Flipped Top-Down is Systematic Bottom-Up

Zaytsev, V.

Publication date
2016
Document Version
Final published version
Published in
EduSymp 2015 : MODELS Educators Symposium 2015

Flipped Top-Down is Systematic Bottom-Up

Vadim Zaytsev, vadim@grammarware.net

Instituut voor Informatica, FNWI, Universiteit van Amsterdam, The Netherlands

Abstract. The paper presents an experience report in course design for a versatile group of computer science students where their needs were surfaced and met by the combination of strict top-down exposure to course material and the flipped classroom model of lecturing.

1 Motivation

Computer science departments in general have little to no trouble attracting students to their undergraduate (Bachelor-level) programmes. However, some suffer from a syndrome when a fraction of somehow misinformed students start their CS education only to gradually increase their disappointment up to the level of switching to a more suitable programme. When such leaks happen at later stages, they can be considered harmful to both students who lose their time figuring out the right match, and to the departments that invest teachers and other resources into students that never complete their education there. One of the ways of preventing such leaks involves moving hardcore computer science courses into the first year to deliver the right message and allow students an earlier chance of making up their minds.

At the University of Amsterdam, such a strategy in curriculum design leads to first year Bachelor students following courses on linear algebra, automata theory, discrete mathematics, as well as introductory software courses that teach them programming in-the-small and turn all too easily into introductions to various paradigms, programming languages and operating systems. Those students that know programming on a more advanced level, are left to their own devices and may develop their skills in their free time; many do.

By the time students reach second year, it becomes necessary to introduce them to the discipline of software engineering. This is accomplished by a course named *Project Software Engineering*. However, several problems block the easy way there, with the largest challenges being the following ones.

– Not a single aspect of MDE had a chance to manifest itself prior to this course. Part of the students were junior programmers experienced with low level tasks that never required any modelling to succeed; the others were non-programmers which were taught programming-in-the-small with throwaway code that was solving one S-type problem (e.g., “balance a tree”) in a very limited time to be left behind and forgotten right after submission.

Mon	Tue	Wed	Thu	Fri
Lecture 1: Intro Lecture 2: Agile	Demo 1: Pitch		Lecture 3: SWEBOK	
Lecture 4: Patterns	Demo 2: MVP	— — — *audits* — — —		
Lecture 5: FP	Demo 3: Working	— — — *audits* — — —		
Lecture 6: Testing Demo 4: Dry run				Demo 5: Final

Table 1. Summary of the course schedule

- The starting level of students varies greatly from those that know basic constructs and may, if the need arises, construct a simple website by copy-and-paste programming, to those that have coded Arduino kits or even worked part-time at software development companies as interns and junior software developers.
- Collaboration skills have received too little attention in the preceding courses, and almost none of it was dedicated to collaboration in technical environments, such as ones commonly found in software engineering projects when a number of people (greater than two) is working on the same codebase for longer periods of time.

The shape of the course was predetermined by the curriculum timeline and could not be changed: one month full time, no distractions, no conflicting activities. The rest of the properties were described in terms vague enough to make it fit any desired shape [18].

2 Course Design

The main principles in designing this course were the goals of full utilisation of the available knowledge (in the form of personal experience, industrial contacts, resources on the internet) and of maximum leverage of students' enthusiasm. The course was aimed to provide something they (felt to have) never received before: learning hands-on skills at own speed and immediately applying them in a real-life(-like) project.

2.1 Summary

Heterogeneous entry level is often mentioned to be a problem solved by the flipped classroom paradigm [12, 14], where typical classroom activities such as lecturing are done at home by consuming prepared videos and contact time is spent on discussions and typical homework activities such as applications of leant skills. Since one of the biggest threats to it were unmotivated students [1, 4] and ours were shining with enthusiasm, it was decided to give it a shot. The first lecture was given traditionally — to be precise, it was a block of two lectures:

one with the introduction to software engineering (as the domain as well as the course) and one explaining the agile methodology that the students needed to follow from the first day. By this the basics were covered and they could start working independently. For lectures 3–5 the material was provided a few days in advance with the obligation of sending at least one question to the lecturer. The actual lecture hours were then used to handle most of the collected questions. Besides the topics, there were distinctions in the methodology: for the third lecture, the questions were collected passively; for the fourth one, each student got a brief answer individually before or shortly after the lecture; for the fifth one, its topic was also left as a choice for the students. The last lecture was given by an external software engineering practitioner invited from the industry.

