Systematic Testing should not be a Topic in the Computer Science Curriculum!

Henrik Bærbak Christensen
Department of Computer Science
University of Aarhus
DK–8200 Aarhus N
Denmark
hbc@daimi.au.dk

ABSTRACT
In this paper we argue that treating “testing” as an isolated topic is a wrong approach in computer science and software engineering teaching. Instead testing should pervade practical topics and exercises in the computer science curriculum to teach students the importance of producing software of high quality. We point out that we, as teachers, are partly to blame that many software products are of low quality. We describe a set of teaching guidelines that conveys our main pedagogical point to the students: that systematic testing is important, rewarding, and fun, and that testing should be an integrated part of any software development process.

Categories and Subject Descriptors
D.1.5 [Software]: Object-oriented Programming; D.2 [Software]: Software Engineering: D.2.2 [Software]: Design Tools and Techniques; D.2.5 [Software]: Testing and Debugging; K.3 [Computing Milieux]: Computers and Education

General Terms
Systematic Testing, Design

Keywords
CS Curriculum, Systematic Testing

1. INTRODUCTION
The title of this paper will hopefully provoke and enrage the reader. On face value, it may seem that our message is that systematic testing should not be taught in the computer science and software engineering curriculum. However, our real point is to read the title with emphasis on the word “topic: “Systematic Testing should not be a topic in the Computer Science Curriculum” — instead systematic testing is so vital that it should pervade most aspects of our teaching.

Systematic testing is not a goal in itself. Reliable software is the goal—and we believe that systematic testing is a good technique to achieve this goal.

The software industry has produced numerous products of questionable quality. This problem is perhaps best illustrated by the fact that customers regrettably have become used to low-quality products and have accepted it as an inherent feature of software; something they would not have accepted for products like cars and televisions.

There are a lot of reasons why software is all too often of low quality. The project management in software industry carries a lot of the blame. The demand for low time-to-market results in optimistic deadlines that force developers to compromise quality. Consumers are also to blame. People are generally more inclined to buy a product with many features than a product of high quality: features are much easier to market than the less visible quality of reliability.

However, the point of our paper is that teachers have a responsibility as well. We are to blame if we do not do our teaching properly. It is well-known that testing is not considered to be interesting by most software developers. We argue below that this viewpoint may directly stem from the way developers are being taught their trade.

2. RELATED WORK
Our approach is inspired by the work on eXtreme Programming pioneered by Kent Beck and others [3]. Kent Beck points out that a software product is basically driven by four parameters: resources $r$, time $t$, scope $s$, and quality $q$. These parameters are related in subtle ways: if you spend more time on a software project, you can broaden the scope and/or heighten the quality; if you put more resources on the team you may be able to get more done in less time, etc. If three of the four parameters are fixed, then very little freedom is left for adjusting the forth parameter. Industrial projects are usually defined in terms of $r, s, \text{ and } t$: “the software must implement features X, Y and Z; it must be delivered on June 1st; and you, Tom and Ann will program it”. Thus, the only parameter left for the team is $q$, quality. As $r, s, \text{ and } t$ is often set very optimistically, $q$ necessarily becomes low in order for the team to survive the project.

Kent Beck argues in favour of a new process, eXtreme Programming, that breaks this evil circle. One of the premises is to let $s$, scope, be the free parameter that the team may adjust. The quality is maintained high throughout the project by rigorous, systematic, testing and refactoring [7] to keep the design sound. As outlined in the next section, we find that both Beck’s observations as well as his
proposals for a new process can put the way we teach in perspective. Our approach follows two of Bergin’s Pedagogical Patterns [5], namely Early Bird and to some extent Spiral. As argued in the next section, we think that systematic testing is a core discipline that must be reinforced in every aspect of computer science teaching. Bergin’s patterns state that important topics should be presented early and be reinforced often.

There are numerous books and papers about testing: however it is difficult to find literature that specifically treats the topic of integrating testing in teaching. This is perhaps not a surprise, considering the low priority that testing (and verification and validation techniques in general) has in curricula [1]. One important exception is the paper by Shepard et al. [10]. They describe how verification and validation techniques, notably testing and inspection, are integrated in the teaching of six out of eight undergraduate courses, and are the subjects of a complete graduate course. Their motivation is similar to ours: “It is important to teach an appreciation and enthusiasm for this wide range of activities [validation and verification], to try to change the prevailing mindset of testing as a necessary evil.” Shepard et al. have used their approach in a number of years; in comparison our work is less evaluated. We find that our work complements their experience as we provide some concrete teaching guidelines and argue how they serve to convey our main pedagogical point to the students.

