A Case-based Approach for introducing Testing Tools and Principles

Frédéric Dadeau
Univ. Bourgogne Franche-Comté
FEMTO-ST Institute/DISC, CNRS
Besançon, France
frederic.dadeau@femto-st.fr

Jean-Philippe Gros
Univ. Bourgogne Franche-Comté
FEMTO-ST Institute/DISC, CNRS
Besançon, France
jean-philippe.gros@femto-st.fr

Fabien Peureux
Univ. Bourgogne Franche-Comté
FEMTO-ST Institute/DISC, CNRS
Besançon, France
fabien.peureux@femto-st.fr

***Abstract*—We present, in this paper, teaching material and experience report on teaching software testing to 3rd year bachelor's degree students. Our approach covers a wide range of techniques from unit testing, to functional testing, through test-first design and black-box testing. To motivate students, we rely on the use of a running case study, that is developed and tested all the way through. This paper presents both teaching material and experience report based on the feedback that we get from the students, and the observations we made regarding the discovery of the software testing aspects.**

***Index Terms*—Case-based Learning, Unit testing, Test Driven Development, Integration, Functional Testing, Agile Approach**

I. Introduction

Testing is one of the most, if not the most widely-spread technique that can be used to ensure software quality in the industry. Over the years, testing has become a first class citizen in software development processes and teams [16]. The recent evolution of software development teams into agile teams and the associated increasing tool support, notably continuous integration, led to an evolution of the practices, about which students have to be educated.

As such, teaching software testing has become a key issue in computer science curricula [14]. In order to have students ready for working into software companies or simply to teach them that software quality is what they will be paid for, it is of primary importance to teach them software testing tools and practices, if possible, in a dedicated course.

The course that we describe here is a third year bachelor course, named *Tools for software development* which focuses on various high-level and low-level aspects of software development: source control software, coding standards, agile methods (with a focus on Scrum), and a large part dedicated to testing. This latter is covered by lectures and tutorials on structural testing, and practical sessions dedicated to a concrete usage of the various tools that students will likely have to use once they will be working as professional software developers.

We present, in this paper, the teaching approach that we have implemented in our university, in the computer science curriculum, to introduce software testing to students. We will mainly focus on the practical sessions, for which we rely on a case study that is:

- small enough for the students to master the software development part, so that they can focus on the testing part of the course,
- realistic, to get students involved on a subject to which they can relate and which will motivate them, and
- tailored to cover all the aspects of software testing, from the discovery of the unit testing drivers until continuous integration and functional testing.

Especially, we intend to cover the following aspects of the software testing activities: unit testing, integration testing, functional testing, and practices: source control, continuous integration, test-driven development.

The contributions of this paper are:

- teaching material used for two years on introducing software testing tools and practices to 3rd year (bachelor) students using a realistic case study,
- teaching material used for several years to introduce agile approaches in a playful way, and
- lessons learned, on both sides, on this experience.

The paper is organized as follows. Section II presents the context of the course, the background of the students, and the teaching objectives. Section III describes the teaching material that we propose, tailored to this year's (2019-2020) case study. Section IV describes how we use testing learning to introduce agile approaches which most frequently promote testing practices. Section V provides a complementary feedback from the students point of view and from the teaching point of view. Finally, section VI concludes and presents the future improvements that we plan to implement.

II. Teaching Testing at the University

At the university of Franche-Comté, the computer science curriculum consists in a Bachelor's degree (in 3 years) that can be completed by a Master's degree (in 2 additional years). The curriculum focuses on software development, with two specialties focused on software engineering, and distributed systems, the two historical research activities of the supporting research laboratory, the Computer Science Department of the FEMTO-ST Institute.

978-1-7281-1075-2/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSTW50294.2020.00077

A. Student's Background

In our university, the students start their computer science curriculum with a general semester in science and technology, which includes general courses on mathematics, physics, computer science, electronics, and mechanics. From the 2nd semester, they then specialize into computer science and mathematics. The two subsequent years of the bachelor's degree are then mostly dedicated to computer science, in which students discover a broad set of concepts (algorithmics, complexity, logics, modelling, design, etc.), programming language paradigms (object-oriented, functional, shell, etc.) and environments (operating systems, networks, web, databases, etc.). The Bachelor's degree count about 1.500 hours of teaching, which is the average in France, and represents, as every bachelor's degree, 180 ECTS (European Credit Transfer Scale).

