Applications of Unit Tests in Computer Science and Software Engineering Education

Atanas Semerdzhiev[0000-0002-7760-1619], Petar Armyanov[0000-0002-4903-8945], Trifon Trifonov[0000-0002-2247-1968] and Kalin Georgiev[0000-0002-6283-1040]

Department of Computer Informatics, Faculty of Mathematics and Informatics, Sofia University "St. Kliment Ohridski", Bulgaria

{asemerdzhiev, parmyanov, triffon, kalin}@fmi.uni-sofia.bg

Abstract. The article discusses the authors' experience with incorporating unit tests in the learning materials and examples used in Computer Science and Software Engineering courses and applying them in the grading process. This approach served several distinct purposes: (1) To teach students how to control and improve the quality of their solutions to training problems; (2) To provide additional means of grading students' submissions; (3) To automate tasks performed by the teaching team; and (4) To improve the quality of learning materials provided to the students. The article describes how unit tests were integrated in the courses led by the authors, what conclusions were reached, and what best practices were established as a result.

Keywords: Education, Unit Test, Sofia University, Computer Science, Software Engineering.

1 Introduction

In the past 5 years, the teaching teams led by the authors of this article, worked towards integrating unit testing into courses they teach. This was done not only as a part of the curriculum (i.e., including unit tests as a topic being taught), but also to improve the quality of the work and to automate certain tasks. To be more precise, unit tests were used: (1) To teach students how to control and improve the quality of their solutions to training problems; (2) To provide additional means of grading students' submissions; (3) To automate tasks performed by the teaching team; and (4) To improve the quality of learning materials provided to the students.

The main factors that motivated this change were:

- the authors' belief that unit tests are an important part of modern software development and should be incorporated as a part of the curriculum;
- feedback from companies in the IT sector (among other prospective employers of the students), which stated that students are lacking knowledge in this area, which is recognized by the companies as highly important;

- feedback from the students, who asked questions about the concept and the mechanics of unit tests and expressed general interest to gain knowledge and skills in the area; and
- internal needs of the teaching teams.

The present study describes how unit tests were integrated in the courses led by the authors, what conclusions were reached, and what best practices were established as a result. The introduction of unit tests as teaching tools has been previously experimented and reported on by other authors as well [1].

2 Use of unit tests

IEEE's Guide to the Software Engineering Body of Knowledge (SWEBOK) version 3.0 describes unit testing as follows: "Unit testing verifies the functioning in isolation of software elements that are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of highly cohesive units. Typically, unit testing occurs with access to the code being tested and with the support of debugging tools. The programmers, who wrote the code typically, but not always, conduct unit testing." [2]. Unit tests have become an industry standard – an indispensable part of software development and are tightly integrated in any modern software development processes, as described, for example, in [3] and [4].

Unit testing practices are traditionally introduced to students as part of a dedicated intermediate or advanced course on software testing. The common perception is that the concept is too complicated for introductory students to grasp and effectively apply. It is clear that there is a certain cognitive cost to be paid when introducing an additional concept at an early stage. We argue that this cost can be offset by applying the concept of unit tests in multiple contexts, thus increasing the gains and the students' confidence in its applicability.

The goal of our study was to examine how unit tests can be introduced into introductory programming courses. Our methodology was to introduce unit testing throughout the entire educational process and demonstrate that it is a multifaceted concept applicable not only to student grading, but as a hinting assistant during an exam, as a validation tool for course and examination materials, and as program analysis tool, among others.

In this section, we describe in more detail how our teams applied unit tests in a variety of educational activities.

