Extreme Programming Modified: Embrace Requirements Engineering Practices*

Jerzy Nawrocki, Michał Jasiński, Bartosz Walter, Adam Wojciechowski
Poznań University of Technology, Poland
{Jerzy.Nawrocki, Michal.Jasinski, Bartosz.Walter, Adam.Wojciechowski} @cs.put.poznan.pl

Abstract

Extreme Programming (XP) is an agile (lightweight) software development methodology and it becomes more and more popular. XP proposes many interesting practices, but it also has some weaknesses. From the software engineering point of view the most important issues are: maintenance problems resulting from very limited documentation (XP relies on code and test cases only), and lack of wider perspective of a system to be built. Moreover, XP assumes that there is only one customer representative. In many cases there are several representatives (each one with his own view of the system and different priorities) and then some XP practices should be modified. In the paper we assess XP from two points of view: the Capability Maturity Model and the Sommerville-Sawyer Model. We also propose how to introduce documented requirements to XP, how to modify the Planning Game to allow many customer representatives and how to get a wider perspective of a system to be built at the beginning of the project lifecycle.

1. Introduction

Extreme Programming (XP for short) is an agile (lightweight) software development methodology [1, 6]. Its popularity is growing very fast. According to Tom DeMarco "*XP is the most important movement in our field today*" [2]. XP proposes many interesting practices (strong customer orientation, planning game, very short releases, test-first coding etc.), but it has also some weaknesses. From the Software Engineering point of view the most important are the following issues:

- In XP there are three sources of knowledge about the software to be maintained: code, test cases, and programmers' memory. If the software remains unchanged for a long enough period of time, the programmers will forget many important things and – what is even worse – some of the programmers might get unavailable (for instance they can move to another company). Then, the only basis for maintenance is code and test cases. That can make maintenance hard. It would be easier if one had the requirements documented. Then the programmers could use them as a guide in reading code and test cases. Thus, the following question arises: is it possible to have an agile methodology and documented requirements?
- XP relies on oral communication as *"the written and e-mail communications (..) are error-prone"* [6]. But oral communication is susceptible to lapses of memory. After some time one can have problems to recall which alternative was finally chosen and why, especially when a system is complex, there are many points of view, many trade-offs and hesitations. Again it would be useful to have some written documents.
- XP assumes there is only one customer representative. If there are many it is assumed they speak one voice [6]. What if they have different points of view? How to modify XP to take this into account?
- In XP the planning perspective is quite short: just one release (if possible, a release takes *"no more than two months"* [6]). Some people consider this a disadvantage [5]. Time and money could be saved if the stakeholders (customer representatives and developers) spend, at the

* This work has been partially supported by KBN grant 8T11A01618.

beginning of a project, a week or two discussing the scope and feasibility of the whole project. How to modify XP to provide a wider perspective of a project?
- XP depends on frequently performed automated testing. Automated test cases must be not only created but also maintained. Maintenance of test cases is a serious problem [17], especially in case of frequent changes. How to support maintenance of test cases?

In the paper we discuss these problems and propose some modifications to XP that aim at solving them. Next section contains a short overview of the XP methodology. Then we assess XP practices from two points of view: the Capability Maturity Model (Section 3) and the Sommerville-Sawyer Model (Section 4). In Section 5 we try to convince the reader that it is possible to reconcile XP with written documentation of requirements. The main idea is to assign the task of requirements documentation and management to the XP tester. Section 6 is devoted to the problem of multiple customer representatives. The proposed solution is the Modified Planning Game that allows multiple customers to make decision about the scope of a release. In Section 7 we propose to introduce to the XP lifecycle the requirements engineering phase at the beginning of a project. That way one can get a wider perspective of a system that the XP team is going to build.

