Integrating Software Engineering Key Practices into an OOP Massive In-Classroom Course: an Experience Report

Integrating Software Engineering Key Practices into an OOP Massive In-Classroom Course: an Experience Report

Marco Torchiano
Politecnico di Torino
Torino, Italy
marco.torchiano@polito.it

Giorgio Bruno
Politecnico di Torino
Torino, Italy
giorgio.bruno@polito.it

ABSTRACT

Programming and software engineering courses in computer science curricula typically focus on both providing theoretical knowledge of programming languages and best-practices, and developing practical development skills. In a massive course – several hundred students – the teachers are not able to adequately attend to the practical part, therefore process automation and incentives to students must be used to drive the students in the right direction.

Our goals was to design an automated programming assignment infrastructure capable of supporting massive courses. The infrastructure should encourage students to apply the key software engineering (SE) practices – automated testing, configuration management, and Integrated Development Environment (IDE) – and acquire the basic skills for using the corresponding tools.

We selected a few widely adopted development tools used to support the key software engineering practices and mapped them to the basic activities in our exam assignment management process.

This experience report describes the results from the past academic year. The infrastructure we built has been used for a full academic year and supported four exam sessions for a total of over a thousand students. The satisfaction level reported by the students is generally high.

CCS CONCEPTS

• **Applied computing** → **Education**; • **Software and its engineering** → *Software configuration management and version control systems*; *Integrated and visual development environments*; *Object oriented development*; *Software testing and debugging*;

KEYWORDS

Object-Oriented Programming, Java, Automated Testing, Configuration Management, Automated Grading

SEEM'18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5750-0/18/05…$15.00
https://doi.org/10.1145/3194779.3194786

ACM Reference Format:
Marco Torchiano and Giorgio Bruno. 2018. Integrating Software Engineering Key Practices into an OOP Massive In-Classroom Course: an Experience Report. In *SEEM'18: SEEM'18:IEEE/ACM International Workshop on Software Engineering Education for Millennials , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3194779.3194786

1 INTRODUCTION

Software development eventually consists in delivering working code [3]. Computer science and the software-related part of computer engineering should teach programming and on top of that provide software engineering skills.

At Politecnico di Torino, Italy, the first course introducing a "*modern*" programming language is the Object Oriented Programming (OOP) course where the language of choice is Java. For historical reasons the BSc degree in computer engineering does not include a Software Engineering course, therefore we decided to provide the basic Software Engineering knowledge in the OOP course.

The students attending the course are Millennials: they were born before 1996. While previous programming courses in the curriculum adopted paper-based exams, we opted for a computer-based exam to leverage the technology familiarity of "digital natives".

The course, in addition to the Java language (version 8), provides an introduction to UML [13], design patterns [9] and basic software engineering practices. It basically follows the indications provided in [1]. The three key practices that we integrated in the course are:

- automated testing: represents a clear step from informally trying the program to a formalized and repeatable verification activity,
- configuration management: introduces a standard way of versioning code and keeping a common shared repository,
- integrated development environment: provides basic features supporting coding, e.g. code completion, language specific presentation, automatic incremental build, error highlighting, and automatic code refactoring [8].

Such practices are meant to develop software testing and software configuration skills as recommended in SWECOM 1.0 [10]. Moreover, automated testing appears particularly suited to responde to the Millennials' need for frequent feedback [12].

A particularly tough challenge in the introduction of such practices is represented by the size of the course: the largest of the three parallel instances counts hundreds of newly enrolled students (330 for a.y. 2017/18). The course is taught in presence and offers practical sessions in the lab facilities of the university. We can define it as a Massive In-Classroom Course (MICC). A MICC, as opposed to a MOOC, exhibits the following characteristics:

- the numbers are smaller than on-line courses, but still large for regular university courses;
- while there are videolectures, they just record the lectures held in classroom therefore they are neither primarily designed nor optimized for autonomous fruition;
- the practical organization must enable anybody to attend in person all the educational activities (both lectures and labs);
- it is not necessarily open, although all materials for the OOP course are freely available online.

This paper reports the experience in integrating a few key software engineering practices in the OOP course by means of specific technologies. In particular we show how the devised solution was able to address both organizational and learning objectives. On one side, such technologies support the management of assignments in the course, on the other side they are required to perform essential tasks thus stimulating the students to acquire the related basic skills.