2.1 As a part of the curriculum

An established beneficial routine in the teaching of classes in Computer Science and Software Engineering (CS/SE) focused on software development is to introduce students, often by example, to as many best practices as possible. This helps

them, especially during the early learning stages, to develop a habit of discipline in programming, conforming to the contemporary expectations for quality in software development. For the students from the Information Systems program in our faculty, it is even more important to achieve knowledge for different types of testing, including unit testing, as they have courses, related to higher level of software design, modeling and testing in the third and fourth year. In such a way, we are attempting to stimulate and develop behavioral competencies, such as, for example, recognizing development of unit tests as a best practice of a seasoned professional, and being able to explain clearly its advantages, and utilizing unit tests as means for intra-team and inter-team communication of the intended purpose of the code [5]. Accordingly, we address the topic of unit testing at an early stage and aim to develop a culture where the code and the tests are considered as one whole – the software unit is only complete when the unit tests for it are also fully developed.

This introduction can be challenging at the beginning of the undergraduate program, especially in the first semester of the first year. One typical obstacle is that unit-testing frameworks typically rely on advanced mechanisms, which the students have not been introduced to at that stage of their education. For example, a typical C/C++ unit test framework relies on the preprocessor and some basic understanding of linkage, translation units and how to organize a solution into multiple projects (for example, cf. [6,7]). It may be argued that students should simply consider the unit-testing framework as a "black box", whose inner workings they do not need to comprehend. In practice, advanced C/C++ knowledge may be required to utilize fully and properly a unit test framework properly, as well as to be able to resolve non-obvious unit test errors occurring at compilation time or at runtime.

That said, we have found it quite possible to teach students to start using unit tests even in the first semester. Our methodology introduced unit tests in a simplified and controlled manner, which may sacrifice efficiency of the unit tests for the sake of simplicity. A parallel can be drawn with how first semester classes in C++ introduce students to input and output using the `iostream` library. At the very beginning, students do not have knowledge of classes, inheritance, operator overloading, or ADL. However, the abstraction allows usage `std::cout` and `std::cin` for output and input and they learn to do so quite well, despite the fact, they will understand the details of the implementation much later, and many will never become aware of the full implementation. In other words, one can learn *how to use* a well-designed programming construct properly, at least on a certain level, even if they do not yet understand *how it is implemented*.

From our experience, the introduction of unit testing early on has many positive effects. Students learn how to properly design units of code and gain a better understanding of some of the processes and requirements of real-world develop-

ment. They understand the benefits and the importance of program decomposition by observing them in practice. Finally yet importantly, they acquire a certain degree of confidence, because they feel they are gaining real-world skills and that they are growing as IT professionals.

Of course, as in actual software development, the benefits of unit tests come at a cost. Firstly, the teaching team must be prepared to work with unit testing in a training scenario. Obviously, at a minimum they must have a good understanding of the topic themselves, so that they can relate it to their students. The less evident fact is that a significant amount of additional work arises for the team:

- Unit tests developed by the students must be reviewed and appropriate feedback should be provided from the teaching team.
- For each assignment, it should be considered what aspects of unit testing should be included in it (for example should the students develop unit tests, what part of the grade comes from the unit tests, etc.).
- Incorporating unit tests in the standard curriculum is not a straightforward endeavor and the team needs to make a conscious and focus effort to identify how to incorporate them in its daily teaching activities.

The second aspect of the cost of introducing unit tests is that, just as in real-world development, unit test development may consume as much time as the development of a given code unit, or even more. Thus, when assigning tasks to the students, it should be ascertained they have enough time to develop both the code units and the tests. As a result, often the team needs to resort to two tactics – either increase the time limit for assignments, and/or simplify the tasks, so that the development of code units becomes simpler, leaving more time to develop unit tests.

Additionally, tasks given to students need to be carefully selected and thought through. As unit tests themselves become a part of the curriculum, there may be tasks that focus solely on the tests. For example, the students may receive an assignment that requires them to develop one or more units of code, which are not challenging in themselves, however covering them with unit tests might be. Respectively, for such assignments it is the unit tests that are being examined and graded, and not the actual solution. For "hybrid" tasks, where both the solution and the accompanying unit tests are graded, a careful advance consideration needs to be made as to what code units has to be developed and what it will take to cover them with unit tests. Last, but not least, unit tests should not be turned into a religion: exceptions could also be made where appropriate. For example, if a class needs to work with the file system, it may not be easy to subject it to unit tests for students in the first year of their studies. At this stage, they may not yet be able to deal with topics such as mocking, stubs, etc. Therefore, unit tests for the problematic areas may be simplified, omitted, or given as an extra credit for advanced students.

