How to Teach Software Testing? Experiences with a Sandwich Approach

Leen Lambers
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
Leen.Lambers@hpi.de

***Abstract*—We report on our experience with teaching software testing to graduate students. This experience was gained within a course offered yearly each winter term from 2015 till today. The course does not solely entail software testing as a topic, but also addresses briefly static analysis first (i.e. before testing) and verification afterwards (i.e. after testing). This is the reason for calling it with a twist from the testing perspective a sandwich approach. We motivate and present the overall structure and format of the course w.r.t. its learning objectives. We moreover describe some illustrative examples for structured interactions with students during the lecture as well as for project assignments, putting these into perspective w.r.t. the sandwich approach.**

***Index Terms*—Software testing, Automatic testing, Software engineering education**

I. Introduction

We aim at sharing our experience of *teaching software testing* to graduate students in a course with the title "Software Analysis, Testing and Verification". The course focuses on software testing, but deliberately presents it *intertwined with other analytic quality assurance techniques.* It addresses briefly static analysis first (i.e. before testing) and verification afterwards (i.e. after testing), which is the reason for calling it with a twist from the testing perspective a *sandwich approach.*

The author offers the elective course with 6 ECTS each winter term since 2015 until today as part of the IT-Systems Engineering study program at the Hasso Plattner Institute for Digital Engineering at the University of Potsdam. It has the following *learning objectives*:

1) get an overview of *state-of-the-art & state-of-the-practice* of software analysis, test & verification techniques,
2) understand *diversity, complexity & limitations* of software analysis, test & verification techniques as well as the *role of automation*, and
3) be capable of discovering, developing and/or adapting, *applying & combining* appropriate software analysis, test & verification techniques to *different use cases.*

The course consists of a weekly *lecture* of 90' (addressing mainly learning objectives 1 & 2) and a weekly *project-based exercise* appointment of 90' (addressing mainly learning objective 3). We report on the *contents and format* of the lecture and project. We thereby concentrate on presenting its *current* contents and format (following the above-mentioned sandwich approach), explaining why it arose based on experiences from previous editions. We also briefly describe some *structured interaction examples* in the lecture as well as a *project assignment example* with testing focus. Thereby we put these examples into perspective w.r.t. the sandwich approach.

II. Lecture

A. Contents

The contents of the lecture (as illustrated in Figure 1) are centered around *two standard text books* in the testing and analysis literature. We in particular rely on the material presented in the book on "Software Testing and Analysis" [1] for conveying the *first and second learning objective*. This is because this is one of the rare and more recent seminal standard text books focussing on analytic quality assurance techniques covering the topics of *static analysis, testing and verification as well as their integration* in a holistic manner. We have experienced that this approach is important in order for students to learn how to make use of all available analytic techniques in an integrated way. In the course testing is treated more prominently compared to the topics of static analysis and verification. The integration of testing with the latter techniques becomes apparent and are considered very relevant.

When it comes to presenting the *topic of testing*, focussing *test design* in particular, we follow the comprehensive coverage-based approach from the book "An Introduction to Software Testing" [2]. We have experienced that organizing test design around the coverage of four structures (input, graph, logic, syntax) is indeed helping our students effectively to understand the diversity of different test approaches as well as to come up with diverse test sets in their projects within a short-time period themselves. The author supports the focus on test design (as propagated also in [2]), since she feels that finding an answer to the related questions "which tests to write" or "why each test is present" is key for coming up with a successful testing approach and seems to be at the same time hard for students. Finally, the topic of *automation in testing* is outlined with a focus on how to automate excise tasks in software testing, in particular in view of regression testing.

Finally, the lecture integrates quite an extensive selection of references to *research papers* from the testing and analysis research community in order to illustrate the diversity, complexity and limitations of corresponding techniques and their

978-1-7281-1075-2/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSTW50294.2020.00076

automation *in practice*. In particular, it presents work reporting on the combined application of analytic quality assurance techniques to different domains (e.g. [3], [4]). In some of the previous editions, we also organized one guest lecture with an *invited speaker from industry*, which emphasized in particular the intertwined practical use of static analysis as well as testing techniques and the role of automation in praxis.