Additional to these 12 hours, around 10 hours were spent on weekly presentations about the progress in developing the actual project; each demo session had a clear purpose announced beforehand, each was graded. On weeks 2 and 3 each team needed to go through a round of audit (for a total of two per team) when they were visited by a "third party expert" that would ask difficult questions about their development process and also assess the progress.

In the centre of the course is the project itself: the students had complete freedom in choosing a topic, but needed to pitch the idea at the first demonstration session and regularly report on their progress. The teams were assembled by students themselves, with size limits imposed by the lecturer (6–8 people per team). All this totalled 41 hours contact time, the rest of 6 ECTS being preparation time for the lecturer and self study for the students, mutually supported by Slack communication.

There was no explicit examination at the end of the course: all intermediate stages were graded and the final grade emerged from the accumulation.

2.2 Lectures

Lecture 1: Introduction to software engineering was made intentionally *personal*: the concepts of imperative, structured, functional programming were explained with concrete examples from the personal history of the lecturer. Computational methods received examples from railway engineering [36], low level programming exemplified with methods of hacking and disassembling, mainframes — with COBOL migration projects, metaprogramming — with Rascal [13], databases and AI aspects demonstrated by successful Master student projects that the lecturer supervised. Yet, the lecture remained an overview. As a result, students remembered only those parts of it that interested them and asked many precisely targeted questions later.

Lecture 2: Agile software engineering [22] was introduced next at the first day so that the students would know where and how to begin working on their projects. This was done in a reasonably straightforward fashion.

Lecture 3: SE knowledge areas [23–28]

Software engineering was presented as a domain: its *domain model* was the Software Engineering Book of Knowledge [5]. Its chapters 1–5 and 10 were taken

and turned into 5–9 minute overview videos each. For the sake of correct counting these clips were not made publicly available until the course ended: videos received 100, 58, 47, 45, 40 and 46 views. Out of 51 students, 34 participated and asked 53 questions total through Slack and email — more than enough to fill in two hours of the lecture. Half of the time during the lecture was dedicated to the Master-level programme in software engineering[1], which is also designed in alignment with SWEBOK and at the same time allows to give very concrete examples of topics and contents.

Lecture 4: Paradigms, patterns, antipatterns [29–31]

The traditional way of introducing students to programming paradigms is exposure to a typical programming language that exemplifies it, such as Prolog for declarative programming / constraint solving. Instead of that, we went for a *megamodel* [19, 20] that introduced *all* known paradigms by explaining relations between them and decisions involved in designing different languages and choosing among them. A renarration [32] of this model was a 13 minute screencast.

For the patterns video, no such material was available, so the video was improvised based on the contents of the books on design patterns [8], implementation patterns [3], architectural patterns [3], with some attention devoted to millipatterns, micropatterns and nanopatterns as well [33]. The third part concerned negative patterns like code smells and bug patterns. The videos received 66, 64 and 50 views and yielded 76 questions from 47 students. This time each question was answered individually and the lecture itself was dedicated to the chosen few that occurred most often. Subjectively this was much better received by students who connected easier to direct answers sent through Slack and email than to handling them during the lecture.

Lecture 5: Practical functional programming [7]

The topics of the remaining two lectures were not fixed in advance. Instead, students could vote on what they wanted to see and could propose their own topics. Participation in voting was much lower than expected (14 voters, 27% participation), and as a result two topics emerged. The topic of practical functional programming had its roots in one of the discussions at an earlier lecture about our experiences in FP: the students had been exposed to Haskell and Erlang in a course meant to introduce non-traditional paradigms and were genuinely puzzled when told that FP languages are being used in practice and that FP constructs are present in almost every language (apparently writing decent Java 8 code was not a part of their Java programming course). The lecture itself consisted of a discussion on the chosen subset of 50 questions sent in by 31 people, as well as of code examples in various languages. Some time was dedicated to introducing various languages by explaining their use cases in the industry: Scala at LinkedIn, Clojure at Amazon, Groovy at Netflix, Erlang at WhatsApp, Arc for Hacker News, F# in Halo, etc.

