Process Model Extreme Programming

Jan Dielewicz
jadi04@student.bth.se

Kai Petersen
kape04@student.bth.se

Sebastian Stein
sest04@student.bth.se

Kashif Ahmed Khan
kakb04@student.bth.se

Abstract

Today software is developed by applying a defined process - a software process. The application of a software process shall ensure to fulfill the needed quality within a fixed time-frame and budget.

In the science of software engineering several software processes were developed and defined in the last 30 years. All software processes have specific advantages and disadvantages. In the last years the software process Extreme Programming got high attention from the software engineering community, because Extreme Programming denies the validity of several software engineering dogmas.

Taking a short look Extreme Programming's radicalism can be explained with the extreme application of known software engineering practices. A more detailed investigation shows instead, that Extreme Programming is based on a complete different world view.

This report starts by describing this new modern world view. Afterwards the software process Extreme Programming will be described in detail. At the end there will be a discussion of Extreme Programming's advantages and disadvantages. This discussion will show, that Extreme Programming can not be applied in all software projects, but that Extreme Programming has some valuable ideas. This discussion will be completed with an example how to apply Extreme Programming in large-scale projects.

1. Introduction

Software is the product of a software vendor. The software vendor develops software for a customer. Thereby it does not matter if it is an external or internal customer. Mostly the software development is conducted in projects. The characteristics of a project are presented in the following [17, p. 4]:

1. A project has a purpose.
2. A project is unique.
3. A project is temporary, it has a limited time-frame and the project organisation is disbanded at project end.
4. A project involves people from different departments and it cuts across organisational lines.
5. A project comes always with a risk to fail, because a project deals with uncertainty.

The in point 4 mentioned involvements of different people are labelled as the project team. Every project member can act in different roles, e. g. as a project manager.

Customers and software developers mostly fix budget, time, requirements and quality standards at the beginning of the project. For the success of the project on the part of the software developer it should fulfill the requirements and quality standards. Furthermore the project should be completed within time and with lesser costs than agreed.

For the successful conduction of the project the software developer mostly follows a software development process model. A process model describes a set of activities and their inputs and outputs. Inputs and outputs are referred to as artifacts. Furthermore each activity is linked to a role which has to be filled out by one of the team members.

Actually a lot of process models like the waterfall model, V-model as well as a set of incremental and plan driven approaches exist.

Extreme Programming is a further process model for the arrangement of the software development process. Extreme Programming is characterised by its radical nature. It lays aside generally accepted dogmas of software engineering such as late changes occur significantly higher costs, detailed documentation as a factor of success and up-front-design. A lot of articles about Extreme Programming can be found in expert databases. According to [1] there is few empirical evidence for the value of Extreme Programming in practice, but the method itself is often discussed in journals and proceedings. Unfortunately often the philosophy

behind Extreme Programming is not considered, but without this consideration there is no whole picture of what Extreme Programming really is. Therefore the philosophy behind Extreme Programming is presented first. After that the planning process is described. At the end the advantages and disadvantages of Extreme Programming are discussed.

2. Philosophy behind Extreme Programming

2.1. Mechanical World View

Over centuries, the science is determined by the mechanical world view. The basis for this point of view was laid by Isaac Newton in 1687 with his work "Philosophiae naturalis principia mathematica". According to this work the world is a complex machine which is based on a few laws of nature. Furthermore the present condition can be examined exactly at any time. With the entire knowledge of nature laws and the actual condition of the world the possibility of calculating future and past conditions is given. The implication of this is that the world's future development can be calculated exactly.

This theory was reflected in all scientific disciplines. Julien Ofray de la Mettrie for example presents the human being as a machine in his work "L'homme machine". In the field of economics this theory was also accepted, e. g. by Adam Smith or Frederick Winslow Taylor[1]. At last the most complex machines built by humans are computers. The development of the computer can be led back to the calculating machines constructed by Bacon and Pascal. Those machines are using mechanic mechanisms for calculation, similar to clockworks.

In the field of software engineering the mechanical world view can be found as well. In general it can be assumed that software can be constructed and developed successfully on the basis of predefined software development processes. As a result of this software development can be planed and all risks can be reduced significantly.

2.2. Modern World View

The mechanical world view is a valuable basis for sciences. Developments in all scientific fields like medicine and technique gained a lot from this theory. Nevertheless the science explores new territories where the mechanic law of nature is not valid. An example for this can be found in physics. An exact description of the movement of electrons and neutrons based on the mechanic law of nature is not possible. Therefore, since the 20th century has begun, quantum mechanics, Einstein's theory, quantum physics and quantum mechanics turned away from the mechanic law of nature. No name exists for this new point of view, therefore it is just called modern world view.

The modern world view is of course not only applied within physics and chemistry, but also in all other sciences. Management has overcome the taylorism and has a totally different point of view nowadays. Another example is stated by Arthur [2], who says that the information technology industry doesn't follow classical market theories as postulated according to Adam Smith.

The modern world view is based on the assumption that the present condition of a system can not be determined exactly and that the development of a new system processes discontinuously. This leads to the conclusion that it is not possible to calculate the future condition of a system. Different theories describing such systems exist like:

- Autopiesis (social science) [14]
- Synergetik (physics) [11]
- Chaos Theory [12]

The modern world view is the basis of the Extreme Programming philosophy. The subtitle of the original Extreme Programming book [4] shows that the author accepts change and that he is willing to utilize it. Therefore it can be assumed that this modern world view describes the philosophy of Extreme Programming. The author accepts the instable environment around the software development process. As a consequence approaches have to be discovered which are appropriate referring to the instability of the environment. As Kent Beck said: "Change is the only constant." [4, p. 28]

After the presentation of the philosophy behind Extreme Programming the ideas and principles of Extreme Programming are presented.