First of all, section 2 presents the context of the course and the detailed motivation that lead us to this course implementation. Then section 3 provides the details of how such key SE practices have been realized in the course. After that, section 4 discusses the educational implications, the main issues encountered, and the lessons learned.

2 CONTEXT AND MOTIVATION

The Object-Oriented Programming course is located in the second year of the Bachelor degree in Computer Engineering[1] at Politecnico di Torino. The first year encompasses fundamental topics for all engineering disciplines (maths, computer science, physics), the second year introduces general ICT and computer engineering topics: circuit theory, algorithms and data structures, object-oriented programming, and databases. The third year focuses on more advanced topics, e.g. operating systems, computer networks, communications, electronics.

The OOP course introduces Object-Oriented programming using the Java programming language and provides basic knowledge of software engineering. The main contents are:

- Basic OO features (1 ECTS[2]) including the OO paradigm, Java, classes and attributes, visibility, basic types, and practical skills concerning the Eclipse IDE.
- Inheritance, interfaces, and advanced features (2 ECTS), including functional interfaces, lambda expressions, exceptions, and generic types.
- Standard libraries (3 ECTS) including the Collections framework, streams, files, dates, threads, and GUIs.
- Software Engineering principles (2 ECTS), including the Software life cycle, UML, Design Patterns, Configuration management, Testing. The latter two subtopics provide basic skills in Subversion[3] and JUnit[4].

The course consists of over 70 hours in the classroom, including both lectures introducing the topics and live coding sessions presenting and discussing programming assignment solutions, and 20 hours in the lab dedicated to the development of programming assignments. While one could argue that the hours in the lab should be much more, this is not practically possible since there are limited lab facilities that are shared with several other courses, e.g. there are 20 replicas of the basic Computer Science course with about 200 students each.

The OOP course is run in three replicas, two in Italian and one in English, with 330, 270, and 115 enrolled students respectively.

The exam process is sketched by the activity diagram shown in Figure 1 and encompasses a few steps:

(1) the teacher prepares an initial project and uploads it;
(2) during the exam, the students develop a small program[5] in two hours, while sitting in the lab;
(3) at the end of the exam the students must submit their program;
(4) meanwhile, the teacher has prepared an acceptance test suite;
(5) after the exam, the teacher assesses the functionality of the program versus the test suite;
(6) the students have to fix or complete the program in order to make it pass all the tests in the suite;
(7) the teacher grades the work done by the students.

The assignment consists of:

- a requirements document, usually made up of four or five sections that are designed to be implemented incrementally, because the features required in a section make use of the ones defined in previous sections. The requirements describe a set of classes and their methods;
- an initial project, containing skeletal classes, i.e. classes with the methods called by the tests but with minimal bodies returning fixed values (e.g. `null`); the project can be opened with the reference IDE and is syntactically correct;
- an example class, containing a `main()` method that exercises the most relevant methods described in the requirements. It is intended to clarify the requirements and to provide the students with a basic testing tool.

[1] Computer Engineering BSc. syllabus: https://didattica.polito.it/pls/portal30/gap.a_mds.espandi2?p_a_acc=2018&p_sdu=37&p_cds=10&p_lang=EN

[2] ECTS stands for European Credit Transfer and Accumulation System and is a standard means for comparing the "volume of learning based on the defined learning outcomes and their associated workload" for higher education across the European Union

[3] https://subversion.apache.org

[4] http://junit.org/junit4/

[5] A few examples of the required program are available at: http://softeng.polito.it/courses/02JEY/exams/