Two people have explicitly voted to have no topic for the lecture and use the available time for their project — both have been informed that they are

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

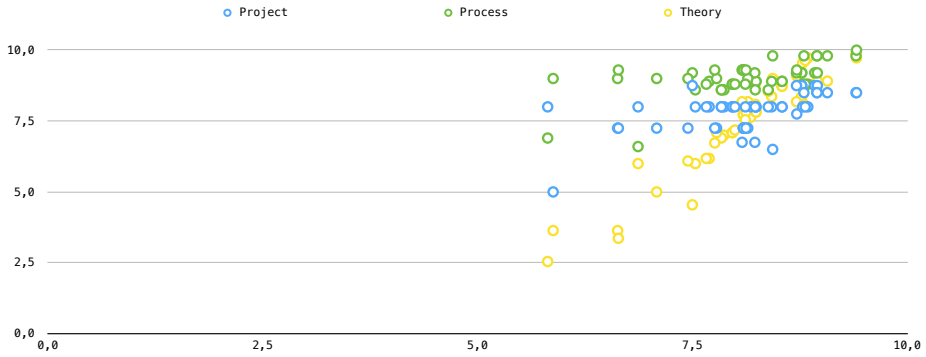

Fig. 1. Plotting the grades of three core course components against the final grade: project (blue), process (green), theory (yellow).

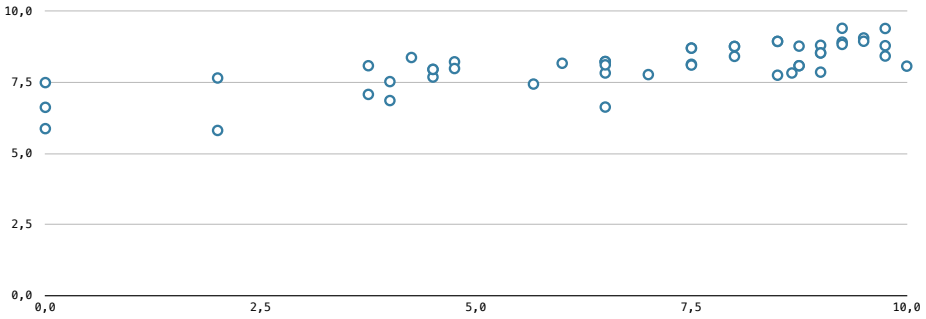

Fig. 2. Plotting the final grade (Oy) against the grades for the questions (Ox).

free to do so without consequences for the final grade; one of them still actively participated, sent good questions and attended the lecture.

Lecture 6: Automated software testing

The last slot was filled by a guest lecturer from a company specialised in model-based testing: he received 60 questions, structured his lecture with some regard to them, explained how his company works and gave a short live demo with Ruby, Rspec and Cucumber; all these parts were greatly appreciated and received warmly by the students — a typical reaction to industrial guest lecturers.

2.3 Demos

There was one preliminary meeting before the start of the course where students attended a specially organised pitch session at one of the local startup incubators. They had six weeks between the session and the official start of the course to think about their project and form teams of Scrum-compatible size. Seven teams

of size 5–8 were made, one student asked for an exemption and was granted it after an interview.

Demo 1 was on the second day of the course, a day after the first lectures. The teams were supposed to pitch their ideas in front of the audience which consisted of all the other students and several invited lecturers. Since they have seen real startupers pitch before, this went without much trouble. One team "mispitched": after having presented their idea, they proceeded to run experiments and those shown that the prototype was infeasible; they promptly switched to another idea.

Demo 2 was much more challenging: exactly one week later they had to present their Minimum Viable Product (MVP, the concept was explained at Lecture 2). The main challenge for them was to focus on something directly presentable right away instead of waterfalling their way from backend configurations and protocols eventually toward the user. This requirement was meant as a disruptive influence and as a way to kickstart thinking about possible software process models.