3. Core Concept and Planning

3.1. Introduction

Extreme Programming is not a result of research, this approach was invented by Kent Beck who is a practitioner. Therefore, Extreme Programming is based on his practical experiences and the experiences of people who worked together with him. People who support Extreme Programming especially refer to the C3 software project for Daimler Chrysler as a successful Extreme Programming project. However, the hardest opposers of Extreme Programming also refer to this project because today it is not clear if the project really was a success.

Beck's first published work about Extreme Programming [4, p. 28] mainly consists of topics like the concrete development process, requirements engineering, design and test.

[1] The principles of scientific management

Only a few hints are given to the planning process. The planning process is presented in more detail in his second published book [5] about Extreme Programming. According to this published work of Kent Beck the following section is structured. First, some fundamentals and the Extreme Programming process are presented. After that the planning process is discussed.

3.2. Four Basic Values

Extreme Programming consists of four fundamental values [4, p. 29ff]:

- Communication
- Simplicity
- Feedback
- Courage

Extreme Programming is totally based on this four fundamental values. They can be referred to as the fundamentals of a project culture and it should be assured that all team members internalise these values.

Extreme Programming tries to assure the *communication* between developers, customer and management by formulating a set of core practices. Communication is essential for successful cooperation and teamwork. Since communication is so essential it has to be supported and maintained because it can be disturbed easily. A so called coach is responsible for frictionless communication. If the communication is disturbed somehow the coach should interfere actively.

Extreme Programming demands *simplicity*. The aim is to produce most simple solutions for problems. Developers should not consider in advance what kind of program details could be necessary in the future. This anticipation may lead to wrong conclusions which occurs higher costs. The easiest way to solve a problem can be recognised by frictionless communication. Furthermore the architecture should be a simple construction because: “The simpler the system, the less you have to communicate about it.” [4, p. 31] Extreme Programming itself denies complex process models like Rational Unified Process and requires a simple set of rules. Therefore the principles and fundamentals of Extreme Programming can be presented on a few pages.

Furthermore Extreme Programming demands *feedback*. Feedback can be given by conducting tests. Tests help to inform the developer on the current system’s status. As a consequence of this, defects can be detected and corrected immediately. Also the customer should get reports about the project’s status frequently. This gives him the opportunity to interfere if something is going wrong. Another advantage is that the customer gets feedback on his demands, how far the developers understand his demands and what efforts are necessary to satisfy the customer’s demands. The customer gets early feedback in the form of a running system. This gives gives the customer the opportunity to influence the implemented solution in a direct manner.

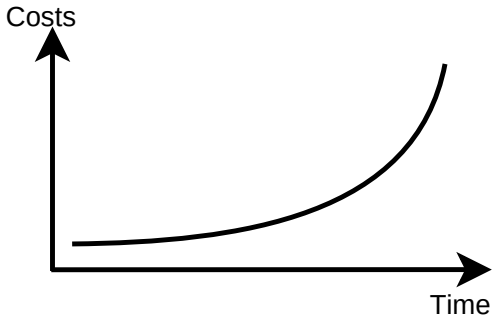


Figure 1. Dependency cost of change over time by Boehm [3]

Courage is needful to conserve the before mentioned principles consistently. A lot of rules defined by the own organisation have to be broken. Extreme Programming claims to destroy not working code and to start from the beginning. Furthermore a trustful and honorable relationship should be established between customer and developer. The customer has to be informed if delays occur. This proceeding surely requires courage.

These four fundamental values are working together and they support each other. Surely it would be possible to formulate the fundamental values with other expressions. Besides the fundamental value the hypothesis exist that late changes occur significantly higher costs than early well known changes. This hypothesis is presented in the next section.

3.3. Cost of Change

As mentioned before[2] Extreme Programming declares some dogmas of software engineering as invalid. One of those dogmas states that for late changes in a project the fixing costs will be much higher compared to early changes. The exponential correlation is shown in figure 1 on page 3. This theory was proclaimed by Barry Boehm [3]. A logical consequence from this theory is to try to identify all necessary and potential changes as early as possible to prevent late changes. Therefore many different process models include an intensive phase for analyses in the beginning of the software project. If software development is compared to house construction, it is easy to understand this theory.

[2] section 1 page 1

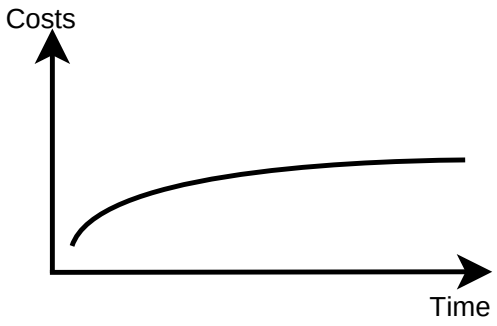


Figure 2. Dependency cost of change over time by Beck [4]

If the base-plate is once set up, the construction of an underground parking will cause tremendous additional costs compared to the approach which would have included the underground parking in the beginning.

In opposition Beck [4, p. 21ff] assumes that the costs depending on the time only increase slowly and asymptotic reach a maximum level. This is shown in figure 2 on page 4. Beck motivates that software can be changed, whenever the effects of the changes can be analysed by automatic testing. Especially when the design of the software is kept simple, the possibility for changes is increased. In this the four fundamental values at least partly find a representation: Simplicity of the design, feedback through automatic testing and courage to accept and realise changes are necessary.

When this ratio between time and costs is valid, the software development process can be designed in a total different way. An intensive analysis at the beginning of the project is not necessary anymore. The design of the software can be developed step by step and furthermore changes can be accepted throughout the whole project.

In the following the Twelve Core Practises of Extreme Programming are presented. Those Core Practises take advantage of the reversed cost coherency and include the four fundamental values.

3.4. Twelve Core Practices

Starting from the four fundamental values Extreme Programming defines 12 Core Practices [4, p. 53ff]. These Core Practises have to be applied in total to make Extreme Programming working. Beck [4] several times points out that these Core Practises are not new at all, but were successfully applied for many years in software development already. Indeed their combination and their consequent (extreme) use in the context of Extreme Programming are unique. The 12 Core Practises are:

- Planning Game
- Small Releases
- Metaphor
- Simple Design
- Testing
- Refactoring
- Pair Programming
- Collective Ownership
- Continuous Integration
- 40-Hour Week
- On-Site Customer
- Coding Standards

At this place the Core Practices will be described in detail.

