Teaching Agile Software Development through Lab Courses

Andreas Schroeder, Annabelle Klarl, Philip Mayer, Christian Kroiß
Institut für Informatik
Ludwig-Maximilians-Universität München
Munich, Germany

***Abstract*—With the days of the lone coder long gone, it is critical in our education of young computer scientists to lay particular emphasis on the "softer" spots of software development: How to organize a development process, how to deal with teams of software engineers with different skills and motivations, and how to produce outstanding software despite hard deadlines and (ideally) a 40-hour-week. In this paper, we report on the setup, execution, and results of two software development labs with a specific focus on agile methodologies conducted in 2010 and 2011 at our university. Not only are agile methods widespread in practice today; with their focus on human interaction and work-life balance, we believe that experiencing a full agile product development cycle in the risk-free academic environment is a benefit not only for our students' technical skills, but to their social skills as well.**

I. INTRODUCTION

Technical topics such as data structures and algorithms, software modeling, and programming are pervasive in today's computer sciences curricula, and rightly so. However, while these topics are key to understanding the technical side of software development, we believe that the human side – how to deal with collaborative software development processes and their need for self-organization, motivation, and work coordination – can greatly benefit students in their transition to, and success in, today's job market.

A lab course specifically tailored to the use of a concrete software development methodology – as opposed to just modeling and programming – can address these shortcomings. However, care must be taken to properly raise the attention of students with a one-sided computer science background, who are used to solving small algorithmic problems on their own, to the issues arising in working as a team. In particular, it is crucial for students to understand the importance of following a real software development process – an ill-executed lab with a development process gone wrong will only strengthen the all-too-common view that "processes are just a waste of time".

In the past years, we have experimented with different software development processes and different software products to be created by students in our labs. Of particular interest are two issues which we have identified to be especially detrimental to the success of the labs. Firstly, academic instructors tend to introduce cutting-edge research topics into their labs; mostly, this is done in the hope of recruiting students or creating working prototypes. We feel that in many cases these topics are too far removed from students' experience, and thus require them to spend a lot of time on getting up to speed on the topic – which is not the focus of a software development lab. Secondly, while we do value software modeling, "getting the model right" can consume so much time that the inevitable coding part at the end of the project is more like a hacking frenzy than a coordinated development approach. This must be avoided at all costs. Thirdly, the software development process in use must be tailored to the time available, and the number of students involved. Performing a generic RUP process with a small number of students, for example, is not going to benefit anyone.

Two years ago, we devised a new software development lab to alleviate these problems. As we discuss in this paper, we have chosen an agile software development approach to the organization of our lab course – first, to avoid process bias; second, to match the number of students; and third, to allow for multiple feedback loops through the highly iterative nature of agile processes. We have chosen Scrum [1] as the actual process.

Furthermore, an particular emphasis was placed on an easy-to-understand, fun challenge which was highly motivational for students, yet offered many of the challenges of real-world software products – especially, the requirements of producing a clean architecture, code comprehensiveness, and rigorous testing; all of which are very important problems, although they are not algorithmically complex. We have chosen a networked, multiplayer card game with a challenging graphical user interface as the product.

Finally, we believe that the actual development process, not the project setup, should be the focus of the lab. Therefore, we have provided and introduced a proper tooling infrastructure (based on Eclipse) and a product skeleton for the students to build upon.

We were able to organize and manage two particularly successful lab courses in the years 2010 and 2011. The selected set of organizational, technical and product features seems well balanced, and therefore it appears beneficial to share the knowledge acquired.

II. LAB COURSE SETUP

In previous installments of the software development lab course, we experienced severe problems: often, the courses

978-1-4673-1456-5/12/$31.00 ©2012 IEEE

were characterized by a prolonged phase of "analysis paralysis" (i.e. staying in the analysis phase indefinitely) that blended into a hacking frenzy as the course deadline approached. Both phases led to a rapid degradation of student morale, focus and motivation, and several software development lab courses ended with a fairly sub-optimal result and learning outcome. We had to expect that as a result, students primarily learned that (a) analysis is only delaying the required coding phase in the end, and therefore must be avoided if possible, and (b) that the project organization means and infrastructure taught in the course are ineffective for leading a project to success.

Our goal in the new setup of the software development lab course is to improve on past installments, and to teach students the cornerstones a software development endeavor requires to be successful. For this, we decided to teach by example and hence to provide a healthy and effective organizational frame. In this frame, we rely on empirically validated and proven techniques and technologies. Specifically, we rely on agile processes, and state-of-the-art tooling support. At the same time, we put less emphasis on the completeness of the product that the student create during the lab course, and more emphasis on the process of product creation and its continuous improvement during the lab.

A. Product

In previous courses, the students got involved in cutting-edge research topics and developed a product realizing innovative, but not yet well-established ideas (e.g. a visual scene rendering engine with an easy-programmable animation interface). The students were therefore heavily occupied by catching up with the research topics to understand, refine, and realize the main concepts of the product. The scientific content and its realization demanded a significant part of students' attention.

To concentrate more on learning the main activities for managing and carrying out software development, we instead suggest developing a simple and easy to grasp product. The product vision must be clear enough to fully explain and understand it immediately, and must allow focusing on the actual process of software development right from the beginning. To keep motivation among the participating students high, the product has to be appealing despite its simplicity.

In our lab course, we decided to develop a simple card game for software engineers called "The Bug is a Lie". The game is a parody of the way software is developed in industry with a sarcastic view on the individual roles people take (manager, honest developers, evil code monkeys, and consultants) and how they, more often than not, work against each other.

This card game is well suited for a lab course since (1) it is easily and shortly explained, and (2) it is not only a virtual game, but also has a real counterpart: prototype card decks can be handed out, and attendants can directly get into touch with the game. However, the game shines not only due to its practicability, it also holds enough requirements and extensions to challenge students. "The Bug is a Lie" is a multiplayer game which requires the (3) implementation of a robust networking layer, authentication facilities and a database layer. For illustrating the game, the students have (4) to sketch a graphical user interface according to customer's needs. One of the most convincing advantages of this card game is that it is (5) easily decomposed into self-contained work packages or user stories. After the initial setup of the basic functions and graphics, each card or rule can be introduced separately (e.g. the coffee machine card that allows to draw two cards). The game can therefore be developed in iterations, each providing an executable and "playable" fraction of the game. To highlight the advantages of Scrum for a product with unclear and changing requirements, user stories can be uncovered gradually over the course of the lab. Additionally, the initial set of user stories only serves as a rudimentary basis which has to be revisited and extended after each iteration to account for the current state of the product and the capabilities of the team.