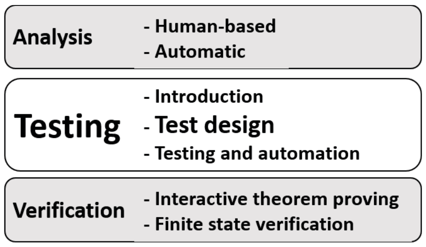

Fig. 1. A Sandwich Approach to Teaching Software Testing

B. Format

The lecture format is as follows (cf. Figure 1): it starts with a general introduction (two 90' lectures) to software verification and validation emphasizing the importance of quality assurance in software engineering in general and introducing basic terminology. It continues with the topics static analysis (three 90' lectures), testing (five 90' lectures), and verification (two 90' lectures). The ordering of these topics used to be different in the first three editions: testing, static analysis, and verification. Swapping the order of the first two topics and having testing as a topic in the middle (*sandwich approach* from the testing perspective) was initiated by a yearly informal evaluation of the course with the students during the last lecture, where a corresponding hint came up during discussion. Indeed it turned out to be very natural to start the lecture with the *topic of static analysis (human-based as well as automatic) before introducing the topic of testing*. This was mainly related with the fact that students in this course apply all techniques presented during the lecture to a real world software project (cf. section III). In particular static analysis techniques support to get more acquainted with a software project, learn more about its requirements descriptions, design as well as its code base. This gained knowledge from *static analysis* can then be used further for *steering testing activities* in different ways: complex modules of the project identified during static analysis can be tested more thoroughly, refactorings initiating from code smell detection need to be tested for behavior preservation, alerts generated by bug detection tools (including pessimistic inaccuracy [1]) can be identified as real alerts by developing tests demonstrating failures arising from the alerts indeed, etc. These *practically motivated work flows* help to transition smoothly to the topic of testing and motivate as well as frame the need for testing. This is strengthened by presenting at the end of the static analysis part of the lecture empirical work demonstrating that testing can focus on finding fault classes (functional/algorithmic [5]) that were not in the focus of static analysis. During the testing part of the lecture its optimistic inaccuracy [1] then inherently becomes clear *motivating the need for verification* (of specific parts or properties) of the system. Moreover, the use of contracts (previously encountered as test oracles) together with the idea of symbolic execution (previously encountered in the context of automated control flow graph coverage based test value generation) facilitates the transitioning to interactive theorem proving. Finally, during the verification part of the lecture it becomes clear that verification can find bugs that are hard to test for and that testing is often still useful in the form of *conformance testing* [1].

In general, each single lecture introducing a new analysis, testing or verification technique follows a similar format. Each technique is introduced using some kind of *structured interaction with the students* challenging their current knowledge of the technique as well as motivating them to learn more about it (cf. group brainstorming example in subsection II-C). It proceeds with *presenting the basic technical concepts* of some technique on small examples interleaved with minor and major active classroom exercises. At the end of each lecture the *technique is presented in action* in an industrial or research context (referring to case studies described in the testing and analysis literature) in order to illustrate the corresponding current strengths and limitations of each technique.

C. Structured Interaction Examples

We describe some selected examples of structured interactions during lecture. In larger groups these interactions can be supported by a *life-feedback system* such as e.g. Pingo [6].

a) Group Brainstorming - Coverage Types: This interaction takes place just before introducing the concept of coverage-based test design [2] (cf. Figure 1). The question shown to the students initiating the brainstorming is the following: "Which types of coverage do you know?" Students brainstorm for a few minutes with their neighbour some possibilities after which this list of possibilities is collected and discussed within the complete student group. The aim of this interaction is on the one hand to *assess which types of coverage students know* already, and on the other hand for each student to *imagine and get curious about more types of coverage* by listening to the answers of fellow students and some further examples given by the lecturer. The variety of different coverage types obtained as a result of the brainstorming motivates the previously mentioned *consolidated approach on coverage-based test design with four basic structures* (input, graph, logic, syntax) [2]. Note that the preceding lectures and project-based exercises on automatic *static analysis* have already introduced e.g. control flow graphs as key artefacts, now reused for test design. Similarly, human-based analysis of other software artefacts such as e.g. requirement specifications or documentation prepared for reusing them within test design. Focussing logical expressions within test design on the other hand prepares e.g. for the subsequent lectures on *verification*, where logical expressions reoccur in the form of contracts.

b) Group Discussion - Test Approach vs Test Classification: This interaction takes place during the testing introduction lecture (cf. Figure 1) after outlining testing classifications such as 1) unit, integration, system testing, 2) validation, defect testing [7] 3) white-box, black-box testing. It was moreover explained from which point of view (1: scope, 2: testing goal, 3: internal vs. external description) the test classifications arose. Then some particular examples (such as reliability testing, behaviour-driven testing, or exploratory testing) from the plethora of test approaches are presented and students are asked to position each of them w.r.t. the presented classifications. The aim of this interaction is for students to *learn how to assess the focus of a new test approach* (making apparent its inherent optimistic inaccuracy [1]) they are confronted with and to understand its strengths and limitations in order to complement it with some other test approach where applicable. Often one test approach focuses merely on one part of each classification. E.g. reliability testing focuses on system testing as well as on validation testing and is a form of black-box testing. As opposed to the preceding and subsequent lectures on *static analysis and verification*, resp., based on this discussion students learn to distinguish here different aspects mainly related to evaluating software by *observing its execution*.

