On How We Can Teach – Exploring New Ways in Professional Software Development for Students

Walter Kriha, Tobias Jordine

***Abstract* — Requirements and approaches for introductory courses in software development at universities differ considerably. There seems to be little consensus on which languages are a good fit, which methodologies lead to the best results and especially which goals should be chosen. This paper takes a look at current approaches and difficulties at our own faculty – computer science and media at the Stuttgart Media University – and explores a combination of teaching techniques which seem to make a difference. The most important change was to switch to a project-based approach instead of the usual exercises given to students after a lecture. The second one is the flipped classroom approach with micro-exams at the beginning of lectures. The third one is an emphasis on professional tools to be used during the project. We also try to achieve a concept-based approach using e.g. modelling techniques to get a better understanding of source code control and build. And finally, we work as a team of two lecturers which allows us time to reflect on how we do things and creates new ideas frequently. None of those approaches is without problems as we will show, and we have met with some critique in our own faculty. The paper is explorative, based mostly on observations and feedback from students, but we intend to get some quantitative results as well in later publications.**

***Index Terms*—Computer science education, Object oriented programming, Object oriented methods**

I. Introduction

In this paper, we describe several technological and organizational changes we made to software development at HdM (Hochschule der Medien, Stuttgart, Germany) in general and in a second term advanced software development class specifically. We will report the observations we made and the feedback from students and the faculty.

The first chapter describes the history of software development classes at HdM and how they fit into the big picture of the curriculum and the forces from the industry. The second chapter gives a more detailed view of a second term class in software development and the changes that were made during roughly three years.

The following chapters describe some of the changes in detail, like the process orientation, flipped learning, professional tool chain, concept-based education and finally team-teaching effects. The final chapter tries to create the big picture of courses and approaches toward software development in our faculty.

II. Setting the stage: Software Development at HdM

Roughly three years ago the lecturers for the first and second term software development classes changed and the lecturer for the first term (SD1, Software Development 1) switched from C to Java and introduced so called flipped learning techniques. Standard handbooks for Java were selected [3], [4], [5] and students have to read a chapter for every lecture in advance. The lecturer also uses Maven, unit-tests and Git to organize exercises and also the final exam is written test on a PC.

This is followed by an advanced Java class in the second term which is an 8 ECTS/6 hours class with 2 hours lecture and 4 hours exercises in the computer lab. It has two lecturers – an experienced professor and a PhD candidate. Due to the switch of the programming language from C to Java in the first term, the second term class had to be re-designed anyway and both lecturers made their areas of expertise part of the curriculum. The first plan for the new class looked like this:

- Repetition from the first semester
- Objects and classes
- Inheritance
- Collections
- Exceptions
- Inner classes
- UML
- Git
- Threads
- GUI with JavaFX, Event Handling
- Streams
- Generics

For every topic special lab exercise was initially designed. The class soon followed the flipped learning approach of the first term. To ensure that students read the chapters of the handbook, micro-exams of about 15 minutes were conducted at the beginning of some lectures.

The team-teaching approach of both lecturers quickly showed its strength and resulted in major improvements for the class as the lecturer who is not currently active can observe the students and how they react on certain things. He can support the active lecturer and add additional information. Things which do not work are quickly detected and fixed.

The fusion of two lecturers' expertise resulted in a course with much more emphasis on software architecture, scalability and professional development techniques than before. Initially,

it was not even obvious to the lecturers, how big a change this would finally be. But it turned out that the word “advanced” for the second term course did not mean advanced syntax but syntax following design following software architecture. While still very practical due to the project work, the course itself is more top down starting with architectural issues and going over to design and implementation. This sounds like overkill initially, but many modern features of programming languages and tooling only make sense when they are used in the context of architectural requirements.

And it quickly came apparent, that the usual exercises were not really able to support those topics. Interfaces – originally a topic of the first term – had moved together with others like collections into the second term where they fit much better. But how and especially why do students learn about interfaces without the need of an extensible software architecture?

It became clear, that the only kind of exercises that would fit to the advanced topics of the second term were real but small projects. The students would form teams of ideally three to four persons, choose a project from a set of project ideas and implement it.