Altogether, the game "The Bug is a Lie" is an easy-to-understand card game which nevertheless poses enough challenges to students. While the game logic is rather straightforward to be implemented since it is given through game rules, particularly realization of the networking layer and graphical user interface calls for clean software development including the design of an elaborate architecture, comprehensive coding and rigorous testing.

B. Process Organization

The purpose of the lab course is to teach how to successfully develop software. The main focus is therefore not placed on the product, but on the introduction of an up-to-date software development process and on the use of state-of-the-art tools to organize and support this process. In the frame of the introduced process the students learn how to balance customer's needs and development time, how to cope with changes and how to organize themselves in a team.

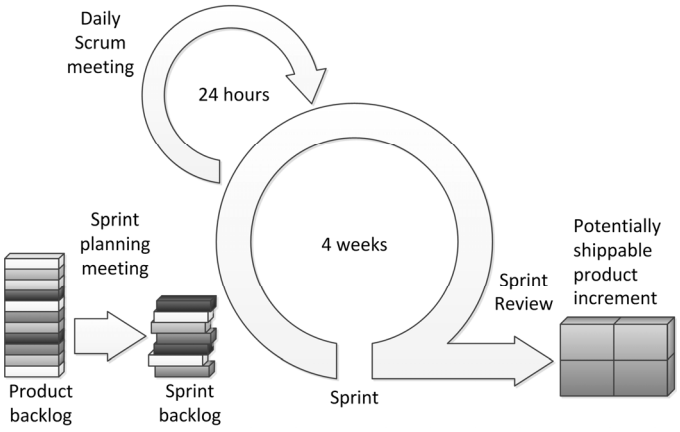

Fig. 1. Scrum Process Loop

Since agile processes are a well-known approach to flexibly cope with unanticipated changes in customer's requirements, the lab course is based on the agile process Scrum [1]. In Scrum, the product is developed incrementally in time-framed *sprints* of about four weeks (cf. Fig. 1) where the development team is working on a fixed set of requirements – called *sprint backlog* – producing a running part of the product. It is

important that the sprint backlog remains unchanged during the sprint; not yet treated user stories, changes or extensions are collected, but remain in a separate *product backlog* to be introduced in future sprints. To monitor the progress during a sprint and to make it visible to everyone, a *burn-down chart* capturing the remaining efforts for the current sprint is created and regularly updated.

One important property of the Scrum process is the *velocity* of the Scrum team. In an ideal world, each team member is working on software development around forty hours per week. However, the real world is intruding with organizational duties like installing software or doing paperwork, as well as delays due to sickness, holidays, hardware breakdowns, and the like. The velocity parameter is team-specific, and captures the ratio of effective work hours spent on developing software to hours invested. Knowing this parameter is essential for not cluttering up a sprint with an excessive workload.

In our lab course, all participating students take on the developer role, while two tutors take the roles of Scrum master and product owner, respectively. Since tutors are more experienced and not directly involved in the development work, they can better supervise the process and ensure that the Scrum values are respected. Furthermore, organizational problems (like providing meeting rooms) can only be solved by university staff. Tutors are also more familiar with the product to be developed; thus, questions about the product can be answered more accurately. However, we take care that the two roles are clearly separated between the two tutors, such that students always know which person to contact for which concern.

The original Scrum process needed to be adapted to fit the frame of a students' course. In our setting, students work only part-time for thirteen to fourteen weeks on the lab course; they usually are not familiar with Scrum and have varying programming knowledge and experience. Therefore, we start the lab course with an introduction phase of two weeks and a development phase with three sprints of four weeks.

During the lab course, we limit the workload according to the credit points of the course – students are expected to work on the lab course for only 180 hours in total. This means that we only assume a 13-hour-week instead of a 40-hour-week. The students are urged to put neither more nor less effort in the lab course, ensuring a sustainable pace and mimicking professional work without overtime or crunch time. This is mostly based on an honest reporting of spent effort per student for each user story. The spent working hours are visible to everybody so that the students can monitor themselves; additionally, the tutors may supervise the workload distribution.

In the two week introduction phase, two lessons are given. One lesson explains the Scrum process in detail. It focuses on a practical view and introduces some additional means to support the Scrum process in practice. Inspired by Pilone and Miles [2] and XP [3], we use *Planning Poker* to estimate user stories, test-driven development to ensure the quality of developed software and continuous integration to monitor the integrity of the product. The first lesson closes with the vision of the product "The Bug is a Lie" and a hands-on session for playing the card game. The second lesson introduces state-of-the-art tools and libraries to work with during the development phase. It also gives an overview about the code skeleton made available to students to kick-start the project. The second lesson closes with a live coding session, where students get familiar with the skeleton and receive advice on where to start.

Each sprint in the development phase is structured as in the original Scrum process with the meeting times scaled down to the timeframe of the course. To consider overhead for communication, setting up the development environment, and getting familiar with tools and libraries, we assume an initial team velocity of 0.7, and recompute the velocity after each sprint. Each of the three sprints starts with a sprint planning meeting where the product owner presents the whole product backlog. All user stories are prepared, so that the students do not have to elicit them in requirements meetings. Trainings with participants unfamiliar to Scrum [4] showed that user story decomposition is too complex for beginners and decomposition is better trained through task decomposition of user stories. The students themselves are responsible for estimating user stories and assembling the sprint backlog. By letting them independently select the workload and putting the responsibility for the realization of the selected sprint backlog on them, we aim at achieving a high team commitment to the sprint. In the selection process, the students are encouraged to use Planning Poker and UML sketches to discuss ideas and transfer knowledge during their estimation. At the beginning of estimation, tutors help developing task decompositions and documenting ideas by UML sketches to guide the students to a reasonable use of tools. In the four weeks of development, the students also have to organize themselves, assign tasks, create a design and architecture for the product, recognize impediments and delays, and meet with the tutors (i.e. Scrum master and product owner) for Scrum meetings once a week (which corresponds to one meeting every thirteen working hours) for a fifteen minutes standup meeting report on progress, plans, and impediments in the fashion of a weekly Scrum meeting. Other tutors' meetings are scheduled as needed during the standup meetings. At the end of each sprint, the product is presented in a sprint review meeting. Not only the product, but also all requested documentation such as architecture diagrams has to be demonstrated. In the review meeting, the team gets feedback on the product, its quality, and the implemented features. Each sprint closes with the sprint retrospective, where the team reflects about the execution of the last sprint, and identifies areas of improvement guided by the Scrum master.

