being it one of XP's key practices. This context makes this data a good reference point for comparing our results.

Comparing the mean productivity cited in the literature to the mean productivity we measured in our study we find that in our case the productivity is higher.

However, our measure of productivity does not consider testing and debugging that is, for most students, completed after the exam session. However, including also an estimate of this figure the productivity is still higher.

4.4. Defect density

The third metrics we consider for our course is the defect density of the program V1 delivered by the students. We measure the density of defects in terms of number of defects per thousand of LOC.

We consider as defect a JUnit test method that fails. Even though a finer granularity based on assertions is possible, we decided to use this metrics because it is easy to collect by looking at the output produced by JUnit. In addition test methods are fairly independent from each other, while asserts are not. If in a method an assert fails then all the following asserts in the same method are not executed and thus are considered failed.

The main statistics about the defect density are summarized in Table 8.

Table 8. Defect density statistics.

	V1
Min	0.00
Max	109.60
Mean	43.21
Median	40.55
StDev	25.55

The distribution approximates a normal curve but it is slightly skewed (skewness = 0.48) and has a higher peak (kurtosis = 1.49). We can see the graphical representation of the distribution in the histogram of Figure 11.

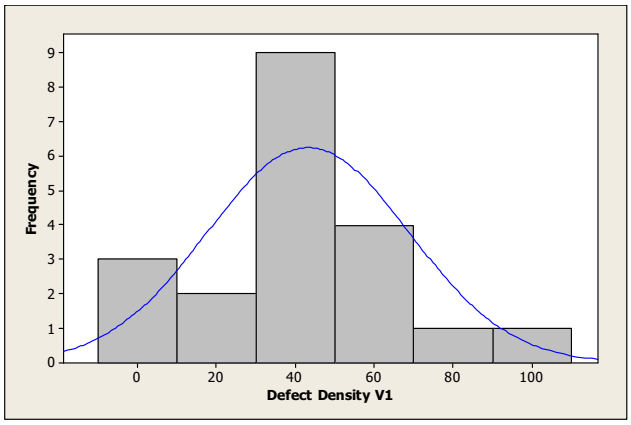

Figure 11. Histogram of defect density.

We can compare the values of defect density with values found in the literature. Industrial data usually report as defects the failure reports during system operation; this is comparable with our definition.

We measured the defect density of V1 because we would measure zero defects in V2. This "anomaly" is caused by the method we used to measure defects: we measure defects against the acceptance tests, but V2 includes the rework to make the program pass all

acceptance tests. Our approach is not able to reveal the residual defects after this stage. Nevertheless we consider the defect metrics we collected on V1 a good approximation of the defects actually present in the code after coding and unit test but before acceptance test.

Table 9. Defect density comparison.

	Source	*Defects/KLOC*
1	JPL Flight Software from [6]	8.6
2	JPL Ground Software from [6]	2.1
3	UK defect rate 1990 from [7]	2.0
4	US defect rate 1990 from [7]	4.0
5	Industry from [14]	0.2
6	Denmark Student from [1]	67.0
7	Our	43.2

Rows 1 and 2 report data from the JPL [6]. They refer to both flight and ground software. The defect density is very low as we would expect from safety/mission critical software systems.

Rows 3 and 4 report typical values for UK and US software [7]. The defect density is very low.

Row 5 reports data from 32 projects from HP and Agilent [14]. They cite 18.8 average defects reported by customers per month over a 12 month period per million LOC. If we consider a year period we have a 225.6 defects per million LOC. This value is very low, especially compared to the other industrial values that refer to defect data collected during several years.

Row 6 reports the defect density measured in an experiment with students using the PSP approach [1]. These values are measured before testing thus they do not include the rework; for this reason they can be more easily compared with ours, which do not include the rework as well.

As we expected the industrial values are far lower than those measured in our course. In particular we measured the defects of V1, therefore before any rework activity.

The metrics collected on students in [1] are higher than ours because their metrics is measured before any test activity. In contrast our measures follow an initial test phase.

In general we can see that, not surprisingly, the defect density of professional developers is lower than what we measured with students. Our defect density is lower than another academic study, but they didn't perform any test. In comparison with industry, students are in an early phase, before completing development.

In V2 defect density is zero, interesting data would be effort to correct defects, and its distribution among students, but we do not have it. This is an issue for future improvement. This, in conjunction with a classification of defect types and gravity, could lead to a better interpretation of the collected data.