3. SYSTEMATIC TESTING IN TEACHING

Software engineering courses are usually organized around a number of topics. Software engineering is a broad field and thus many of the topics are relatively independent. However, most practical topics and exercises require that students have some core knowledge. One obvious example of core knowledge is programming. While topics like “search-algorithms” or “concurrency” can be treated in a theoretical and mathematical way, the students must master basic programming skills to transfer their knowledge of either topic to working programs.

Thus we here make a distinction between “core knowledge” and “topics”. Core knowledge is the more fundamental knowledge that pervades every developer does and has to be mastered in order to fully comprehend and utilize the knowledge represented by “topics”. Many topics are indeed important to teach and important to know about as a developer, but they do not pervade every aspect of software development. As an example, we teach our students about search-algorithms—but searching does not pervade every aspect of our teaching and of the exercises from that day and on.

This distinction appears perhaps somewhat artificial but our main point is that “systematic test” is generally treated as a “topic” in the curriculum rather than “core knowledge”. Our point of view is that testing should be core knowledge that pervades our teaching and the students’ approach to exercises. Our goal is to make our students feel: “I cannot develop software products without a programming language and a compiler. I cannot develop reliable software products without systematic testing and a testing environment”.

3.1 Teaching Guidelines

How do we make students feel that they cannot program without also doing systematic testing? We consider that much of the answer lies in how we structure exercises and how we evaluate the students’ solutions, and have formulated a set of teaching guidelines. We are presently restructuring an advanced programming course to the guidelines and are in the process of assessing the impact, as outlined later.

Guideline 1: Insist on documented high quality rather than full scope in exercises.

A typical software engineering exercise determines the exact same parameters as the traditional industrial project: time (you must deliver next Monday), resources (individual or group assignment), and scope (your implementation must do X, Y, and Z). Thus our teaching inherently sacrifices quality. Our guideline states that we should fix the requirements on quality instead and be willing to sacrifice scope. In practice, this means that we approve the exercise where the group has only implemented feature X and Y and sacrificed feature Z in order to ensure that both feature X and Y work reliably under all foreseeable circumstances, documented by systematic testing; and that we reject the exercise that have implemented all features but cannot document that the implementation is reliable.

Guideline 2: Make quality measurable.

Why do teachers evaluate exercises based upon the parameters time, resources, and scope? For the exact same reasons that industrial salesmen and project managers do: they are easy to assess! Evaluating quality is much harder. Therefore we need to make quality measurable and more manifest to both the students and to the teachers. We use two techniques to accomplish this. For the quality attribute, we teach systematic testing techniques and demand that students always report the principles and arguments they have used to ensure that their testing of a specific programming exercise is complete. To make the quality manifest/visible, we use the JUnit testing framework [9]. We will explain JUnit in the next section.

Furthermore, we ask the students to follow the recommendation of eXtreme Programming, namely the test-driven implementation approach: Write a test that won’t run, then write the code that will make it run [4].

Guideline 3: Make exercises a progression.

Making software reliable by insisting on systematic testing is a long term investment as seen from the developers’ (students’) point of view. Developers pay by spending time on testing—and the customers get the benefit of better software. This is unfortunately not in line with human nature. Humans want to have the benefit of an investment ourselves and the faster we get the benefit the better. Thus it is vital that teachers ensure that the students benefit from their tests. This is the only way we can hope they will continue writing tests as a natural part of programming.

Our proposal is to formulate exercises as a progression. I.e. exercise $n$ is based upon the solution to exercise $n-1$. This way, students are forced to maintain and extend their existing code base instead of starting from scratch every time.

In formulating the progression, it is important that the students are not aware of the precise demands that exercise $n+1$ will pose on the solution they are working on at present. This way the students will be forced to redesign and reimplement parts of their earlier solution in order to accommodate new requirements—and it is here that systematic tests show their power as a tool for programmers. Without systematic tests, it is very difficult to ensure that design changes and reimplementations do not invalidate the functionality within the existing code base. With systematic tests, however, a sound basis is provided for refactoring the system while ensuring that it still provides the required functionality.