When they start the classes, the students have a good knowledge and practice of Java, since it is the language they learn in first year, and which is also the support of the object-oriented programming class in second year. Notice that, until now, there is no computer science teaching in high school. In general, students present good programming skills in various paradigms: object-oriented, system, web, etc.

At this step, the students already know testing because some of practical courses, such as algorithmic, use test suites to evaluate their work. In their mind, tests are something positive and useful, but they do not necessarily see how to do it. In addition, they sometimes separate the activity of testing from the activity of software development. In general, students add some tests after the project is finished instead of testing regularly. Most of the time, they use manual tests, and they do not look for automation in the test execution. The goal of this course is to reshape their opinion on this subject.

B. Learning Objectives and Outcomes

The syllabus presents the learning objectives as follows, we only present here the objectives related to testing.

At the end of the semester, the student will be able to:

- *explain the principles of software testing,*
- *cite and describe the main code coverage criteria,*
- *write and execute unit tests in an xUnit environment,*
- *automatically assign a test verdict using assertions,*
- *evaluate the quality of the tests using code coverage or with a mutational analysis,*
- *develop a piece of software using a continuous integration environment.*

To achieve these objectives, we propose to rely on case-based learning in which a single case study is used as a running example to illustrate the various aspects and concepts of software testing.

C. Organization and Technical Choices

The course has a standard schedule for French universities, meaning that it counts around 55 hours in face-to-face sessions augmented with personal work from the students. This motivates to use freely-available tools, so that student can work at home. The teaching staff is composed of two associate professors, one professional software developer (who presents the source control software and coding standards – which is not the focus of this paper), and one PhD students who teaches some of the practical sessions.

Lectures represent 15 hours and are divided into three categories, one related to testing principles, with a focus on structural testing [9], one related to collaborative work such as source control and coding standards, and one related to software engineering in general, and agile methods [10], in particular.

The first two points are mainly covered by the lectures and tutorials. Most of the practical sessions related to white-box testing are focused on introducing software testing in practice in order to have students ready, at the end of the semester, to master the main tools that can be used at each stage of the software development process. This paper is focused on these particular sessions.

Our approach relies on the use of a simple-but-realistic case study that will motivate the students. The idea is to develop the different pieces of the case study (namely the different classes) and apply the appropriate testing tool, or technique, to validate it.

As we do not want students to be stuck by a technical issue in programming, we have decided to select Java as the language in which development will be made. Indeed, as explained earlier, Java is the first language that the students have learned and it is the one that was the most used during their curriculum. Besides, the tooling related to Java is quite extensive, and free integrated development environment dedicated to this language exist (IDEA IntelliJ, Eclipse, NetBeans notably).

We made the choice not to focus the lessons on writing tests only, as we believe it is important to keep both programming and testing, so that (1) testing is not considered as an activity that is aside from coding, (2) testing has to become a reflex when starting programming.

The techniques and tools that we aim to teach to our students are addressed through successive sessions that we now detail.

III. Teaching Material "Tools for Programming"

Each year, we implement our educational program on a realistic example. Last year, we focused on a supermarket scanner. This year, we focused on a Japanese subway case study. We first describe this case study before detailing the different sessions that use this case study.

A. Chikatetsu, a case study on the Japanese subway

In Japan, the subway (or *chikatetsu* – literaly: "underground railroad") is an interesting system for a case study. The fare of the travel is determined by the distance that the traveler is supposed to travel: the farther he goes, the higher the fare. If the ticket that is used is not loaded with enough money, fare adjustment machines can be used to add some credits to the

tickets, in order to open the exit barrier. Notice that jumping over the barrier would simply not be acceptable.

We took the example of the Sapporo subway network, which is quite representative of such a network, composed of three lines, and around 50 stations.

The case study consists in implementing such a system. The design of the system is already made and is part of the informal specification. The set of classes that compose the system, along with their API, and a description of their operations is provided. Figure 1 shows the class diagram that is used to illustrate the description. Similarly to the case studies that we have already used, the *chikatetsu* system contains 4 main entities:

- the `Station` class is used to represent the station and their associated lines that cross the station. Each station is located a given kilometer point for a given line.
- the `Network` class represents the set of stations that compose the subway network. Some properties are expected from the network, for example: it should contain only stations that are all linked to each other, and all station names should be different.
- the `Ticket` class represents transport tickets which may be intended either for adults or for children, and which are initialized with a certain amount of money, which can later be modified.
- the `Barrier` class represents barriers that can be opened, to enter or to exit the station, by introducing a ticket in it.

The data model of the different classes is left unspecified and students are free to implement the methods as they wish. Nevertheless, we expect the methods to comply to the signature that is specified, and to conform to the behavior that is described in the informal specification that is provided.

