Informatics in Education, 2008, Vol. 7, No. 2, 149–158

Crafting, Crafting, Crafting – Extreme Programming in Classroom?

Marcus BITZL

Technical University of Munich, Department of Informatics, Didactics in Informatics
Boltzmannstr. 3, D-85748 Garching
e-mail: marcus@bitzl.com

Received: April 2008

Abstract. *Extreme Programming* (XP) shows several interesting approaches which are very attractive for education. It is centered around early and incremental creation of working software. In the following, the chances XP offers for class are shown – especially for use in a class project, but also for practical phases in all lessons where programming is useful. Finally several common problems which can occur with XP will be shown as well as how to deal with them to make the use in class as smooth as possible.

Keywords: extreme programming, XP, education, pair programming, project class, independent work.

1. Introduction

Extreme Programming (XP) is an agile software development methodology founded by Kent Beck. On a closer examination it shows several very interesting approaches for use in classroom which will be shown in the following. Because of XP's educational values, Brüggemann et al. chose it for their projects in professional education in IT companies (Brüggemann *et al.*, 2006). But scientific papers concerning the use in schools are very rare. Thus, XP will be explained below and interesting aspects for the use in school will be especially emphasized. In addition, different applications and solutions for important problems concerning its use in class will be shown.

But first: what does "use in school" mean? The term school is cleared at once: where this paper relates to a special type of school, the Bavarian Gymnasium is meant because of its large amount of informatics lessons, but everything in this paper can be easily generalized to other school types. When it comes up to "use", there are mainly three possibilities: first, one could apply XP in practical phases in a lesson; second, one could use XP in a class project, and finally, one could make an own elective creating software with XP (e.g., an "XP-Team").

The main source for this paper is "Extreme Programming Explained. Embrace Change" from Kent Beck (Beck, 2000), in which he presents XP. All statements on how XP works are from this book.

2. Roles

For XP, Beck defines several roles (Beck, 2000) which are important for its understanding: the central role is the one of the *programmer* and is similar to the one of other programmers, but with slightly shifted priorities. The programmers' main objective is communication between each other and with all other roles as shown later. The *customer* is in XP also part of the team – no matter if it is an external customer or the marketing section of the same company. It is crucial that the customers are real users which would really work with the software that is built.

Beside these, there are more roles: the *tester* helps the customer to create function tests and is responsible for the regular execution of these test, if it is not already automated (unit- and integration-tests are part of the development and therefore tasks of the programmers). The *time manager* is responsible for the project's schedule. The time manager is kind of the team's conscience as he (or she) is responsible for adhering the schedule or for its correction if adhering is impossible. He also watches the team's performance to help them making better effort estimations[1]. The time manager should stay in the background for not to disturb work more than necessary.

The *Coach* is responsible for the overall process. His job is to direct people in a certain direction if necessary. His interferences should be as few as possible. Moreover, it is recommended to prefer an indirect guidance over a direct one, so people (mainly programmers) could find a better way by themselves.

As XP programmers are rather rarely specialists in their customers' area of application such specialists are taken for a certain time as *advisors*.

The Big-Boss is responsible for the team's motivation, but also for all organizational concerns like staff or scheduling. The team takes its problems to him and expects his help to solve them. If everything works fine, the Big-Boss should stay in the background – his intervention would perturb the team's productivity.

What do these roles mean for use in class? The students get in any case the programmers's role, some of the could also be customers instead. The teacher should take the other roles. In a class project or a practice class it would be possible for some roles (e.g., the advisor) to invite other persons for some time.

3. Values and Principles

3.1. *Values*

According to (Beck, 2000), Extreme Programming is based on four values which can be found in all aspects: open and honest *communication* has to take place throughout the whole project, between the pair-programming partners, between all developers, the management and also with the customers. This is necessary to produce high quality software.

[1] Here we can already see an important point of Extreme Programming: instead of dictating a schedule and forcing its adherence at any price, one rather aims for the best effort estimation with inclusion of the programmers.