2. Short overview of XP

The classical approach to software development is based on a set of widely accepted standards and techniques, including IEEE Standard 830 for requirements specification, Fagan inspections, effort estimation based on Function Points or COCOMO etc. XP rejects many of those techniques:

- *IEEE standard 830* [20] and requirements specification documents are not needed in XP. They are replaced with a set of very short **user stories** written on index cards. The cards are supplemented with oral communication between the development team and the customer representative. It is assumed that the customer representative is always available to the team – he is called an **on-site customer**. If a user story is not clear or some of them are in conflict, a team member can ask the customer representative and he will solve the problem very fast.
- *Classical inspections* [25, 26] have little value from the XP perspective. They are based on a kind of *off-line review*: first the author writes a piece of code then reviewers read it and make comments. In XP software production is based on **pair programming**. Two people are sitting in front of one computer. One person has the keyboard and writes the code while the other person looks at the screen and tries to find defects. That is a kind of **on-line review** but there are no forms, no facilitators, no checklists etc. They are replaced with conversations. Some programmers consider pair programming more enjoyable than individual one [22, 16].
- *Function Point Analysis* [27], *COCOMO* [21] and other software estimation models are of little importance from the XP point of view. The XP practitioners are very skeptical about long-term plans. The planning horizon in XP is just one release and the development strategy is based on **small releases** (each release is usually not longer than two months [6] and it is split into **iterations** taking usually not longer than 3 weeks each [2]). That provides a rapid feedback and makes planning simpler.
- In the classical approach the development team "*uses the allocated requirements as the basis for software plans*" ([12], page 130), i.e. the plan is to answer the following question: *how long will it take to implement a given set of requirements*? In XP one tries to answer a different question: *what valuable features can be implemented in a given period of time*? To answer that question and to maximize customer's satisfaction the **Planning Game** is used at the beginning of each release and iteration. The customer brings a set of user stories. The development team estimates the effort required to implement each story, assesses the technical risks associated with each story and specifies how many hours they will be able to spend on the project (without interruptions) in a given release or iteration (that estimate is based on the observations coming from the past). Knowing the effort for each story and the total time available to implement them, the customer decides which stories will be implemented in a given release or iteration.
- Some development methodologies suggest to write code first, then to think how to test it [23]. XP recommends the **test-first coding** principle (write a set of test cases first, then write the code).
- To practice XP the programmers do not have to know UML. One of the XP principles is **'travel light'**. The only required artifacts are test cases and code. If one needs to discuss the design, he can use the **CRC cards** (Class-Responsibility-Collaboration) [24]. But those cards do not have to be stored and managed.
- In the waterfall model *'the software design is realized as a set of programs or program units'* and then *'the individual program units are integrated and tested'* ([28], page 10). XP is based on **continuous integration**: the programmers integrate a new piece of software with the latest release several times a day and each time *'all the tests must run at 100%'* [1]. When testing is performed so frequently, it makes sense to use **automated testing** and tools such as xUnit [29].

3. XP vs. Capability Maturity Model

Perhaps the most popular reference model for software process assessment is the SEI's Capability Maturity Model (CMM for short) [12]. That model consists of five maturity levels and each level contains a number of Key Process Areas (KPAs). At Level 2, called Repeatable, there are six KPAs and one of them is Requirements Management. Recently Paulk presented an interesting evaluation of XP from the CMM point of view [13]. In his opinion requirements management, as defined in CMM, is "*largely addressed in XP*". His arguments are on a high abstraction level and we are going to confront in more detail the XP practices with the abilities and activities recommended by the Requirements Management KPA of CMM.

According to Paulk, *"XP addresses Level 2's requirements management KPA through its use of stories, onsite customer, and continuous integration"* [13]. A user story is made up of two components: the written card and the series of conversations that take place after the card is written. The written cards are just *"promises for conversation"* [6] and they do not have to be complete nor clearly stated. If something is not clear, it will be clarified during the conversation with the on-site customer. Once a story is successfully implemented, the story card can be destroyed [2]. Even if the stories are not destroyed (some organizations keep them on the web [30]), they are not a replacement for requirements as they do not have to be clear and complete. The third element, *continuous integration*, is more important for quality assurance than for requirements management. What matters in the requirements context is small release, which allows fast feedback from the end users [1] – that is the best way of requirements validation.