Planning Game: The Planning Game has been revised since its initial publication [4] and will be discussed in section 3.5 on page 5 in more detail. During Planning Game the requirements are described in a story-like way[3]. For those stories the effort according to time and costs is estimated. Together with the customer the prioritisation is done. Furthermore the customer and vendor together define which stories are included in the next release of the software. On this basis the release plan is set up. Since the customer may change his requirements at any time, a Planning Game has to be conducted for every larger changes.

Small Releases: Each new release of the software shall be as small as possible, and shall include the most valuable stories. It is preferable to have new releases every one to three month. The frequent releases shall make it possible for the customer to use the most valuable features as early as possible. In addition those frequent releases shall ensure feedback from the customer.

Metaphor: Extreme Programming does not request for a distinct design phase. To be able to have a shared model of the software, the Metaphor is used. The Metaphor represents the basic idea of the system. On its basis the communication is build up as, e. g. terms and phrases used in the context of the Metaphor are used by the developers.

[3] similar to use cases

Simple Design: The design of the software shall at any time be as simple as possible. In detail this means that all test cases are met, there is no redundancy in the implementation, the Metaphor is implemented and classes and methods are reduced to a minimum. An anticipation of future requirements is not welcome, since it is never for sure that these requirements would not be changed or completely dismissed. In this case the implementation would have caused unnecessary costs.

Testing: During the implementation always a test is developed first, before the underlying source code is developed. This development approach is named as Test Driven Development. Extreme Programming distinguishes between two types of tests. Unit tests are written by the developer themselves and cover single methods or classes. Usually it is necessary to have many tests for a complete class. Functional tests are written by the customer, if necessary, with support of the developer. Functional tests cover complete features. After each change to the program code, each test has to be conducted newly. Only if all tests were passed, the new code is allowed to enter the system.

Refactoring: Since Extreme Programming renounces an explicit design phase, the software design has to be continually revised and improved. Very often it is necessary to simplify specific parts or source out program code into new methods or classes. These activities only change the internal structure of the source code, but never affect the functionality of the program [10, p. 53f]. This process is called Refactoring. In the last couple of years, many standard methods [10] were developed, to assist in identifying useful changes in design and implementing them.

Pair Programming: Every source code implemented in use of Extreme Programming has to be developed in pairs. This means that always two developers work at one problem at the same time. Usually both developers use a single computer. It might be necessary to adapt the workplace environment [20, p. 67ff]. Pair Programming is not about monitoring each other, but about developing solutions together. Ideally the pairs will be new arranged at least once a day. Pair Programming guarantees the knowledge transfer in between the developers.

Collective Ownership: Every developer has all rights according to the source code. He is allowed to make changes everywhere in the code. Therefore every developer is responsible for the whole program code.

Continuous Integration: As soon as a pair of developers has finished a feature, the new or changed code is integrated in the overall system. The integration is successful, when the system still passes all tests. There has to be at least one integration per day. Program code that could not be integrated at the end of a day has to be thrown away.

40-Hour Week: In Extreme Programming there is no overtime allowed for more than one week. When overtime is necessary during one week, there must not be any overtime in the following week. A permanent overtime indicates a general problem with the project that can not be solved by overtime. The necessity for overtime is only a symptom of a problem, but not its solution.

On-Site Customer: Extreme Programming demands to have the customer, or at least representative of the customer, in every team. The customer shall be accessible for the developers at any time. Furthermore he is the one to specify the functional tests and the stories, as well as to prioritise the different stories. It might be necessary for the customer to be supported by the developer to be able to do this.

Coding Standards: Since all developers work on the entire code, it is necessary to have a common style of writing the code. This means it shall not be possible to identify the person, who wrote a specific part by looking at the code. This includes that everyone shall be able to understand any part of the code.

3.5. Planning and Requirements Engineering

In this section the Extreme Programming software development process is described in detail. This includes the demonstration of the use of the 12 Core Practices. A graphical representation is given with figure 3 on page 6.

At the very beginning of each project, the project's scope has to be defined. For that purpose the system metaphor and a vision for the project has to be set up [5, p. 35ff]. During this so called architectural spike the technologies that may be applied in order to implement the vision, are investigated. The budget and the main areas of activities (big plan) have to be specified. This is followed by a rough effort-estimate of those activities. Furthermore all needed tools and systems like test-system and configuration management are set up.

The next step includes the initial Release Planning [5, p. 39ff]. Outgoing from the big plan the customer has to develop the requirements represented by the stories. The size of each story is estimated by the developers. Each story is not allowed to take longer than 2 or 3 weeks. If a story can not be implemented within this time-frame, the story has to be divided up into smaller pieces. In the other case it might be necessary to sum up some smaller stories into one story.

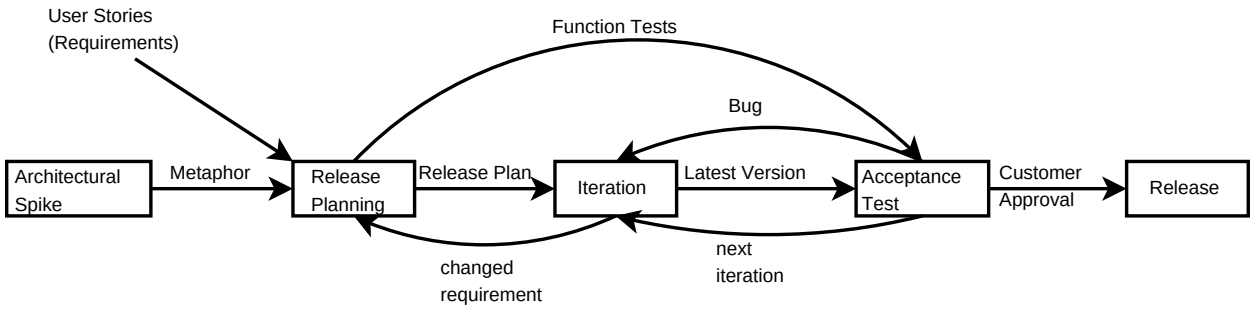


Figure 3. Global process flow of Extreme Programming [19, figure based on]

The story is described in the words of the customer. Technical issues are not included. The stories shall be described in only a few sentences. Furthermore for every story a test (functional test), which allows the validation, has to be described as well.