Demo 3 was meant to show clear and tangible progress and out of all demos looked the closest to a real Scrum weekly review.

Demo 4 was scheduled on the first day of the last week and was positioned as a dry run of the final demo. This was appreciated considerably, and the adjustments most teams made between the last two demo sessions, were very apparent.

Demo 5 was the last day of the course, where all complete teams presented the results of their month work.

The following projects were completed within the course:

- **Audioflame**, http://audiofla.me, an online audio editor frontend that logs in to the user's cloud storage, loads files from there and allows to adjust them in the browser and finally save the project or export it back to the cloud.
- **Payclouds**, http://payclouds.net, https://youtu.be/5VBcwrKkQLY, a webapp for distributing expenses within a group of friends.
- **Doko**, https://github.com/JoerivVuuren/Doko#doko, a gamified mobile platform for recording small scale debts within a group of friends.
- **Qluzr**, https://qluzr.nl, a gamified web app for distributing household duties and earning points and badges for their timely completion.
- **Vlakbijles**, https://github.com/Vlakbijles, an app for searching tutors based on their expertise, location and reviews.
- **Ariana**, http://ariana.pictures, a deliberately simplified webapp for user-friendly editing pictures in the browser.
- **Smart Address Book**, a heuristic-based contact data synchronisation platform among Google Contacts, LinkedIn and Podio.

2.4 Audits

Audits have held their place in software development for many years [6]. We have used them as an opportunity to expose the design, architecture, technological

decisions and implementation details to third party experts. Audit sessions varied in content: some were based on ISO/IEC 25010:2011, some involved code review, etc. The main role of an auditor in all these cases was to provide fresh informed perspective to the students and to ask difficult questions exposing weak sides of their projects: what is the state of your documentation? how did you test that your components work together? why did you use this framework? Six audit sessions were done by the present author, the remaining eight delegated to colleagues and Master students of software engineering.

2.5 Assessment

All the questions were graded individually: no question at all meant 0 points, a simplistic question that could have been answered by spending ten minutes on Wikipedia was 6 points, a reasonable question 8 and a perfect one 10, with ± 1 deviations for particular details and circumstances. Laconic summaries of the questions can be found on Table 2.

The final grade was the average of three grades for the core course components: the project (assessed as the quality of the final product, adjusted by auditors' feedback), the process (the average of the assessments of weekly demonstrations) and the theory (presence and participation at lectures, measured by the quality of answers provided per lecture). Figure 1 and Figure 2 relate the final grade with its components and with an average grade for questions.

After the course has ended and all the components have been graded, as usual, there were some students dissatisfied with their grades. They were given a chance to compensate for each of the components for which they got zero points, by elaborating on one of the problems briefly mentioned in the flipped lecture videos, by writing running code demonstrating the issue.

2.6 Evaluation

As typical for any course at the University of Amsterdam, it ended with an evaluation form filled in by students (40 submitted their forms). In general, the results were positive, with most items scoring around 7.0 out of 10 or 3.5 out of 5. On Figure 3 we present a part of the results in somewhat more detail and offer the following observations:

- The in-classroom lectures were not deemed very useful, even though their level was appreciated. In our case the classroom hours were indeed not crucial for consuming the material, but if they are to be dropped in the adjusted version of the course for the next year, there should be other ways found to enforce material consumption.
- The questions that the students had to ask after watching the flipped lectures, were assessed much more positively, but many people found them rather dull. Indeed, that was one of the worries we had during course design. Alternative ideas that seemed more exciting (such as tweet-summaries of the lectures), ended up not being implemented because they were too hard

to explain. Perhaps next time instead of (or together with) asking questions, students should answer them by recording their own videos? Active participation has been generally known to increase involvement, and there is evidence for student-generated content, specifically video content, to be particularly effective [11, 15, 17] and positively perceived by the students themselves [16].
- Weekly demos were not seen as incredibly useful. We dare attribute it to the general computer engineer mentality of preferring uninterrupted work in a cave to explanations and open discussions.
- Two aspects were noticeably different from the evaluation of the last year: the course felt a bit more practical (in the sense of learning skills that feel like they will be used later in practice) and somehow it felt more academically challenging and research-intensive. Last year the students were also left to their own devices for the project development part, but only received two lectures on rather randomly chosen topics (maintenance [35]) and entrepreneurship [34]). Apparently systematic top down exploration of software engineering domain starting with its areas into the paradigms, languages, patterns and processes, left the students feeling more sciencey.