The main weakness of the XP approach to requirements management is lack of documentation of requirements. From this point of view XP resembles evolutionary prototyping, where a prototype rapidly grows into a finished product. Casper Jones points out the following hazards associated with that approach [7]:

- *"Written specifications are missing or perfunctory.*
- *Maintenance costs are usually well above average.*
- *Litigation probability is alarmingly high."*

Written documentation of requirements is also important in CMM. At Level 2 there are, among others, the following criteria concerning a sound requirements management process [12]:

- **The allocated requirements are documented** (Ability 2). Unfortunately, in XP there is no requirements documentation. A story card is like an iceberg: what you see is only a small part of what you will get (you have to wait till conversations).
- **Changes resulting from changes to the allocated requirements are documented** (Activity 3, Subpractice 2). In XP changes are not documented. *"If Business realizes it needs a new story during the middle of the development of a release, it can write the story. Development estimates the story, then Business removes stories with the equivalent estimate from the remaining plan and inserts the new story."* [1]

Recently another model, called CMM Integration (CMMI for short), has been developed that integrates maturity models for System Engineering, Software Engineering (SW-CMMI), and Integrated Product and Process Development [4]. CMMI has two representations: staged (traditional) and continuous (it resembles ISO 15504). The staged representation of SW-CMMI is similar to the old CMM (it consists of five maturity levels and each level contains a number of KPAs). At Level 2, called Managed, there are seven KPAs and one of them is Requirements Management. It was a surprise for us that the description of the Requirements Management KPA does not explicitly state that requirements must be documented. However, there are two specific practices that implicitly demand documentation of requirements:

- **Manage changes to the requirements as they evolve during the project** (Specific Practice 1.3). Typical work products suggested by SW-CMMI include requirements database and the recommended subpractice is to maintain the requirements change history with the rationale for the changes.
- **Maintain bi-directional traceability among the requirements and the project plans and work products** (Specific Practice 1.4). A typical work product suggested here by SW-CMMI is a requirements traceability matrix and it is recommended to maintain traceability from a requirement to its derived requirements as well as to work products (code, test cases etc.).

Thus, the XP approach to requirements management, which relies almost completely on oral communication, is not acceptable from the CMM/CMMI point of view. The lack of documentation most impacts the maintenance. There are two kinds of maintenance: continuous and discrete. **Continuous maintenance** means that defect reports and change orders appear quite frequently and it makes sense to have a team designated to the maintenance job. They can produce new patches and release new versions of the software on an almost periodic basis. Many off-the-shelf tools are maintained that way. If the software is not too big, XP can be a good choice (a success story is described in [30]). In **discrete maintenance** defect reports and change orders appear rarely. When a need arises a project is launched. Many of the 2000-year-problem projects were discrete maintenance projects. Moreover, customer-initiated software systems are frequently maintained on the discrete basis. If the distance from one maintenance project (episode) to another one is long enough, then the lack of documentation may become a serious problem.

4. XP vs. Sommerville-Sawyer model

Maturity of the requirements engineering processes applied in an organization can be assessed according to the Sommerville-Sawyer model [15]. This is a simple 3-level model based on good practices. Sommerville and Sawyer have identified 66 practices and they split them into 3 groups: basic, intermediate and advanced. Each practice can bring from 0 to 3 points, depending on how wide it is used in an organization (3 points if the practice is an organization's standard, 2 if it is frequently used, 1 if its usage is discretionary, 0 if the practice is never used). The highest maturity level is called **Defined**. Organizations at that level have more than 85 points in the basic practices and more than 40 points in the intermediate and advanced practices. The intermediate level is called **Repeatable** and organizations at this level have more than 55 points in basic practices. The lowest level is called **Initial** and an initial organization has less than 55 points in basic practices.

We have used the Sommerville-Sawyer model to assess the approach to requirements management proposed by XP. There are only seven basic practices of the Sommerville-Sawyer model that are fully supported by XP (see also the appendix):

4.1 Assess system feasibility.
4.6 Use business concerns to drive requirements elicitation.
5.4 Plan for conflicts and conflict resolution.
5.5 Prioritize requirements.
7.3 Model the system architecture.
9.1 Uniquely identify each requirement
9.2 Define polices for requirements management.

From the estimation presented in the appendix follows that an XP team can earn up to 31 points in basic practices. It is much lower than 55 points required to be classified as Repeatable. Thus, XP teams are at the Initial level. This is an important risk factor.