The customer has to order the stories by their business value. Doing this, the priorities are set. Technical interdependencies are of lower interest in this case. Further on the deadlines for the different releases are defined. The developers, relating to their experiences, define how much development can be done between two releases. On this basis the stories are related to the different releases. As the result the release planning delivers the release plan which displays all releases and their stories, ordered according to their prioritisation.

In contrast to a release, which is supposed to be published every one to two months, the development work is done in iterations, which are supposed not to take longer than one to three weeks. To reach the next release, couple of iterations are necessary. At the beginning of each iteration there is the iteration planning meeting [5, p. 87ff]. Within the iteration planning the stories are subdivided into tasks. The tasks represent the technical specifications, which are developed together with the customer. The developers sign up for the different tasks and distribute them to each other. Each developer estimates the needed time for the task he signed up for. The overall sum does not necessarily match the rough effort-estimate. In case a capacity overload is obvious, this is discussed with the customer. The customer then decides which stories have to be delayed to further releases. The iteration planning is always only done for the next following iteration. In Extreme Programming this is the detailed planning.

Now, the underlying iteration, shown in figure 4 on page 7 takes place. Each developer seeks for a partner to implement the task he signed up for. The project manager is not coordinating this process, but leaves the coordination to the developers. The pair first of all defines the unit test for their task. In case of any uncertainties, the developers confer with the customer. Not until the unit test is developed, the underlying code is implemented. It is possible that this makes changes to existing code necessary. In this case refactoring is applied. After a couple of hours the tested code is integrated into the overall system. By passing all tests, the integration is ensured.

During the iteration the progress is measured. Doing this makes it possible to determine the likelihood of being able to implement all stories planned for this iteration. When larger deviations occur, a new release planning has to be conducted.

As previously pointed out, the customer is allowed to change existing stories or add new ones. The developers estimate the necessary effort for the changes or the new stories. It might happen that the customer has to decide, which other stories has to be dropped according to the changes or new stories.

3.6. Organisational Aspects

Extreme Programming does not state how the organisation of the project should be structured. Every development team consists of team members having equal rights. This kind of organisation is not new; it was developed by G. Weinberg in 1971 and is called egoless programming teams.

Normally egoless programming teams or decentralised teams consisting of about ten or fewer members with different qualifications [13, p. 107]. Two principles of Extreme Programming were already applied by using egoless teams. First, the source code is not owned by the person who programmed it, each line of code is owned by the whole team. This implies that the source code is produced to achieve the project's goals, not to satisfy the ego of one programmer. Second, the principle of direct communication is applied in egoless programming teams. Everybody can communicate with everybody about every concern without any formal restrictions. The leading rotates within the team. The team leader becomes who has the best qualification to solve a current problem appropriately.

One problem of egoless teams is that they are hard to handle if the size of the team increases. Each new team

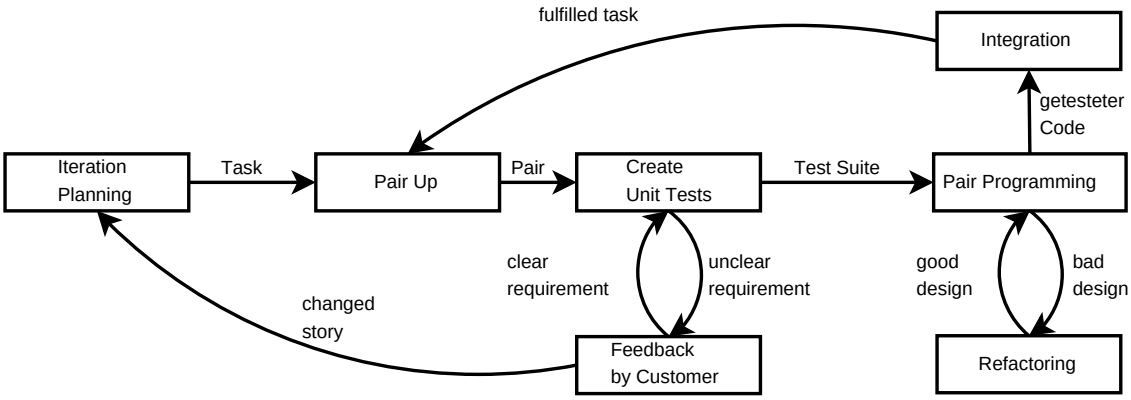


Figure 4. Process flow of an iteration in Extreme Programming [19, figure based on]

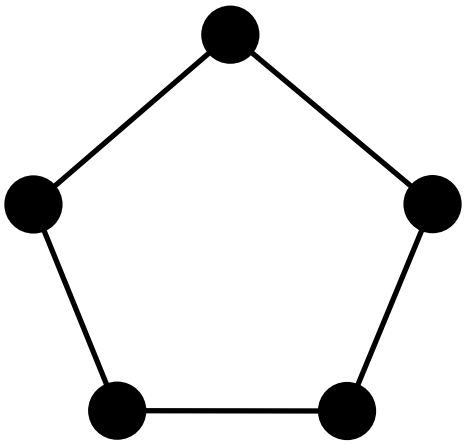


Figure 5. Organisational structure of egoless programming team [13]

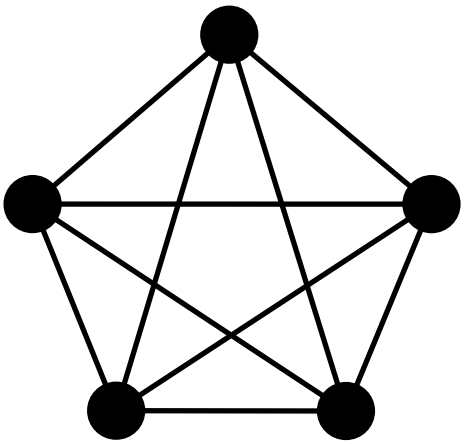


Figure 6. Communication paths in egoless teams [13]

member generates a large quantity of new communication channels. A egoless team of 40 programmers has already 1560 communicate paths. The number of communication paths can be calculated by the following formula:

$$communication\ paths = n * (n - 1)$$

In the formula above n is the number of team members.