This guideline serves several pedagogical purposes. First, students learn that tests counteract architectural erosion of their existing code base. Second, students learn that not focusing on quality...
early may become very costly in the long run, as they are stuck with their previous solutions. Third, the processes they are going through resemble the industrial reality of software development much more.

4. EXPERIENCE

We have some initial experience with the guidelines and approach from teaching the course “Programming of Large Systems” [6]. The course is part of open education at University of Aarhus. The focus of the course is to cover techniques for handling the development of large, object-oriented, programs, and includes topics like design patterns, frameworks, concurrency, and advanced language constructs. The programming language used is Java. Most of our students are software developers in the software industry that take the course to sharpen their programming skills. A major part of the course is a large compulsory programming exercise that counts as part of their exam.

4.1 The Compulsory Exercise

The course has been taught three times, but it is the first time that we have fully tried to follow the proposed guidelines as much as possible. The primary vehicle has been the compulsory exercise formulated as a programming experience with three deliveries. The topic of the exercise is the board game Backgammon and the deliveries are:

- **Delivery 1: Backgammon Board**: Design and implement the basic domain: board, board-locations, and checkers, and methods to move checkers and determine board state.

- **Delivery 2: Backgammon Validation**: Design and implement validation of moves; the logic to control game flow (rolling of dice, which player is in turn, how many moves are left); undo of moves; and ability to switch at runtime who controls the two players (AI or human player).

- **Delivery 3: Graphical User Interface**: Design and implement a GUI based on JHotDraw [8], and link it with the developed domain model to provide a full backgammon game.

Using a board game as exercise has a lot of advantages. The domain is well-defined and generally known to the students. The rules of backgammon are complex and demand systematic testing techniques to ensure that e.g., move validation is correct. And finally, it requires a graphical user interface that suits the functionality provided by JHotDraw well. JHotDraw is used as example of a large object-oriented framework in the course.

4.2 Quality precedes Scope

In the specification of all deliveries, we explicitly stress that Quality precedes Scope. That is, in case that the students are running short of time then they should choose to implement less functionality but keep the quality high.

The means to achieve high quality is systematic testing. We teach standard techniques such as equivalence testing, boundary testing, etc., and ask the students to document their testing practice. We demand that they use the JUnit testing framework, described next, to organize and document their test cases.

4.3 Measurable Quality: JUnit

We introduce JUnit [9] as testing framework to the students in the course. JUnit is a powerful yet simple testing framework that significantly lowers the effort required to setup a testing environment. A snapshot of the JUnit framework is shown in fig 1.

One major advantage of using JUnit is that students can focus their effort completely on programming the backgammon model in the first two deliveries: there is no need for writing “drivers” or user interfaces in order to run the model. All testing and debugging can be done from within the JUnit framework. As is also apparent from fig 1, JUnit provides very detailed information about tests that break; here the expected value was 4 but the implementation produced 7 in test “testMoveClass”; the failure occurs at line 115 in class TestAI as seen in the lower window.

JUnit may also be run as a console-based tool, writing output to the console. This makes it a nice tool to integrate with the Apache Ant build management tool [2]. Ant is a Java-based replacement of traditional build tools like Make. What makes the integration interesting is that running the test suites becomes a natural part of the development process. Ant automates an “edit-compile-test” cycle: students modify their Java source files and invoke Ant that compiles the code and thereafter runs all the tests. Thus tests become a natural step, just as compiling.

4.4 Progression in Exercise

A key point in having three deliveries on the same theme is that students are not told what the requirements are in the next delivery. This way, they cannot design and implement the backgammon model in anticipation of particular, later, requirements like for instance our demands for the graphical user interface based upon JHotDraw.

Thus, students are almost certain to make design decisions during the first deliveries that are invalidated by requirements of later deliveries. While the students find this tedious, it has two important pedagogical aspects.

First, students learn that their JUnit test suits are an important help in refactoring their implementation to accommodate new requirements. Without the tests to ensure that the functional requirements of the previous deliveries are still kept, it is very difficult to change and redesign the existing code. We find this experience important for the student: those tests are their tool, and for their ben-
eft; not something they reluctantly do to serve the requirements of management or customers.