C. *Tooling Infrastructure*

The tooling infrastructure is primarily designated to support the Scrum process discussed above. The tooling infrastructure is intended to provide our students with a solid, ready-to-use, state-of-the-art infrastructure for their lab course. In particular, the aim is to provide enough room for individual tailoring through students, but at the same time allowing students to get quickly up to speed with development activities. Providing a fixed tooling infrastructure helps avoiding lengthy negotiations and discussions on topics that require substantial investigation time for proper and conclusive decisions.

The tooling infrastructure we use in the lab course consists of an integrated development environment (IDE) with unit testing support, a version control system, a web-based issue tracker and wiki for the management of the product and sprint backlogs, a continuous integration solution, and a UML tool. In the selection of the tools, we put special emphasis on tools that integrate well with each other, provide a consistent experience, and are easily accessible.

The tools in use are changed for each lab course based on previous experiences. For the 2011 course, we selected Eclipse [5] as IDE, as it provides an extensible platform with good tool integrations, Trac [6] as web-based issue tracker and wiki, and Subversion (SVN) [7] as version control system. All three tools integrate well with each other. Eclipse and Trac offer both sophisticated means for linking and navigating between related resources, allowing for a quicker and better understanding of the code base and the project activities (e.g., Trac allows to link to wiki pages, tickets, source code lines, and revisions).

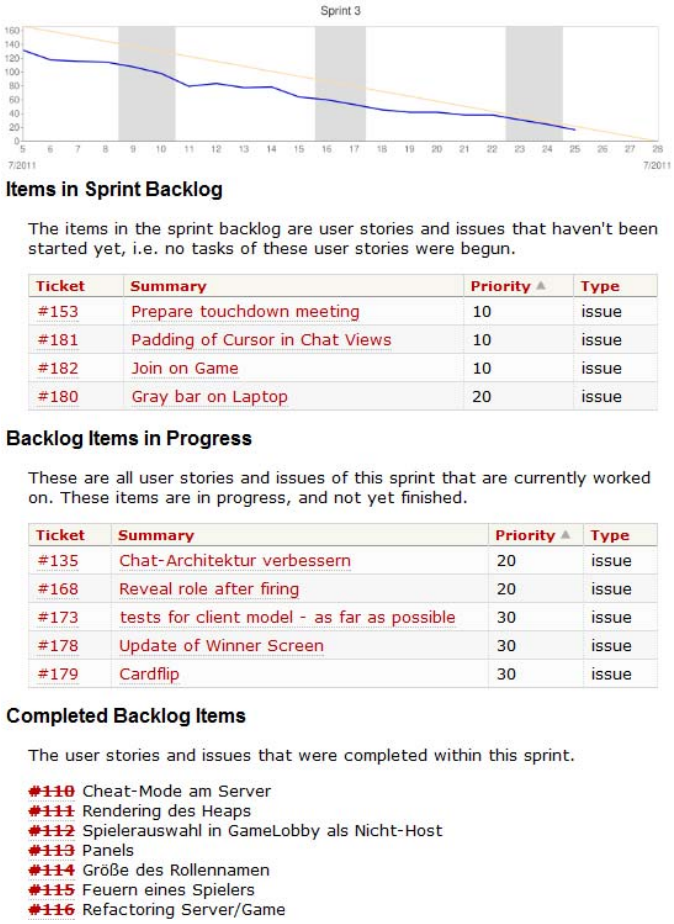

Fig. 2. Sprint Whiteboard

Trac itself is a minimalistic issue tracker that is extensible through plug-ins. For supporting the Scrum process, we add the plug-in EstimationTools [8] that draws burn-down charts based on workload estimations attached to tickets. Fig. 2 shows the sprint whiteboard wiki page featuring a burn-down chart as well as automatically updated ticket lists (generated from ticket queries) that show sprint backlog items (i.e. items not yet started), items in progress, and completed items related to the sprint.

Furthermore, we add the Bitten [9] continuous integration plug-in to Trac that builds the product, runs all tests every six hours, and displays results on build success, test success, code size, and unit test coverage on the lab course's Trac site. Our Eclipse setup also includes test supporting tools. While JUnit test runners are included in the basic Eclipse distribution, we also add EclEmma [10] as a coverage tool, and Infinitest [11] as a continuous testing tool – a tool that runs all affected unit tests on code changes.

Finally, we introduce MagicDraw [12] for the creation of UML diagrams. We periodically encourage students to use simple and communicative UML diagrams as means for discussing and clarifying designs, and for documenting design decisions at the end of each sprint. Therefore, we try to lead by example, and deliver all introductory code with proper UML documentation, demonstrate the use of UML sketches during the sprint planning meeting, and support the design sessions with clearly represented UML diagrams.

D. Libraries

Similar to the lab course tooling infrastructure, we focus on state-of-the-art, exemplary libraries that feature well-designed programming interfaces from which the students can learn, and which help our participants in creating the product we require them to produce. In the selection process, it is sometimes necessary to balance between practicality and quality of the libraries. For example, we decided to use JDBC [13] and Swing [14] although both libraries have known design issues. We furthermore provide students with libraries for networking applications (Java NIO [15], JBoss Netty [16]), logging and utilities (Log4J [17], Google Guava [18]), and concurrency (Java Concurrency Framework [19]). For creating unit tests, we supply JUnit 4 [20] and Mockito [21] for the creation of test mocks.