2.2 As an aid for students during exams

In the courses, which are in the focus of this paper, students are often subjected to a particular type of exam. It is of relatively short duration, usually between 1 and 3 hours. At the beginning of the exam, the students receive one or more problems they are required to provide a solution. For example, they may need to create a computer program as a solution to the exam problems. In other situations, students are not required to implement an entire program, but a single class or a single function or a combination of such program units. Each student solves the exam on their own, using a computer with preinstalled IDE, compiler, etc. At the end of the exam, the students submit their work in the form of one or more source files. Submissions are uploaded to our faculty's e-learning system, which is an instance of the popular LMS Moodle [8].

In such exams, we have found it helpful to provide the students with a comprehensive set of unit tests that cover the major use cases of the code units they are required to develop. Students are free to add unit tests of their own, if they see fit (this may be useful if they choose to develop additional code units, apart from the ones mandated by the exam text).

The unit tests are provided simultaneously with the problem statement of the exam. Revealing them in the days before the exam may compromise it: based on the unit tests it is often straightforward to determine what the exam problem is. This may allow the students to prepare their solution in the days before the exam has started, using external help, and only be able to reproduce it verbatim during the actual exam. This defeats its purpose, as the exam checks the ability of the students to solve a problem in a limited period, while the teaching staff can monitor their work and can make sure they do so independently and do not receive aid from third parties.

It is important to note that while the ideal conditions for conducting such an examination is in a controlled environment with a university-provisioned machine on which the student has restricted rights, the authors have often been compelled to work in a less suitable examination environment. As an example, some examinations need to be performed simultaneously for a large number of students during a time slot with limited availability of university-provisioned machines or under the conditions of restricted or unavailable Internet access. One potential solution for such scenarios is to allow students to use their own computers and distribute the unit tests in advance as an encrypted archive. The students are required to download the archive in advance and are provided with the decryption key only at the exam. For increased security, we prefer ZIP archives encrypted with the industry-standard AES-256 algorithm as opposed to the legacy and less secure ZipCrypto algorithm [9]. One drawback is the lack of native support for AES-256 in Windows, macOS, and Linux, which can be overcome by utilizing

the open source 7-zip archiver [10]. For testing purposes, students may be provided with a test encrypted ZIP archive with a known password so that they can test their ability to decrypt successfully such an archive.

The unit tests are provided for several reasons. Firstly, they serve as an aid to students and can help focus their attention to specific corner cases, which they may otherwise omit. In addition, they will know well before the exam is graded whether their solution works correctly or not, based on the outcome from running the tests. Secondly, they help the students work in an environment, which is closer to that in real-world development, where they will have all kinds of tests to help them reduce the risk of introducing bugs in their code. Thirdly, it also aids our goal to help students build a habit of using unit tests and think of them as "assistants" in solving their problem. In this case, the tests are provided by another person (members of the teaching staff) and the students need to use them properly.

Since the exam is of short duration, the teaching staff do not want the students to waste time trying to configure their projects to run the unit tests (unless that activity is a part of the exam, of course). For this reason, an empty template is provided to the students. Such template could even be provided in advance of the exam, as it reveals nothing of substance about the actual examination. The template consists of one or more files, which the students download. It may consist not only of source code files, but also project configuration files that bundle all of the solution together and can also contain an empty "placeholder" file for the unit tests. In the day of the exam, the students simply overwrite it with the actual set of unit tests they are provided.

Depending on the language being used, the template should be organized in a way to make it possible to develop the solution without the unit tests interfering. For example, in a C/C++ project, a Visual Studio solution with two separate projects inside it may be provided, or a `CMakeLists.txt` file with two separate targets, etc.