All solutions have to follow the principle of *simplicity*: an implemented solution may be only as complex as necessary to meet the requirements. If an existing solution is too limited a refactoring, that is a change of the code structure, is applied.

To ensure high-quality solutions, some *feedback* is needed: from the customer to meet his requirements, from the other developers to find the best solutions and to learn. And also from the software itself feedback is demanded by tests to ensure that all requirements are fulfilled and to avoid defects. The fourth value is *courage*: it takes courage to address problems and conflicts frankly and openly. It also takes courage to replace already working parts of the software with new ones when it is necessary. XP always takes courage for change and for experiments, if an appropriate solution is not clear enough, and at last courage to take the responsibility for the different tasks.

If one considers these values together with the constitution of the free state of Bavaria, or the profile of the Bavarian Gymnasium, one can find several interesting parallels: the constitution already emphasizes the importance of a sense of responsibility and the disposition to accept responsibility as an educational objective. This is taken up in the profile of the Bavarian Gymnasium and supplemented with skills like the ability to communicate, the ability to work in a team, conflict handling skills and the ability to make sound judgements.

3.2. *Basic Principles*

To use the values above practically, Beck derives five basic principles: mainly based on behavioristic learning theories, but also to recognize problems as early as possible, *rapid feedback* is demanded. Technically, one gets this with automated tests. For people, stand up meetings and personal discussions are used.

Second, the developers should *assume simplicity*, that is solving any task only as it is given at the moment – not as it will be later or even as it could be later. This is exactly what the customer wants and pays for in the next release. Also, this is the easiest solution to understand and to maintain. For class, this is good because of three reasons: the students get solutions in less time, realize an immediate relation between their work and the resulting use (there is nothing that is not used at the moment) and with a latter extension, they can see the limitations of their present solution and learn new approaches.

As the project is under constant development, the third principle is *incremental change*: large and extensive changes are error prone and therefore difficult to control. If large changes are split into sequences of many small changes, every single one could be very well controlled and realized. This is also very useful for school, as it supports students in realizing changes and understanding their consequences. Moreover, they get working versions even before all changes are implemented.

This can only work, if everybody *encourages change*. This should also be visible in the way how solutions are chosen: the best ones are those solving the problem (without violating another principle) and offer the most options for future development.

In every single aspect, *high quality work* is demanded: if the created software is of poor quality, it is no fun for the developers to work on it. This has negative consequences for the quality of their further work and therefore for the success of the whole project.

In school it would surely be the same: software of poor quality (as one would see from many errors and instabilities) would harm the motivation of all participants seriously.

3.3. *More Principles*

Besides the five base principles, there are ten more principles from which some special ones get further explained: software development is always also considered as a learning process. Therefore, *teaching to learn* is a crucial task for the coach. *Systematic experiments* are also part of the XP's everyday life helping to understand concepts or to select solutions (thus, learning again). Furthermore, each developer has to take responsibility, a competence already demanded by the Bavarian constitution.

Technically, small initial investments are demanded: the project is naturally quite small when it is started (as it didn't exist before). Therefore, starting with too much resources can lead to problems. How to solve this will be shown later.

The principle "*playing to win*" has to be understood in contrast to "playing not to loose": our goal has to be to develop a project in intensive communication with our customer and delivering it in high quality. Everything else would harm the project and therefore also the developers' motivation and the effectiveness of software development. This is also very important for school because students expect a well working result at the end of a project. A low quality product would be certainly demotivating, and a failed project would be disastrous, regardless if the rules of a certain process model have always been followed correctly.

In every aspect of the development process, we need *open and honest communication*, also one has to *adopt XP to local conditions*, use *people's instincts* (not work against them) and *travel light* – that means, not to use any unnecessary processes or software.

When using tests, metrics or simply when speaking with developers and customers *honest measurement* is important: if measurement results or information retrieved from discussions are not "nice", this is an important hint that something doesn't work properly or could be done better.