Similarly to a real-world exercise, we put them into the situation where a developer has to implement a set of features that have to be integrated later on with others pieces of software. It is thus of utmost importance that the students respect the methods' signatures (name and parameters) and do not modify them (spoiler: they will modify them).

We now detail how we make use of the different classes that have to be implemented as a support for different exercises aiming to discover and apply testing tools and methodologies.

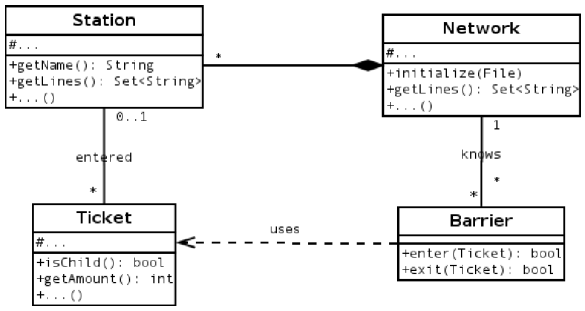

Fig. 1. The class diagram of the case study

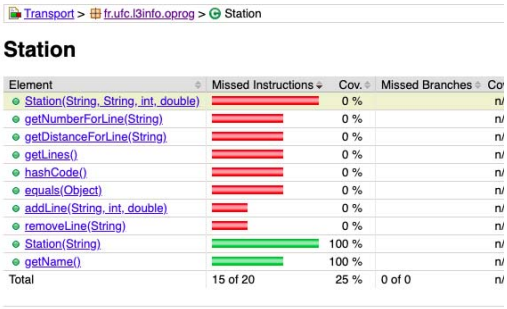

Fig. 2. An example of the JaCoCo HTML report

B. Educational Sequence

We describe here the different sessions that we carry out with the students. For each, we describe the theme and the objective of the session, and we present it on the running example. Each session is a face-to-face session of three hours, in computer rooms. However, most of the session require students additional time to finish the work at home.

1) Unit Testing: The first session is dedicated to discover the basic tools used for unit testing.

This session is also the setup of the development environment. We rely on the IntelliJ [4] IDE (Integrated Development Environment), coupled with a Maven [1] architecture, so that the source files can be organized in a standard manner.

a) Objective: The goal of the session is to write the code of a simple Java class, write JUnit [6] tests to validate it, and finally use the JaCoCo [5] code coverage tool to evaluate the "completeness" of the test suite. We illustrate the generation of code coverage reports, and show them the usefulness of the coverage report integrated into the IDE. We expect the students to write test cases with appropriate observations (assertions) in order to establish a test verdict. At this stage, this latter is mainly based on using getters that will retrieve some of the internal values of a subset of the data model.

b) On the case study: The students are asked to develop the `Station` class which is the most basic unit of the system. In general, we try, when possible, to add one method that requires some algorithmic issues, such as a checksum, so that students may rely on the code coverage criteria that they have seen during the lectures and tutorials. As the case study is not very complex, students should easily reach a 100% code and branch coverage, relying on the JaCoCo tool, whose reports can be shown in Fig. 2.

2) Test-Driven Development: The second session is dedicated to improving the knowledge of unit testing, and discovering Test-Driven Development (TDD) [11].

a) Objective: The goal of this session is to write a class that reads from a file to instantiate a (sort of) database. This kind of exercise is a good support for Test Driven Development. Indeed, the specification describes a file format (generally, a Comma-Separated Values file), which is supposed to describe a set of instances of the class developed during the

```
...
Namboku, 5, Kita-Jûni-Jô, 3.9
Namboku, 6, Sapporo, 4.9
Namboku, 7, Ôdôri, 5.5
Namboku, 8, Susukino, 6.1
...
Tôzai, 8, Nishi-Jûitchôme, 7.5
Tôzai, 9, Ôdôri, 8.5
Tôzai, 10, Bus Center-Mae, 9.3
...
```

Fig. 3. Sample of the Sapporo network CSV file

previous session. In this context, the specification provides the passing cases, and all the possible error formats that have to be captured by the code and that will raise exceptions. We provide a first example file, which contains a valid data set, which can easily be modified by the students to introduce invalid records. It thus makes full sense to, first, design the test file that will be used by a test case, and then design the test case that will load this file, before implementing the code that will read the file and treat the possible error that is supposed to be contained in the test file. As errors are signaled by exceptions, this session shows how to establish a test verdict based on the presence or absence of an exception.

b) On the case study.: In this session, the students have to develop the `Network` class which initializes a subway network by reading a set of data that will be used to instantiate the stations that compose the network. Figure 3 shows an excerpt of the file which describes the network of Sapporo. For each line (in the CSV file), we have the name of the subway line, the number of the station for the line, the name of the station, and finally the kilometer position.