We provide a short introduction for each library in the Trac wiki, as well as links to detailed documentation. In this way, the students are able to read up on more advanced topics.

E. Skeleton

With the substantial amount of libraries imposed on the project, there is a considerable danger of overwhelming the students with unfamiliar technology and tools, even though introductory materials are supplied. We therefore provide a code skeleton that already supplies a basic project setup and exemplary code for library usage, coding style, and tests. Additionally, the skeleton allows speeding up the parallelization of development activities. If the lab course started from scratch, the first weeks of development work would be characterized by a constant overlap of activities and students getting in each other's way – a bad start for a course that is (a) short and (b) depends heavily on the students' motivation.

In our setup, we decide to invest ten person days (a week, two developers) worth of time for the creation of the basic skeleton on each installment. This time was sufficient to create

a fully operational client-server networking layer, a database access layer, a registration/login facility, and rudimentary UI shell (consisting of background, header, footer, and login, logout, and registration screens), all with JavaDoc, architecture documentation, and unit test coverage required from course participants. Fig. 3 shows the UML class diagram of the skeleton server that was provided together with two pages of explanatory text.

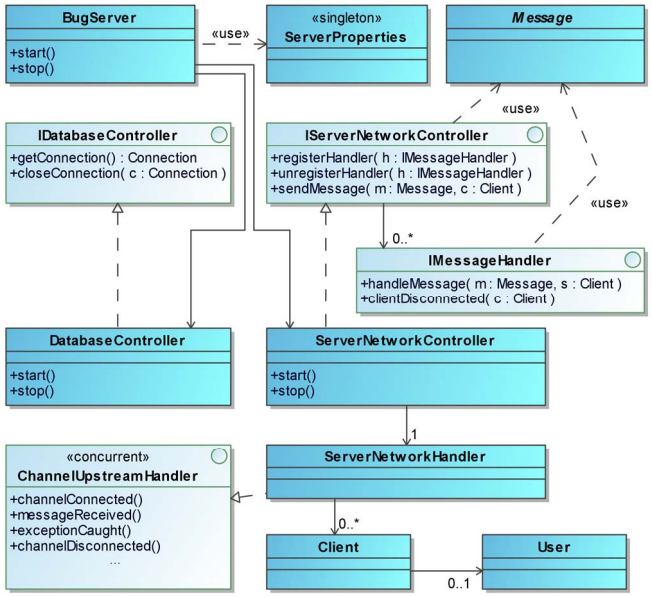

Fig. 3. Server Skeleton Architecture

This skeleton formed the basis for extension and improvement in the three sprints of the course.

III. Results and Evaluation

We implemented the lab course during the summer terms of 2010 and 2011 with slightly varying setups, different velocities, and slightly different time lengths (14 weeks in 2010 vs. 13 weeks in 2011). In both courses, six students with varying levels of programming skills took part. All students had taken part in a programming lab that focused on programming activities, and followed no development methodology. Still, the lab outcome was the creation of a working product, which was achieved by putting in long hours of hacking effort. The students also had heard lectures on UML and object-oriented analysis and design. The low number of students allowed us to focus heavily on the fine-tuning of the course setup, although we would have favored a larger number of students.

From the student questionnaires that were created specifically for the evaluation of the new lab course, we can deduce that the lab course setup is very well received. Slight negative feedback was found on imposing the dictum of "good enough design", i.e. to only design a system as far as the currently considered user stories require it to be designed and to look no further ahead; students did not experience the value and the potential of a clean and simple design, as it seems they had a hard time at creating and capturing proper system designs in general.

From a bird's eye view, both courses successfully created a restricted, but working and playable version of the game. The created versions were robust, featured satisfactory unit test coverage, and were of acceptable quality with regards to readability and maintainability.

A. Product

In the execution of both lab courses, we found that the product "The Bug is a Lie" is very well accepted. The game is explained quickly, and especially the hands-on sessions where the students can play the real card game are highly appreciated and increase the understanding of the game. The students generally examine the game so intensely that they ask very precise questions about its rules.

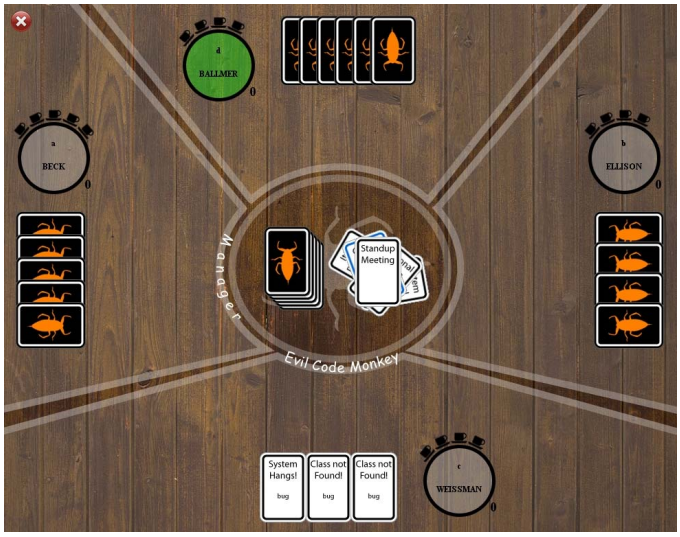

Fig. 4 "The Bug is a Lie" Game Board

Concerning the functional challenges of the product, we discovered that the students are able to grasp the main intentions. However, students are generally challenged by the implementation, specifically by the creation of multiplayer facilities and game UI. In the end, however, students manage to realize a sophisticated and appealing UI (Fig. 4 shows the result of the 2011 course). Although the game UI itself was a success, fundamental issues in user story definitions arose in the second sprint of the two courses: Usually, students tend to separate units of work along the layers of the product: if not directed, students would create and work on user stories like "user interface creation" and "server-side logic", with the consequence of largely omitting integration of the created layers. Thus, students would fail in creating a satisfactory product. Thus, we advise that tutors pay close attention that user story partitioning is performed in accordance with recommendations from the literature [22], i.e. in features with real costumer value. Still, students tend to create a "game UI kernel" user story, with the problem that as long as the game UI kernel is under development, user stories concerning the realization of game rules need to be queued. However, the card game can indeed be decomposed into user stories according to