The final project evaluation is done with a spreadsheet containing 10 required techniques (interfaces and inheritance, package structure, documentation, testing, GUI, logging and exceptions, UML, threading, streams and collections, factories) which each project needed to implement to get points (0-3 points per required technology). The points from the project form 30% of the final grade with the rest coming from an end term written test (no online exam, no programming on a paper sheet, just concept questions). In effect, this means that with a decent project result it is almost impossible to not pass the complete exam, considerably reducing anxiety and fear for the students. The lecturers like the fact that a professional attitude shown by the students gets rewarded in a transparent way.

The introduction of projects instead of exercises caused many changes with respect to organization, participation, content, order and evaluation of the course. It turned out to be absolutely crucial that the current state of the student-projects is in lockstep with the topics handled in the lecture. The mistakes the lecturers made are described in the chapter on “Team Projects instead of Exercises” below.

Feedback from students indicates that this was the most successful change made to the class.

The current order of sessions in the course looks like this:

- UML
- Git
- Inheritance & Interfaces
- Collections
- Unit testing
- Logging
- Exceptions
- GUI with JavaFX
- Inner classes & Event Handling
- Threads
- Java Streams
- Generics

As a consequence of the new project structure, reviews are frequently conducted during lecture time and after some initial reservations students quickly realize how much a public discussion can improve the model of an application and the application itself.

The course gets very positive evaluations from students and mixed comments from faculty members. Some critique is basically unavoidable due to the fact, that the focus is on Java and C skills were abandoned. It was recognized that lecturers often tend to underestimate how quickly things are forgotten after a three-month break, especially practical skills and students frequently underestimate what they learned in a previous class and often claim “that they didn’t learn anything”. This is part self-defense as topics get repeated instead of new topics introduced and part self-assessment problem [6], [7]. In the second term class the repetition-sessions were dropped completely. They were found to be much too passive and the projects automatically forced students to close knowledge gaps from the first term. The critique from faculty staff certainly needs further investigation and also some quantitative research.

The following chapters discuss some aspects of the new class in detail.

III. Flipped Learning in Software Development

The lecturers decided to try “peer instruction”, an approach similar to the “flipped classroom” concept. Peer Instruction is a specific pedagogical practice defined by Eric Mazur at Harvard University [8]. More information and teaching materials can be found at [9].

What were the reasons behind such a didactic change? The SD1 lecturer had started to use a flipped classroom concept and turned into an evangelist for it. A practical reason was the lack of a lecture script for the new topics in SD2 (Software Development 2) and the lecturers started to question their roles in the course: Is it really worth to create a new script for a basic course in software programming when a lot of excellent handbooks and teaching materials already exist? And even more important: Is the time spent in telling basic know how in a lecture really well spent?

“Flipped classrooms” emphasize pre-lecture work done by students. Usually they need to read a chapter in a book to prepare for the next lecture. If done properly, the lecture time turns into the discussion of problems students were facing. Time spent clarifying concepts behind problems is much better spent than with reading basic information to an audience.

The essence behind is also clear: How do you get the students to read the material in advance? And how do you evaluate the quality of the preparation?

The lecturers decided to use micro-questionnaires with 6-10 short questions about the chapter. At the beginning of the lecture students got between 15 and 20 minutes time for the answers. As this is all quite experimental, no digital support platforms like Ilias or Moodle are used yet.

The lecturers observed different behaviors which turned out to be quite stable across terms. Initially, a larger number of students turn in empty questionnaires. Obviously, they didn’t

read the material and were unable to answer the questions. It quickly became obvious as well, that they were not able to follow the discussions during the lecture. With a flipped classroom concept, the lecture does not repeat the content of the handbook chapter. The lecture has its focus on conceptual problems.

As soon as some sessions skip the questionnaires, morale drops even further. This points to some fundamental problems with the approach and they mostly have to do with resources. As student-tutors are not allowed to grade other students work in Germany, the two lecturers are unable to grade the micro-questionnaires. The answers are only used to get a feeling for which topics the students might have problems with. Many students try to understand the major topics by simply following the discussions during the lecture – which turns out to be impossible with difficult topics like multi-threading. The fact that due to project work it is quite hard to fail in the final written examination, many students gamble and try to get by without much pre-lecture work.