4. Organisation

4.1. *Activities*

According to Beck, the four main activities for XP programmers are *coding, testing, listening* and *designing*. Coding is the central point in XP, because it quickly leads to measurable results, one can easily quantify implemented features and clean code is suitable for communication about solutions. The implementation of an idea, an algorithm or a solution can be tested if they meet the requirements. Different solutions can be compared. This enables the programmers not only to find the best solutions or to improve the existing ones, but also to learn with every comparison, selection or solution they make. In school, the students can sample theoretical concepts with concrete problems and solutions. The short times till students achieve results and the fact that they can easily quantify their work and understand their code, support a positive feeling of success and therefore supports independent work.

Permanent testing ensures that all implementations fulfil their requirements. Therefore, one creates tests before creating the corresponding implementation. The execution of these tests should be automated, so the programmers can execute them easily and often. In addition, Extreme Programming uses these tests as parts of the documentation, because one can clearly see there the expected behaviors. A meta-analysis done by Jeffries und Melnik (Jeffries and Melnik, 2007) shows the use of this approach: out of 16 studies 11 showed a considerable improvement of software quality. Only one states a negative influence on software quality and only four do not show any effect at all.

For school, this approach is interesting for different reasons: first, the students get a positive (or negative) feedback on their implementations immediately. Therefore, their erroneous work is still present and an immediate feedback after they corrected the mistake leads to an increased learning effect according to behavioristic learning theories (see also the principle of *immediate feedback*). With unit tests, errors get narrowed down to the smallest possible code regions. Feedback from tests also supports the teacher and also encourages the students' independent activities with concrete information on errors. Moreover, the exact way of working as demanded in the profile of informatics education (see (Fachprofil Informatik G8, 2004)) is supported as tests only accept exact solutions to their requirements.

The third activity – *listening* – is based on a basic problem in software engineering: misunderstandings between the customer and the developers. Therefore, developers have to learn how to listen and have to learn listening and how to do this actively in a discussion, to individuals as well as to large groups. Part of this is to ask questions and to develop a basis for common communication. Moreover, developers help the customers to understand which things are hard and which ones are easy to achieve. In school, this strong focus on communication supports students by improving their communication skills as for example demanded in Bavarian curriculums.

These activities alone could help us only for a short time, because if good design is neglected, code soon gets unmaintainable and almost impossible to change. Therefore, the last activity is *design*. This is not a phase, but a continuous process interrupted by other activities or melted with programming to one unit. This is very useful for class as implementations immediately follow on design, changing in very short cycles. This allows students to experience the consequences of their good (or bad) design immediately and to realize how appropriate design simplifies their solutions.

Designing in XP is different from other software engineering methods: first, programmers implement a test case and a piece of code solving the current task in the simplest possible way. For the next task, they extend the test case and change the existing code to pass the tests. Doing so, they take care that the code doesn't contain anything which is not necessary at the moment. For the third, fourth and any further tasks they do the same, until all tasks are done. This is also expedient for use in lessons: breaking up the whole problem into many pieces reduces it to a set of small problems which are easy to solve, and every one is independent from the following changes. Therefore, students can focus on their problems without being distracted by much stuff not necessary at the moment and whose purpose is yet hard to understand. Astrachan et. al suggest to give students (in

his case from university, but for this technique this doesn't really matter) a task for which they should find a solution which is as simple as possible. The following tasks demand a gradual extension of the existing solutions leading slowly to a certain, very elegant design. He reports a strongly improved learning success compared to assignments solved with a design the teacher had only presented his students (Astrachan *et al.*, 2003).

4.2. *Workflow*

An XP-project's workflow consist of two parts: during the *planning game*, the project plan is created or adapted together with the customers and so the software development is controlled. On the planning game follows the creation of the software. If requirements change we repeat the planning game for the changed requirements (also during a creational phase) and the project plan is adapted. Afterwards, the creation can be continued without any problems. If a new version of the software should be developed, the planning game is done again.

How is the extent of a version determined? According to XP, this can only be done with coarse communication with the customers. This happens in the so called *planning game*, a process consisting of three phases (see Fig. 1) which will be further explained in the following: in the *exploration phase* the customers create story cards describing the desired features. The developers estimate for every story card, how long it would take to implement the features. If this is not possible, they talk to the customers which will explain their ideas in a more detailed way or split the story card into several smaller ones.

