Model-based Student Admission
(Position Paper)

Vadim Zaytsev

Institute of Informatics,
Universiteit van Amsterdam,
The Netherlands

Abstract. We should test people the same way we test software.

During the last decades, the field of software testing has matured into a solid sector of software engineering with a wide variety of available techniques, empirically supported usefulness and applicability claims, a sophisticated ontology of terms and approaches, as well as an arsenal of tools. The main contribution of this paper is a proposal for reuse of the domain model of software testing for assessment of students. The proof of concept used in the paper is that of conditional admission of graduate students to a software engineering programme.

Keywords: student admission; education; assignment generation; automated assessment

1 Motivation

There are many established ways to test software [6]. Most of them could be boiled down to forming expectations about the behaviour of a system under test, encapsulating them in some kind of model and exploiting the model to test the actual system — by proving properties over it, generating test data from it, and in general confirming that the behaviour observed from the system is consistent with the behaviour expected by the model. The discussion topic we would like to raise with this paper is the possibility to test students the same way.

In the next section (§2) we go through a number of testing techniques, as far as the space constraints permit, in an endeavour to draw parallels between testing software and assessing students. Then, in §3 we get acquainted with the Master programme in software engineering and explain its admission procedure, which involves examining self-studying students. We also notice how assessment based on learning objectives is similar to testing a software system based on its model, and propose an infrastructure that is useful at least under our particular circumstances, and perhaps in a much wider context as well. We take a moment in §4 to discuss possible drawbacks of the proposal and allow ourselves to draw some conclusions in §5.

2 Testing techniques

Unit testing is a technique to test whether one unit (a method, class, module, package, etc) works as intended. There are various traditions within unit testing and different terms used for variations: a "programmer test" is intended to validate the last atomic change (a commit or even a smaller edit), a "developer test" is a unit test which success would mean total satisfaction of the programmer, a "customer test" which has the same role for the client, etc [5,12,16]. Although the question of mapping these concepts to teaching has already been raised [22,24,27], the focus has remained on teaching programming (essentially, extreme programming), but Conway's Law does not have to have its power here. Any teaching process with regular (say, weekly) exercises that focus predominantly on validating understanding on one isolated topic, is a form of unit testing administered on students. Traditionally, unit testing of programming exercises, if automated, is done by differential testing [11,21].

Acceptance testing refers to an advanced form of customer testing and system testing. Basically an acceptance test contains the most important functional tests for the client to accept a software system — typically covering programmer/developer tests, but not covering some customer tests like stress tests [23]. Even if we step over the obvious observation that final examinations can be seen as acceptance tests (which is, under closer consideration, arguable, since teachers are not really the "clients"), many courses or even guest lectures start with either a formal quiz or a informal collection of data about students' starting level of knowledge. This is an adaptable form of acceptance testing known in software development as behaviour-driven development [20]: a TDD-like setup to guide the process of improving a system until it is in an acceptable state. To the best of our knowledge, there is no systematically developed universal framework for a similar approach in education, even though its application is not uncommon. However, there are striking similarities even in small details: for instance, the range of different techniques used to assess students' starting level in a course (a quiz on the first lecture, a regular test taking the first 10 minutes of every lecture, an informal discussion, wiki-based accumulation of questions, apps, clickers, etc) mirrors the situation in software testing where acceptance tests may not always be automated (unlike unit tests).

Stress testing is an interesting case: it is a technique of deliberate intensification of demands to test software system's robustness, commonly accepted in education as well as a theory around "the comfort zone" [4] which basically boils down to the empirically validated observation that anxiety always improves performance, until a certain level is reached. However, since software and humans[1] are different, the ultimate goal of stress testing of software is to find bottlenecks that can be liquidated and to determine safe usage limits, yet in education we are more concerned with improving the students' performance while still push-

[1] The "comfort zone" effects have also been observed on other animals [32], but we make an excusable assumption that all students are humans.

ing the limits further and in general keeping the challenges and the raising level synchronised.