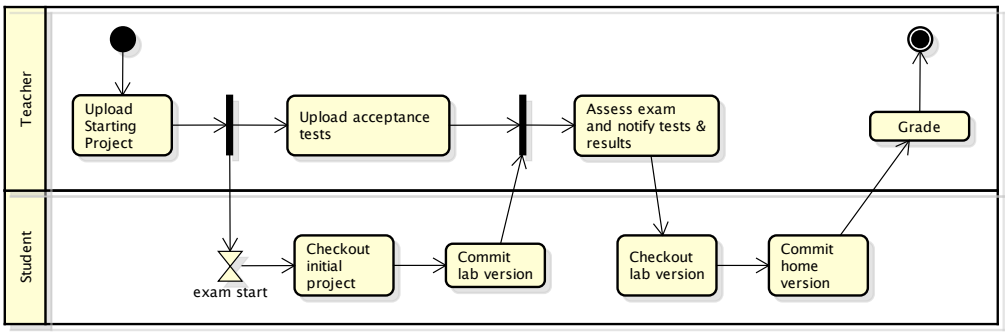

Figure 1: Exam procedure

The evaluation is computed on the basis of the functional compliance – both in terms of correctness and completeness – of the program. Such an approch has been inspired by the agile manifesto principle "*Working software is the primary measure of progress*" [3].

More in detail, the tests are packaged into a `.jar` file containing both class files and source files. This is done to avoid both unintended and malicious modifications to the test suite.

In practice the grade is computed on the basis of two indicators:

- Percentage of acceptance tests passed by the lab version (S),
- Code churn (M) applied to make the program pass all the tests.

Code churn [11] is the amount of added and modified lines of code; it is a very simple measure of the quantity of code modification.

The former indicator provides a coarse grained assessment of the functional compliance from an end-user point of view, the latter represents a fine grained evaluation and is a proxy measure of the rework needed to fix defects (correctness) and to complete unimplemented features (completeness).

The basic formula to compute the grade is:

$$Grade = c_0 + c_1 \cdot \left(S + (1 - S) \frac{c_2}{c_2 + M} \right)$$

Where the constants c_0, c_1 and c_2 are adjusted case by case based on the difficulty of the exam.

Given the above formula:

- when a large amount of modifications is applied the grade is essentially defined by the percentage of tests passed
- as the percentage of passed tests get lower the component inversely proportional to the modifications gets a higher weight.

An important aspect of this evaluation approach is that the completeness and correctness of the program delivered in the lab is evaluated by comparison. The reference program is the fully working version submitted from home, after the exam; it is a natural evolution carried out by the same student who wrote it initially in the lab. A possible alternative would be to use a predefined solution developed by the teacher as a reference, but its adequacy could be low for the following reasons:

- since there is no single solution for any given problem, the comparison with a predefined solution could penalize different – possibly even better – solutions;
- the amount of work needed to complete the program and to fix defects can reasonably be estimated only by comparing the original one with the evolved version.

The approach also encourages the students to understand the requirements, identify a design and then work on the requirements, one by one, developing fully working code (possibly just for a subset of the requirements) rather than write a complete solution in a single *big-bang*, which typically does not work.

The rationale behind such an approach is that, in real-world terms, it is better to have a program that performs correctly on a subset of requirements that a program that is almost complete but crashes at the beginning and eventually does nothing.

The above assessment method is fully automated and can be applied to large numbers of delivered projects producing objective and unbiased grades.

The current approach presented in this paper is an evolution of the one developed originally in 2003 and described in [16].

The approach was updated in response to several challenges:

- the number of students enrolled raised significantly in the latest years from around 200 per year to roughly 700 in the current academic year, making this course a real Massive In-Classroom Course; therefore the solution must be scalable and robust;
- the teaching staff is very limited: three teachers lecturing three parallel tracks (to fit lecture halls hosting 250 students at most), plus three teaching assistants supporting the students in the labs;
- the lectures are video recorded and this encourages the students to attend the course remotely. In particular the students should be able to work autonomously on

their assignments – e.g. at home – due both to personal reasons and to crowded labs; therefore the assignment management framework must be based on tools that can be easily installed on their PCs;
- the instruments and tools should enable the students to acquire skills directly usable in a real-world setting; therefore the tools should be widely adopted in practitioner communities;
- the course content was extended to include basic SE practices, so as to encourage the students to adopt or at least become acquainted with basic software engineering practices, i.e.:
  – automated testing,
  – configuration management,
  – integrated development environment (IDE).

To better characterize the learning outcomes, we can refer to taxonomies that describe curricula objectives in terms of topics and levels of understanding. In particular, Bloom's taxonomy [2, 5] classifies learning achievements into six different cognitive levels: knowledge, comprehension, application, analysis, synthesis and evaluation. While the educational goals in the programming part of the course clearly address all the six levels of the taxonomy, the software engineering part only addresses the lower levels of the taxonomy.