Fig. 1. Phases of the planning game.

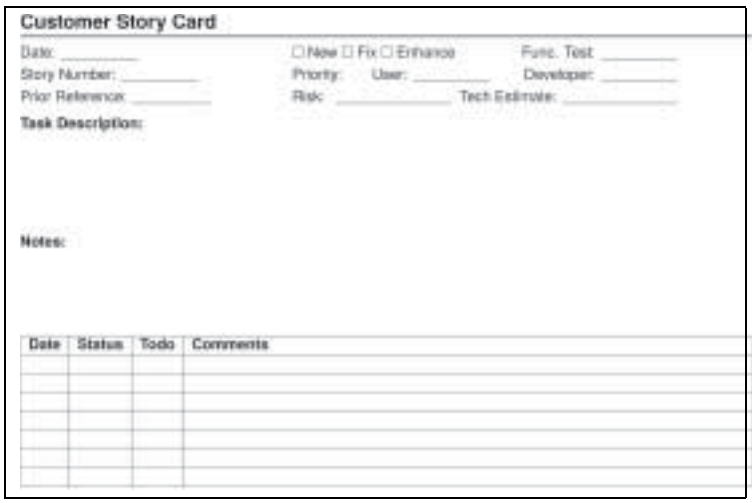

Fig. 2. A story card.

In *commitment phase* all story cards are sorted into three stacks: first, by the customer by value (absolutely necessary, important and nice, but less important), and second, by the developers by risk. This is measured by how closely the necessary time can be calculated (almost certain, closely and hardly to not at all). Afterwards, the developers tell the customer, how fast they can be in ideal development time per month.

At last, the *steering phase* follows: the customers choose from the selected story cards those, which should be implemented in the next iteration (usually a period from one to three weeks). After the first iteration the system has to work, but doesn't have to be fully developed. If the developers realize that they have overestimated their development speed, they can correct the schedule together with the customers and remove less important features from the plan. If the customer already needs some feature for the next release which is not in the plan yet, he can write a story card for this and replace another story card needing the same effort. If the programmers get the impression that the plan might lose touch with reality, the plan has to be estimated again. Therefore, in XP the customer is always a member of the team and has always to be available for developers' questions. This is called 'customer in place', although physical presence is not needed all the time.

For the creation of software several techniques are used: a *metaphore* is used for communication, for an intuitive understanding of the project's aim and also for a common design. For an accounting software it could look for example like the following: "a virtual accountant collects and manages all accounting orders, evaluates them and makes nice charts". The metaphor should support a *simple design*. In the case of our virtual accountant it could be like this: one package for managing transactions (the "accountant" himself), one for analysis and one for making charts. After some time it could prove useful to move the "accounts" from our accountant to a new package of its own – refactoring simplifies design even more. If this software would need interfaces to other accounting solutions, the introduction of a "translator" package could solve this.

To do *refactoring* easily without accidentally destroying other features, continuous testing (with unit tests and continuous integration) is necessary.

A very central aspect of XP is *programming in pairs*: at any time, two programmers do the programming together. One of them has the active part and writes code, the other one follows his line of thought, asks questions and tries if he (or she) can find a solution that is simpler. If they are not sure if a solution is good, they give it a try and maybe they also try another one until it is clear which solution is the simplest. The partner helps to identify problems earlier, which saves time and energy. Together, pair programmers might find solutions each of them alone wouldn't. The partners can switch their roles anytime.

For school all these aspects are advantages, from debugging to creative problem solving and, of course, learning from each other. Werner et. al. describe a pair-programming-session with middle school girls and introduce some rules for successful pair programming with students (Werner *et al.*, 2004). A survey from McDowell and Werner (McDowell *et al.*, 2003) examines the effects of pair programming in programming courses for beginners. The participants using pair programming showed better results than those who did not.