The grade given to the instructors by the students was in reverse correlation with the instructors' teaching capacity: the ones scored the highest (upwards of 8.0 out of 10) were Master students supervised by the present author who were asked to serve as auditors; teachers scored lower, down to 6.0 for the most experienced one in educational matters.

Another interesting piece of feedback was collected from the "special remarks" section as well as direct conversation: students perceived the "post on Saturday morning — expect questions by Sunday night" model as much more relaxed and less invasive than "post on Wednesday — expect questions by Friday"; the expectations were opposite. We thought that giving assignments during weekends could be stress-inducing, but it was seen as preferred, possibly because watching educational videos was more compatible with a typical weekend schedule of Netflix and chill than with a typical workday.

3 Conclusion

Instead of attempting explicit elicitation of the needs of individual students in the hope to cater to them, we have designed this course in a strictly top-down fashion: the introduction was followed by an high level model of the domain (the highest official model we have of the entire software engineering [5]) and proceeded by handling other increasingly more detailed yet still irrevocably abstract models and megamodels of programming paradigms and good and bad practices. Yet, by combining such a design with the flipped classroom paradigm (where usual classroom activities like lecturing are completed individually at home and the traditional homework is treated during contact hours), we have achieved a good fit with the needs of a versatile group of students. We claim that this would not

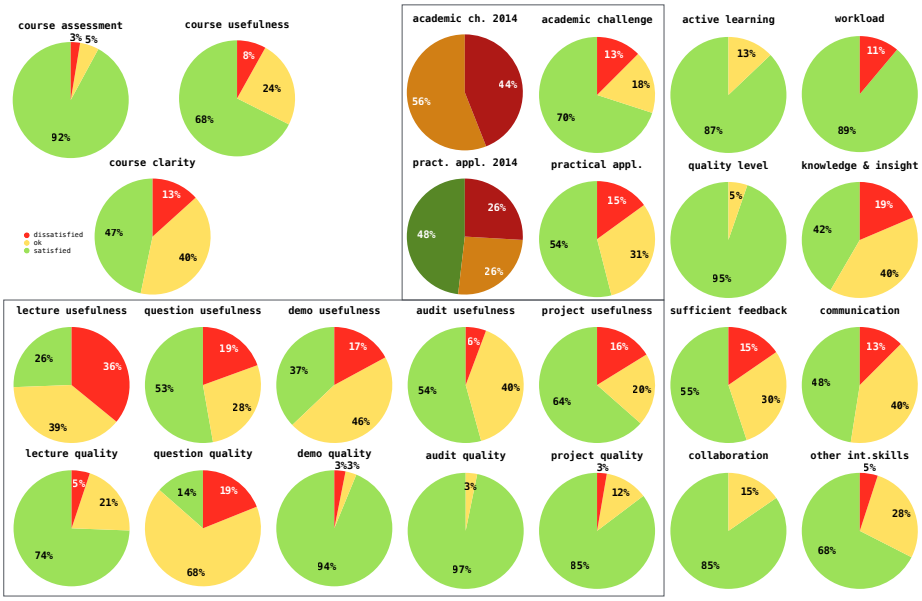

Fig. 3. Summaries of university-provided student evaluations of the course: main accumulated measurements in the top left corner; the lower selected block is usefulness/learnability and quality/level of course components; the right part refers to the skills learnt/improved during the course (so "communication" means learning to communicate your ideas to others, not communicating with instructors); the top middle block selects two aspects that differed most from the previous year. Green means "satisfied" or "deeply satisfied", red means two opposite options, yellow is the middle.

have been possible to achieve so easily otherwise, because students' initial lack of real life knowledge would not allow them to make good choices of the learning direction, and a straightforward let's-first-cover-the-basics approach would bore a considerable fraction of the students even if there was enough time for that.