5. Reconciling XP with documented requirements

At first glance, XP and written documentation do not fit together at all. XP is a lightweight methodology and it strongly prefers oral communication over written one. However, a question arises what does it mean that a development methodology is lightweight. To whom is it lightweight? XP is certainly not a 'lightweight' methodology from the customer point of view since XP requires *on-site customer*, working all the time with developers (classical approach is much lighter from that point of view). The main beneficiaries of XP are programmers – the only artifacts they have to write are test cases and code. Thus, if one wants to reconcile a lightweight methodology with written documentation of requirements, he cannot put the burden of documentation onto the programmers. Our proposal is to make the tester responsible for requirements documentation and management.

According to Beck, an XP tester is "*responsible for helping the customer choose and write functional tests*" and running them regularly [1]. Requirements and acceptance tests are on the same abstraction level, so tester seems to be the best person to take on the job of analyst responsible for requirements documentation and management. We will call that person a **tester/analyst**. Merging the tester and analyst roles seems reasonable because test cases must be managed anyway. It is especially difficult when requirements creep and the system is complex enough (the more complex system the more requirements creep, as no body is able to produce them at the beginning of the project). A good practice is to link the test cases to the requirements. If the requirements change, one knows what test cases should be reviewed. Thus, a well-organized tester will manage requirements anyway. If requirements are not documented, his job is harder.

Here are our tips for the XP tester/analyst:

- **Do not be afraid to use advanced tools**. *Travel light* (that is one of XP principles – see Sec. 2 and [1]) does not mean that you have to go on foot. If you go by car 30 km in wrong direction, it is a small problem; if you go by foot 30 km in wrong direction, it is a big problem. Such tools like Rational RequisitePro [14] may be very useful.
- **Organize the requirements into multiple layers**. Treat the information put by the customer onto story cards as a feature description and place it on the highest layer (one story card is one feature/ requirement). Place the clarifications on a lower level. The test cases should be a layer below.
- **Make the collected requirements available to the team and to the customer through the web** (in RequisitePro there is a tool called RequisiteWeb).
- **Carefully choose attributes for your requirements**. The set of attributes should be as small as possible. Then it is much easier to find information and maintain the attribute values.
- **Transfer all enclosures provided by the customer to an electronic form**. All the drawings, pictures, tables or law acts provided by the customer on a paper keep as files (e.g. gif or jpeg files) using a scanner. Then you can make them available through Internet. Sheets of paper can easily be lost. Electronic files can be copied and printed as needed.

A question arises how requirements documented by the tester influence other XP practices. Will the presented modification (i.e. extending the role of tester to tester-analyst) introduce conflicts or inconsistencies to the methodology? When one reads the XP practices (see e.g. Sec. 2 or [1]), he will find that none of them contradicts with the proposed modification. Moreover, we are convinced that augmenting XP with requirements documented by the tester will strengthen the methodology and from the programmers point of view it will be as lightweight as the original XP.

We have been experimenting with XP since 1999. The subject of the first experiments was pair programming [9]. Then, in the next academic year (2000/01) we run another experiment as a part of the Software Development Studio [19]. Five 1-year projects have been assigned to ten teams. Five teams worked according to XP and five other according to CMM Level 2. The CMM teams used IEEE Standard 830 for requirements engineering, the requirements elicitation was supported by the FAST meetings [18], the Delphi method [8] was used for estimating the effort and developing the plans, progress tracking was based on the Earned Value method, and CVS was used for version management. Each team (both CMM and XP) consisted of eight students coming from the 3rd, 4th and 5th year. The XP teams were in a very difficult position. The main problem was the customers. It proved impossible to have an on-site customer (more details can be found in [11]). For academic year 2001/02 we decided to modify XP. We introduced to XP documented requirements, the students were equipped with document templates, we structured the meetings with agendas and forms, and we have found sponsors who helped us to buy the Rational Suite. The projects are not finished yet but students, customers and staff members are much more satisfied than a year before.

6. Multiple customer representatives

One of the XP practices is on-site customer. He is a real customer (end user) and he is expected to sit with the team, answer the questions, and make business decisions [1]. Particularly, he participates in the Planning Game through which the scope of the next release and the next iteration is decided.