The lesson learned is clear: the micro questionnaires need to be evaluated every time. And there is another reason to do so, which became very clear in a different seminar [10] that used a journal-club approach: Even students who read the pre-lecture materials complain frequently, that they are unable to answer many of the questions. Why is that? And how should the questions look like? There needs to be some kind of reward for reading the materials and so students should be able to give answers to some questions. If you did not read the material, you should not be able to answer the questions and that is usually the case. But there needs to be some questions which challenge the understanding of difficult parts of the material. Those parts which beginners easily skip until they learn, that there will be questions exactly about those parts. This is a training process and according to our experience students need to read around 5-6 papers or chapters until they learn to actively question the content. For a detailed description of such a process in a course on concurrency and parallelism see [10].

The difficult questions are also a hint toward the final written test and make students aware of complicated concepts which need to be understood.

Having constrained resources, the experiences with a flipped classroom concept in software development are mixed: The concept shows its strength better in small journal-club seminars, where a small group of like-minded students and their lecturer try to understand new topics. Still, for SD2 saved valuable time by using the flipped classroom concept. They did not have to write a new script or explain the absolute basics in the lecture. The time saved is invested in the possibly most successful change that has been made to SD2: team projects.

IV. Team Projects instead of Exercises

Before projects as a means in learning software development are introduced, a few statements on programming are in order. Both lecturers of SD2 were convinced, that developing software in itself is a sensually satisfying activity if done in the right context. Time pressure or single exercises do not provide the right context: An exercise sheet might belong to the context of a specific lecture, but over the whole term it is completely standalone and out of context and this can affect motivation Learning how to develop software also means to learn the syntax of a programming language. This can only happen through the use of the language. No amount of syntax demonstrated in lectures will turn somebody into a programmer and that is the reason why the lecturers decided to leave the learning of syntax as a task to students – supported by stack-overflow, the support of lecturers and the newly introduced practical projects. And finally, software development is teamwork and that is another reason why it can be a very satisfying experience.

The final push towards the use of projects in software education came from recognizing the limits of exercises in advanced software topics. The syntax of e.g. Java Interfaces is trivial, their use goes deep into software architecture. How could one learn the use of Interfaces through single exercises? And the same goes for logging, exceptions and many other topics of advanced programming language use.

Students liked the idea of projects immediately and several open issues needed to be solved right away: Team building, project type, deadline and how the project result would influence the final exam.

Team building was and still is a rather ad-hoc process which starts already in the first session of the term and needs to be completed quickly. 2-3 students per group are the norm because larger groups encourage “free-riding” students. Mostly, this process works quite well, and students manage by themselves.

Initially, we had a small list of project types (note taking app, address management tool., customer relationship app, sports competition manager etc.) to choose from, but we learned to be less restrictive and let the students basically choose whatever type of project they want. This led to much more game development and advanced project ideas. Frequently, students notice quickly that they had taken more than they could achieve, which is an important lesson to learn in software development. The lecturers ensure, that not too much time is wasted and shortcuts to the project goals are suggested where needed.

The deadline became the day of the final written examinations at the end of term. At this day, the groups send a link to the Git repository to the lecturers and further development is frozen. The grading process starts now and the time the lecturers saved by not writing their own lecture notes is now spent several times, as all the projects need to be evaluated. The whole process is very transparent and guided by spreadsheet, introduced in the above. This spreadsheet contains ten required features and the degrees of fulfillment for each. Zero to three points per feature are possible. Three points are usually related to an architecturally sound solution e.g. for exceptions and logging or building an extensible base framework with interfaces. The lecturers check out the projects and run a couple of shell scripts to detect certain features in the code. The applications are started to evaluate the GUI development. In most cases the students know the final result already in advance, as they can do the same evaluation since they can fill the same spreadsheet by themselves. This is certainly a lot of work for the lecturers in addition to the final written examination of one-hour length. But both lecturers

consider it worthwhile as looking at the source code of the projects makes deficits in the lecture visible.