By adding just one more person to a team of 40 programmers, 80 new communication paths must be added. The growth in communication paths can be described by big-O-Notation: $O(n^2)$. This means that communication paths grow exponential by a factor of two with growing n. In figure 5 on page 7 the structure of the egoless programming team is illustrated.

The needed communication paths in a egoless team of 5 persons are shown in figure 6 on page 7. It can be seen, that each person in the team establishes 4 communication paths.

3.7. Extreme Programming and classical approaches

Although Extreme Programming is totally different compared to classical process models, it seems as if the existence of some activities within software development and therefore in the world of projects is invariant. Within this section those activities, and how they relate to the 4 fundamental values and 12 core practices of Extreme Programming, are presented. As basis for the discussion the systems development cycle, as presented by Nicholas [17, Part II] is used:

Conception: The initiation of the project is the main event in the first phase, the conception phase. In this phase the customer or user identifies the problem and comes up with the idea to have a solution based on a software product. Since this phase includes a feasibility study and the

contract negotiation, the involved organisations at least have to have an idea what the problem exactly is about and how a solution could look like. Usually process models do not include this early stage of the project, and so does Extreme Programming.

Definition: The definition phase of the project includes the activities requirements engineering, system design and planning according to costs and schedules. The result is a detailed plan, which is basis for the customer either proceeding with the project or cancelling it. Now, in Extreme Programming this phase and the detailed plan is missing. Nevertheless, the planning game and the development iterations with their iteration planning meeting take the role of most of these elements. But in contrast to the techniques and methods suggested by Nicholas [17, Part III] Extreme Programming has a different approach: One will most likely not find any network scheduling, program evaluation and review techniques or resource allocation approaches in Extreme Programming. Extreme Programming is not relying on heavy weight methods, just as it tries to keep everything simple, the methods used should also be simple and as uncertainty is accepted, nothing is planned, which very likely is matter of change anyway. But nevertheless, planning *is* done. Extreme Programming will not present a detailed plan for the complete project with large and important milestones. Instead, those small development iterations take place. They are more like inch pebbles, not milestones.

Execution: During the execution phase the system itself is implemented. Referring to classical process models, this includes the implementation, testing, teaching and the rollout. These activities again, of course, are part of Extreme Programming. But they are split by those small releases. So it is not one phase, but several. Every development iteration in this meaning is one execution phase and every release delivers a software product to the customer.

Nicholas puts a lot of emphasis on the control process [17, pp. 340-378]. Again in Extreme Programming this is covered by the iteration planning meeting, but without using sophisticated systems or methods.

Operation: In the operation phase the customer uses the product. The vendor might stay being involved by maintaining and supporting the delivered system. Usually classical process models do not spend too much attention to this phase again. And it is the same with Extreme Programming. Nevertheless, in Extreme Programming maintenance is very easy to conduct. It is nothing else, but just adding one or some additional releases with all related steps, especially the development iterations.

3.8. Common Reasons for Project Failures and Extreme Programming

Nicholas identifies a couple of potential reasons for projects to fail [17, p. 537]. In this context some of the aspects belonging to the "Level III" shall be discussed:

Inadequate communication: On of the fundamental values is communication. Planning game, pair programming, on-site customer and the development iteration meetings are the answers Extreme Programming gives to this problem.

Noninvolvement of user: Feedback and communication are fundamental values of Extreme Programming. The small releases and the on-site customer shall improve the user involvement.

Inadequate Definition: The planning game and the on-site customer as well as the user stories and the metaphor try to deliver a precise definition. The small (and often) releases shall give frequent feedback in order to find out whether the chosen direction will fit to the customer's needs.

Numerous changes: Extreme Programming does not try to prevent changes to occur, but takes them into account right from the beginning. Refactoring and simple design shall weaken the changes effects.

4. Discussing Extreme Programming

4.1. Introduction

The literature on Extreme Programming is very much based on practical experiences. It is easy to find many hints on particular approaches, usually based on anecdotes [16, e. g.]. There is only little literature on the formal aspects of the Extreme Programming process. This may hinder the implementation of Extreme Programming in organisations, since usually the development divisions will not have the time to study the original literature and look into the problems with Extreme Programming.

Extreme Programming is not applicable for all kinds of projects of course. In addition it must be clearly stated that Extreme Programming most probably is not useful for the majority of projects. The use of Extreme Programming is only making sense, when the requirements are very much uncertain. If it is possible to easily identify the requirements at the beginning, it would not make any sense not to do this [15, p. 10ff].