Error cases can be of two kinds. Format errors represent all errors that are related to the CSV file format: empty fields, incorrectly-typed fields, wrong number of fields, etc. Invalid network errors relate to the structure of the network itself: all stations should be numbered with an ascending numbering, for a given line there are no gaps in the numbering, all kilometer positions are ascending as the station number increases, no station is isolated on the network, etc. Both kind of errors are signaled by dedicated exceptions.

3) Specification-Based Testing: In order to show students that testing can also be a standalone activity, in which they may have to validate someone else's code, this session is dedicated to the writing of test cases only, based on the specification of the class.

a) Objectives: The goals of the session is to force students to read, analyze and exploit the specification for writing test cases. The idea is to see the code as a black box, which is provided as a compiled Java class (one for each student). The objective is to find out if the class contains an error, and to explain why there is an error. We do not expect the students to locate the error, but rather refer to the specification and explain why the assertions they wrote have failed.

b) On the case study: This session is dedicated to the `Ticket` class which has a simple lifecycle that can be described using a state machine, as depicted in Fig 4. The ticket is first issued when it has been initialized with a given amount of money. When passing through an entry gate, the ticket records the station in which the gate is located. When passing an exit gate, or if the ticket is reused, for re-entering a station, it becomes invalidated. It is possible to question the current amount of money on the ticket, and the station name, but it is impossible to know the current state of the ticket, which forces students to design more complex observations, in which one must attempt to perform a given action to check if it is authorized or not. This session is the occasion to make the students aware of situation in which they do not control the code that is written and have to deal with the (limited) observation points provided by the system. Each student receives an implementation of the class which may (but not necessarily) contain an error.

4) Mocks and Spies: As unit testing is performed, it is, by essence, a key issue to test classes in isolation, by mocking other classes [17]. To illustrate this to the students, we rely on two concepts, namely mocks and spies. A *mock* is a fake object that is setup to behave as needed for a considered unit test case. A *spy* is similar to a mock but it is used to fake a part of an existing object. This may notably be used to replace some methods of the object under test.

a) Objectives: The goal of the session is to master the mocking tool and its principles to write test cases in isolation. To achieve that, we introduce the Mockito library [8] that is well-known in the Java community and makes it possible to define both mocks and spies.

b) On the case study: This session consists in designing and testing the `Barrier` class which is central in the system. Indeed, this class is related to the station. A barrier is located in a given station. It is also related to the network, as we choose to delegate the computation of the fares to the barrier. Finally, the barrier interacts with a ticket, as the barrier may be used to enter the station (in this case, the barrier should open with an issued ticket and it should record the station on it), or to exit the station (in this case, the barrier should open if the ticket has enough money compared to the fare that is computed based on the shortest path between the entry station and the current station). The other classes have to be mocked, so that the methods of the barrier (one for entering and one for exiting) can be invoked in the different interesting situations (normal usage, re-entry with an already used ticket, etc.)

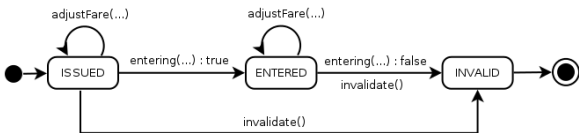

Fig. 4. The state machine associated to the `Ticket` class

5) Integration Testing: After having tested classes in isolation, the next step is to test the correct integration of classes.

a) Objectives: The goal of this session is to make students understand the difference between unit and integration tests. This session aims to write tests that use the system classes in integration. The idea is to remove the mocks that were introduced previously and add new test preambles (to set up the different objects) and assertions (to evaluate if the effects of methods of the previously-mocked objects –which could not be observed– have correctly been applied).

b) On the case study: The session focuses, once again, on testing the `Barrier` class. The students start from their previous test cases, but they have to design a full network, and instantiate tickets with the appropriate parameters and put them in the right state. Thus, students have to check that the effects of the methods that were invoked are effectively applied and propagated through the system. Especially, they need to make sure that the ticket, which was faked before, has recorded the entry station once the entry barrier has been opened.