Common responsibility is another instrument for quality assurance and part of a XP working culture. Every developer is responsible for the whole project. Regularly changing the partners of the programming pairs ensures that all programmers get knowledge of the whole project by and by. If one sees something which he (or she) could do simpler or nicer, he (or she) has to make this change (except somebody else is working on the same piece of software). Especially there are no "private" parts of the software which are off-limits for other programmers. Because of this and also because of design through refactoring, it is very common that parts of the software are completely replaced by others within a few weeks. The tests ensure that changes can be performed without any danger of breaking the software. For students, this is, of course, a chance to learn, but also to take responsibility.

The rule "*40-hour-week*" demands a project plan which allows the programmers to fulfill it without making overtime regularly. This would reduce their manpower as well as their motivation and therefore harm the project. For school, this is very important as overtime is hardly possible while the schedule is very tight.

As many programmers work on the same project, XP demands *coding standards*. These can help students to develop a clean structure for their code.

5. Problems and Solutions

Several circumstances at school make the use of Extreme Programming difficult: the *group size*, the available amount of *time*, the *experience* the developers have, the *customer in place* and the *immediate assignment of all developers*.

The group size has an important role in XP, because the integration process is a bottle neck. With estimated four minutes for each integration process, ten pairs (twenty pupils) would need 40 minutes to integrate their results. Smaller groups and projects with low interference between the developer pairs can speed this up. The elective solution has one crucial advantage: perhaps the group size one can expect is in acceptable limits, and if not, one could control these factors from the outset.

Another critical factor is *time*: from the 45 minutes of a lesson, a certain amount of time is already lost for other activities (pupils have to come to the computer room, take their places and make pairs, there might be organizational work to do from the school, etc.). In the remaining time the pupils have to take a story card, extract their tasks, implement their tests and solutions and finally integrate their solutions. The teacher as coach supports them when there are problems, but otherwise stay in the background.

Here it becomes apparent that the working techniques cannot be introduced only for a class project, but have to be familiar to the students at least partly. For this, there are different approaches: in practical phases, the complexity can be easily controlled, for the used working techniques as well as for the tasks. In class projects, this is more difficult, but with an appropriate control of the planning game it is possible in certain limits. Another solution is to split the class and give together with another teacher of this class two double periods at the same time. This way, one could get double time and half group size. To use this method, it has to be checked how to deal with the teachers' overtime.

The pupils' lack of *experience* could be faced with appropriate tasks and a stepwise introduction of the XP-techniques. Before pupils can work independent in a class project, they should have made first experiences with pair programming and writing software tests in practical phases in class. In an elective, they could make first experiences with software tests together with the teacher and first experiences with pair-programming directly. For the introduction of pair programming, a version of pair programming proposed by Astrachan et al. (Astrachan *et al.*, 2003) would be appropriate. In this version, the teacher works as active and the whole class together as passive programmer in one programming pair. The class would take part in the solution with ideas, proposals, but also with questions.

The *customer in place* could be replaced with students taking the customer's role. Doing so, it is also possible to make pairs with an odd number of students and perhaps to slightly lower the number of developer pairs (probably by one or two pairs). It also gives a chance to assign certain, more appropriate tasks to some students (e.g., for motivational purposes).

An *immediate assignment of all "developers"* is hard to avoid in school as we always have to deal with whole classes (or at least large parts of them if they are split), where all students have to be present. Obviously, each of them (or rather, each pair of them) needs an expedient task. One solution might be to give a small code basis to the students at the beginning of the project to avoid that they would disturb each other. Part of this code basis could be developed in practical phases of past lessons or through the teacher-class-pair-programming described above. It would also be possible to start with a part of the students programming, while others do some investigation work (or other not interfering tasks). At last, projects with low coupling between its parts are better.

6. Conclusion

Examining the values, the principles, the basic working techniques up to the whole project flow, it is obvious that Extreme Programming offers many interesting aspects for class. XP emphasizes learning as an important part of the principles and practices in many points and therefore is very appropriate for use in schools. Above three basic scenarios have been presented and will now be summarized shortly.