Compatibility testing is a complementary approach devoted to checking consistency between components of a software system, not within them [9,17]. (Its variation are also sometimes referred to as “interoperability tests” when the focus is on dynamic properties or “protocol tests” when the software-linguistic aspects are stronger at play). The importance, relevance and even necessity of compatibility testing has been pointed out not just in software, but also for information systems in general [3]. For education, compatibility testing reflects to any group activity, and in anything close to software engineering, to project work of solving bigger assignments in a group. Initially group assignments were introduced as a way to reduce teacher/grader workloads, but they were quickly reassessed and praised for the amount of attention they bring to commonly overlooked “soft” skills like communication, planning, task distribution [10]. Since the ways collaboration can fail between humans are prominently more numerous that the ways interoperability between pieces of software can, such educational aspects have been a rather active research area, perhaps even more active than the corresponding software testing one. We face many challenges there like uncollaborative behaviour and combativeness, free riders and procrastinators [30], up to the point where some have proposed to completely redesign of the very concept of homework [8]. However, this is quite possibly the only form of testing/assessment where there is nothing to learn for educators from testers.

Regression testing is about checking whether introducing a new feature has not broken the existing functionality, while **non-regression testing** is about checking whether introduction of a new feature has indeed led to new previously unobserved systems behaviour [25]. Given its place right between unit tests exercising one particular topic (regular assignments) and system tests (final graduate projects), it is relatively easy to find a corresponding approach for (non-)regression tests in midterm/final examinations — not to say they are always designed that way, but they could/should systematically check that the newly acquired knowledge is indeed present and that it does not interfere with previously present expertise.

Conformance testing is done to ensure that a software systems complies with the requirements [15] usually encapsulated in a (semi)formal specification which can reach considerably high levels of complexity, depending on the types of systems under test [31]. Final assessment in each course is also a form of conformance testing with respect to claimed exit qualifications. However, most research from the educational point of view concerns not the assessment models as such, which are usually thought-through but comparably straightforward, but side aspects like deliberate cheating [1] or the impact of the kind of examination (oral vs. written) on the result [13].

Installation testing is a technique found closer to software deployment than to software development as such — it concerns the effort needed and the feasibility per se of getting a software system up and running at the client side, complete with all the configuration steps, but without the very last step that separates such a dry-run from the production. In many cases, and definitely in our case which will be covered in more detail in the next section, such a dry-run corresponds to having a Master's student leave the lab conditions of the university in order to perform the final graduate project at a software engineering company as an intern.

3 Software Engineering in Amsterdam

Master of Science in Software Engineering[2] is an intensive one year programme of graduate education at the University of Amsterdam. It graduates around 60 students yearly and beside the final project contains six courses for 6 ECTS each:

– *Software Architecture* (high level system design and system modelling)
– *Software Specification and Testing* (type systems, testing, verification)
– *Requirements Engineering* (elicitation, analysis and negotiation)
– *Software Evolution* (metaprogramming and static analysis of source code)
– *Software Process* (management, integration, deployment, maintenance)
– *Software Construction* (MDE, design patterns and programming styles)
– *Preparation Master Project* (experimentation, reading, writing, planning)

The last mentioned course runs in parallel with everything else; the other six core courses are run in blocks of two, two months per block, some with an examination at the end, which, even if present, comprises only a part of the grade, with the rest determined by the laboratory work and written reports — all courses are very hands-on, and their educational tasks reflect that. The final project itself (18 ECTS) involves a replication study of a practically applicable academic paper and takes upward of four months.

3.1 Intake procedure

Admission to the programme is based on having a Bachelor degree or its equivalent, as well as on demonstrating sufficient levels in skills like software development, logic, term rewriting, compiler construction and algorithmics. At the entry level, students are expected to be able to manipulate simple mathematical formulae and have some basic knowledge of (semi-)automated testing techniques, have some experience in constructing, versioning, packaging and maintaining software. Given the admission guidelines and the Dutch environment around the University of Amsterdam, we have at least these five large categories of applicants well represented:

– **University students** following the classic educational road by obtaining a Bachelor degree from the same or a neighbouring university, in computer science, software engineering, business computer science, etc., and moving on to a (allegedly more practice-oriented) Master programme.