3 TECHNOLOGICAL PLATFORM

The technological solution we developed is based on a few technologies that both implement software engineering best-practices and cover a key role in the exam and assignment management process described above.

The SE areas we decided to cover with technologies and assignment related activities are:

- Automated Testing using JUnit
- Configuration Management using Subversion
- IDE as Eclipse

In addition we had to set for a robust method for authentication and authorization to be used during exams. We decided to use SVN authentication as the basic technology.

3.1 Automated Testing

Testing is a key technique for the Verification and Validation phase in any software development process [14], in particular automated unit testing has gained much attention in recent years.

JUnit is the de-facto standard for writing automated tests in Java [4]. While its original purpose was to write unit tests, it is also widely used as the basis for UI testing and end-to-end tests.

In our approach, JUnit is used to evaluate the functional compliance of assignments. The basic measure is the proportion of passed test cases. The JUnit execution report is the standard feedback the students receive both when they complete their lab assignments during the course and right after the exam.

In terms of test automation, the main challenge is that we need to test a huge number of programs. The peculiarity

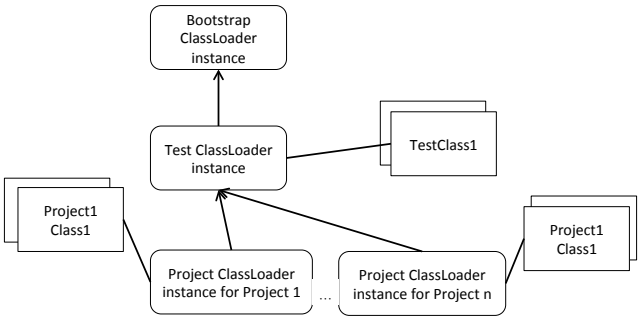

Figure 2: Hierarchy of loaders used for testing.

is that while in a regular industrial setting we have large test suites to be executed on a single program (or parts thereof), in our course we have a single (small) test suite to be run against several hundred similar programs that provide different implementations of the same classes.

A technical obstacle is that – at least in theory – we ought to start a new Java Virtual Machine (JVM) for every project, load the tests and the classes making up the program, and run the tests. Unfortunately the VM startup and class loading are very heavy tasks.

The solution we devised to achieve a reasonable scalability is to use a hierarchy of Java class loaders as shown in Figure 2.

Class loaders are classes responsible for finding and loading classes whenever the VM needs them. A bootstrap class loader is always present and it searches the classes in the predefined *classpath*. Class loaders are generally organized in a delegation hierarchy, so that if a specific class loader is not able to find a class, recursively delegates the search to its parent. In our approach a dedicated class loader class has been developed to load the test classes from a given `.jar` file. In addition we developed a project class loader that loads classes from the project path; an instance is created for each project.

The test execution starts from this more specific class loader; when a test class is needed it delegates the test class loader to get it. Since test classes are common to all the assignments, in most cases the test class loader finds them in the cache. Such a solution has several advantages:

- the test classes are loaded only once by the test class loader,
- any project specific class loader is isolated from the others so that classes with the same name can be loaded in the projects without any interference,
- overall a single VM can be used for testing several projects.

Owing to this approach, testing projects can proceed at a rate of two projects per second on a standard Ubuntu VM with 4 cores and 8GB RAM. Each project typically counts five to eight classes while the test suite includes 20 test cases at least.

In terms of acquired skills and knowledge, the students are not required to write tests – the main reason being the

short time available for the exam, i.e. 2 hours – though they must be able to: (i) import a test suite, (ii) run the tests, (iii) understand the tests results, and (iv) identify the cause of the test failures.

In particular, for the latter ability, the students have to know what the assert statements mean, how an expected exception is tested, and in general they must be able to read a failure or error message, as well as to understand a stack trace in order to locate the origin of a failure, and also to interpret the test code to figure out the conditions that led to the failure.

The implementation of the infrastructure includes 67 Java classes for a total of 6700 LOCs.

3.2 Configuration Management

Subversion (Svn) is a widespread centralized version control system [6]. Although its adoption has recently decreased in favor of more modern distributed systems, such as *git* [15], Svn is still widely used in industry and as far as our course is concerned it is easier to use, thus less error prone, and simpler to manage. In our approach Subversion is used to give the assignments to the students as well as to collect their implementations, and this takes place both during the course and at the exam.