The use in practical phases in a lesson takes the least time and gives the teacher the strongest control. Here, the students can learn the practices together with their teacher. With design through refactoring, students can learn the advantages of certain designs, and programming in pairs supports the students in finding good solutions. The creation of tests supports them in recognizing their mistakes.

For the use in a class project, one needs more time and the students have to know the working techniques. The teacher's control is much smaller and in many cases limited to support at problems. Software tests ensure always a working version of the program. Small iterations allow a fine risk-control and a smooth change of the project plan, so one can ensure that the students will have working software. An XP-elective offers several other opportunities: because it has longer period than a class project and smaller group sizes, the techniques can be learned more intensively and also can be applied much better. This allows larger projects.

No matter in which form Extreme Programming is used – the student's activities will always be the most important aspect.

Acknowledgments. I would like to thank Daniela Drexler and Elisabeth Rothe for their valuable comments on a manuscript of this paper, and also Prof. Dr. Peter Hubwieser for guiding me to research.

References

Astrachan, O., Duvall, R.C. and Wallingford, E. (2003). Bringing extreme programming to the classroom. In M. Marchesi, G. Succi, D. Wells and L. Williams (Eds.), *Extreme Programming Perspectives*. Addison Wesley, Boston, MA, 237–250.

Beck, K. (2000). *Extreme Programming Explained. Embrace Change*. Addison-Wesley, Reading, Mass.

Brüggemann, A., Rohs, M. and Schwill, A. (2006). Extreme programming – extreme learning? Erfahrungen aus dem Grenzbereich zwischen Arbeiten und Lernen. In C. Schulte and M. Thomas (Eds.), *Proceedings Didaktik der Informatik, 3. Workshop der GI-Fachgruppe "Didaktik der Informatik"*. 19–20 Juni 2006. Kölle, Potsdam, 7–16.

Cleland, S. and Mann, S. (2003). Agility in the classroom: using agile development methods to foster team work and adaptability amongst undergraduate programmers. In A. Williamson (Ed.), *Proceedings of the 16th Annual NACCQ*, 49–52.

Fachprofil Informatik G8 (2004). `http://www.isb-gym8-lehrplan.de/contentserv/3.1/g8.de/index.php?StoryID=26380`

Jeffries, R. and Melnik, G. (2007). TDD: The art of fearless programming. *IEEE Software*, **24**(3), 24–30.

McDowell, C., Werner, L., Bullock, H.E. and Fernald, J. (2003). The impact of pair programming on student performance, perception and persistence. In *Proceedings of the 25th International Conference on Software Engineering*, 602–607.

Schulartprofil Gymnasium: Das Gymnasium in Bayern (2004). `http://www.isb-gym8-lehrplan.de/contentserv/3.1/g8.de/index.php?StoryID=26350`

Verfassung des Freistaates Bayern (1946). `http://www.verfassungen.de/de/by/bayern46-index.htm`

Werner, L., Denner, J. and Bean, S. (2004). Pair programming strategies for middle school girls. In *Computers and Advanced Technology in Education*, 301–305.

M. Bitzl is a teacher student at the Technical University of Munich. He has studied computer science, physics and educational science and is now doing his first state examination for qualifying as a teacher at Bavarian Gymnasium. His main research interests include the implementation of independent learning and ontologies both for e-learning and semantic web.

Mokymas, mokymas, mokymas – ekstremalus programavimas pamokose

Marcus BITZL

Ekstreminis (kraštutinis) programavimas (anglų k. santrumpa XP) turi įdomių savybių, ypač tinkamų mokymui. Šis programavimas iš esmės susijęs su programinės įrangos, kūrimo darbais. Straipsnyje aptariami būdai, kaip taikyti ekstreminį programavimą mokant klasėje, ypač atliekant klasės projektus, tačiau tuo neapsiribojama – parodoma, kaip panaudoti ir kitose pamokose, kur tik programavimas galėtų būti naudingas. Aptariamos kelios svarbiausios problemos, kurios gali iškilti naudojant ekstreminį programavimą, taip pat pateikiami šių problemų sprendimo būdai.