Extreme Programming comes along with two basic assumptions. Those two assumptions have to be accepted

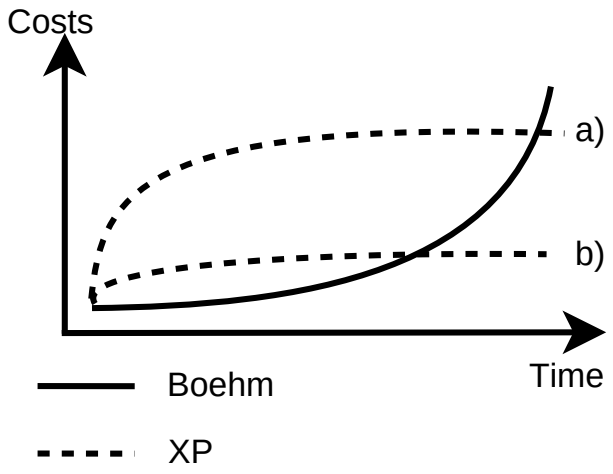


Figure 7. Comparing cost graph by Extreme Programming and Boehm

to understand Extreme Programming. Those two basic assumptions are:

- Code is easy to modify
- Good code is easy to understand

4.2. Costs of Change

The core practices of testing and refactoring are used in Extreme Programming to ensure the easy changeability of the code. Only if this is the case, then the changed ratio between time and costs[4], as postulated by Kent Beck, is valid. Until now it is not stated how this ratio between time and costs actually behave [15, see also p. 119ff]. It may be assumed that within Extreme Programming the costs for changes are higher compared to early identified costs within a classical model. It is not clear, at which point both graphs (representing the respective ratio between time and costs) intersect. This problem is shown in figure 7 on page 9. If in Extreme Programming the graph b) would be relevant, its use will pay off very fast: Throughout the entire development time-frame the costs for changes are much lower or only little higher compared to the classical model. But if graph a) would be relevant for Extreme Programming, then it is very unlikely that the changeability will pay off financially.

Up to now none of the shown graphs a) or b) in figure 7 is approved. It is a matter of fact that due to better software development tools and clearer software design the costs for changes decreased. Especially a clear software design is in the context of Extreme Programming in dispute. Extreme Programming asks for a simple design and ongoing revision and improvement of bad design through refactoring. Some feedback from the use in practice [18, e. g. p. 90] shows that refactoring is neglected in the daily routine and therefore no clear software design is formed.

It may assumed that the effort for refactoring increases during the project's progress and therefore less resources can be used for the implementation of new features. In this situation a valid question is, whether the effort for refactoring may reach a level where it is impossible to implement new features at reasonable costs.

Extreme Programming shows its strength in those cases, when it is impossible to anticipate late change requests [15, see p. 122]. By explicitly allowing changes within the process, Extreme Programming found a good approach to deal with this common problem.

The unit testing may be another major source for costs. Extreme Programming demands to have unit test for all methods and classes. If, e. g. due to refactoring, changes are made to the underlying class, then all affected unit tests have to be adapted as well [15, p. 74].

4.3. Good Code is easy to understand

The idea of easy to understand code is based on the idea of Literate Programming [6]. Literate Programming assumes that program code can be written in a way that it is understandable without any comments included in the sources. This may be achieved e. g. by smart usage of names for methods and variables [15, p. 75]. Since there are many tools[5] that deliver a complete API documentation or the according UML diagrams from a documented source code, it is not easy to understand why to renounce any documentation of the source code. Furthermore Literature Programming makes refactoring difficult, since not only the functionality must be kept, but also the readability must be ensured.

When a developer wants to get an overview on a specific module or subsystem, he often does not want to read the source code, but have a look into the design documentation. Extreme Programming does not guarantee that this design documentation is generated or maintained.

The ambition for an easy to read source code is certainly assisted by the use of coding standards. But using the source code as the only documentation seems to be very problematic.

4.4. Requirements Engineering

Within Extreme Programming requirements are defined as user stories. This is mainly done by the customer. It is

[4] see section 3.3 on page 3

[5] like JavaDOC, Doxygen or documentation generator integrated in Microsoft .NET

doubtful that the customer is able to formulate accurate user stories [15, see p. 60ff]. Very often customers have undefined ideas about the requirements. Usually it is an essential part of each project to stepwise investigate the problems and develop the solutions together with the customer. Extreme Programming tries to support the customer to formulate the user stories in giving frequent feedback. It is questionable, if this is enough.

In addition it seems to be difficult to formulate the user stories in a way that their realisation is possible during one iteration of three or four weeks. In this context it should be mentioned that the acceptance tests are based on those user stories. This implies that the requirements are founded on a functional basis. The direct measurement of qualitative requirements as reliability or safety is much more difficult.

4.5. Estimates and Monitoring Progress

Extreme Programming demands the developers to deliver the estimates. This may lead into a high commitment of the developers. On the other hand studies [9, p. 29] have shown that estimates made by experts rather than by developers are more precise. In addition it may be assumed that not all developers have equally developed abilities to give good estimates.

The estimates in Extreme Programming are only done for small time-frames. Due to this a high degree of precision for the short term planning can be reached. On the other hand the long term planning is not assisted through this. Furthermore only the requirements with the highest priority are implemented. It is easily possible to loose focus concerning the time-frame and the budget.

Extreme Programming postulates contracts between the customer and the vendor that do not relate to a fixed price but to a specific time-frame. Again, it seems to be questionable if the customer agrees to this and would accept those contracts.

4.6. On-site Customer

The immediate inclusion of the customer into the project team is problematic [15, p. 81ff]. In many projects this might be impossible. The customer likely is not willing to put the needed financial effort into the project to have his own employees at the vendor's site. Furthermore this is not the usual procedure within software development projects.

Further more it is doubtful that the customer (respectively the employee at the vendor's site) has the necessary domain knowledge, the necessary overall overview and suitable decision-making authority. Very often the customer would have to send out several employees which would increase the overall effort and cause additional costs.

To avoid these problems, it might be useful to install analysts and requirements engineering professionals. But they could not completely substitute the customer, since they do not have all necessary information and knowledge. An analyst will have problems to identify political aspects of a project since he has no knowledge on the informal network at the customer's site.