The course was perceived by students as activating but there are a lot of options to optimise that aspect: peer assessment [17, 21], deeper integration of Master and Bachelor student activities [2], doing systematic pedagogical code reviews [10], using competition-inspiring benchmarks [9], etc. We have not decided yet which of the options to include in the next year's design and collecting such ideas is one of the main reasons to present this experience at the educators symposium.

Acknowledgement

Hans L. Dekkers was a coordinator of the *Project Software Engineering* course for many years and conducted many experiments of his own — they are not

reported here, but the form the course had in 2014 has left implicit impact on the current design. Robert Belleman and Robert van Wijk also shared their ideas on (re)design of this course, some of which were implemented. Robert van Wijk, Magiel Bruntink, Alan M. Berg, Carlos Cirello, Hans Dekkers, Leonard Punt and Timon Langlotz have graciously agreed to serve as external auditors for this course. Machiel van der Bijl from a model-based testing company Axini gave a guest lecture. Jeroen van Duffelen from startup accelerator ACE Venture Lab facilitated and sponsored the first pitch/brainstorm session. Aimy Eyzenbach conducted the evaluation by codesigning the survey form with the present author and processing the collected results.

51 UvA students of teams Audioflame, Clownvissen, Kite, Pannenkoek, Softwear, Swaggerboys and Team Two have ultimately shaped this course and made it educating and fun.

References

1. K. Ash. Educators View "Flipped" Model With a More Critical Eye. *Education Week*, 32(2), Aug. 2012.
2. G. Bavota, A. De Lucia, F. Fasano, R. Oliveto, and C. Zottoli. Teaching Software Engineering and Software Project Management: an Integrated and Practical Approach. In M. Glinz, G. C. Murphy, and M. Pezz, editors, *ICSE*, pages 1155–1164. IEEE, 2012.
3. K. Beck. *Implementation Patterns*. Addison-Wesley, Nov. 2007.
4. J. Bergmann and A. Sams. Before You Flip, Consider This. *Phi Delta Kappan*, 94(2), Oct. 2012.
5. Guide to the Software Engineering Body of Knowledge, Version 3.0, 2014. http://www.swebok.org.
6. S. G. Crawford and M. H. Fallah. Software Development Process Audits — A General Procedure. In M. M. Lehman, H. Hnke, and B. W. Boehm, editors, *ICSE*, pages 137–141. IEEE Computer Society, 1985.
7. N. Ford. Introduction to Functional Thinking. http://player.oreilly.com/videos/0636920030416?toc_id=152336, https://secure.trifork.com/dl/goto-ams/2015/Slides/Neal_Ford_Functional_Thinking.pdf.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable OO Software*. Addison-Wesley, 1995.
9. T. Hassner and I. Bayaz. Teaching Computer Vision: Bringing Research Benchmarks to the Classroom. *ACM ToCE*, 14(4):22:1–22:17, 2015.
10. C. D. Hundhausen, A. Agrawal, and P. Agarwal. Talking About Code: Integrating Pedagogical Code Reviews into Early Computing Courses. *ACM ToCE*, 13(3):14:1–14:28, 2013.
11. P. Karppinen. Meaningful Learning with Digital and Online Videos: Theoretical Perspectives. *AACE Journal*, 13(3):233–250, 2005.
12. A. King. From Sage on the Stage to Guide on the Side. *College Teaching*, 41(1):30–35, 1993.
13. P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the Ninth International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 168–177. IEEE Computer Society, 2009.