Mistakes were made, too. The term was started with a repetition of first term topics over roughly 3 weeks. Then new topics were introduced in the lecture and finally the projects were started. This made the projects very late in the term and caused massive overload at the end. We realized that the repetitions were actually not needed or would happen automatically as part of the project work. In addition, it was realized that the order of the topics was required to be reorganized to start project work early on in the term. Ideally, a topic that gets treated in a lecture should be implemented right afterwards in a project. By using such an approach, the topic is still fresh in the mind of the students and a short and successful implementation afterwards can be fun. It is also very beneficial to do this in the regular exercise time of the course, as both lecturers and tutors are available during that time.

But not all topics are equal. While the approach described above works very well e.g. for implementing and using a logging framework or starting with an exception architecture, building the GUI takes quite a while and effort and tends to separate lecture topics from implementation for several weeks. The solution was to start with project planning work and modeling in UML. Afterwards, all topics that can be implemented quickly are covered and only after every team has a baseline implementation of the model we start with GUI programming. It is recognized that this will need more time for the implementation and during that time we do reviews and more theoretical concept work in the lecture. After several improvements, the projects terminate in time and without overload. The results are usually quite good, sometimes exceptionally good.

It needs to be stated that the use of threads in a shared state context are still a problem with this approach. It was noticed that while the students are able to answer conceptual questions on threads in the written exam, they are unable to transform those threading concepts into their projects in most cases. Two different approaches will be used in the next term: First, a special code review session after the threads lecture will be conducted. This way students will be able to see and discuss different implementations. Second, a short code example with a deadlock caused by nested synchronize statements will be given to them to fix. Past experience has shown that frequently sleep() statements are used to delay execution of one thread or that the nested synchronized statements are separated, thereby destroying the transactional safety.

Some interesting observations were made over roughly six terms: With regular exercises, the computer lab tended to be quite empty towards the end of the term. Often several weeks before term end students did no longer show up for exercises and concentrated on preparations for the final exams. This changed considerably with projects. A constantly good attendance is observed e.g. because some teams need help for their implementation. In addition, experienced teams trying to improve their project even further and sometimes the didactic purpose of the project needs to be made clear again. Those better students would possibly not attend the exercise sessions without projects at all. They find the general exercises either boring or trivial and tend to attend only the lecture. But since they are able to choose their own project, they can now pick advanced topics (e.g. some distributed client-server idea) and still get help when it is needed.

Regular public reviews are done during lecture time. Students quickly learn how valuable those are and lose their shyness when presenting their UML models or code.

In course evaluations students claim a much better motivation due to the project structure of the class. The amount of free-riding that happens is still unclear. By keeping the groups small and making the written exam a bit more important again it is tried to counter this effect. This – and the overall effects and side-effects on the computer science curriculum will be discussed below.

A disclaimer on the use of the term “project”. The term is used here in a very special way. In this paper, the “project” term means a tightly guided goal for a team of students with a purely didactic purpose and with fixed time slots in the lab where support is always available. Completeness (besides the ten required features) is not required nor beauty (even though students spend quite some time and emotion on graphical designs). The projects have nothing to do with projects in later terms, where teams develop independent solutions. This type of loosely running project would not work in early terms.

Introducing new things in a course is not side-effect free. Student workload and motivation need to be monitored carefully across classes in a term. In discussions with the operating-systems lecturer it was identified that most of the students were using OS-exercise time to work on their software development projects. Students show a very high motivation in projects but there is only limited time available in a single term. This means that projects need to be kept small and few, otherwise students won’t be able to cope with the workload and fail in other courses. Project-oriented courses leave room for purely theoretical courses as well and there is no reason to inject every course with a project structure. In case of workload problems, it is recommended to start a discussion in the faculty about shifting courses.

V. Professional Tooling

It has already been mentioned that SD2 works its way top-down from architecture and model to syntax and implementation along the lines of a real project. The lecturers of both SD1 and SD2 decided to expose students early on to a set of standard tools and processes. This included GitLab for source control and teamwork, Maven for automatic builds and environment control and unit-tests (JUnit 4) for better code quality. The second term adds usage of a logging framework (log4J 2), UML 2 modeling and a GUI framework (JavaFX).