4.7. Pair-Programming

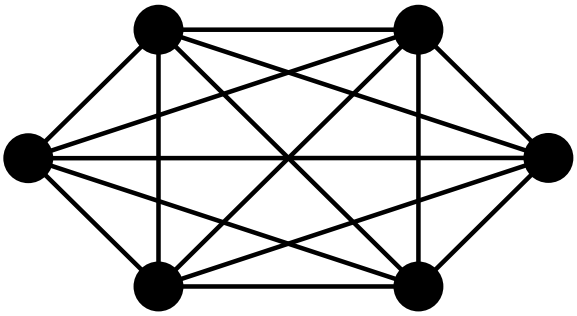
Another point that led into controversial discussions is the constraint to apply pair programming in any case. Very often coding tasks include easy to solve aspects, e. g. the interface design, but also very hard tasks as the implementation of complex algorithms. To apply pair programming in any case is not necessary. Instead it is better to find a good balance.

The human factor might be another problematic aspect of pair programming. The pairs have to be formed carefully [20] in order to avoid negative effects, e. g. caused by personal conflicts. The self-coordinated pairing as suggested by Extreme Programming appears unrealistic and dangerous.

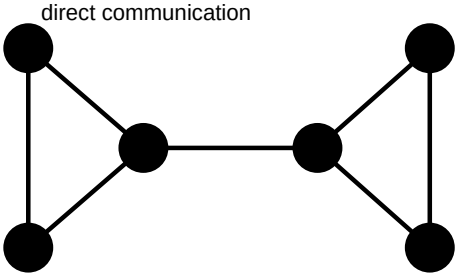
4.8. Direct Communication and Knowledge Transfer

Extreme Programming very much relies on direct communication. The knowledge transfer between the employees mainly takes place by 1:1 communication during pair programming. The formal description of the system is neglected or even prevented. The direct communication is not scalable to any team-size [8]. The problem is illustrated in figure 8 on page 11 In part a) only direct communication between all 6 team members is applied. In order that everyone can communicate directly with all other team members a total of 15 communication paths is needed. In part b) the team is divided into two sub-teams. Between both sub-teams only one communication path is established. By implementing this indirect communication path, the overall sum is reduced to 7 communication paths.

Since Extreme Programming relies on having the complete project know-how within the underlying source code and represented by the team members' knowledge, Extreme Programming is to be rated negatively according the knowledge management. The loss of only few team members might stop the entire project. Extreme Programming tries to avoid this by the use of pair programming and collective code ownership. During a project with several team members and a time-frame of several months, it seems unrealistic that every developer can build up a complete understanding of the entire system. But the developer needs this complete understanding so that he is able to make useful changes at



a) 15 communication paths using direct communication



b) 7 communication paths using mixed communication

Figure 8. Number of communication paths between 6 persons

any place in the source code. Only then the developer can cope with the responsibility derived from collective code ownership. It is obvious that this postulation, formulated in Extreme Programming, is not accomplishable. Otherwise the entire system could be developed by only one developer. Another disadvantage of the lack of design documentation occurs when introducing a new employee into the project. A new employee could not get familiar with the system by only reading documentation of course, but still this helps very much.

Extreme Programming's exclusive focus on direct communication is regarded to be Extreme Programming's major disadvantage. It should be mentioned that methods and procedures exist to weaken these negative effects. In the following section a case study is presented which tried to apply Extreme Programming in a large-scale project.

4.9. Extreme Programming in large-scale Project

This section refers to the proceeding of Cao et al. [7] who discuss the application of Extreme Programming in large-scale projects. As mentioned before, communication problems may occur when having a high amount of participants and stakeholders in the project. But this is not the only problem. Furthermore software architectures are hard to design for complex software and consequently this planning should take place at the beginning of the project. From the other point of view agility is also essential in large-scale projects, because of dynamic environments and frequently changes concerning requirements. Cao et al. [7] conducted a case study where they propose a set of techniques for the introduction of Extreme Programming in large-scale projects.

The studied company develops complex enterprise systems for the finance sector and 22 developers were involved in the project. Core practices were used, but in an adjusted form instead of the pure form developed by Kent Beck [4]. A selection of changes in core practices, according to [7] is presented in the following:

Upfront design: An upfront design was conducted, even if it is not applied by Extreme Programming because of frequently changing environments. But in the case study, Cao et al. found out that upfront design supports other agile practices like pair programming and refactoring. Design patterns, defined by the customer's needs, helped to make the communication between pair programmers more efficient. Also the knowledge transfer between pairs is enhanced by using design patterns. Furthermore existing functions which were based on the patterns could be reused frequently.

Short development cycles: Customers always want something changed. Therefore short release cycles were also used in the large-scale project, but with adaptations. The upfront design itself took full six month. After that the tasks were implemented as end-to-end functionalities where the duration was not fixed like it is normally done within Extreme Programming. In this case the duration was aligned to the complexity and nature of each individual task.

Surrogate customer engagement: A continues customer involvement is essential for large-scale projects. The company in this case study held frequent meetings to negotiate problems, requirements and specifications. This often led to changes in these areas. Often it is problematic to have the customer always available if he is needed. In this case the customer was surrogated by product managers and business analysts who had good knowledge of the customer's needs.

Flexible pair programming: Pair programming was applied for analysis, design and test case development. For all other tasks like coding it was in the hand of the developers if they would share the monitor and keyboard. As stated by one of the project managers who worked for the studied company, pair programming works better when the developers have a choice. Cao et al. [7] formulated the following benefits of pair programming:

- Pair programming reduces development time, two programmers together achieved the goal in only 80% of the time they would have needed if they worked alone.
- Pair programming reduces training time, programmers with different experience levels learn from each other. An inexperienced programmer becomes productive very fast. Furthermore knowledge is spread across the development team because of pair changes. In the studied company a pair worked together for maximal 6 weeks and in average for 2 weeks.
- Pair programming improves the quality of code, when programmers negotiate about their coding they discover inconsistencies or defects. Pair programming creates a supportive environment where developers find always somebody they can ask and talk to.

Reuse with forwarding refactoring: Based on design patterns, they applied reuse in the form of forwarding refactoring. This means that new features are implemented, based on existing architecture. In Extreme Programming developers normally focus on their current needs, they do not think about the reusability of their code. Thinking about reusability will speed up the development process.