Second, this situation is similar to many industrial projects where the introduction of a software system triggers the imagination of the customers leading to numerous new requirements and changes not foreseen at the project start-up. This observation is also one of the reasons that Kent Beck argues in favour of making scope the free parameter for the development team: Software changes its own requirements.

4.5 Student Evaluation

Students have been asked to report their experience of test-driven implementation as part of their reports detailing their effort. The evaluation is based upon 22 students in 9 groups.

Many groups pointed out that writing tests first/early is actually a great help in understanding the functional requirements and issues such as class responsibilities and method parameters. The same observations are reported by Beck.

Many groups also found tests a benefit when they needed to refactor their solution because of new insight or new functional demands. As this was a key insight—that tests are a technique for the benefit of the developers themselves just as much as for the customers—we are of course happy that this was reported.

Some groups reported that the tests written made them reluctant towards changing interfaces as this requires the tests to be modified as well. This is of course a liability of having test suites.

One group reported: “...especially as test and build becomes one step in the ant build script, testing feels as a natural part of the translation of the program”. We hope they will carry this experience into their professional lives.

Another group reported that they felt that they moved forward every time their JUnit bar went green and felt that they were “99% sure that all functions worked”.

One thing that stroked us was how deeply the “we must satisfy all functional requirements” mindset is rooted in some people. Two groups handed in a first delivery that fulfilled all functional requirements, but had almost no testing; one group attributed this to a late start. Obviously the students had felt it more as a failure to leave out functionality than to leave out the tests, though contradictory to the explicitly stated project requirements!

In the second delivery, the scope demands were set too high, and most groups reported it necessary to leave out specific requirements. Though it was not planned so, we may argue (in hindsight) this is a very direct way to convey the quality precedes scope point to the students—and force them to prioritize requirements.

In the third delivery, some groups reported a great satisfaction and even some puzzlement that “...when we combined model and GUI, the two systems worked together right way.” Thus the model testing effort was rewarded by a painless integration and improved modularity.

4.6 Teachers Evaluation

We have found a great additional benefit from the “quality over scope” guideline that was not obvious from the start. One problem that we are facing as teachers is: “Is my compulsory exercise too easy or too hard?” Especially a young teacher without much experience will have this problem. Requesting quality over scope lessens the burden of finding the right level—you can write an assignment that contains enough challenges even for the brightest students without overwhelming the average or poor student. The average students make a quality implementation of 80% of the functional requirements while the bright students may achieve high quality of all requirements. Students can focus their energy more on selected requirements instead of becoming panic that they do not have time to get everything working.

One issue that needs further work is the question of making quality measurable. Students were required to A) report the techniques used to define test cases and B) provide the actual JUnit test cases. However, it was difficult to assess aspects such as coverage and how the testing systematism was transferred to test cases from the reports.

5. CONCLUSION

In this paper, we have argued in favour of making systematic testing techniques pervade the practical assignments in the teaching curriculum. The motivation is that teaching our students that testing is as natural as designing and coding in making software is an important contribution of our profession to make software more reliable and of higher quality.

The problem is that rigorous testing is more costly than beneficial in the small-scale projects and exercises that are usually given in software engineering courses. We have outlined some guidelines that make testing more beneficial, more natural, and more fun in a teaching context. We have argued that these features can be achieved without sacrificing anything except perhaps some traditional thinking.

Our guidelines are much inspired by ideas from the eXtreme Programming community. A key guideline is to favour “quality over scope”. To achieve this we must make quality easier to measure and have described how JUnit is a tool that goes some way of allowing this.

We have reported some early evaluation in an advanced programming course. The observations are not scientifically complete but do show some interesting statements. Primarily we are happy that many groups report findings that are in line with the pedagogical goals that we had with our guidelines. We are thus confident that the guidelines are basically sound. Many of our students on the course are full-time industrial developers and we hope they will gain insights about testing and how testing benefits the developers themselves that they will carry with them into their professional lives.

We have also reported some additional advantages of interest in a teaching context. Most importantly the focus of quality over scope allows students more room for maintaining a reasonable work-load that fits their intellectual capacity.

6. REFERENCES