[2] http://www.software-engineering-amsterdam.nl

– **Students switching** their focus from another programme: mathematics, artificial intelligence, computational science or bioinformatics, for either growing to dislike their original choice or seeking some fresh technical topics.
– **International students** from all over the world with degrees administratively equivalent to the ones from the first two groups, but having occasionally followed a very different curriculum.
– **Technical college students** which have also received a Bachelor diploma (typical for the Netherlands) yet followed a considerably more technical curriculum without much attention, if any at all, to formal methods or any other underlying theory for that matter.
– **Software engineers** who obtained their Bachelor degree in the past (sometimes decades ago) and have been working as practitioners of software engineering ever since, until they decided out of curiosity or as a strategy to boost their CV, to relearn their basics at a more advanced and modern level.

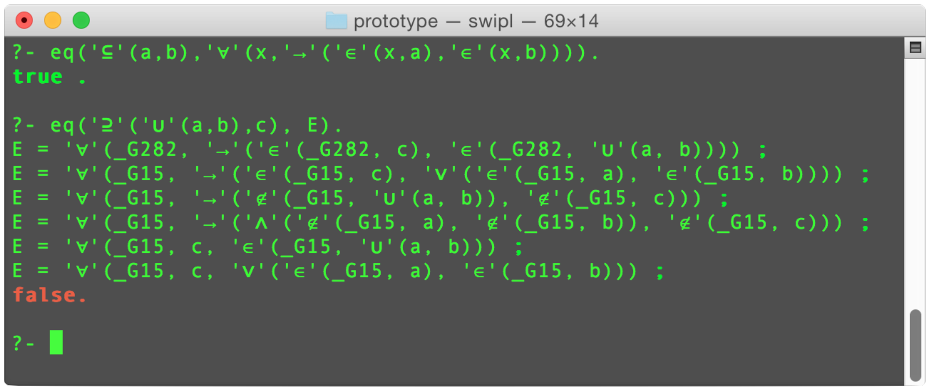
3.2 Premaster courses

Each intake procedure is done individually. It resembles a job interview, occasionally involves a formal test and is performed after receiving, processing and inspecting the application which includes a letter of motivation. Within approximately an hour, a programme coordinator assesses the skills of the applicant and converges to a verdict which can be in a form of rejection, acceptance, conditional acceptance or redirection to a different programme.

Conditional acceptance is the most interesting outcome, because it means that one or more premaster courses are assigned to the student. Each premaster course typically involve many hours of self study, occasional interactive supervision and finally an examination.

An example of a premaster course is *Mathematical Logic* which is meant for technically strong applicants with a weak or absent formal background. The learning objectives revolve around quantifiers, induction proofs, propositional logic, syllogistic reasoning, grammars and automata — all on a very basic level, just enough for potential students to not struggle when they see a formula or discuss infinitely large objects. The material is not limited: we advise the students to use *Logic in Action* [26], Wikipedia articles, Coursera courses, Khan Academy material and any other resources googleable from the internet or borrowable from the local library.

Another example is *Compiler Construction* which is assigned to applicants strong in modern software engineering techniques such as web app development with no prior exposure to system software and/or DSL/MDE. A student involved in this course is advised to get acquainted with the first chapters of the Dragon Book [2] and expected to be able to list the components of a typical compiler backend, assess their usefulness and role in performing a particular software language engineering task. Many students also turn to other books, YouTube videos, Coursera courses, etc.



```
?- eq('⊆'(a,b),'∀'(x,'→'('∈'(x,a),'∈'(x,b)))).
true .

?- eq('⊇'('∪'(a,b),c), E).
E = '∀'(_G282, '→'('∈'(_G282, c), '∈'(_G282, '∪'(a, b)))) ;
E = '∀'(_G15, '→'('∈'(_G15, c), '∨'('∈'(_G15, a), '∈'(_G15, b)))) ;
E = '∀'(_G15, '→'('∉'(_G15, '∪'(a, b)), '∉'(_G15, c))) ;
E = '∀'(_G15, '→'('∧'('∉'(_G15, a), '∉'(_G15, b)), '∉'(_G15, c))) ;
E = '∀'(_G15, c, '∈'(_G15, '∪'(a, b))) ;
E = '∀'(_G15, c, '∨'('∈'(_G15, a), '∈'(_G15, b))) ;
false.

?- 
```

Fig. 1. An example of checking an equivalence of two formulae (top) and generating possible formulae equivalent to a given one (bottom). The program behind both is shown on Figure 2.