Controlled empowerment and organizational structure: In the studied project a flat hierarchy was used. This led to a flexible and responsive environment which helped to react fast to changing environments. Furthermore decentralised decision making was applied which led to motivation on the part of the developers.

Summarised it can be said that pure Extreme Programming can not be used for any project. The modification of the core practices showed that they sometimes have to be aligned to the project environment. But, as mentioned before, it is also important to have the interdependencies of core practices in mind. It is important to be careful with leaving one practice away, it may support another practice. Such relationships become obvious by looking at the example of dependencies between upfront design in form of design patterns and pair programming. The study showed that a design pattern positively influences pair programming.

4.10. Discipline and Culture

As Beck [4, p. 151ff] points out, the realisation of Extreme Programming in an organisation is not easy. Extreme Programming postulates that *all* 12 core practices[6] have to be applied entirely and consequently. The reason for this is that the different core practices assist each other and weaken the other's negative effects. The compliance with all core practices demands significant discipline within the project team [15, p. 66]. For example the simple design can only be applied if later the design is revised by refactoring. If the refactoring is neglected, the source code gets complex and badly structured easily.

The use of Extreme Programming is always linked to the risk that developers only adopt the pleasant core practises such as the 40-hour week or the simple design and neglect the less pleasant core practices such as refactoring and test driven design. A project conducted in this manner most likely is doomed to failure. This might cause the project management to put a lot of pressure on the project team to guarantee that all core practices are applied. This might override many of the positive effects on the human level.

Implicitly Extreme Programming draws up some demands to the overall organisation. The higher management looses power due to e. g. the complete delegation of any technical decision to the level of the developers. The same problem occurs related to the shifting of the responsibility for the estimates to the level of the developers. The project management is disempowered due to this.

The integration of change as a fundamental part of a project is a difficult intention for an organisation as well. It can be understood as the avowal that every long term planning is useless because of the lack of precision. In many companies a lot of employees are engaged with planning and controlling. Changing the planning and controlling process according to Extreme Programming withdraws the legitimacy of these subdivisions. It is doubtful that such a change to the corporate culture is very likely to happen.

Simple design and collective ownership are a potential source for critical conflicts among the developers. Simple design enforces the experts to choose a simple design against their knowledge in cases where they recognise that refactoring will be needed in the next couple of days or weeks. From a financial view this is senseless as well [15, p. 66]. Following the theory of collective code ownership, everyone shall feel responsible for the source code. But it might turn exactly the other way round. In this case nobody feels responsible for the source code anymore. Within Extreme Programming only the complete team can be awarded, since it is not possible to identify individual merits. This withdraws some aspects of the management's power.

5. Conclusions

The report showed that Extreme Programming is more than its twelve core practices; it is a philosophy which focuses on the modern world view instead of the mechanical world view. Furthermore Extreme Programming is a toolbox which consists of interdependent core practices. There-

[6] see section 3.4 on page 4

fore, the model can only modified hardly. But it is not impossible to modify the model successfully according to project needs as shown by the study on the usage of Extreme Programming in large-scale projects.

Extreme Programming is not the solution for all problems; it also has its disadvantages and drawbacks. Therefore it Extreme Programming can not be applied successfully in every kind of software project. When a company decides to use Extreme Programming it should have in mind that this process model deals best within projects labeled with uncertainties.

References

[1] P. Abrahamsson, J. Warsta, M. T. Siponen, and J. Ronkainen. New directions on agile methods: A comparative analysis. In *Software Engineering, Proceedings 25th International Conference*, 2003.

[2] W. B. Arthur. Increasing returns and the new world of business. *Harvard Business Review*, (July-August):100–109, 1996. online available http://www.santafe.edu/arthur/.

[3] B. Barry. *Software Engineering Economics*. Prentice-Hall, 1981.

[4] K. Beck. *Extreme Programming Explained: Embracing change*. Addison-Wesley, 1999.

[5] K. Beck and M. Fowler. *Planning Extreme Programming*. Addison-Wesley, 2000.

[6] J. Bentley and D. Knuth. Literate programming. *Communications of the ACM*, 29(5):364–369, May 1986.

[7] L. Cao, K. Mohan, P. Xu, and B. Ramesh. How extreme does extreme programming have to be? adapting xp practices to large-scale projects. In *Proceedings of the 37th Annual Hawaii International Conference*, pages 83–92. IEEE, 2004.

[8] A. Cockburn. *Agile software development*. Addison-Wesley, cop., Boston, 2002.

[9] T. DeMarco and T. Lister. *Peopleware: productive projects and teams*. Dorset House Publishing Co., New York, 2nd edition, 1999.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, 1999.

[11] H. Haken. Synergetics. *IEEE Circuits and Devices Magazine*, 4(6):3–7, 1988.

[12] R. Lewin. *Complexity: Life at the Edge of Chaos*. Macmillan, 1992.

[13] M. Mantel. The effects of programming team structures on programming tasks. *Source of Communications of the ACM-Archieve*, 24(3):106–113, 1981.

[14] H. R. Maturana. Organization of the living: a theory of the living organization. *International Journal of Man-Machine Studies*, 7(3):313–332, 1975.

[15] P. McBreen. *Questioning Extreme Programming*. Addison-Wesley, Boston, 2002.

[16] J. Newkirk and R. C. Martin. *Extreme Programming In Practice*. Addison-Wesley, cop., Boston, 2001.

[17] J. M. Nicholas. *Project Management for Business and Engineering*. Elsevier Inc., Burlington, MA, 2004.

[18] M. Stephens and D. Rosenberg. *Extreme Programming Refactored: The Case Against XP*. Apress, 2003.

[19] D. Wells. *Homepage ExtremeProgramming.org*. 1999. some figures based on figures in http://www.extremeprogramming.org/.

[20] L. Williams and R. Kessler. *Pair Programming Illuminated*. Addison-Wesley, 2002.