6) Continuous Integration and Software Evolution: Continuous Integration (CI for short) consists in a tool-supported environment which combines a source code repository, a build server, and test suites, so as to continuously build the software on the repository and run the tests each time a developer commits its code.

a) Objective: The goal of this session is to introduce the principles of continuous integration, through the setup of a CI environment, and the collaborative development of a set of additional features to the current version of the application.

b) On the case study: The first part of the session is dedicated to the set up of the continuous integration environment. We have decided to work with the Gitlab [3] source control environment, which provides the possibilities of scripting the execution of test cases. Gitlab integrates well with Maven builds, and makes it possible to easily build the application and run the tests in a Docker container on the server side. In addition, Gitlab CI keeps track of the different builds, notifies the developers in case of failures, and makes it possible to retrieve the last build and get the console log of the build and the test execution to detect where errors are located.

In addition, we ask the students to implement an evolution of the software. Due to a lack of time, this year, we did not implement this part, but we had planned to ask the students to add a limitation in the ticket validity (e.g. the traveler has one hour to exit the station before invalidating the ticket), and to implement the use of a travel card to pass the barrier.

7) Functional Testing: Once a whole system has been designed, it is mandatory to perform functional tests, at the system level. As the system is relatively small, integration tests can be seen as very close to functional tests as both may involve all the classes of the system. To tackle this misconception, we propose to the students that they test the system in a web-based simulator.

a) Objective: The goal of this last session is to both teach the difference between integration testing and functional testing, and to make students discover test execution robots, namely Selenium IDE and Selenium Driver [7]. To achieve that, we design, each year for each considered case study, a web-based application, of the form of a simple video game that simulates the real-world application on which the students have worked all the semester.

b) On the case study: We have designed a web-based video game in which the player can manage characters that enter a subway station, buy tickets, take trains to reach other stations, adjust the fare of the ticket, and exit the station. Figure 5 displays the main screen of the application in which it is possible to handle a character and have it buy tickets (see Fig. 7), pass through the barriers, and reach a train to go to another station (see Fig. 6). The application is available at: https://fdadeau.github.io/chikatetsu

The students have to discover the Selenium IDE plug-in which is installed on a web browser, and can be used to record test sequences and replay them in the context of regression testing [15]. Then, their objective is to write Java unit tests that drives the browser using the Selenium Driver library that has the ability to launch a web browser and input commands in it. At the same time, they both practice functional testing, and discover the application they know in real-life situation. Notably, they can discover that some behaviors that were possible in the code of the application is not necessarily possible in practice. For example: tickets are invalidated once they are used to pass an exit barrier, and an invalidated ticket can not be used to pass a barrier. An interesting test case is to invalidate a ticket before using it to try to enter a station. However, in practice, it is impossible to retrieve a ticket after having passed the exit barrier, as the barrier "keeps" the tickets when exiting.

C. Evaluations

We describe here how the practical sessions are evaluated. In order to better motivate the students we ask, before the next session, to upload their code and their associated tests to evaluate them. The evaluation usually consists in the following three steps.

a) Running a reference test suite on students code: The goal of this step is to evaluate if the students have correctly implemented the requirements that are described in the specifications. To achieve that, a full test suite, designed by the teaching team, is run on the students code. The objective is, for their code, to pass all the tests.

b) Running students tests on a reference implementation: The goal of this step is to evaluate if the students have correctly tested their implementation. By detecting test cases that fail on the reference implementation we are able to detect false positives, typically tests that are run correctly on their code, but which are not correctly designed, notably due to an erroneous interpretation of the informal specification.

In addition, we weight the score obtained at this step by the number of tests that were designed. The idea is to force