This is a lot to digest and it has raised some critique by staff members which claimed that students are simply overwhelmed and without a proper understanding of those tools. There is certainly some truth behind this critique. Students start using tools without an understanding of the technology behind or sometimes even the purpose they serve. This is countered with a concept-based approach in the second term

The question is: Can professional software development be

taught with a focus on the language only – or is it required to include architecture, tooling and process from the beginning? In other words, there are two very different views on learning software development about. The first one, the traditional one, starts with language and adds tools, design patterns, architecture, distribution and databases later on. In the context of this paper this is called the "burger model" because it adds things like the stack of a hamburger. The other one, the "progressive image model", starts with everything at once but with a low resolution resulting in a coarse-grained picture. And piece by piece more detail is added and the whole picture becomes visible. Both models can come to the same end, but they take a very different way.

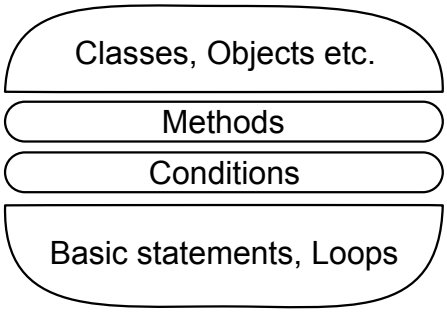

Figure 1: Example for the burger model

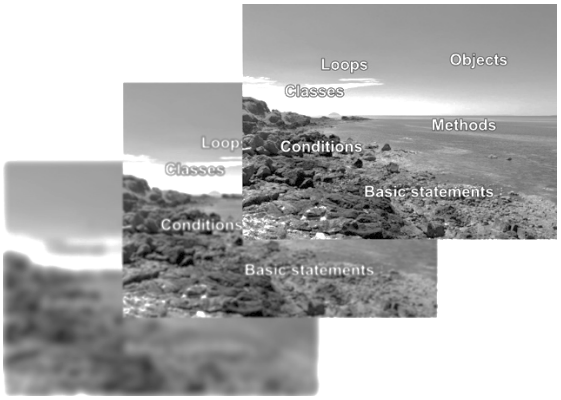

Figure 2: Example of the progressive image model

The burger model is probably much easier to organize as it requires less cross-cutting topics between classes. An integrated view of all topics might be easier to achieve with the progressive image model. Here is not the place to discuss the advantages or disadvantages of both models. In reality, there will be probably anyway a mixture of both approaches. But it has to be clear that whichever you chose will have a major impact on the organization and structure of later classes in your curriculum.

VI. Concept-based Teaching and Learning

The choice of language – in this case Java – is only a tool. It is used to demonstrate the OO-paradigm and in the first term also some procedural ideas. Java streams are used to introduce the functional paradigm and at no time we claim superiority of any paradigm. Instead, it is pointed out to the students, that they should invest in the functional and procedural paradigm as well and offer classes in functional programming and C/C++. And that is probably the only way to keep other lecturers happy as well who need different paradigms in their courses. Peter van Roy and Seif Haridi give an example for concept-based understanding in [13].

When teaching and using Git, the concept of generic source control is explained by building a model of Git in UML. Some design patterns (e.g. Singleton, Factory) are introduced early in the second term – one could almost say that students find them by necessity when discussing interfaces etc.

Overall, the second term software development class has turned into an incubator for advanced concepts which need to be extended in later classes.

A. Team teaching

From a managerial point of view, spending more than one lecturer on a class is a waste of resources. While this may be true in economic terms, our case has shown that team teaching can improve a class in major ways. Different expertise and experience allow different point-of-views for both students and lecturers. Lecturers can observe the process better and make suggestions. They can even play devil's advocate and express different opinions and preferences. And the overall success of the class can be discussed and regulated in case of problems.

From the experiences gathered in six semesters, it is suggested that at least one of a professor's classes should be held in team teaching form to allow sharing of didactic approaches and the joint development of new ideas and experiments.

The new structure of SD2 would not have been possible without team teaching.