While Svn can support concurrent development with a Copy-Modify-Merge approach, and can manage different threads of execution using branches, the course makes use only of the basic versioning features.

In practice the assignment life cycle is supported by Svn as follows:

(1) the teacher commits an initial version of a Java project together with an acceptance test suite to a *master* repository,
(2) the initial project is committed to all student repositories by the teacher using a simple script,
(3) the students check-out the initial project and start working on it,
(4) the students commit the results of their work to their own repositories,
(5) the teacher checks out the latest version of the projects available in the repositories and runs the tests on them.

The latter step is performed using the multiple classloaders approach described in the previous sub-section.

The main challenges faced in customizing Svn for the purpose of the course were as follows:

- the students must have isolated personal repositories, so that no interference can occur by mistake;
- during the exam, the students must not be able to access other students' repositories, to avoid plagiarism;
- during the exam, the students must be able to access their repositories as soon as the exam begins; therefore the repositories must be created in advance;
- the students must not be able to keep working after the exam deadline has elapsed.

The isolation can be obtained by means of a single repository containing one subfolder per student and adequate permissions. Alternatively, one repository per student can be created with the student having access to her own repository only. While the former is more efficient, it is less isolated: every time a student performs a commit, the revision number is incremented for every other students too. For this reason, even if it is more expensive we opted for the one repository per student solution.

During the course, a student sharing his credential with a colleague is generally not a problem and can foster collaboration. But, such behavior must be prevented during the exam. The solution is to create a new repository for each student who signed up for the exam. Then each repository is populated with a copy of the initial project and the credentials for the repositories are handed to the students at the beginning of the exam.

The creation of an Svn repository, on our server, typically takes 4-5 seconds, therefore the repositories must be created in advance, at the beginning of the course and before each exam session.

During the exam, students are allowed to commit their projects as many times as they wish. At the end of the exam, the teacher annotates the actual end time; only commits performed before the end time are taken into account.

In addition Svn was used to make the tests available to the students on a dedicated test repository. Sometimes errors can be found in tests after the reports have been sent to the students. By using Subversion we can update the test `.jar` inside the repository and notify the students via email.

In terms of acquired skills and knowledge, the students must learn a few basic tasks:

- performing the check-out of a project from a repository,
- performing the commit of a project to a repository.

In addition, during the course the students are encouraged to perform frequent commits when developing a project. During the exam, they are invited to commit after implementing each requirement and explicitly instructed that is *safer* to commit 10 minutes before the deadline. The goal of the course is to make the students familiar with the elementary configuration management operations that are at the basis of any workflow they will adopt in the future.

The management of the repositories has been implemented using 16 scripts in bash and python, for a total of 910 and 827 LOCs respectively.

3.3 Java IDE

The usage of an IDE is often an implicit assumption when writing code. In our course we opted for Eclipse[6] because historically it was one of the most widespread IDEs and due to the fact that it is an open-source product.

The Eclipse Java IDE is the reference IDE that is taught during the course. The configuration management and testing tasks are performed by the students using the plug-ins for this IDE.

Eclipse is installed in all labs and the students are encouraged to install it on their machines. We observe that while

[6] http://www.eclipse.org

Eclipse comes with a built-in JUnit plug-in, – oddly enough – it has no default built-in plug-in for Subversion. Therefore an additional plug-in (Subversive) has to be installed on top of the default Java IDE.

4 DISCUSSION

4.1 Learning objectives

Table 1 reports the six taxonomy levels and the corresponding capabilities addressed with respect to the three key SE areas included in our course. In the table, the capabilities addressed by the course are shown in **bold**, those partially addressed in *italic*, and the others, not addressed, in a regular font.

The cognitive levels addressed are first needed in the lab assignments the students have to perform during the course and then they are required in the exam. Therefore we are confident that the students passing the exam achieved those levels to a good degree of completeness.

The *Analyze* level for configuration management is only partly addressed because no concurrent development is used in the course, therefore no conflict will take place: this is an activity students learn in lectures but never experience in practice. For this reason, the *Evaluate* level is not addressed either.