XP assumes, that there is only **one customer representative** for the project. In case of many customer representatives it is assumed that they speak one voice. Unfortunately, in some cases there are many customer representatives and they do not speak one voice. Here are some guidelines proposed by Sommerville and Sawyer [15] that show the need for considering **many customer representatives**:

- **Identify and consult system stakeholders**. It is important to make people feel they participate in creating the system.
- **Collect requirements from multiple viewpoints**.
- **Be sensitive to organizational and political considerations**. If a system is to serve several departments of an organization, a common practice is to include into the decision process a representative of each department.
- **Plan for conflicts and conflict resolution**. Conflicts are inevitable if a system is to serve many people with different expectations and fears.

Here is our proposition how to include many customer representatives into the development process:

- **The tester/analyst answers programmer's questions**. If there are many customer representatives they cannot sit with the team all the time waiting for questions asked by the programmers. The tester analyst is the best person to do it.
- **The tester/analyst has everyday contact with the customer representatives**. He is developing with them acceptance tests and presents them progress reports (failed/passed acceptance tests).
- **The customer representatives participated in a modified Planning Game**. That game is described below.

7. Modified Planning Game

We assume that at the customer side there are many customer representatives and one senior customer. Each **customer representative** has some domain knowledge and he will be an end-user of the system. It is possible that his requirements will be contradictory with the requirements of the other customer representatives.

The senior customer represents interests of the whole customer organization. It is possible that he does not have the **domain knowledge** the customer representatives have and **time** to discuss details of the system to be build. Because of this, both senior customer and customer representatives are necessary. The main role of the senior customer is to resolve conflicts between the customer representatives whenever necessary.

Like in the original Planning Game, the **first move** belongs to the Business, i.e. to the customer representatives. They bring their story cards.

The **second move** is done by the Development. As in the original Planning Game, the developers estimate effort in Ideal Engineering Time (hours) and assess risk associated with each story. One can use here the Delphi method [8, 3].

In the **third move** the Business decides about the scope. That move is modified since there are many customer representatives instead of one. Each customer representative gets some budget representing the time the XP team can spend working for him. The budget is expressed in hours. It is assigned to each customer representative by the senior customer. Each customer buys the most valuable stories from the developers. If *n* customer representatives purchase a story, each one pays 1/*n* of the price.

If a customer representative is not able to buy a valuable story (all his stories are too expensive), he can return his budget to the senior customer. Then the senior customer offers that "money" to the remaining customer representatives. It is a kind of **auction**. The person who offers the highest price will get the "money" and will be able to buy another story. The price will be paid at the next Planning Game to the person who resigned from them.

If there are many customer representatives it seems reasonable to have a longer releases (e.g. 3 months long) to be able to implement stories proposed by more than one person. At the increment level it would be useful to schedule for one increment stories coming from the same customer representative.

8. Modifying the XP Lifecycle

In the original XP the planning perspective is very short: not longer than two month (one release). Some people consider this a disadvantage [5]. Time and money could be saved if the stakeholders (the customer representatives and the developers) spend, at the beginning of a project, a week or two discussing high-level goals, constraints, scope, risks and feasibility. Pressman warns that "*there is a tendency to rush to a solution, even before the problem is understood. This often leads to elegant software that solves the wrong problem!*" [18]. XP is not that bad. If there is an on-site customer and the releases are short enough, the probability of solving the wrong problem is very low. But, as pointed out earlier, it is difficult to have a competent **on-site** customer, who is also in a position to make business decision on behalf of his organization. Moreover, it makes no sense to rush to coding when nobody understands the application domain and some basic decisions have not been made yet.

Our proposition is to modify the XP lifecycle and to introduce a requirements engineering phase at the beginning of a project. That phase does not have to be very long. A few weeks would be enough. During that phase the XP team could do the following things:

- **Collect the use scenarios**. Scenarios concerning the current system are collected from the customer representatives and they describe problems and motivations for building a new system. This is a simple form of feasibility study. This is also a good starting point for collecting domain knowledge and understanding the problem. Scenarios concerning a new system can be written by the customer representatives and/or by the XP team. They show the vision how the system could work.
- **Assess system feasibility**. One of the questions that should be answered is if it is the right decision to develop the system according to the XP methodology. Other questions concern business and technical risks.
- **Identify system stakeholders**.
- **Roughly describe the system's operating environment.**
- **Look for main domain constraints.**
- **Prototype poorly understood requirements**. At this stage the prototype can be simple and should mainly focus on human-machine communication.