3.3 Model-based admission

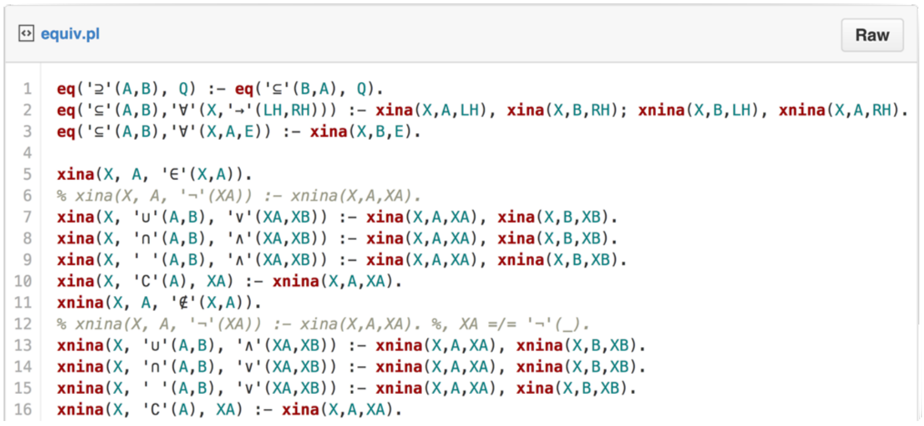
Consider the premaster course on logic briefly introduced above. Given that we can rely on technical knowledge of our potential students, we ultimately want them to see a connection between familiar activities and their formal methods counterparts. This is the case for different topics within the premaster material:

$$\forall x,\ x \in A \wedge x \in B \Rightarrow x \in C \qquad \Longleftrightarrow \qquad A \cap B \subseteq C$$

A quantifier-based formula on the left is universally equivalent to a set-based formula on the right, and there are many exercises in our examinations to either recognise such pairs or infer one if given the other. However, this is also the case for bridges between other areas and formal methods: e.g., substitution in a mathematical formula has always been a somewhat difficult topic, unless explained in terms of programming (bounded variables are local variables, unbounded are global, and thus it is much easier for practising programmers to come up with the idea of renaming in case of clashes on their own). Cardinality estimation skills, especially for infinite ($\aleph_0$ or $\aleph_1$) sets, is also usually assessed based on questions about the number of possible programs in a language, or chess moves, or sorting algorithms of particular structure.

Hence, we can systematically list these equivalence relations that we expect a student to confirm, and generate enough exercises (according to time constraints) to test them. Such exercises can be checked automatically. Furthermore, we can also make solid claims about coverage of the studied material by the assignments, both generated and answered correctly. Figure 1 and Figure 2 show a simple example.

Hence, we achieve drastic decreases of time spent on preparing the examination assignments (they are generated), better examination service in general



```
eq('⊇'(A,B), Q) :- eq('⊆'(B,A), Q).
eq('⊆'(A,B),'∀'(X,'→'(LH,RH))) :- xina(X,A,LH), xina(X,B,RH); xnina(X,B,LH), xnina(X,A,RH).
eq('⊆'(A,B),'∀'(X,A,E)) :- xina(X,B,E).

xina(X, A, '∈'(X,A)).
% xina(X, A, '¬'(XA)) :- xnina(X,A,XA).
xina(X, '∪'(A,B), '∨'(XA,XB)) :- xina(X,A,XA), xina(X,B,XB).
xina(X, '∩'(A,B), '∧'(XA,XB)) :- xina(X,A,XA), xina(X,B,XB).
xina(X, ' '(A,B), '∧'(XA,XB)) :- xina(X,A,XA), xnina(X,B,XB).
xina(X, 'C'(A), XA) :- xnina(X,A,XA).
xnina(X, A, '∉'(X,A)).
% xnina(X, A, '¬'(XA)) :- xina(X,A,XA). %, XA =/= '¬'(_).
xnina(X, '∪'(A,B), '∧'(XA,XB)) :- xnina(X,A,XA), xnina(X,B,XB).
xnina(X, '∩'(A,B), '∨'(XA,XB)) :- xnina(X,A,XA), xnina(X,B,XB).
xnina(X, ' '(A,B), '∨'(XA,XB)) :- xnina(X,A,XA), xina(X,B,XB).
xnina(X, 'C'(A), XA) :- xina(X,A,XA).
```

Fig. 2. The program behind Figure 1 is a trivial universal declarative model in several lines of Prolog: `http://gist.github.com/grammarware/813374043858030b2059`.