The *Create* level is not addressed for any of the three key areas. As to testing, writing tests is a time consuming activity that cannot fit in the tight schedule (2 hours) allowed for the exam. As far as configuration management is concerned, the lack of concurrent development makes it impossible to apply merge operations. Regarding the Java IDE, all assignments start with students importing pre-defined Eclipse projects from Svn, therefore the project set-up phase is not put into practice.

Concerning the basic skills we observed one important point: even though most students are able to perform correct Svn operations, they tend to apply a minimalistic workflow. Students are encouraged to perform a commit after completing each requirement section, nevertheless most of them tend to perform fewer commits, just the barely minimum to abide by the exam rules: a commit at the end of the exam session and a commit after the session.

Given an assignment whose requirements contain r sections, the recommended process entails at least $r + 1$ commits: one for each requirement section plus one from home after the exam. This is a very simple, though approximate, criterion to identify compliant students.

We analyzed the number of commits performed by the students on their exam repositories. Out of 1008 repositories – corresponding to the bookings – we found that 25% of them contained only the initially project and no student commit. These are untouched projects: students that either booked the exam but did not show up or decided to quit during the exam.

Excluding the untouched projects, the distribution of the number of commits for the students who actually attended the exams is shown in Figure 3.

Table 2: Process compliant students

Session	Students	Dropout	Compliant
June 2017	334	7.5%	44.3%
July 2017	258	6.6%	41.5%
Sept 2017	101	15.8%	22.8%
Jan 2018	63	0.0%	23.8%
All	756	7.7%	38.8%

There is a small percentage of students (7.7%) who performed just 1 commit, i.e. they committed a version in the lab during the exam but did not completed their programs at home; they are the exam dropouts. A larger share of the students (92.3%) performed at least two commits, i.e. one in the lab and one from home.

Table 2 reports, for each exam session, the number of touched repositories and the proportion of dropouts and compliant students. We observe that overall 39% of students complied with the recommended process. The first two exam sessions – closely following the end of the course – exhibit a higher compliance, 44% and 41% respectively.

4.2 Issues

The first instance of the course using the infrastructure described above was given in a.y. 2016/17. The set-up was used both during the course (from March to June 2017) and for the exam sessions. We managed four exam sessions (June, July and September 2017, and January 2018) for a total of 629 exams graded. Given the huge number of students we encountered several problems.

We summarize here the main issues that emerged during and after the exam sessions:

- Several students after checking-out realized that Eclipse did not provide editing support (e.g. code completion). This is typically due to the fact they did check-out the whole repository and not just the folder containing the Eclipse project; as a consequence, Eclipse is not able to recognize the folder as a Java project and thus cannot provide all Java-related supporting features.
- After the exam some students got a test report showing many failures they could not find in their projects. The cause for this lies in a late commit, i.e. a commit performed after the exam deadline.
- After the exam, some students got no test reports because the projects submitted contained errors that prevented a successful compilation. Despite the invitation issued 10 minutes before the end of the exam, several students continued to work on the code rather than checking the code for errors.
- Sometimes students get compilation errors they were not able to see in the lab within their IDE. In our experience this is due to a few causes:
  - the Eclipse uses its own (incremental) compiler that in a few cases – e.g. type inference for generic types

Table 1: Cognitive level and specific capabilities addressed for the key SE practices.

Level	Testing	Configuration Management	IDE
Remember	**JUnit framework elements**	**Svn operation**	**Eclipse features**
Understand	**Semantics of test methods and assert statements**	**Semantics of commands**	**Main tasks (e.g. compile, run, etc.)**
Apply	**Execute test suite**	**Perform *check-out* and *commit***	**Develop and run**
Analyze	**Understand test results**	*Understand outcome of operations*	**Understand error messages**
Evaluate	**Identify failure causes**	Identify conflict causes	**Identify defects or problems**
Create	Write tests	Merge conflicts	Set-up a project