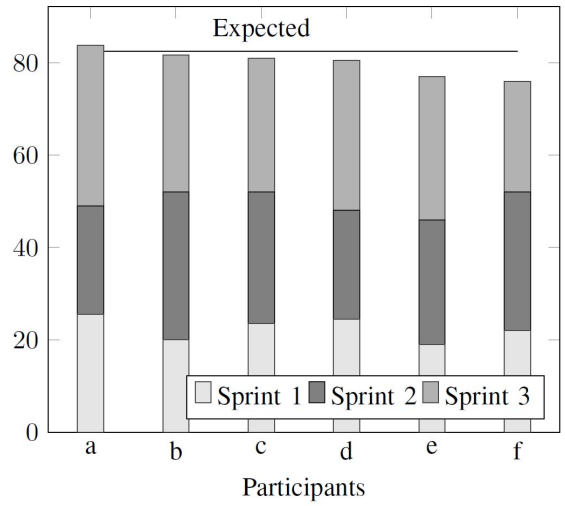

Fig. 5. 2010 Hours Worked

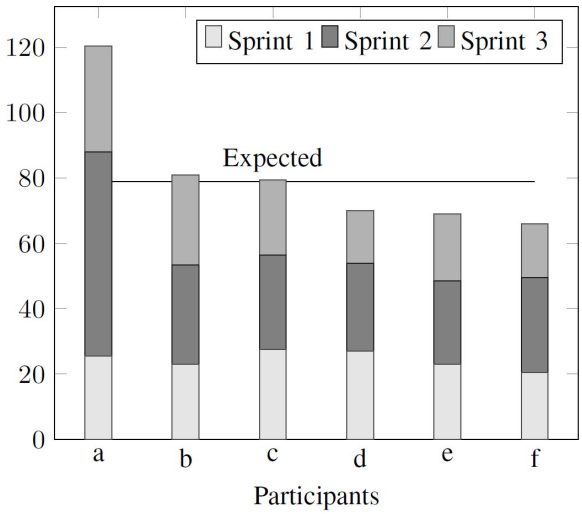

Fig. 7. 2011 Hours Worked

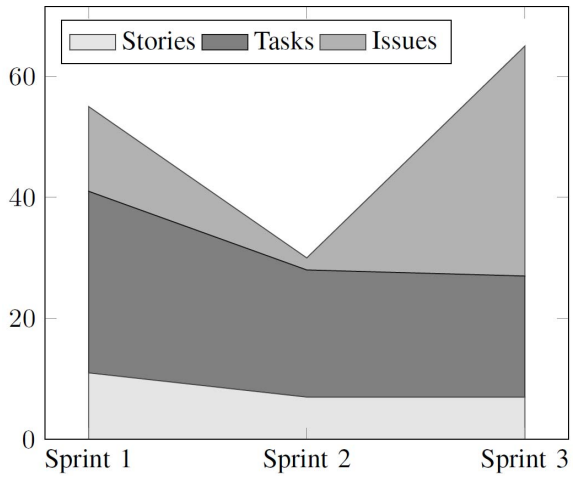

Fig. 6. 2010 Lab Course Tickets

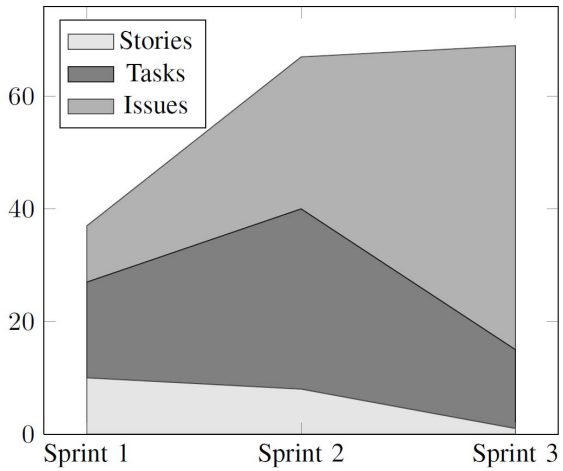

Fig. 8. 2011 Lab Course Tickets

each card or rule, i.e. stories without a technical focus. The product structure allows tutors to tune the amount of work per sprint depending on the success and velocity of the Scrum team with relative ease.

The product itself is appealing to the students. Most of the students indicate that given the choice, they would value a fun product more than a product relevant in research or industry. However, some students criticized that the specific game "The Bug is a Lie" is only fun for software engineers and fails to attract non-professionals. They express that they would appreciate the game more if they could share it with family and friends.

B. Process Organization

The chosen process fits the settings of the lab course quite well. It creates a high commitment of the students to the product and the lab course itself. The students are able to create a satisfactory and working product while learning the cornerstones of effective software development.

Since the participating students have varying development experience and programming skills, the introduction phase is particularly useful. A 4-hour-lesson on Scrum is sufficient to communicate the principles for the practical application of the process in the lab course. After this introduction, all students are able to put the Scrum rules into action in an effective way. The second lesson on tools, libraries and the provided skeleton allows an immediate kick-start of the project. However, tutors must take care that enough time is reserved for intensive coding and Q&A sessions since, due to the size and complexity of the provided skeleton, difficulties in understanding and extending it may arise.

One of the most important advantages of the outlined process organization is the workload limit imposed on the development of the product. On the one hand, this workload limit creates great relief among the students since the lab course has a sustainable pace and manageable workload compared to other lab courses. Additionally, the introduction of the team's velocity to account for overhead meets general approval (velocity is computed by dividing the sum of the user story estimates by the effort put into the sprint) – it is a good means for allocating time for initialization and organization. On the other hand, the workload limit makes time a valuable resource that has to be managed carefully. Students hence learn to develop software with a limited time budget in an effective

way. As a result, students are highly motivated and invest the requested time, as can be seen from figures Fig. 5 and Fig. 7[1].