14. 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):pp. 30–43, 2000.
15. M. J. W. Lee, A. Chan, and C. McLoughlin. Students as Producers: Second Year Students' Experiences as Podcasters of Content for First Year Undergraduates. In *ITHET*, 2006.
16. A. Luxton-Reilly, P. Denny, B. Plimmer, and R. Sheehan. Activities, Affordances and Attitude: How Student-generated Questions Assist Learning. In *ITiCSE*, pages 4–9. ACM, 2012.
17. J. Tritz, N. Michelotti, G. Shultz, T. McKay, and B. Mohapatra. Peer Evaluation of Student Generated Content. In *LAK*, pages 277–278. ACM, 2014.
18. Universiteit van Amsterdam. Project Software Engineering. http://studiegids.uva.nl/xmlpages/page/2015-2016/zoek-vak/vak/19112, 2015.
19. P. Van Roy. The principal programming paradigms. https://www.info.ucl.ac.be/~pvr/paradigms.html.
20. P. Van Roy, J. Armstrong, M. Flatt, and B. Magnusson. The Role of Language Paradigms in Teaching Programming. In S. Grissom, D. Knox, D. T. Joyce, and W. Dann, editors, *SIGCSE*, pages 269–270. ACM, 2003.
21. A. Vozniuk, A. Holzer, and D. Gillet. Peer Assessment Based on Ratings in a Social Media Course. In *LAK*, pages 133–137. ACM, 2014.
22. V. Zaytsev. L2: Agile. http://grammarware.net/slides/2015/pse-agile.pdf.
23. V. Zaytsev. L3: Construction. https://youtu.be/40_9Py8L7y8, http://grammarware.net/slides/2015/pse-swebok-construction.pdf.
24. V. Zaytsev. L3: Design. https://youtu.be/R7kZZbmx1CE, http://grammarware.net/slides/2015/pse-swebok-design.pdf.
25. V. Zaytsev. L3: Maintenance. https://youtu.be/WOvfJH4uizs, http://grammarware.net/slides/2015/pse-swebok-maintenance.pdf.
26. V. Zaytsev. L3: Quality. https://youtu.be/A6MWv19HlU0, http://grammarware.net/slides/2015/pse-swebok-quality.pdf.
27. V. Zaytsev. L3: Requirements. http://youtu.be/9kkmr_aVdx0, http://grammarware.net/slides/2015/pse-swebok-requirements.pdf.
28. V. Zaytsev. L3: Testing. https://youtu.be/q3kydDeIj6g, http://grammarware.net/slides/2015/pse-swebok-testing.pdf.
29. V. Zaytsev. L4: Antipatterns. https://youtu.be/6lO8qQp4w14, http://grammarware.net/slides/2015/pse-antipatterns.pdf.
30. V. Zaytsev. L4: Paradigms. https://youtu.be/lqmMqtgWpms, http://grammarware.net/slides/2015/pse-paradigms.pdf.
31. V. Zaytsev. L4: Patterns. https://youtu.be/lBCEVFbB6Yo, http://grammarware.net/slides/2015/pse-patterns.pdf.
32. V. Zaytsev. Renarrating Linguistic Architecture: A Case Study. In *MPM@MoDELS 2012*, pages 61–66. ACM, Nov. 2012.
33. V. Zaytsev. Micropatterns in Grammars. In *SLE*, volume 8225 of *LNCS*, pages 117–136. Springer, Oct. 2013.
34. V. Zaytsev. Software engineering in startups. http://grammarware.net/slides/2014/startups.pdf, 2014.
35. V. Zaytsev. Software maintenance. http://grammarware.net/slides/2014/maintenance.pdf, 2014.
36. L. Zaytseva and V. Zaytsev. Methods and Tools for Predicting Working Modes of Railroad Power-supply Systems. In *RCM*, pages 63–66. IET, Nov. 2006.