B. Evidence

So far, some observations from the lecturers and some critique from colleagues in our faculty about "students unable to program a loop" in later terms were collected. Students feedback will be evaluated below. Empirical validation of statements in software development are notoriously difficult [12]. Given the two models above for learning about software development it seems to be obvious that they will produce different results at different points in time. The burger approach with its clear boundaries might produce better test results per layer, the progressive image approach provides a better integrated view. In other words: Both will prefer and generate different questions with different qualities. How can these two approaches be compared?

Currently, with a mix of burger/progressive image style classes, with new courses in Functional Programming and C/C++ and Cloud-Computing, it is almost impossible to compare results.

But what about results in written exams? The SD1 lecturer uses a fully automated e-test with every task embedded into a unit-test template. The students embed their solution in the unit-test and when it turns green, they get full point for the task. Soon, the time for the exam had to be doubled, because now

every typo needs to be fixed to make the unit-test work. This costs extra time. If the solution is almost perfect but the unit-test does not work, the student gets no points at all. This causes a high negative stress level and runs counter to the idea of software development being a sensually satisfying activity. But it can get even worse: To pass such a test, the syntax and programming necessarily need to be really well established to get results in such a stressful situation. In other words, there needs to be a limited pool of possible tasks and the correct answers need to be trained well. In SD2, the exam contains only concept-based questions without requiring active programming. Code problems are shown, and students are asked to identify and fix them. This way of doing the exams fits to the concept-based teaching approach.

1) *Methodology*

For the evaluation of the teaching concept, students positive and negative feedback was collected from winter term 14/15 till winter term 17/18. Students had the opportunity to provide textual feedback after two thirds of the term has passed. Each student who was registered to the lecture had the chance to participate anonymously by using an online form. Participating students were mainly enrolled to the computer science and media and the mobile media bachelor's degree study programs. Both lecturers were evaluated separately. For the analysis of the comments, a quantitative content analysis [14] is used. As the result, categories are derived based on the feedback material, the inductive approach is chosen (ibid.).

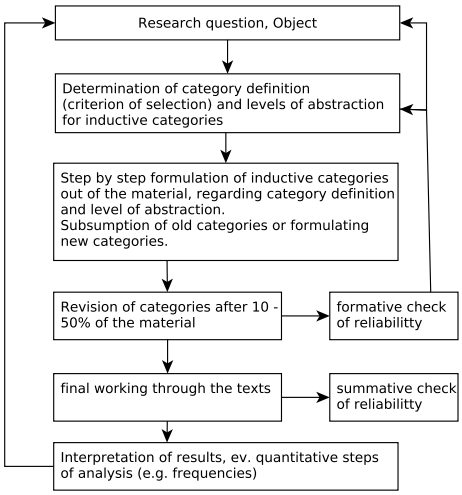

Figure 3: Quantitative Content Analysis [14]

About 350 students attended the lecture during seven semesters. 70 individual comments (positive and negative) were submitted by students.

2) *Results*

Next, the results of the feedback analysis are shown and interpreted. The top five mentioned categories for both the negative and positive comments are presented in the following.

The results show that besides the success of the concept-based teaching approach is not only dependent on the structure of the approach. In fact, personal aspects, such as lecturer's ability to teach, their motivation, competence is a pre-requisite for a successful course.

VII. A Framework for Teaching Software Development

TABLE I
POSITIVE FEEDBACK

Manifestation	Category
Ability to teach (N=50)	Personal aspects
Motivating lecturers (N=29)	Personal aspects
Competence of the lecturers (N=25)	Personal aspects
Positive atmosphere (N=23)	Personal aspects
Teamwork (N=9)	Personal aspects

TABLE 2
NEGATIVE FEEDBACK

Manifestation	Category
Stray from the subject (N=8)	Personal aspects
Lack of a test exam (N=4)	Lecture
Inappropriate lecture time (N=3)	Lecture
Too extensive reading assignments (N=2)	Reading assignments
Unstructured (N=2)	Personal aspects