5. Discussion

We discuss now the findings presented above and the qualitative experiences collected during the study in reference to the teaching and research goals.

5.1. Teaching goals

The majority of students has accepted with enthusiasm the new procedure for exams. The main reason is the possibility of producing the program with a powerful development environment like Eclipse, including debugger, language documentation, and automatic code completion.

All students are exposed to coding and testing procedures similar to those present in an industrial environment. The teachers play the role of the test group that writes the acceptance tests, while the students play the role of developers and unit/system testers.

Another advantage is in the improved clarity of the specification of the program. In comparison with plain language specification, misunderstandings by the students due to ambiguities have dropped sharply. A related point to be addressed here is the number of acceptance test cases to be published with the specification. Up to now, we have published in the specification a small subset (typically 50%) of test cases. In principle we could publish all test cases. The pro would be that misunderstandings would be virtually avoided. The con would be the reduction in analysis effort required to the student, who simply receives a complete specification, not a real world case. We intend to experiment on this point in the next exam sessions, varying the percentage of test cases published with the specification.

From the point of view of grading, the procedure is now much more transparent to the student. The student himself/herself runs the acceptance tests and checks the result. Apart the improved fairness, the student has clear evidence of what was wrong in his program. Besides, (s)he learns by example the test conditions that should be considered and that (s)he may have overlooked.

Another advantage lies in the evaluation of the program. A typical problem is to give in all exam sessions programs of similar size and complexity. We have now two quantitative means to assess them. A priori we have the number of acceptance test cases, which can be considered a proxy of size and complexity of a program. However, this measure can be biased because acceptance tests are defined by the teacher. A-posteriori we have the difference in size between the programs V2 and V1 (see figures 4, 5, 6). If this difference is larger than average, the program may be more complex or longer than average. In these cases the grading can be adjusted accordingly.

Right now, grading is performed manually, starting by the number of acceptance tests passed. On the long run,

Table 10. Goals – Measures – Results.

GOAL	MEASURE	RESULT
TG1 Improve grading procedure	diffLoc defect density	Very good. Procedure is now more quantitative, equal for all students, less dependent on specific evaluator.
TG2 Assess teaching effectiveness	NOC, AMC, LOC Compared between teacher and student s' versions of program	Applied only to one course. To be further verified over several courses.
TG3 Improve coding and testing process learnt/used by students	all the examination procedure	Very good. Students satisfied for tools used. Students apply test by doing.
TG4 Assess quantitatively the difficulty of exams	diffLoc between V1 and V2 of students' program	Applied only to one exam session. To be further verified over several sessions.
TG5 Improve the specification of exams		Very good. Number of ambiguities raised by students sharply dropped.
RG1 Collect basic process measures	LOCH defect density	Very good, measures are collected. To be extended with other measures.
RG2 Characterize student population to discuss external validity of experiments with students	LOCH defect density	Good. Although the data set is for the moment small, the results are reasonable, with productivity higher than industry figures, and higher defect density before acceptance test.

we also want to investigate the issue of automatic grading taking into account other factors like diffLOC and the gravity of defects. The idea is to define a formula to compute the grades, compare a posteriori its results with manual grading, and tune it until satisfactory. The advantage is the fairness for students; the open issue to solve is how to evaluate the internal design of a program and not only the external functionality.

5.2. Research goals

We collected data about student's productivity and defect density. We argued that the methodology we used is valid also from an educational perspective. In addition we claim that the metrics we collected are valid and useful for comparison with other academic settings, but also industrial ones. Further, we are enlarging the pool of data points at each new exam session.

In the future we will add new measures, such as defect classification, defect gravity and possibly rework effort per defect; and we will analyze the student's process with tools such as Hackystat [11].

The extremely high productivity, also in comparison to industrial data, can be explained by different factors:

- the exam situation is very stressful and stimulates the students,
- the specifications are unambiguous and require a very small effort for interpretation,
- the programs are very small therefore the design requires a reduced effort.

In addition we measured the productivity for a short period when the students just do programming. In an industrial settings programmers have several occupations in addition to programming such as meetings, social interactions, and communications (e.g. email), which make productivity lower.

The high defect density in our case study was expected especially if compared with industrial results, which are collected after an extensive testing activity. The only data that not included any testing activity (Table 9, row 6) is higher because on the contrary our data are collected after an initial test and rework activity (the subset given at the exam).