(assignments are individual and are not reused through the years), less time spent on assessment (automated grading) and better learning analytics (extensive use of coverage criteria can also eventually lead to developing an interactive system that keeps exploring knowledge areas and localising white spots in order to make the fairest estimation possible of the skills of the student under test).

The same model-based approach is useful in a flipped classroom scenario [18], where some technique is appreciated by students to test their knowledge at home after presumably acquiring new knowledge, but before coming to class to use it for training new skills and for getting detailed feedback from the instructors [29].

4 Threats to validity

Having praised the proposed method in previous sections of the paper and focused on its best sides, we can now finally identify several threats to its validity.

Creation effort of the models needed for model-based student assessment and admission might turn out to be higher than the effort usually spent on just getting to the same objectives with manual labour, especially if the quality standards allow reuse of the same course materials over the years. There is some evidence that similarly complex or even more sophisticated projects can succeed [14], but they still require a lot of resources to set up in the beginning.

Validation of specifications remains an issue reserved for future work of unpredictable difficulty. Estimating the challenge level of a particular set of tasks is one thing, but validating the claim that all tasks generated from a particular model, belong to a predefined complexity class, requires special attention.

Complete automation of model-based student admission may not be possible, even if deemed desirable. At this point we run all our intake procedures individually, investing several hours per student candidate. We also do most of the premaster course examinations orally because we feel that yields more fair assessment — the belief is supported both by student evaluations and by research published by other institutions [13].

Throughout the paper we have been talking about conditional admission and other ways to **filter out** students that are deemed unsuitable in one way or another for the Master's programme. The obvious possible flaw in this way of thinking is the chance of getting a false negative — rejecting a potentially valuable and talented student. An alternative would be a completely different approach: to accept everyone and challenge everyone individually — and it has been done before [28], demonstrating feasibility and being very reasonable for first years students [19]. Even though we are committed to having a consistent one-year programme with high work load, state of the art technical content and a pragmatic and critical view on the scientific foundation — a set of demands making intake filtering inevitable, we must acknowledge the fact that the design principles of the programme might not be perfect.

5 Conclusion

In this paper, we have considered borrowing expertise and techniques available in the field of software testing, to improve the quality of assessment of students enrolled in premaster courses. From the constructive alignment point of view [7], premaster courses are unique in a sense that they have loosely defined learning objectives, a virtually non-existing or at least unknown set of educational tasks and an absolutely crucial assessment procedure. As a proof of concept, we propose a setup where an equivalence model is formed around the learning objectives in such a way that a set of test cases is generated to be used as an input (i.e., exam) for the system under test (i.e., student). The automation enabled by this approach allows us to provide better testing services (i.e., fair admission) with less performance sacrifices (i.e., teacher time).

There are a lot of open issues at play here: the traditional lack of difference between installation testing and system acceptance testing (i.e., technical compatibility vs. operational compatibility) in educational research; the level of attention and caution one needs to exercise when attempting anything even remotely related to destructive testing and exception handling; formal specification and validation of coverage criteria; etc. However, given all the similarities spotted and reported in §2, we are tempted to exploit them, to align software testing methodologies with student assessment practices and borrow useful experience both ways?

In recent decades we have become good in software testing, so we should really test people the same way we test software.