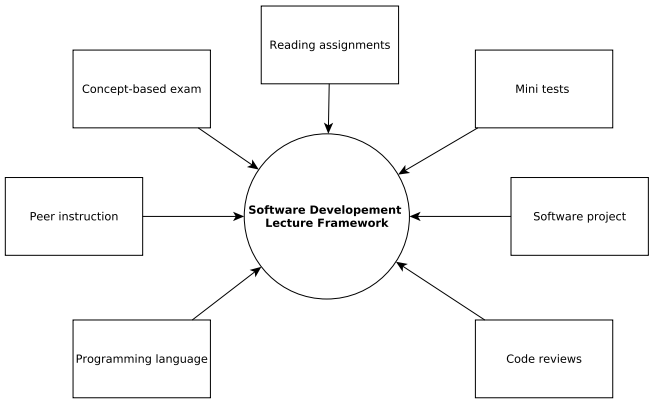

Figure 4: Software Development Lecture Framework

Based on the experiences described in the above a software development lecture framework can be derived. This framework can be used and tailored by others for teaching in any programming language. The following explains each part of the proposed framework.

A. *Reading assignments*

The reading assignments are sent via e-mail in advance of a

lecture, usually a week before a specific topic is discussed. The sent links refer to existing tutorials, API documentations, articles and blog posts – expertise from professional developers and practitioners is reused. It is not necessary to create own reading assignments. Additionally, further readings can be provided for the more experienced students, which allows them to get deeper insights into topics they already know. Further, contrasting articles can lead to a discussion during lecture time.

B. *Mini tests*

Mini tests are handed out at the beginning of a lecture and contain questions about the topics covered by the reading assignments. The questions are mainly aiming at concepts and not syntax. Students have about 15 minutes to fill the tests and afterwards, the sheets are collected by the lecturers. A brief review of the returned tests indicates in which sub-topics comprehension problems appeared and serves as a foundation for the subsequent discussion. During the discussion these problems are explained e.g. by diagrams or code snippets. As far as possible a link between the current topic and the students' projects is established. Some topics require deeper insights and examples (e.g. GUI development) for which presentation slides and live coding is prepared in advance. After each lecture, the mini test as well as the results of the live coding examples are published online.

C. *Software project*

The software project servers as a replacement for traditional programming exercises during lab sessions. Students are encouraged to implement the topics that were previously discussed in the lecture. This motivates them to think about how they can implement and adjust a specific topic for their project. The project also allows the use of professional tools (e.g. version control, modeling tools etc.) in a natural manner, which is hardly to achieve and artificial for traditional programming exercises. It has been observed that students are intrinsically motivated when they are allowed to freely choose their type of project (e.g. sports manager, address manager, board game). Before students start with programming, they need to ask the lecturers if the project scope is realistic to achieve the predefined requirements. Each project must match ten grading items which are linked to the topics presented in the lecture. This ensures that each topic was practiced by actual programming. It has been shown that a team size between two and four project members is ideal as group effects can be practiced, too. During the semester, students have at any time the chance to track their project progress as grading sheet is available for them and they are able to check which of the items are already implemented. Having a software project instead of programming exercises addresses both experienced and beginner students: experienced students have the opportunity to add more features than required to their project (e.g. database access) whereas beginning students only have to fulfil the requirements to get a very high grade. The software project counts one third of students' final grade.

D. *Code reviews*

Code reviews, conducted at least two times during lecture time, link the lecture with the previously introduced software project. Each project team is required to present their current progress. Afterwards, the project is discussed with both students and lecturers. The reviews are aiming to ensure students' progress and to improve their ability to discuss about code. The first review should be conducted during the first weeks of a semester right after students developed their first models with UML class diagrams as this allows to make changes easily. The second review should be conducted in the second half of the semester. It has been shown that most project groups have started with implementing most of the required topics and have concrete questions from which other groups can benefit. The presentation of the review should not be graded as it must be a "save environment" so that students can talk open about their implementation problems.

E. *Programming language*

The selected programming language should be taught for two semesters. This allows to introduce and focus on the syntax and the main concepts in the first semester. Having these main concepts as a foundation, allows to get more in detail by teaching advanced concepts and techniques during the second semester.

F. *Team teaching*

It is suggested to have two lecturers for teaching the advanced programming topics. Ideally, one of them is an experienced lecturer that is able to provide insights into real-world projects and the other one is a PhD student. This team constellation allows both providing experiences feedback and managing the increased workload compared to traditional teaching approaches.