6. Conclusion and future work

We have presented an approach to grade students in a programming course, heavily based on using a test based approach and a set of advanced, openly available tools such as Eclipse and JUnit. This approach allows, as a side effect, the collection of productivity and defect density measures over a large population of students. The rationale for introducing this approach and the results obtained are summarized in Table 10.

The students like the course because it is based on cutting edge development environments; in addition the exam and the evaluation criteria are clear and unambiguous.

The teachers appreciate the approach since it couples programming with measuring and testing, students learn

measurement and quantitative assessment by doing it.

Currently we are applying the tools and techniques presented in this paper to provide fully automated feedback for lab assignments. This is particularly useful since we are dealing with circa 200 students.

Our approach is repeatable on the basis of the information provided in this paper.

The results presented in this paper refer to the initial data we collected; we plan to continue the collection of data in the next exam sessions during the future years and to collect other measures. In perspective we aim at building a baseline of student data, the potential is good given an average of 200 students per year. The baseline of data can be shared with the research community for many purposes (like www.isbsg.org.au do with industry).

The research community is invited to add data points; it would be possible to compare different realities and nations.

We want to state clearly that we did not expect the students to match professionals' performance. The purpose of the comparison is to build a knowledge base of the relationship between the features of students and professionals. This will enable the research community to draw better conclusions from experiments conducted with students.

7. Acknowledgements

We wish to thank Giorgio Bruno, teacher at Politecnico di Torino for his cooperation and Hakan Erdogmus for the useful discussions that led to the definition of the measures and tool infrastructure of the course.

8. References

[1] P. Abrahamsson and K. Kautz, "The personal software process: experiences from Denmark" Proc. of 28th Euromicro Conference, 4-6 September, 2002, pp. 367-374.

[2] P. Abrahamsson, "Extreme Programming: First Results from a controlled case study" Proc. of EUROMICRO'03, 2003

[3] K. Beck, *Extreme Programming Explained: Embrace Change*. Reading, Massachusetts: Addison-Wesley, 2000.

[4] K. Beck, *Test-Driven Development : by Example*: Addison Wesley, 2003.

[5] B. Boehm, *Software Engineering Economics*: Prentice Hall, 1981.

[6] M. W. Bush, "Getting started on metrics" Proc. of 12th ICSE, 1990, pp. 133-142.

[7] M. Cusumano and C. Kemerer, "A Quantitative Analysis of US and Japanese Practice and Performance in Software Development" *Management Science*, 36 (11): 1384–1406, November 1990.

[8] Eclipse Consortium, "Eclipse Platform Technical Overview" Object Technology International, February 2003 available at http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[9] H. Erdogmus and M. Morisio, "Evaluation of combinations agile practices with controlled experiments" Proc. of EEAP, 2002

[10] W. S. Humphrey, *Introduction to the Personal Software Process*: Addison-Wesley, 1997.

[11] P. M. Johnson, H. Kou, J. Agustin, C. Chan, C. Moore, J. Miglani, S. Zhen, and W. E. J. Doane, "Beyond the personal software process: metrics collection and analysis for the differently disciplined" Proc. of 25th International Conference on Software Engineering (ICSE 2003), Portland (OR), USA, May 10-13, 2003, pp. 641 - 646.

[12] C. Jones, *Applied Software Measurement: Assuring Productivity and Quality*: McGraw-Hill, 1991.

[13] JUnit.org, *www.junit.org*.

[14] A. MacCormack, C. Kemerer, M. Cusumano, and B. Crandall, "Trade-offs between Productivity and Quality in Selecting Software Development Practices" *IEEE Software*, 20 (5): September/October 2003, 2003.

[15] L. Prechelt and B. Unger, "An experiment measuring the effects of personal software process (PSP) training" *IEEE Transactions on Software Engineering*, 27 (5): 465 - 472, May 2001.

[16] QSM, available at http://www.qsm.com/FPGearing.html.

[17] P. Runeson, "Experiences from teaching PSP for freshmen" Proc. of 14th Conference on Software Engineering Education and Training, 19-21 February, 2001, pp. 98-107.

[18] P. Runeson, "Using Students as Experiment Subjects – An Analysis on Graduate and Freshmen Student Data" Proc. of 7th International Conference on Empirical Assessment & Evaluation in Software Engineering (EASE'03), April 8-10, 2003

[19] C. Wohlin, "The PSP as a context for Empirical Studies" *Software Process Newsletter* (12): 7-12, 1998.