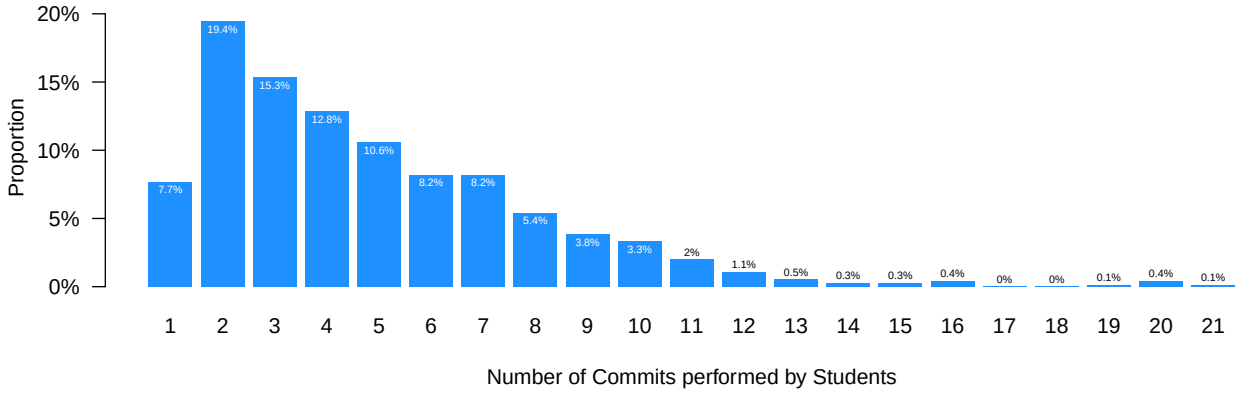

Figure 3: Distribution of student commits.

  - behaves differently from the Oracle JDK compiler we use to compile the project before testing;
  - the Eclipse IDE, when suggesting imports in case of undefined classes or interfaces, usually provides a list of all compatible elements, e.g. for `Collections` it includes of course the `java.util.Collections` class as well as, e.g. `com.sun.xml.internal.ws.policy.privateutil.PolicyUtils.Collections`. The latter class is usually not present in a clean JDK installation and the corresponding `import` is marked as an error during compilation.
- During the first exam session, the students in a lab were not able to connect to the Svn repository. This problem was caused by a misconfiguration in the web proxy and firewall in just one lab.

As a side note, we also encountered some weird issues unrelated to the topics covered in the course. A few students in every exam session typically call for help because suddenly the editor in Eclipse is overwriting their code instead of inserting new characters: this is due to the fact that the students inadvertently pushed the *Ins* button on the keyboard thus switching from insert to overwrite mode. We speculate that such *magic Ins key* problem is due to some students being used to small factor laptop keyboards that do not have a dedicated *Ins* key.

Another issue that emerged while discussing with colleages is the suitability of the Eclipse IDE. In the last year the Eclipse market share[7] (40%) appears to be shrinking in favor of IntelliJ IDEA (46%), which according to colleagues provide a more modern and usable environment.

4.3 Lessons learned

We collected a number of critical issues that we intend to overcome in the next version of the course.

Students are not able to use the basic tools: this is particularly true for Subversion (as reported above) but sometimes it happens they are not familiar with Eclipse or even with the PCs available in the university lab. The lesson we learned is that the countermeasure is to force or provide incentives for the students to get familiar with the tools *before* sustaining the exam. Currently the assignments proposed to the students during the course are not mandatory. A possible mitigation to this problem may be to give additional points in the final grade if the students complete a specific assignment that requires basic skills (e.g. Subversion).

[7] http://www.baeldung.com/java-in-2017

The development environment might differ in part from the testing environment: this is typically due to the compiler (Eclipse own compiler vs. JDK javac), the *classpath* (Eclipse Java project vs. clean JDK), or the operating systems (Windows in the lab vs. Ubuntu for the test server). The consequences of this issue can be significantly reduced by implementing a simplified *Continuous Integration* [7] infrastructure. Every commit goes through compilation and testing and the results are reported back to the students. Such a feedback would enable the students to understand what the problem is in the testing environment.

Students assume they can work in a new environment just because they used a similar one: several students – because of the crowded lecture rooms and labs, the availability of video recorded lectures, and the possibility of performing assignments on their own PCs – tend not to attend all lectures and labs. As a consequence, the day of the exam turns out to be the first time they use the lab equipment. The (presumed) tech savyness and confidence of Millennials apparently bring them to overestimate their knowledge.