An important aspect in the setup of the development process is the assignment of the roles of Scrum master and product owner to the tutors and to two different persons, leading to a clear separation of concerns. The students have several questions about the rules, design and realization of the product that are best answered by a product owner familiar with the game. Conversely, the students can approach the Scrum master with impediments or organizational problems. In a students' course, the Scrum master has a highly important supervisor role. She needs to pay special attention that the students stick to the Scrum values because they often experience difficulties with self-responsibility and self-organization. The Scrum master has to remind the students that they not only report on technical advances in the weekly Scrum meetings, but also on delays, impediments and next steps and that they critically analyze their best practices. They have to establish openness and honesty as a common value so that problems can be openly addressed. In our 2010 lab course, closeness, ignorance, suspicion and mistrust among the students and towards the tutors caused the degradation of software quality in the second sprint. Issues were not reported until the review meeting as can be seen in the lack of reported issues in Fig. 6. This installment clearly demonstrated that students must be encouraged to act on their own responsibility and not wait for any instructions – for example, when discovering ambiguities in user stories. In 2011, the students used an open and honest communication channel where issues were reported immediately (see Fig. 8) which led to a higher quality of the final product.

We regard the Scrum process as suitable for the software development lab course. The sprint planning meetings on the development team's responsibility and authority create a high commitment of the students, the weekly Scrum meetings give continuous feedback on the current status of the product, the review meeting gets the customer and the development team into touch and the retrospective exposes favorable and unfavorable methods and practices. However, at the beginning the estimation process is hard for the students. We recommend that the tutors assist the first estimations by partitioning user stories into tasks and providing solution approaches, but do not overwhelm the students with hints since this has a high impact on the estimations. As soon as the students become more confident – normally not later than in the second sprint, the tutors have to make sure to abandon their assistance to not further influence the process. Depending on the progress of work, it might also be better to hold the status report meetings more often than once a week to give students more possibilities to arrange the next steps when progressing fast. Due to the lack of a constantly available shared working place, the students also suffer from the problems of distributed development and reduced availability. Therefore, they must be encouraged to organize regular pair programming or working meetings to benefit from quick communication channels and high collaboration. In the same way, the students have to be urged to make use of UML sketches. During the planning meeting and the development work, UML sketches help to visualize and fix ideas on the architecture of the product. We demonstrate the advantage of UML for sharing knowledge by providing UML diagrams for the skeleton, for recording ideas by documenting the sprint planning meetings, and for clarifying designs by supporting design discussions with UML. Unfortunately, the students often have difficulties in formulating their visions in a fast and simple way since they are either not used to creating a proper system design at all or are only familiar with extensive modeling instead of "good enough design". Nevertheless, the students generally benefit from the advantages of Scrum: short sprints and an executable product at the end of each sprint keep development on-time and effective while the students are trained in self-responsibility and self-organization.

Conversely, self-responsibility and self-organization pose challenges for grading. As previously mentioned, the focus of this lab course is not on the completion of the product, but on the realization of the process. In grading process acceptance, care must be taken to only make demands that are measurable or verifiable. Code coverage, architecture documentation and conformity with user stories are good means to evaluate the quality of the product without assuming a certain progress in the completion of the product and can be easily supervised. To rate the realization of the process, we monitor the participation of each student in the meetings and the number of issues detected by the team since this reflects the degree of commitment to the lab course. In the same way, we do not recommend to assign roles to students that involve management activities (e.g. supervision of code coverage or documentation), as it is difficult to grade students for results achieved by others. Common responsibility and shared code ownership lead to a high level of commitment and quality, but need to be achieved through other means than grading.

C. *Tooling Infrastructure*

The tooling infrastructure is readily accepted by our students. Due to the proficiencies in programming and in the use of Eclipse they acquired in other courses, students are able to leverage the provided introductions and documentation made available in the Trac wiki. The students are generally able to start working effectively right from the start of the course, and to implement a significant part of their project already in the first sprint. Hence, the expected kick-starting effect and prevention of lengthy setup phases are both achieved.

However, as the students become more and more familiar with the tooling infrastructure, progress can still be impeded due to design and code quality issues of the created code. Luckily, acquired familiarity with the infrastructure allows students to cope with these issues.

Also, the visibility and browsability of the code base, progress and remaining workload that Eclipse, Trac, and SVN offer enables students and tutors to discover impediments and problems early enough to remedy them within the lab course. The immediate and persistent feedback provided by continuous integration – specifically concerning unit test coverage – also

[1] In the second installment of the lab course, one student exceeded the requested effort by far. This violation of the workload limit was based on the high commitment of the student to the lab course and was only tolerated after agreement with the whole Scrum team and the tutors.

allow the tutors to point the finger on issues that are relevant for project success.

On the negative side however, we discovered that students have a hard time getting used to specifying and updating remaining efforts on Trac tickets, which is mostly due to neglect. During the first sprint, the Trac tickets created by students need to be reviewed, and immediate feedback must be given.

D. Libraries

The proposed approach to library use is largely embraced by the students. While the more advanced students find good use of the libraries provided, the less advanced students make use of the available examples in the skeleton and the provided documentation. Both seem enough to eventually cope with the unknown APIs, although pair programming sessions may help to transfer knowledge far quicker.

Furthermore, we experienced the value of clear and accessible documentation of libraries, once a student introduced a library with documentation that is not freely available. Students struggled heavily with the new library, and as a consequence, a large part of the code became only maintainable through a single student. This incident also shows that course supervisors need to be aware of the consequences and impact of student's actions and general approach.

E. Skeleton

The provided skeleton is well adopted by students. In the first sprint of both course installments, however, students had unexpected difficulties in understanding the provided skeleton, and in finding the proper starting points. Students need very extensive and clear documentation of interfaces and starting points, and it seems like the provided explanations were insufficient. Also, students tend not to maintain the coding and testing standards of the skeleton by themselves. Several gentle reminders in sprint retrospectives and dedicated design patterns and design principle sessions are necessary to achieve an acceptable software quality. On the other hand, by letting the quality degrade, students get first hand experiences on the consequences of code quality degradation for their project's progress.

Nevertheless, the kick-starting effect of the provided skeleton is clearly achieved, as the general project progress is characterized by a very productive first sprint, and a significant slow-down in the second sprint due to degrading software quality (as Fig. 6 shows, this effect was also visible in the amount of tickets of the 2010 course installment). It can be expected that if the students started from nothing, the first sprint would be characterized by slow progress. Additionally, as expected, the skeleton provides useful examples of library usage, and reduces the amount of conflicts in collaboration. Furthermore, in the case of code quality degradation, the skeleton provides presentable code examples that are up to standard.