SWEBOK	Q3	Paradigms, patterns, antipatterns	Q3	Practical FP	Q5	Automated testing	Q6
testing objectives	6	patterns vs paradigms	10	best FP language	8		0
	0	is singleton antipattern	10	monads	8		0
purpose of unit testing tools	7	subconscious antipattern use	9	FP smells	10	automated vs manual testing	10
validation vs verification	7	agile vs dirty	9	FP vs sideeffects	10		0
	0	classify C++	8		0		0
slicing, terminology	7	patterns vs copy-paste	9	transition from imperative to FP	10	risk-based, techniques vs tools	6
	0	avoid dependency	8	FP performance, debugging	10	race condition, test limits [late]	8
enough testing, deploy often	8	copy-paste vs reuse, config mgmt	10	when to FP	8	quickcheck for non-FP languages	10
reverse vs maintenance, debugging	10	90/90, patterns outside OO	9	why erlang at whatsapp	8	test to find requirement gaps	10
	0	why FP unpopular, pattern choice	9	when to FP, how to combine	8	GUI testing	9
designer vs code	8	patterns vs antipatterns	10	need for FP	8	how much of time is testing	6
security, performance vs simplicity	10	Dijkstra vs patterns, big teams	10	functional vs logic programming	8	test coverage claims	10
scrum roles vs SE domains	6	MVC downsides, clone detection	10	FP vs SE, FP vs AI	7	tests vs quality, black-box metrics	9
	0	dev time vs cpu time	10	provers / solvers	10	FP antipatronen, C test framework	10
	0		0		0		0
UX vs long actions	9	Strategy vs Command	7	recursion in FP vs in imperative	10	*(absence approved)*	
SE master	8	FP frequency is large projects	9	*(absence approved)*		proofs vs tests	10
security	8	Strategy vs State, refactoring	8		0		0
reuse public code commercially	9	patterns turning into antipatterns	10		0		0
test cases vs product quality	9	premature optimisation	8		0		0
costs of process adoption	9	clones vs feature envy	7		0		0
	0	patterns in PSE, 90/90	9	tasks for FP in the industry	8	product testing	9
design vs agile	10	premature optimisation	8	FP for normal people	7	testing vs guarantees	9
quality standards, patterns vs reuse	10	non-determinism, paradigm choice	10	when FP, what FP, code style	10	frameworks, terminology, strategy	9
quality of testing	9	patterns vs antipatterns	10	FP performance	10	properties to test for	10
test limits as key issue	8	state vs stateless, pattern choice	9	FP pro et contra	8	terminology, test limits	9
unit vs integration vs system	9	paradigms vs languages, anti vs anti	10	FP speed, utility, examples	9	test automation	10
architecture: sw vs hw, unit tests	10	monkey patch	10	FP performance	10	pos vs neg testing	9
best doc in agile	10	limits of paradigms	10	FP learning curve	9	test coverage metrics	8
	0	FP performance	10	immutability & free concurrency	10	self-testing testing software	10
	0	AbstractF vs FMethod, API smells	8	FP & concurrency	10	automation vs coding, big data	10
	0	antipattern: negative or positive	8	*(absence approved)*		QA in the process	9
	0		0		0		0
	0	milli vs micro vs nano	7	hassle of immutable data	8		0
	0		0		0		0
mutation, GUI, tech debt, bugs	10	monads, new paradigms, vs patterns	10	arrows, FP & scale, FP vs proofs	10	GUI, strong typing vs testing	10
	0	why choose to use antipatterns	6	dev speed in Java vs Haskell	9		0
internal vs ext testing, dev reqs	10	UML diagram, pattern mashup	8	FP & memory management	8	GUI testing	8
user interaction modalities	9	learning patterns	8	easiest language to learn FP	9		0
formal analysis of requirements	8	cargo cult programming	10	moving parts & OO vs FP	8	testing visual output	9
devs vs SWEBOK	10	logic sub-paradigms, antipatterns	10	FP typical tasks, optimisation	10		0
webapp testing	9	code smells in large projects	10	is FP often used?	7	GUI automation	9
	0	FP advantages	7	FP performance (recursion)	10	test coverage: when is enough	9
how much testing is enough	8	wrong pattern conseqs, clone size	9	expressiveness FP vs imperative	10	manual vs (manual) unit testing	8
	0	pattern choice	8	FP performance (immutability)	10		0
when to refactor, how to postpone	10	pattern choice	8	when to FP, lambda functions	9	testing vs product quality	10
conceptual modelling, testability	9	sequential coupling	9	FP vs imperative (abstraction)	10	regression testing	8
verifiability vs readability	10	why Prolog close to C	8		0		0
formal analysis, PSE specifics	8	nanopatterns, rule of credibility	8	FP vs imperative (popularity)	7	when is enough	7
	0	pattern choice	8		0		0
regression & guarantees	10		0	FP expressiveness, FP in Java	9		0

Table 2. Question summaries and grades for all students; reported per team.