9. Conclusions

In the paper we have proposed three modifications to the XP methodology:

- Written documentation of requirements that is managed by a tester/analyst.
- The Modified Planning Game that allows to have multiple customer representatives (in the original Planning Game there is only one customer representative).
- The requirements engineering phase at the beginning of a project that provides a wider perspective of a system that is to be developed.

Our early experience shows that those modifications are simple enough and the resulting methodology is almost as lightweight as the original XP.

An important indicator of improvement is increase in the number of Sommerville-Sawyer points (see Section 4). If an organization is using the original XP it will earn at most 31 points in the basic practices and will be classified as Initial. After introducing the proposed modifications it can earn up to 78 points in the basic practices (see the appendix) and it would be classified as Repeatable. That is a good indicator of improvement.

References

[1] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Boston, 2000.

[2] K. Beck, and M. Fowler, *Planning Extreme Programming*, Addison-Wesley, Boston, 2001.

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

[4] *Capability Maturity Model Integration*, Version 1.1, Staged Representation, CMU/SEI-2002-TR-004, Carnegie Mellon University, 2002, http://www.sei.cmu.edu/cmmi/products /v1.1ippd-staged.pdf, January 11, 2002.

[5] R.Glass, "Extreme Programming: The Good, the Bad, and the Bottom Line", *IEEE Software*, November/December 2001, pp. 111-112.

[6] R. Jeffries, A. Anderson, and C. Hendrickson, *Extreme Programming Installed*, Addison-Wesley, Boston, 2001.

[7] C. Jones, *Software Quality: Analysis and Guidelines for Success*, International Thomson Computer Press, London, 1997.
[8] H.A. Linstone, and M. Turoff, *The Delphi Method: Techniques and Applications,* Addison-Wesley, 1975.
[9] J.Nawrocki, and A.Wojciechowski "Experimental Evaluation of Pair Programming" **in**: K.Maxwell, S.Oligny, R.Kusters, E. van Veenendaal (Eds.) *Project Control: Satysfying the Customer*, Shaker Publishing, London 2001, pp. 269-276. (see also http://www.escom.co.uk/conference2001/papers/nawrocki.pdf)
[10] J.Nawrocki, B.Walter, and A.Wojciechowski "Towards Maturity Model for eXtreme Programming", *Proceedings of the 27th EUROMICRO Conference*, IEEE Computer Society, Los Alamitos, 2001, pp. 233-239.
[11] J. Nawrocki, B. Walter, and A. Wojciechowski, "Comparison of CMMI Level 2 and eXtreme Programming", **in:** J. Kontio, R. Conradi (Eds.), *Software Quality – ECSQ 2002*, Lecture Notes in Computer Science 2349, Springer, pp. 288-297.
[12] M. C. Paulk et al., *The Capability Maturity Model: Guidelines for Improving the Software Process,* Addison-Wesley, Reading MA, 1994.
[13] M.C. Paulk, "Extreme Programming from a CMM perspective", *IEEE Software*, November/December 2001, pp. 19-26.
[14] Rational Software Corporation, *Using Rational RequisitePro*, Version 2001.03.00.
[15] I. Sommerville, and P. Sawyer, *Requirements Engineering: A Good Practice Guide*, John Wiley & Sons, Chichester, 1997.
[16] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries "Strengthening the Case for Pair Programming", *IEEE Software*, July/August 2000, pp. 19-25.
[17] M. Fewster, and D. Graham, *Software Test Automation: Effective use of test execution tools*, ACM Press & Addison Wesley, Harlow, 1999.
[18] R.S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, 4th edition, 1997.
[19] J.Nawrocki, "Towards Educating Leaders of Software Teams: a New Software Engineering Programme at PUT" **in**: P. Klint, and J.R. Nawrocki, *Software Engineering Education Symposium SEES'98*, Scientific Publishers OWN, Poznan, 1998, pp. 149-157.
[20] "IEEE Recommended Practice for Software Requirements Specifications. IEEE Standard 830-1998" **in:** *IEEE Software Engineering Standards Collection. 1999 Edition*, Vol. 4: *Resource and Technique Standards*, IEEE Press, April 1999.
[21] B. W. Boehm et. al., *Software Cost Estimation with COCOMO II*, Prentice Hall, Upper Saddle River (NJ), 2000.
[22] J.T. Nosek, "The case for collaborative programming", *Communications of the ACM*, vol. 41 (1998), No. 3, pp. 105-108.
[23] W.S. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, Reading, 1995.
[24] K. Beck, W. Cunningham, "A laboratory for teaching object-oriented thinking", *Proceedings of OOPSLA 89*, SIGPLAN Notices, vol. 24 (1989), No 10, pp. 1-6 (see also http://c2.com/doc/oopsla89/paper.html).
[25] M. Fagan, "Design and Code Inspections", IBM System J., vol. 15, no.3, 1976, pp. 182-211.
[26] J.C. Knight, and E.A. Myers, An improved inspection technique, *Communications of the ACM*, vol. 36 (1993), No.11, pp. 51-61.
[27] *Function Point Counting Practices Manual*, IFPUG, http://www.ifpug.org/publications/manual.htm, June 2002.
[28] I. Sommerville, *Software Engineering*, Addison-Wesley, Harlow, 5th edition, 1995.
[29] K. Beck, *Simple Smalltalk Testing: With Patterns*, http://www.xprogramming.com/testfram.htm, June 2002.
[30] C. Poole, and J.W. Huisman, "Using Extreme Programming in a Maintenance Environment", *IEEE Software*, November/December 2001, pp. 42-50.