IV. Areas of Improvements and Further Ideas

The lab course setup, with its product, process, infrastructure, libraries and skeleton has proven to be adequate for successfully teaching agile software development practices. Nevertheless, the two instantiations of the course have disclosed shortcomings that should be addressed in future installments. These areas of improvements can be grouped around knowledge transfer improvements and process improvements.

Concerning knowledge transfer, we experienced that the introduction to the skeleton code needs to be extended. For this, plenary sessions with live code inspection, discussion and coding may be helpful. In addition, precise and simple-to-read written instructions on how to extend the skeleton code should be provided.

Furthermore, we believe that on top of the regular Scrum meetings happening during the lab course, weekly plenary sessions should be established for discussions on design patterns, design principles, programming techniques and tool usage. These sessions should be held in the tradition of coding dojos [23], i.e. with focus on learning by example, learning from each other, and helping to improve the programming and design skills of the students. Following our experience with test-driven development coding dojos, it is important to create an open, motivating and positive learning environment for maximum knowledge transfer.

On the organizational side, we observed in both installments of the lab that the students were struggling for proper communication channels. Students tried to establish the use of forums, or instant chat clients with varying success. We cannot provide a conclusive and well-founded recommendation on the best electronic communication means, but a group solution like Google groups or Yahoo groups may be a worthwhile addition. The problem of proper communication channels in both installments is obviously due to the lack of a constantly available shared working place. In fact, students quickly began to organize regular pair programming or working meetings to mitigate the effects of a distributed team. At the end of the course, they expressed their regret for not having started these practices earlier. It is hence wise to suggest teams to establish these practices right away, or to provide a meeting place and fixed hours.

Also, we discovered that the use of SVN for version control can cause significant issues in collaboration. Students that need longer for finishing tasks, and therefore refrain from committing often to the repository, can find themselves in the challenging situation of having to merge a considerable amount of conflicts. To alleviate this problem, we consider using a distributed version control system like Git [24] as a replacement for SVN. Git would allow students to commit small increments to their local repositories, and therefore make merging easier when pushing changes to the shared repository.

A big concern in establishing coding/design dojos and regular shared working sessions is the lack of familiarity and trust that newly formed teams experience. One can expect that students may be reluctant to committing to these activities without knowing their team-mates well. To help the teams start

establishing healthy practices, it is therefore necessary to introduce team-building activities.

Fig. 9. Lego4Scrum City Example

One team-building activity with the beneficial side effect of serving as a practical introduction to Scrum is Lego4Scrum [25]. Lego4Scrum is a small project for consolidating and deepening knowledge of Scrum techniques using as product a Lego city consisting of houses, cars, and utility buildings and vehicles (e.g. trucks, tower cranes, bridges, and car ports). Fig. 9 shows a Lego city in construction, in which one house with garden, a bridge, a tower crane, a car and a too low-ceilinged car port (on the bottom right) is already installed. The Lego4Scrum project can be completed in a 3-hour-session, since all meetings and the sprint duration itself are reduced to five minutes. Lego4Scrum sessions allow newly trained Scrum developers to experience Scrum techniques and practices such as workload estimation with Planning Poker, process improvement through sprint retrospectives and project status visualization through sprint whiteboards. During the supervision of five Lego4Scrum team projects in 2011, we observed that due to the time pressure and vivid interaction, the team members get to know each other. Especially, they get a better understanding of how the team operates under pressure, and how the team may self-organize during the full lab course. This information may also be useful for supervisors to identify issues in team structure and communication, and may allow enough time for reflection and introduction of respective coaching measures early in the course. Therefore, we expect that using Lego4Scrum will be beneficial in future installments of the software development lab course for a further acceleration of project initialization and initial team building.

Another issue that was not yet investigated is managing larger numbers of students in the lab course. Due to the low number of participants in both installments, it was not necessary to split up the participants into separated teams. While this allowed us to focus on other issues in the course organization, managing large student numbers needs to be feasible if the course is intended to reach a larger audience. From our experience, we see two possible courses of actions. The first one is to create collaborating Scrum teams working on different areas of the same product. For the game "The Bug is a Lie", it would be possible to accommodate two parallel teams by putting one team on the development of the actual game, and another team on the development of functionality that a multiplayer game needs (such as e.g. game creation, game browsing, game invitations, chats, game statistics, and player statistics). We experienced that a single Scrum team of six students was not able to implement all features and functionality required in both areas of the product. Another way of scaling up to higher numbers of participants is to put teams in competition to each other and to let them build the same product. This is a classical approach to the organization of large programming courses, but it also involves delegating the roles of Scrum master and product owner to students or tutors to keep the number of required mentoring staff maintainable. This would entail that the teams are basically left to work on their own, which also requires additional efforts from the tutoring staff to ensure they keep adhering to Scrum practices. In general, students that are exposed to a coherent process for the first time need more guidance and tutoring for following that process than less.

V. Related Work

Using agile methods in a software development lab course is nothing new. Since the declaration of the agile manifesto, several university courses were designed and held on the basis of agile principles. Reichlmayr reports in [26] on one of the first course setups based on elementary agile principles (e.g. frequent work delivery, regular reflection, and software as measure of progress) with a very rudimentary course infrastructure compared to what we found necessary. Bower and Hughes focus in [27] on the learnability of industry-grade test-driven development (TDD) and continuous integration, and conclude that they are worthwhile for student projects in academia as well. Our experience with continuous integration and TDD is in line with their statements. Pinto et al. report in [18] on a Scrum-based NXT robot project executed by a student team of five members. The team used Google groups and Google documents to collaborate; our infrastructure is more elaborated and geared towards software development in comparison. Pinto et al. identified Scrum as suitable for student projects, and highlight the benefits of Scrum as increased progress visibility, and increased student focus and motivation. In [29], Rico and Sayani report on the result of the adaptation of capstone courses to agile methods. They conclude that proper tutoring and coaching of teams with respect to agile methods is a key factor for a project's success; we are in line with these findings, and provide more details about the lab infrastructure that can be used. Lingard and Barkataki focus on teamwork learning in [30], but they also stress that the self-organization principle of Scrum is beneficial in the organization of large courses comprised of several parallel teams. In [31], Scharff discusses a globally distributed implementation of an agile process with separate auditor teams recruited from the students themselves. The auditors reviewed process adherence as well as project progress. In her findings, Scharff stresses that Scrum helps students to structure both development and learning work that a project requires, and that students are initially overwhelmed by the amount of activities and discipline that Scrum requires – a finding that indicates that students are often lacking crucial skills and knowledge initially. Devedzic and Milenkovic present their insights and advices on teaching agile software development in [32], namely: training using practice, having clear role assignments,