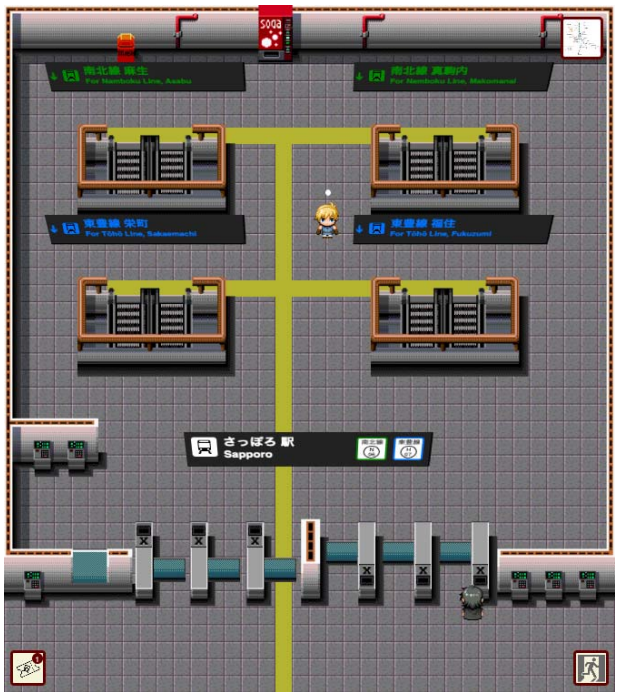

Fig. 5. Station playground

Fig. 6. Travel inside the subway

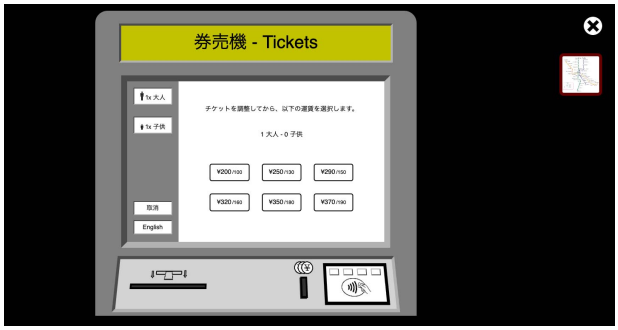

Fig. 7. Ticket vending machine interface

the students to have shorter tests, each focused on a specific feature, rather than long tests, that check several features at once. This is considered as a good practice for locating errors and debugging code.

c) Running students tests on mutants of the reference implementation: The goal of this step is to evaluate the relevance and the completeness of the students test suites. Before running their test, we first remove those which failed on the reference implementation. Once the test suite passes on the implementation, we run it on a set of mutants.

The mutants are manually designed, based on the reference implementation, and represent erroneous implementations of the informal specification. To ensure that the mutants are not equivalent, we make sure that our reference test suite is able to kill all the mutants. This is also, as a side-benefit, a way to strengthen the reference test suite.

After each session, we take some time to run automated scripts that perform the evaluation on the students code. A feedback message is sent to the students to inform them of their results and the common mistakes that they have done, and how to improve their code/tests.

IV. Connection to Agile Practices

The last part of the course aims to put testing into perspective and evaluate its strengths and weaknesses in an operational development context.

This objective is addressed by introducing software agile development and underlying the benefits expected by the intensive use of the test, as promoted by the agile approaches. This topic gives rise to a more general introduction to software engineering methods and practices. This lesson firstly consists in presenting the motivations of the Software Engineering and introducing a brief history of methods, i.e. from waterfall cycle to current agile approaches. Secondly, it focuses more specifically on the motivations and global principles of agility as described in the Agile Manifesto [12]. Finally, the agile development method *Scrum* [19] is used to explain agile project management and team organization, while the 13 practices defined by the *eXtreme Programming (XP)* [13] method are presented to illustrate development practices. Obviously, a particular attention is given to the practices studied by the students during the previous sessions of the course, namely acceptance testing, unit testing and continuous integration that automates and makes systematic the execution of tests.

In order to make students aware of agile principles in a playful way, a practical teamwork, inspired from the "Marshmallow" challenge, is organized over a dedicated 3 hour session. This playful game basically consists, by team of 4 students, to build a free-standing Eiffel tower only using sticks of spaghetti, adhesive tape and one marshmallow, which must be placed on the top of the structure. Figure 8 depicts some of the resulting Eiffel Towers that were built. Teams have to apply an agile approach to build their tower. Concretely, it consists in conducting 5 to 7 sprints including the sprint planning (2 minutes), the building period (5 minutes), the sprint review and retrospective (2 minutes). Previously, each team populates during a 30 minute release planning the product backlog by identifying and defining the building tasks they decide to do. After the game is finished, the rest of the lesson, about 1 hour, is dedicated to a general debriefing discussion.