G. *Concept-based exam*

The exam at the end of the semester should focus on asking about concepts and not syntax. Students ability to code is checked by their implementation of the software project. Questions in the exam are aiming on how specific concepts can be used, their advantages and disadvantages as well as interpreting a code snippet (e.g. code that forces a dead lock). The exam counts two thirds of the final grade. When the grading is done, students get an automatically sent e-mail that contains detailed information on how they performed in each topic of the exam and the software project.

VIII. Summary

This article introduced a concept-based teaching method for learning programming in the second term of computer science programs. The main differences to traditional head-on teaching are the switch from exercises to a software project, team teaching, mini tests, code reviews and reading assignments. Based on the experiences gathered in seven semesters of

applying and improving this approach, a software development teaching framework was derived, which allows others to use, tailor and improve this way of teaching.

REFERENCES

[1] H.Abelson, G.J.Sussmann, Structure and Interpretation of Computer Programs, 1996
[2] J.Dean, Large-Scale Deep Learning for Intelligent Computer Systems, Google Tech Talk with Jeff Dean at Campus Seoul, https://www.youtube.com/watch?v=QSaZGT4-6EY
[3] J.Goll, C.Weiß, F.Müller, JAVA als erste Programmiersprache. - vom Einsteiger zum Profi, Teubner Verlag
[4] Christian Ullenboom, Java ist auch eine Insel, Rheinwerk Verlag
[5] Budi Kurniawan, Java: A Beginner's Tutorial, Brainy software
[6] G. W. Scott (2017) Active engagement with assessment and feedback can improve group-work outcomes and boost student confidence, Higher Education Pedagogies, 2:1, 1-13, DOI: 10.1080/23752696.2017.1307692
[7] Boud, D., & Falchikov, N. (1989). Quantitative studies of student self-assessment in higher education: A critical analysis of findings. Higher Education, 18, 529–549.10.1007/BF00138746[Crossref], [Web of Science ®], [Google Scholar]
[8] Mazur, Eric. Peer Instruction: a User's Manual. Upper Saddle River, N.J.: Prentice Hall, 1997. Print.
[9] http://www.peerinstruction4cs.org, a site run by Cynthia Bailey Lee of the Computer Science Department at Stanford University, and Beth Simon of the Department of Computer Science and Engineering at the University of California San Diego.
[10] W.Kriha, flipped concurrency – Concurrency and how to improve reading and understanding, https://kriha.de/flippedconcurrency.html
[11] Competences after C-Crash Course: compile and run on console: 90%, basic language features like variables and control structures: 50-90%, function declarations and calls: 66%, pointers and references: 33%, using syscalls 33-50% (from B.Binder, HdM)
[12] G. Wilson, what we actually know about software, http://vimeo.com/9270320
[13] P. Van Roy, S. Haridi Concepts, Techniques, and Models of Computer Programming, MIT Press
[14[P. Mayring, Qualitative Inhaltsanalyse, Forum: Qualitative Social Research, 2000, http://dx.doi.org/10.17169/fqs-1.2.1089

Walter Kriha Since 2002 Walter Kriha holds a professorship in Distributed Computing and Internet Technologies at Hochschule der Medien, Stuttgart. His research projects included virtual lab technologies and secure systems for smart-energy grids. Besides his academic work he spent many years in the industry, developing software for Unix kernels and embedded control, object-oriented frameworks and Internet portals. In 2017 he won the prize for best teaching at HdM.

Tobias Jordine Tobias Jordine received the B.Sc. and M.Sc. degree in computer science and media at the Stuttgart Media University in 2009 and 2011. In the beginning of 2013 he started his PhD studies in computer science education in cooperation with the University of the West of Scotland and the Hochschule der Medien. He finished his PhD in November 2017. He presented at the Frontiers in Education conference (FIE), Madrid, the European Conference in the Applications of Enabling Technologies, Glasgow and the European Conference on Games-based Learning, Paisley where he presented his PhD topic. Tobias Jordine is currently responsible for the technical development of a learning analytics project.