A variation of this technique is to provide only a subset of all unit tests that would normally be used to cover the entire solution. It can further be extended to two more sub-variations. On the one hand, the students may be required to complete the set of unit tests to provide optimal coverage. In this case, the unit tests they develop are too subject to evaluation. The second option is to make the development of additional unit tests optional. In this case, the students can choose not to invest time in developing unit tests at the higher risk of missing significant errors in their solution.

2.3 As a grading aid

When grading exams, the teaching teams often face several issues summarized below:

- There are repetitive checks that need to be performed on each submission. For example, if the task requires the students to write a function *f*, it may be required to test how *f* behaves when called with certain inputs (does it throw an exception, does it calculate proper results, etc.). As another example: students may be required to implement a class for which the exam text describes its contract. This class may need to be tested in multiple scenarios. If executed manually each time, these checks consume a lot of time.
- The checks need to be performed with all submissions and no checks must be left out to ensure objective and equal grading. In addition, each check must be executed fully, faithfully, and under the same conditions. When ran manually, it is easy for a person to forget a certain check, or to degrade its quality.
- The checks should be graded in the same manner for each submission to ensure fairness. When executed manually, it is easy to miscalculate the number of passed and failed checks.

As can be easily seen, all issues described above could be addressed by utilizing unit tests. In fact, the rationale is the very same as for any software development team, which in itself is instructive for the students. However, one potential setback here must be stated. In regular software development scenarios, at a given point in time, the source code may fail to build, or may raise runtime errors, or not implement all the requirements of a given class' contract. Over time all those issues are typically overcome, or, if for some reason it is decided that some of the requirements will be omitted from scope, then the unit tests need to be changed accordingly. In short, both the code and the unit tests are not seen as static, but as evolving as the needs addressed by the software evolve. In contrast, during an examination, this is not the case. The students work on their solution for a limited period and after they submit it, it cannot be modified anymore. Additionally, both the requirements and the unit tests are provided to students in the beginning of the examination. They remain static throughout the course of the exam. Thus, a student may submit a solution, which implements only a part of the required units/functionality, and which contains bugs and does not build correctly. It may be impossible to run the original unit tests against it (if it does not build), or there may be such a bug inside it, which causes most tests to fail, due to runtime errors, while at the same time there may be valid solutions to certain parts of the exam text, which should be graded positively.

For the reasons described above, it is necessary to view unit tests as a complement to, rather than a core part of, the examination. In particular, all solutions submitted need to be reviewed and graded manually and appropriate feedback to be given to the students, akin to a peer code review in a regular software development process. The unit test results are thus seen as assisting tools and providing

additional information to the reviewer with minimal effort. Also, for submissions which do not implement all required units and/or contain bugs (and thus cause the tests to fail), the teacher should decide whether to adapt the unit tests, so that they can run for the given submission, or to leave them out altogether, if this requires more effort than grading the submission manually.

2.4 As means to improve the quality of teaching materials

Often, when a teacher supplies learning materials to their students, they are bound to contain errors and omissions. Sometimes such errors are easy to spot and may be considered as a mere inconvenience to the students. However, they may also be subtle and manifest in such a way as to change the meaning of the original document. In such a case, they may cause the students to internalize falsehoods. Another problem, albeit hugely dependent on the cultural context, is how those errors affect the reputation of the teacher. In some workplaces, if the materials supplied by a teacher, contain errors, they may be perceived as a professional with poor skills and practice. In our experience, this issue may have a significant impact on the self-esteem and confidence of the teacher and cause considerable amount of stress.