However, forcing the students to work on their assignments in the lab, would restrict their freedom, and possibly overload both the facility and the teaching assistants. It is important to make sure the environment the students re-create on their machines is as close as possible to the lab environment. This can be achieved by defining very well the reference environment – IDE and JDK version – as well as ensuring the latest version – the one that the students will download most likely – is installed in the lab too.

Scalable and reliable automation requires a lot of effort for the infrastructure: even if the three cornerstone technologies are quite sound and mature, their usage in the course is peculiar and requires dedicated workflows to be designed as well as a suitable infrastructure to be developed. For this course, during several years, over 10KLOC of code were written mostly in Java but also in Python, Bash shell and Html. The recommendation is to use existing tools as far as possible but also to be prepared for a large effort in infrastructure development.

Whenever you rely on a server, never underestimate the network: we performed tests in two (out of six) labs used for the exam, but not in the one that turned out to have the issue. The recommendation is of course that extensive testing must be performed in the field.

5 CONCLUSIONS

This paper presented a report on the experience in integrating three key Software Engineering practices – automated testing, configuration management, and integrated development environment – into a large OOP course. The key practices play a twofold role: first, they are instrumental to achieve a set of educational goals, second, they are the cornerstones of the infrastructure supporting assignment management both during the course and at the exams.

The resources required to run the course consist in a linux server hosting the Subversion repositories, the scripts, and runnning the test correction procedure. In addition labs large enough are required with PCs hosting the Eclipse IDE. All the required software is open-source and the additional custom software can be provided upon request.

The course, as reported, has been run once in a.y. 2016/17, although it builds on almost 15 years of experience. The anonymous student satisfaction questionnaires resulted in 90% students being overall satisfied for the educational part. The global satisfaction level – also including the logistics – is at 83%, mainly due to the crowded classes and labs.

For the next edition of the course we plan to put into practice the lessons learned, the most important being the introduction of a light-weight continuous integration feature.

Moreover for future editions we will have to consider a possible evolution of the adopted technologies (e.g. IDE and configuration mangement), taking into account both the ease of use and the popularity.

REFERENCES

[1] ACM/IEEE-CS Joint Task Force on Computing Curricula. 2013. *Computer Science Curricula 2013*. Technical Report. ACM Press and IEEE Computer Society Press.

[2] Lorin W Anderson, David R Krathwohl, P Airasian, K Cruikshank, R Mayer, P Pintrich, James Raths, and M Wittrock. 2001. *A taxonomy for learning, teaching and assessing: A revision of Bloom's taxonomy*. Longman.

[3] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. 2001. Manifesto for agile software development. (2001).

[4] Kent Beck and Erich Gamma. 1998. Test infected: Programmers love writing tests. *Java Report* 3, 7 (1998), 37–50.

[5] Benjamin S Bloom et al. 1956. Taxonomy of educational objectives. Vol. 1: Cognitive domain. *New York: McKay* (1956), 20–24.

[6] Ben Collins-Sussman, Brian Fitzpatrick, and Michael Pilato. 2004. *Version control with subversion*. " O'Reilly Media, Inc.".

[7] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.

[8] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[10] IEEE-CS. 2014. *Software Engineering Competency Model*. Technical Report. IEEE Computer Society Press.

[11] T. M. Khoshgoftaar, E.B. Allen, N. Goel, A. Nandi, and J. McMullan. 1996. Detection of Software Modules with high Debug Code Churn in a very large Legacy System. In *Proceedings of International Symposium on Software Reliability Engineering*. 364–371.

[12] Karen K. Myers and Kamyab Sadaghiani. 2010. Millennials in the Workplace: A Communication Perspective on Millennials' Organizational Relationships and Performance. *Journal of Business and Psychology* 25, 2 (01 Jun 2010), 225–238. https://doi.org/10.1007/s10869-010-9172-7

[13] James Rumbaugh, Ivar Jacobson, and Grady Booch. 2004. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education.

[14] Per Runeson. 2006. A survey of unit testing practices. *IEEE software* 23, 4 (2006), 22–29.

[15] Travis Swicegood. 2008. *Pragmatic version control using Git*. Pragmatic Bookshelf.

[16] Marco Torchiano and Maurizio Morisio. 2009. A Fully Automatic Approach to the Assessment of Programming Assignments. *International Journal of Engineering Education* 24 (4) (2009), 814–829.