References

1. A. Adams. Student assessment in the ubiquitously connected world. *SIGCAS Computers and Society*, 41(1):6–18, Oct. 2011.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
3. D. K. Allen, S. Karanasios, and A. Norman. Information Sharing and Interoperability. *European Journal of Information Systems*, 23(4):418–432, July 2014.
4. J. M. Bardwick. *Danger in the Comfort Zone: From Boardroom to Mailroom — How to Break the Entitlement Habit That's Killing American Business*. AMACOM, 1995.
5. K. Beck. Extreme Programming: A Humanistic Discipline of Software Development. In *FASE*, pages 1–6, 1998.
6. A. Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In *Future of Software Engineering*, FOSE '07, pages 85–103. IEEE CS, 2007.
7. J. Biggs and C. Tang. *Teaching for Quality Learning at University*. McGraw-Hill and Open University Press, 2011.
8. D. Blaheta. Reinventing homework as cooperative, formative assessment. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 301–306. ACM, 2014.
9. G. V. Bochmann and A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '94, pages 109–124. ACM, 1994.
10. G. Button and W. Sharrock. Project Work: The Organisation of Collaborative Design and Development in Software Engineering. *Computer Supported Cooperative Work (CSCW)*, 5(4):369–386, 1996.
11. C. Douce, D. Livingstone, and J. Orwell. Automatic Test-Based Assessment of Programming: A Review. *Journal of Educational Resources in Computing*, 5(3), Sept. 2005.
12. S. Fraser, K. Beck, B. Caputo, T. Mackinnon, J. Newkirk, and C. Poole. Test Driven Development (TDD). In M. Marchesi and G. Succi, editors, *Proceedings of the Fourth International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2003)*, volume 2675 of *LNCS*, pages 459–462. Springer, 2003.
13. H. Gharibyan. Assessing Students' Knowledge: Oral Exams vs. Written Tests. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, pages 143–147. ACM, 2005.
14. B. Heeren and J. Jeuring. Feedback Services for Stepwise Exercises. *Science of Computer Programming*, 88:110–129, 2014.
15. R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using Formal Specifications to Support Testing. *ACM Computing Surveys*, 41(2):9:1–9:76, Feb. 2009.
16. R. Jeffries. What is Extreme Programming? *XProgramming*, Nov. 2001.
17. J. D. Kindrick, J. A. Sauter, and R. S. Matthews. Improving Conformance and Interoperability Testing. *StandardView*, 4(1):61–68, Mar. 1996.
18. M. J. Lage, G. J. Platt, and M. Treglia. Inverting the Classroom: A Gateway to Creating an Inclusive Learning Environment. *The Journal of Economic Education*, 31(1):30–43, 2000.

19. R. Lister and J. Leaney. First Year Programming: Let All the Flowers Bloom. In *Proceedings of the Fifth Australasian Conference on Computing Education*, ACE '03, pages 221–230. Australian CS, 2003.
20. R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.
21. W. McKeeman. Differential Testing for Software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.
22. G. Melnik and F. Maurer. A Cross-program Investigation of Students' Perceptions of Agile Methods. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 481–488. ACM, 2005.
23. R. Mugridge and W. Cunningham. *Fit for Developing Software: Framework for Integrated Tests*. Robert C. Martin series. Prentice Hall, 2005.
24. J. F. Naveda, K. Beck, R. P. Gabriel, J. L. Díaz-Herrera, W. S. Humphrey, M. McCracken, and D. West. Extreme Programming as a Teaching Process. In D. Wells and L. A. Williams, editors, *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods*, volume 2418 of *LNCS*, pages 239–239. Springer, 2002.
25. A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression Testing in an Industrial Environment. *Communications of the ACM*, 41(5):81–86, May 1998.
26. Open Course Project. Logic in Action. http://www.logicinaction.org/.
27. D. H. Steinberg. Extreme Teaching — An Agile Approach to Education. In D. Wells and L. A. Williams, editors, *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods*, volume 2418 of *LNCS*, page 238. Springer, 2002.
28. P. Strooper and L. Meinicke. Evaluation of a New Assesment Scheme for a Third-year Concurrency Course. In *Proceedings of the Ninth Australasian Conference on Computing Education*, ACE '07, pages 147–154. Australian CS, 2007.
29. J. van Eijck and V. Zaytsev. Flipped Graduate Classroom in a Haskell-based Software Testing Course. In *Pre-proceedings of the Third International Workshop on Trends in Functional Programming in Education (TFPIE 2014)*, May 2014. http://wiki.science.ru.nl/tfpie/File:Tfpie2014_submission_16.pdf.
30. W. M. Waite, M. H. Jackson, A. Diwan, and P. M. Leonardi. Student Culture vs Group Work in Computer Science. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, pages 12–16. ACM, 2004.
31. M. Woehrle, K. Lampka, and L. Thiele. Conformance Testing for Cyber-physical Systems. *ACM Transactions on Embedded Computing Systems*, 11(4):84:1–84:23, Jan. 2013.
32. R. M. Yerkes. *The Dancing Mouse: A Study in Animal Behavior*. The Macmillan Company, 1907. https://archive.org/details/dancingmousestud1907yerk.