One particular type of teaching materials, that are seriously affected by the issues described above, are those used in exams. For example, in certain programming exams, students are given a specific problem and they are required to solve it in a limited period (e.g., a couple of hours). After the exam, they are provided a sample solution to the exam. If this solution contains errors (i.e., compilation errors, bugs, etc.), this raises a question about the validity of the exam. The thinking usually is along the following lines: "If the teaching staff cannot provide a proper solution, when they have more time on their hands and a greater experience and expertise than the students, is it reasonable to expect that the students can solve the problem flawlessly in a couple of hours?" Of course, if the situation is such that there are significant errors in the solution, or the exam was inappropriately difficult, this line of thinking may be justified. However, even a simple and inconsequential oversight may provoke such statements. There are situations when the students with a biased viewpoint and are caused by emotions such as resentment and anger, ignited after receiving a poor grade. In such cases, it is difficult to rationalize that in a programming exam simple oversights are not the underlying cause for a poor grade. Students may fail to appreciate that, in fact, educators understand that under the stressful conditions of an exam it is easy to make errors and, as a result, are tolerant of such oversights. Furthermore, the students need to be taught that human programming errors are an integral part in the discipline of software development and, consequently, need to be managed by appropriate methods and techniques

instead of simply being punished by a poor grade, or, similarly, by a financial deduction for a professional. It should be made clear and explained that it is the more significant errors, that speak of poor understanding of the contents of the curriculum, that are indicated with lower scores.

Based on the above, it is easy to see how unit tests may be beneficial. For example, when providing source code examples that accompany the lectures, the teacher can use unit tests to provide some level of guarantee that they work correctly. In programming exams, a working solution can be developed before the exam. This serves two purposes. Firstly, it helps the teacher validate the task that will be given to the students. They can see how large (as volume of source code) the solution will be; whether it requires the development of additional units of code, which were not foreseen, while writing the exam text; whether the solution involves topics not yet covered in the classes, etc. Secondly, this solution may be released to students immediately after the exam. To validate this solution, the teacher may cover it with unit tests. This will not only help them fix any errors and oversights, which could be introduced in the solution, but it will also greatly reduce the stress experienced by the teaching staff, as the proper test coverage will give them a peace of mind and increase their confidence in the correctness of the supplied solution. Finally, the solutions will help validate the unit tests themselves. Invalid or incorrect unit tests may cause students even greater distress than an erroneous reference solution. Even if they are considered complementary material, our experience shows that unit test validity is as important to the students as the correctness and clarity of the statements of the examination problems.

3 Practical considerations

This section contains certain best practices that were discovered by our teaching teams in the process of integrating unit tests in our work.

3.1 Exam preparation

Step 1: Develop the exam assignment.

While the text of the exam is being developed, in parallel, a solution to it is also being written. Of course, it is also possible to achieve this in a waterfall style – first develop the exam text and after that the solution, but it seems more natural and easier, when they are done in parallel, or even if the solution precedes the text of the exam. This may seem strange at first, but here is a viewpoint, which helps understand the underlying principle. In reality, the text of the exam is only a physical externalization of a concept, a set of requirements that the solution must fulfill. If this concept is kept only in our heads, it is easy to omit one or more of its aspects. By externalizing it as a fully working, assembled software system, we

can consider it in its entirety and be able to describe better it. On the other hand, if we first develop the solution and only then begin to describe it, there is an increased risk of missing one or more of the important points.

During this process, unit tests may be written in parallel too, but often this leads to a more complicated workflow. They can also be implemented in a test-driven development (TDD) style [11], or after the solution is fully developed. This depends a lot, on what the teaching staff is comfortable.

During this process, it may be discovered that the exam task is too complicated or too easy, that it contains one or more elements that should be removed, that it does not address properly the curriculum of the course, etc. If this turns out to be the case, there is an excellent and timely opportunity for the problem to be adjusted appropriately.

Step 2: Reevaluate the scope of the assignment.

After the solution is completed and properly covered by unit tests and they all pass, it may be considered valid. At this point, it should be considered if the amount of source code that needs to be written is proper for the time limit of the exam, whether the required knowledge and skills correspond to what the students have been taught in the course, etc. The statement of the exam problems should also be given a final revision. Our experience shows that during this final revision process it is crucially important that the three exam components: problem statement, unit tests, and reference solutions, are all kept in sync, and any change made to one of them is immediately propagated to the other to avoid confusion and misalignment.