allowing and fostering self-organization of teams, and keeping teams small and sprints short. As all other related work, they report that the implementation of agile methods in lab courses was successful. However, none of the work stressed planning of work capacity management and team velocity computation, or focused on the integration of development process and tooling infrastructure, and the presentation of an infrastructure template for lab courses.

VI. Conclusion

Planning, designing, and implementing today's software systems require a higher level of collaboration and teamwork than ever before. It is therefore crucial to expose these topics to tomorrow's software engineers as part of computer science curricula, and do so in a manner which highlights the importance of following software development processes, and not present them as an unnecessary burden which does not remedy the seemingly inevitable "code-and-fix" phase at the end of a project.

In this paper, we have presented the setup, implementation, and results of two highly successful software development labs conducted at our university in 2010 and 2011. The labs are based on agile development methodologies (specifically Scrum), permanent collaboration and feedback, and keeping an (adapted) work-life balance. We have found the easy-to-understand, accessible Scrum methodology to be ideal for introducing software processes – not to mention its widespread industry acceptance. Other key lessons learned from our labs include using a fun challenge for student motivation and providing a skeleton and development environment for a quick start. We hope that our findings may benefit and inspire other instructors. We are keen on your input on the presented methods, and will gladly share more information on our setup.

Acknowledgment

The authors would like to thank Martin Wirsing and Rudolf Haggenmüller for enabling the creation of the new software development lab course described in this paper, as well as all students that took part in the labs.

References

[1] K. Schwaber and M. Beedle, *Agile Software Development with Scrum.* Prentice Hall, 2002.

[2] D. Pilone and R. Miles, *Head first software development.* O'Reilly, 2007.

[3] K. Beck, "Embracing change with extreme programming," *IEEE Computer,* vol. 32, no. 10, pp. 70–77, 1999.

[4] K. Schwaber, *Agile Project Management With Scrum.* Microsoft Press, 2004.

[5] J. D'Anjou, S. Fairbrother, and D. Kehn, *The Java Developer's Guide to Eclipse,* 2nd ed. Addison-Wesley, 2004.

[6] D. J. Murphy, *Managing Software Development with Trac and Subversion: Simple project management for software development.* Packt Publishing, 2007.

[7] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version Control With Subversion for Subversion 1.6: The Official Guide And Reference Manual.* CreateSpace, 2010.

[8] EstimationTools, trac-hacks.org/wiki/EstimationToolsPlugin.

[9] Bitten, bitten.edgewall.org.

[10] EclEmma, eclemma.org.

[11] Infinitest, infinitest.github.com.

[12] MagicDraw, magicdraw.com.

[13] G. Reese and A. Oram, *Database Programming with JDBC and Java.* O'Reilly, 2000.

[14] M. Robinson, P. Vorobiev, P. A. Vorobiev, D. Anderson, D. Karr, and J. Gosling, *Swing*, 2nd ed. Manning Publications, 2003.

[15] R. Hitchens, *Java NIO.* O'Reilly, 2002.

[16] JBoss Netty, jboss.org/netty.

[17] C. Gülcü, *The Complete Log4j Manual: The Reliable, Fast and Flexible Logging Framework for Java.* QOS.ch, 2003.

[18] Google Guava, code.google.com/p/guava-libraries/.

[19] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice.* Addison-Wesley, 2005.

[20] P. Tahchiev, F. Leme, V. Massol, and G. Gregory, *JUnit in Action,* 2nd ed. Manning Publications, 2010.

[21] Mockito, code.google.com/p/mockito/.

[22] M. Cohn, *User Stories Applied: For Agile Software Development.* Addison-Wesley Professional, 2004.

[23] D. T. Sato, H. Corbucci, and M. V. Bravo, "Coding dojo: An environment for learning and sharing agile practices," in *Agile, 2008. AGILE'08. Conf.* IEEE, 2008, pp. 459–464.

[24] T. Swicegood, *Pragmatic Version Control Using Git.* Pragmatic Bookshelf, 2009.

[25] Lego4Scrum, www.lego4scrum.com.

[26] R. T., "The agile approach in an undergraduate software engineering course project," in *Proc. 33rd IEEE Frontiers in Education Conf.* IEEE, 2003, p. 5.

[27] J. Bowyer and J. Hughes, "Assessing undergraduate experience of continuous integration and test-driven development," in *Proc. 28th Int. Conf. Software Engineering.* ACM, 2006, pp. 691–694.

[28] L. Pinto, R. Rosa, C. Pacheco, C. Xavier, R. Barreto, V. Lucena, M. Caxias, and C. M. Figueiredo, "On the use of scrum for the management of practcal projects in graduate courses," in *Proc. 39th IEEE Frontiers in Education Conf.* IEEE, 2009, pp. 1396–1401.

[29] D. F. Rico and H. H. Sayani, "Use of Agile Methods in Software Engineering Education," in *Agile, 2009. AGILE'09. Conf.* IEEE, 2009, pp. 174–179.

[30] R. Lingard and S. Barkataki, "Teaching teamwork in engineering and computer science," in *Proc. 41th IEEE Frontiers in Education Conf.* IEEE, 2011.

[31] C. Scharff, "Guiding global software development projects using scrum and agile with quality assurance," in *Conf. Software Engineering Education and Training.* IEEE, 2011, pp. 274–283.

[32] V. Devedzic and S. R. Milenkovic, "Teaching Agile Software Development: A Case Study," *IEEE Transactions on Education,* vol. 5, no. 2, pp. 273–278, 2011.