c) Active Classroom Exercise - Testing and Automation: This interaction takes place when outlining the topic of testing and automation (cf. Figure 1). Students are aware of the four different steps in the testing process (test design, preparation of test values, test execution, and result comparison) and have learned the terms controllability as well as observability (terminology from the testing introduction lecture). This active classroom exercise is performed in small groups. Each group has the task to classify a number of testing tools w.r.t. the four steps of the testing process. Tool examples are selected from a dedicated testing tool website (e.g. https://java-source.net/open-source/testing-tools). The main objective of this interaction is to *learn how to find tool automation for a particular step in the testing process* as well as to *discover strengths and limitations of automation w.r.t. each step in the process*. The exercise precedes and *motivates the focus on test design* within the testing part of the lecture, since test design represents the main step in the testing process that can hardly be automated. Compared to the preceding lectures on *static analysis*, where students learn about the interplay of human-based and automatic static analysis techniques (e.g. distinguishing real alerts from false ones remains usually a human-based activity, which initiates from inherent pessimistic inaccuracy [1] in automatic static analysis), they learn here about the role of automation in the context of testing. In particular they learn that test design remains mainly a manual and creative activity (capturing clearly the inherent optimistic inaccuracy related to testing), setting the stage for further steps in the testing process that can then often be automated. In the subsequent lectures on *verification* students analogously learn about (and compare with) the role of automation in techniques such as interactive theorem proving or model checking.

III. Project

A. Contents

We follow a project-based approach in order to convey in particular the *third learning objective*. To this extent students work in *groups* of two or three people on a common software project. Students are *free to propose their own real world software project* of interest that they want to analyse using all techniques presented in the lecture. The aim of this freedom is on the one hand to support the intrinsic motivation of students. On the other hand usually it turns out that the selected software projects as a whole cover a variety of different application types (e.g. web-based systems or embedded systems), software development processes, programming languages, system sizes, etc. This stimulates the discussion and comparison w.r.t how well different techniques can be applied in an integrated way in varying practical settings. In a weekly (or bi-weekly) rhythm students solve some *project assignment* related to a new technique presented in the lecture. Regular project assignments are similar to classical homework tasks that need to be solved however for the chosen software project (cf. subsection III-C). Few project assignments consist of preparing an individual *presentation* of around twenty minutes, e.g. about a testing tool used to support one of the steps in the testing process for a certain type of tests developed (in the context of the regular project assignments) for the chosen software project.

B. Format

Each week the course comes in addition to the lecture with a project-based exercise appointment. It consists of an *evaluation part* in which solutions of the previously handed-in project assignment are presented by the students and discussed within the group. This evaluation is used to strengthen engagement and understanding with the solutions worked out by the student within the group and simultaneously getting an overview of solutions worked out in different project settings by other student groups. The remaining time of the exercise appointment is used to *prepare the upcoming new project assignment*. First questions w.r.t. understanding can be clarified collectively without circumstances. Single appointments are moreover used for the individual presentations including discussion, or as the case may be some live exercise (e.g. on mutation testing following a game-based approach as presented in [8]).

C. Project Assignment Example: Graph-Based Test Design

This project assignment is handed out after the lecture on coverage-based test design using graphs [2]. Students have learned graph coverage criteria such as *node, edge, and prime path coverage*. They have moreover learned that not only *control flow graphs (CFGs), but also activity graphs (stemming from use case descriptions) or finite state machines* can be used to design tests accordingly. Recall that each project assignment is addressed by a student group working on the same software project selected by the students themselves to be tested.

The project assignment consists of *three tasks*, only roughly described in the following: 1) Select a code snippet of the software project with at least one decision as well as a loop.

Develop and describe a test set satisfying edge coverage, but not prime path coverage as well as a test set satisfying prime path coverage. Describe and explain infeasible test requirements if they occur. 2) Develop some further test design based on some graph (different from the CFG) related to our project. Develop and describe a corresponding test set satisfying a graph coverage criterion suitable to this other graph type and different type of tests. 3) Describe for each of the previously developed test sets how many tests run successfully and how many tests reveal some failure.

During the *preparation phase* for this assignment in the weekly exercise appointment it is possible to brainstorm together with the students which type of graphs other than CFGs can be valuable for test design within their particular software project. During the *evaluation phase* one week later we discuss how successful the related type of testing is w.r.t. finding failures and which types of failures it can discover (or not). Since students work with real world projects, usually a *bug repository* is available that they can use to evaluate if described failures therein could have been detected with this type of test design. Moreover, since they have already applied *static analysis* to their project, they can also evaluate how well it performs w.r.t. finding failures identified during human-based static analysis or issued as alerts by some automatic static analysis tool. We *compare with other types of test design* such as the one based on input domain modelling (topic of the previous exercise). We identify logical expressions related to decisions or loops within the graphs that can be used to refine the test design (in the subsequent exercise). We discuss *implications of infeasible test requirements* for CFG coverage and the difference with having them for example for activity graph coverage. We identify w.r.t. the *test classifications* presented in the introductory lecture to testing which type of testing arises from the example test sets derived for CFG coverage as opposed to, for example, activity graph coverage.