Step 3: Determine whether to supply the solution to students after the exam.

Depending on the situation at hand, it should be decided whether to supply the solution to the students and if so, when (i.e., immediately after the exam, or after a given period, etc.). For example, in certain cases, the students are allowed to attempt a given exam multiple times. Typically, a given time interval must pass before students are granted another attempt (a week, a month, etc.). If one supplies the solution immediately after the first attempt, this may make it pointless to give an option for a second try.

Step 4: Determine whether to supply the unit tests to students after the exam.

It should also be considered whether to supply the unit tests as part of the supplied solution. This may be an excellent opportunity for the students to learn how to cover their solution with unit tests, how to organize them, etc. However, it also means that the teaching staff needs to spend additional time organizing (and probably documenting) the unit tests, so they are in an appropriate form, which can be presented to the students.

Step 5: Determine whether to supply the unit tests to students during the exam.

Another thing to consider is whether to supply the unit tests to the students during the exam and if so – whether to supply the entire suite of tests, or just a proper subset of it. It should also be considered whether the students will be required to develop unit tests themselves and if so, how this will affect evaluation. In this case, the teaching staff, which oversees the exam, may need to be qualified to provide aid and/or to answer any questions that the students may have, related to the unit tests. This however is not always necessary, nor desired. For example, in certain exams in our university, the staff overseeing the examination is not allowed to discuss the topic of the examination and/or to give any advice or guidance.

3.2 Exam grading

If the teaching team has followed the practices described in the previous section, they will have a suite of unit tests that can be used to check the submissions. However, there are some important things to consider.

Let us assume the exam statement requires the students to develop a given set of code units. For example, they may be required to implement specific classes, based on a description of their interfaces. A given student may fail to implement all units. In such a case, there will be unit tests that refer to units not present in the source code and this will most likely prevent us from running a successful build. A possible solution is for the examiner to provide simple, "dummy" definitions for such units, which fail all tests. For example, if students are required to implement a certain function, but it is missing in a certain submission, the examiner can add a version of the function, which does nothing, but fail. Usually, unit test frameworks have a dedicated function/macro for that purpose, or the programmer can simply write an "assert false" statement in the code.

Secondly, it is possible there are errors in the submission, which prevent the program from building properly. In such a case, the examiner can comment out the problematic code and supply a "dummy" unit, as described above.

Thirdly, a very common (at least in our experience) mistake made by students is to misspell a code unit. For example, the exam text may ask them to implement a function called `sortArray`, but they may call it `sort_array`, or `SortArray` in their code. This will also cause the unit tests to fail. An easy solution may be to add a definition, which links the identifier used in the unit tests to the one the student has written. For example, the examiner may define a macro, which replaces `sortArray` with the identifier used by the student, or they may define a new function `sortArray`, which simply delegates to the function developed by the student.

All of the above, however, may cause a significant amount of work for the examiner and for each submission; it should be considered if the effort spent, supplying dummy units outweighs the benefits of using unit tests to grade the submission.

To alleviate this issue, it may be beneficial to design the exam with unit testing in mind. For example, students may be required to organize their code in separate files/modules, so that code units are divided into subsets that can be tested independently. Of course, this depends a lot on the exam and such an approach may not be possible.

In our view, it is important to state explicitly that unit tests should be viewed as an aid, and they can in no way replace proper examination of the submitted solution by a human.

- The examiner should always check the source code, regardless of whether it passes all checks or not.
- The final grade should never be automatically assigned, based on how many tests have passed successfully, but should be determined by the examiner after reviewing the solution. If any grade suggestion is being automatically calculated, all team members must have a clear understanding that it is intermediate and subject to review and change.
- The grading process should be clear and transparent to students. We recommend that the teaching team assure explicitly students that each submission is carefully reviewed and graded by the members of the team.