This game is intentionally not linked to computer science

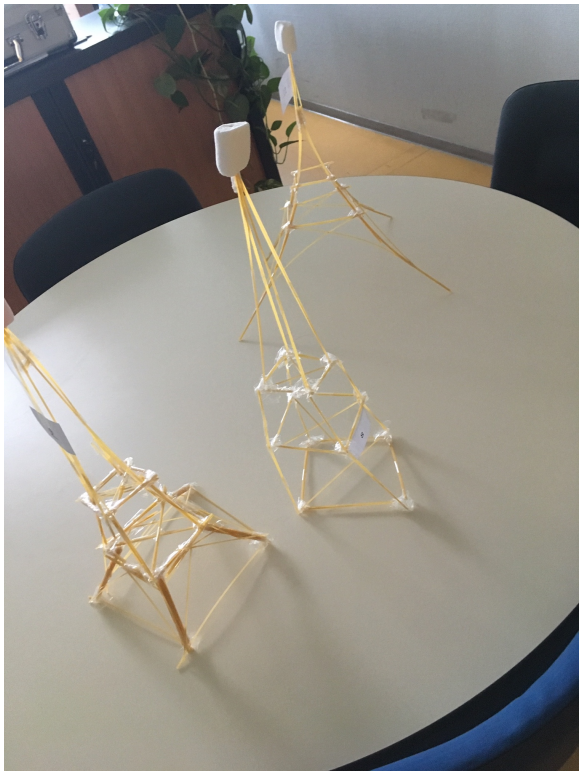

Fig. 8. Results of the Eiffel Tower Game

activities. This choice is motivated by the desire to focus the attention of the students on agile project management by a short experience, and not to disrupt this learning goal with IT techniques. However, reconciling agile and testing issues is performed during the general debriefing. To reach this goal, one the one hand, the debriefing consists for each student to give feedback about the problem his team was faced and how the team succeeded to mitigate them or not, etc. On the other hand, the teacher takes notes during the game and relies on it during the debriefing to make students react or question about decisions or attitudes. They are also used to illustrate what agility promotes or tends to present as a good practice. In this context, a lot of situations can be used to point out the characteristics of testing.

For example, one usually hear, many times in many groups, the exclamation "Don't touch it, you will break it!". This typical comment allows the teacher to highlight, within software development, the role of testing to change this sentence into "Do not be afraid, you can touch it to make it better.". Writing test cases, structural and functional as well, allows developers to validate the code they are producing, but it also makes it possible to gradually build up a battery of tests that can be used to detect regression. Moreover, the continuous integration process enables to automate and systematize their execution (no human intervention is needed), and therefore multiplies their benefits since they are systematically executed whenever necessary. In this context, developers are informed at the soonest that a regression occurs and they can immediately react to fix it. In the worst case, within such a continuous integration process, we can also explain how the source code version control system makes it possible to rollback without damage to the last stable version of the source code.

In addition to these benefits, the teacher can also alert the students that it is necessary to master the number of test cases. Notably, since test case scripts are code, they necessarily require maintenance effort. Time execution can also become a problem. The given message is clearly about the benefits of testing and its capacity to increase the quality of the developed solution. But we also underline that this leverage capacity has a price to pay, and this price has to be managed.

To sum up, the main lesson about testing we want to convey to students during the debriefing is that testing is not only a set of techniques: these essential practices also require a dedicated expertise to be used in a relevant and efficient way. Nowadays, as promoted by agility, developers need to master this skill.

V. Lessons Learned

We present here the lessons we have learned from this two years experience. We start by classical (or frequent) errors made by the students, and we then present the feedback that was given by the students.

A. Observations on the Students Work

We observed two main issues during the students work, which caused some troubles with the evaluation of their code and tests.

First, we had to admit that students do not read the specifications. Indeed, we left some pieces of the software behavior's unspecified, on purpose, without any specific indication on the behavior that was expected. We expected students to ask us questions about them. Surprisingly, students are very reluctant to ask such essential questions, as they seem to be used to have a full description of the application available and thus, they do not seem to conceive that the specification can be erroneous or even incomplete. At the same time, many consider that if something is not specified, they are free to implement it as they like. Thus, they regularly make implementation choices, which, sometimes, do not correspond to the expectations. This experience was the occasion for them to learn that the client does not always fully describe what he wants, and that, in this case, inventing or choosing for him is not an option.

Second, we discovered that sometimes, students are tempted to modify the classes, either by adding additional methods in the classes, or by directly changing the API (i.e., the operations signatures) of the classes they had to develop, just because they thought they "would not have done it like this". In these cases, it is simply impossible to evaluate their code (as our reference test cases do not compile with the modified signatures) or their tests (as our reference implementation does not comply with the invocations of the API that is performed in the tests). These experiences were the occasion for the students to learn that once a development team has agreed on a design, this design can not be changed by an individual without the consent of the rest of the team, as other developers may rely on this design.

Even though these two negative experiences happened while we expected students to simply follow the instructions, we were grateful that we had the opportunity to teach them that they made a mistake, and why they made a mistake. We believe that the use of such case studies, makes it possible to encounter representative issues that a developer may face during the testing phase.