IV. Conclusion

A number of quite recent *experience reports* on university courses *focusing on software testing* exist. It is described and evaluated, for example, in [9] how to add a pragmatic perspective following key elements such as, for example, stimulating interaction with practitioners, or design systems for testability. In [10] a *real world project* is selected by the students themselves (as in the course presented here) increasing student enthusiasm. A recent systematic literature review [11] on testing education gives a further overview on experience reports published between 2013 and 2017.

With the aim of *sharing and stimulating further the discussion* around best practices, experiences and perspectives on how to *teach software testing* we presented briefly the characteristics of an approach to teaching software testing to graduate students, where the testing topic is *surrounded with the topics of static analysis as well as verification* (as in a sandwich from the perspective of testing). Our approach emphasizes *practically motivated work flows* as applied in *real world software projects* and clarifies the *role of automation*. The approach crystallized from an elective course running in its fifth edition now, following a *holistic approach* to the presentation of *analytic quality assurance techniques* within software engineering with the focus on testing. We sketched the *structure and format of lecture* (promoting structured interaction) and *project* (the topic of which selected by the students themselves). We moreover presented illustrating examples of *structured interactions* as well as *project assignments* putting these into perspective w.r.t. the sandwich approach.

Acknowledgment

The author is very grateful to all students visiting one of the editions of the course described in this paper for their engagement, enthusiasm and feedback.

References

[1] M. Pezzè and M. Young, *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

[2] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[3] A. Groce, K. Havelund, G. J. Holzmann, R. Joshi, and R. Xu, "Establishing flight software reliability: testing, model checking, constraint-solving, monitoring and learning," *Ann. Math. Artif. Intell.*, vol. 70, no. 4, pp. 315–349, 2014. [Online]. Available: https://doi.org/10.1007/s10472-014-9408-8

[4] M. Harman and P. W. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*. IEEE Computer Society, 2018, pp. 1–23. [Online]. Available: https://doi.org/10.1109/SCAM.2018.00009

[5] J. Zheng, L. A. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Trans. Software Eng.*, vol. 32, no. 4, pp. 240–253, 2006. [Online]. Available: https://doi.org/10.1109/TSE.2006.38

[6] W. Reinhardt, M. Sievers, J. Magenheim, D. Kundisch, P. Herrmann, M. Beutner, and A. Zoyke, "PINGO: peer instruction for very large groups," in *21st Century Learning for 21st Century Skills - 7th European Conference of Technology Enhanced Learning, EC-TEL 2012, Saarbrücken, Germany, September 18-21, 2012. Proceedings*, ser. Lecture Notes in Computer Science, A. Ravenscroft, S. N. Lindstaedt, C. D. Kloos, and D. Hernández Leo, Eds., vol. 7563. Springer, 2012, pp. 507–512. [Online]. Available: https://doi.org/10.1007/978-3-642-33263-0_51

[7] I. Sommerville, *Software engineering, 8th Edition*, ser. International computer science series. Addison-Wesley, 2007.

[8] J. M. Rojas and G. Fraser, "Code defenders: A mutation testing game," in *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*. IEEE Computer Society, 2016, pp. 162–167. [Online]. Available: https://doi.org/10.1109/ICSTW.2016.43

[9] M. F. Aniche, F. Hermans, and A. van Deursen, "Pragmatic software testing education," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE 2019, Minneapolis, MN, USA, February 27 - March 02, 2019*, E. K. Hawthorne, M. A. Pérez-Quiñones, S. Heckman, and J. Zhang, Eds. ACM, 2019, pp. 414–420. [Online]. Available: https://doi.org/10.1145/3287324.3287461

[10] D. E. Krutz, S. A. Malachowsky, and T. Reichlmayr, "Using a real world project in a software testing course," in *The 45th ACM Technical Symposium on Computer Science Education, SIGCSE '14, Atlanta, GA, USA - March 05 - 08, 2014*, J. D. Dougherty, K. Nagel, A. Decker, and K. Eiselt, Eds. ACM, 2014, pp. 49–54. [Online]. Available: https://doi.org/10.1145/2538862.2538955

[11] P. L. Jr. and A. Arcuri, "Recent trends in software testing education: A systematic literature review," in *31th Norsk Informatikkonferanse, NIK 2018, Universitetet i Oslo, Oslo, Norway, September 18-20, 2018*. Bibsys Open Journal Systems, Norway, 2018. [Online]. Available: https://ojs.bibsys.no/index.php/NIK/article/view/516