It should also be stated that there are limits to what can be checked with unit tests. Obviously, they cannot be used to evaluate the coding style, and may not be able to detect bad practices. It is also neither feasible, nor necessary to check for every possible error that can be introduced in each code unit. On the other hand, submissions usually contain all sorts of errors – memory leaks, improperly organized code, poor architecture, bad complexity, etc. A certain balance must be found: again, an excellent teaching opportunity for students, as such a balance is important in actual software development as well.

Finally, it may be beneficial for the team to keep the unit tests used to check submissions in a centralized place and to improve them as grading proceeds. For example, one examiner may find that many submissions contain a certain type of error, which is not detected by the unit tests. It is very likely that the error may be present in the submissions checked by the other members of the team. Thus, the examiner may implement one or more tests that detect the error and push them back to the central repository. The other examiners can pull the improved unit tests and run them again against the submissions they are processing.

4 Related work

The application of unit tests to programming education has been explored by multiple authors in a variety of contexts. Automated testing of student submissions in programming courses for the purposes of grading has a long history spanning over 50 years and have been studied extensively [12,13]. Most of the case studies rely on end-to-end tests, which test the entire solution on various inputs and validate against an expected set of outputs. Such an approach aids the development of a mindset in the students for considering a wide spectrum of input possibilities for their program in order to ensure obtaining a maximum grade. It also teaches them that the correctness of their solution can be evaluated automatically given the appropriate set of tests, which is something they can do themselves before submitting their programs for grading. With end-to-end tests, incremental grading is typically achieved by developing an appropriate set of tests, which aim to capture corner and error cases (e.g., small inputs, null or invalid inputs, large inputs) or potential logical flaws (e.g., by falsifying a potentially incorrect assumption about the input). This approach, however, rarely focuses on testing individual units of the solution, and a simple omission in the program (e.g., off-by-1 error) can lead to a zero score.

The discipline of writing unit test has typically been taught in intermediate and advanced programming courses dedicated to software testing and production programming, as opposed to introductory programming courses (cf. [14,15]). It is only a recent trend that software-testing practices are increasingly introduced in introductory courses. A useful systematic survey of research related to the application of unit test to programming education [16] offers similar observations to ours: the benefits of introducing unit tests early on can extend significantly beyond grading, but also into curriculum, familiarizing students with testing tools and processes, improving course materials, and developing perceptions and behaviors toward testing.

Some authors, such as Howles [17] and Edwards [18] argued as early as 2003 that fostering a software quality culture should start from the very early stages of programming education. This idea has been taken further by Janzen and Saieidian [19], who coined the team *test-driven learning*, inspired by test-driven development, where students learn by starting from the unit tests before designing their solution. The authors later attempted to evaluate the effectiveness of this approach with a quantitative approach with moderately positive results, although the small amount of data was insufficient to draw broad conclusions [20]. The authors discuss similar benefits to what we have noticed in terms of teaching the underlying concepts of software quality practices and the change in perception of students where unit tests are concerned.

Some authors note the danger of introducing undue cognitive load on introductory-level students by adding concepts such as unit testing and test-driven

development early on [21]. Their proposed solution is the development of a simplified macro language for defining unit tests in function comments as opposed to applying a fully-fledged unit-testing framework. Our approach is slightly different: we insist on using an actual unit-testing framework without the application of intermediate tools, but we carefully select a framework by considering simplicity of usage and apply only a limited set of capabilities, which we found sufficient for an introductory level course.

5 Conclusion

Based on several years of experience in utilizing unit tests in teaching and examinations in CS/SE/IS courses, the authors believe that their early introduction is highly beneficial for students. In addition to the purely practical benefits of providing a degree of safety and confidence in the correctness of the developed code, be it examination solutions or teaching materials, the exposure of students to this practice helps them achieve a first-hand appreciation of its advantages and drawbacks. We have found that it helps them build their own style of approaching unit testing in parallel with finding the programming style that they feel most comfortable. Last, but not least, students appreciate obtaining a level of familiarity with the unit testing practice, which is a valued and beneficial practice in commercial software development, especially of widely used information systems.