B. Feedback from the Students

We collected the feedback from the students at the end of the semester using an online form.

a) Regarding the content of the course: In order to evaluate the learning outcomes, we asked the students to evaluate their knowledge and confidence in their ability to perform the expected objectives (described in Sect. II-B. A large majority of the students considered that they were "able" or "very able" w.r.t. the learning objectives that were defined.

All students were mainly satisfied by the lesson, and acknowledged that this course is useful for their professional life. However, some of them did not quite understand the purpose of the agile exercise, and did not relate it with software engineering.

b) Regarding the organization of the course: We also learned from the feedback of students these two years. Last year, the supermarket scanner case study was quite dense and a large number of students got lost. So, this year, we decided to simplify the case study to make it more accessible to the students. Thus, the students could more efficiently focus on the different concepts that were taught.

In addition, we also reshaped the content of the sessions w.r.t. last year. Initially, in order to show the interest of IDEs, during the first practical session the students were not using an IDE but only a file editor and a terminal to compile the code. Nowadays, students use IDEs naturally and they saw this preliminary session as a regression, so we decided to remove it to focus more on the testing aspects. This year, we had the possibility to dedicate a full session to black-box testing which was successfully understood by the vast majority of students.

VI. Conclusions and Future Improvements

The course that we presented is integrated in the computer science curriculum of the University of Franche-Comté. It aims to be an active pedagogy class in the sense that students are put in real-world situations, using industry-strength tool, and applied to a realistic case study.

We believe that this course make students gain some maturity. First, because they learn tools and techniques that will help them to improve the quality of the software they develop. Second, and more importantly, because it is one of the first courses in the curriculum in which they are not the designer of the system they have to develop. This places them in a real-life situation in which they do not only develop a code for themselves, but also for, and with, other developers.

The feedback from the students is very positive, as it helps them understand the importance of testing. This course is considered as a good introduction to testing techniques, and methodologies, that will be consolidated during the two subsequent years in the Master's degree:

- In the first year of the Master's degree the students follow a course on compiler construction, coupled with a course on software engineering, which will, again, put them in a real-life situation, this time in larger teams (6-8 developers), in order to practice an agile approach in the context of software development.
- In the second year of the Master's degree, closed to research activities, the students can follow an optional course on model-based testing in which similar case studies can be employed. Notably, the Java implementation can be used as a model, coupled with the ModelJUnit tool [18], in order to predict the expected behavior of the system on the web application simulator.

For the future years, we plan to draw a better relation between the structural coverage criteria seen during the lectures and the tutorial and the practical sessions. Especially, we are looking for an integrated tool that could be used to measure paths coverage to be employed in the first practical sessions, such as CodeCover [2].

References

[1] Apache maven project. https://maven.apache.org/.
[2] Codecover – an open-source glass-box testing tool. http://codecover.org/.
[3] Gitlab – the first single application for the entire devops lifecycle. https://www.gitlab.com/.
[4] Intellij idea. https://www.jetbrains.com/idea/.
[5] Jacoco java code coverage library. https://www.eclemma.org/jacoco/.
[6] Junit. https://junit.org/junit4/.
[7] Selenium automates browsers. that's it! https://selenium.dev/.
[8] Tasty mocking framework for unit tests in java. https://site.mockito.org/.
[9] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.
[10] Sondra Ashmore and Kristin Runyan. *Introduction to Agile Methods*. Addison-Wesley, Upper Saddle River, NJ, 2014.
[11] Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
[12] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, February 2001. http://agilemanifesto.org.
[13] Kent Beck and Andres Cynthia. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2nd edition, 2004. ISBN 978-0-3212-7865-4.
[14] Edward L. Jones. Software testing in the computer science curriculum – a holistic approach. In *Proceedings of the Australasian Conference on Computing Education*, ACSE'00, pages 153–157, New York, NY, USA, 2000. ACM.
[15] H. K. N. Leung and L. White. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance - 1989*, pages 60–69, Oct 1989.
[16] William E. Lewis and W. H. C. Bassetti. *Software Testing and Continuous Quality Improvement, Second Edition*. Auerbach Publications, Boston, MA, USA, 2004.
[17] Tim Mackinnon, Steve Freeman, and Philip Craig. Extreme programming examined. chapter Endo-testing: Unit Testing with Mock Objects, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
[18] Mark Utting. How to design extended finite state machine test models in Java. In *Model-Based Testing for Embedded Systems*, Series on Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 147–170. CRC Press, 2011.
[19] Ken Schwaber. *Agile Project Management With Scrum*. Microsoft Press, 2004. ISBN 073561993X.