As part of future discussion, we will aim at mapping the introduction of unit tests to specific learning outcomes (cf. [5]) in order to isolate better their contribution to the overall learning process.

6 Acknowledgements

The authors gratefully acknowledge financial support by Sofia University “St. Kliment Ohridski” grant 80-10-173/05.04.2021.

References

1. Peláez C. (2016). Unit testing as a teaching tool in higher education. In *SHS Web of Conferences* (Vol. 26, p. 01107). EDP Sciences.
2. IEEE Computer Society (2014) Guide to the Software Engineering Body of Knowledge (SWEBOK) Version 3.0. Chapter 4 Software Testing pp.4-2
3. Robert M. (2020) Clean Agile. Chapter 5: Technical practices; section “Test-driven-development”. Pearson.
4. Robert Martin (2009) Clean Code. Chapter 9: Unit Tests. Prentice Hall.
5. Kanabar V., & Kaloyanova, K. (2017). Identifying and embedding behavioral competencies in Information Systems courses.
6. Catch2 Tutorial. https://github.com/catchorg/Catch2/blob/devel/docs/tutorial.md, last accessed 2021/05/05.

7. Googletest Primer. https://github.com/google/googletest/blob/master/docs/primer.md, last accessed 2021/05/05.
8. Moodle. https://moodle.org/, last accessed 2021/05/05.
9. Stay, M. (2001, April). ZIP attacks with reduced known plaintext. In *International Workshop on Fast Software Encryption* (pp. 125-134). Springer, Berlin, Heidelberg.
10. 7-zip, https://www.7-zip.org/, last accessed 2021/05/05.
11. Beck, K. (2003). Test-driven development: by example. Addison-Wesley Professional.
12. Kirsti M Ala-Mutka (2005) A Survey of Automated Assessment Approaches for Programming Assignments, Computer Science Education, 15:2, 83-102, DOI: 10.1080/08993400500150747.
13. Daly C., Livingstone D., and Orwell J. (2005) Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.* 5, 3, 4–es. DOI:https://doi.org/10.1145/1163405.1163409.
14. Garousi V., and A. Mathur (2010) Current State of the Software Testing Education in North American Academia and Some Recommendations for the New Educators, 23rd IEEE Conference on Software Engineering Education and Training, 2010, pp. 89-96, doi: 10.1109/CSEET.2010.29.
15. Allen E., Cartwright R., & Reis C. (2003). Production programming in the classroom. ACM SIGCSE Bulletin, 35(1), 89. doi:10.1145/792548.611940.
16. Passos Scatalon L., Jeffrey C., Carver, Rogério E. G., and Francine Barbosa E. (2019) Software Testing in Introductory Programming Courses: A Systematic Mapping Study. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19). Association for Computing Machinery, New York, NY, USA, 421–427. DOI:https://doi.org/10.1145/3287324.3287384.
17. Howles T. (2003). Fostering the growth of a software quality culture. ACM SIGCSE Bulletin,35(2), 45 – 47.
18. Edwards S. H. (2003). Rethinking computer science education from a test-first perspective. Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA '03. doi:10.1145/949344.949390.
19. Janzen D. S., & Saiedian H. (2006). Test-driven learning. ACM SIGCSE Bulletin, 38(1), 254. doi:10.1145/1124706.1121419.
20. Janzen D., & Saiedian H. (2008). Test-driven learning in early programming courses. ACM SIGCSE Bulletin, 40(1), 532. doi:10.1145/1352322.1352315.
21. Lappalainen V., Itkonen J., Isomöttönen V., & Kollanus S. (2010). ComTest. Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education - ITiCSE '10. doi:10.1145/1822090.1822110.