Appendix. Evaluation of XP and Modified XP based on the basic guidelines of the Sommerville-Sawyer model

In the appendix all the basic guidelines presented in [15] are considered. Each guideline can bring from 0 to 3 points (see Section 4). The actual score depends on how far a given methodology supports the considered guideline. The structure of a guideline description is as follows:

a.b The guideline		
XP:	x	Explanation why the guideline earns x points.
MXP:	m	Explanation why the guideline earns m points.

where:

a.b: The guideline number as presented in [15].
XP: Original XP
x: Our estimate of the maximum score earned by XP
MXP: Modified XP as presented in the paper and aided by the Rational RequisitePro [14].
m: Our estimate of the maximum score earned by MXP

The evaluation is presented below.

3.1 Define a standard document structure		
XP:	0	There are no documents in XP.
MXP:	3	The document structure is defined.
3.2 Explain how to use the document		
XP:	0	No documents.
MXP:	0	We skipped it. We did not need it.
3.3 Include a summary of the requirements		
XP:	0	No documents.
MXP:	0	It can get inconsistent with the requirements.
3.4 Make a business case for the system		
XP:	0	No documents.
MXP:	3	There is a section with the customer's problems
3.5 Define specialized terms		
XP:	0	No documents.
MXP:	3	There is a section with the terminology.
3.6 Lay out the document for readability		
XP:	0	No documents.
MXP:	3	There is a document template.

3.7 Help readers find information		
XP:	0	No documents.
MXP:	3	The contents lists are used.
3.8 Make the document easy to change		
XP:	0	No documents.
MXP:	0	We do not change documents. We produce new versions.
4.1 Assess system feasibility		
XP:	3	User stories + Planning Game
MXP:	3	User stories + Planning Game
4.2 Be sensitive to organizational and political factors		
XP:	2	On-site customer + frequent conversations
MXP:	2	Senior customer and Modified Planning Game
4.3 Identify and consult system stakeholders		
XP:	1	The stakeholders are reduced to one customer.
MXP:	3	Modified Planning Game
4.4 Record requirements sources		
XP:	0	There are no such records.
MXP:	3	The source is an attribute of a requirement.
4.5 Define the system's operating environment		
XP:	0	No such information is stored.
MXP:	3	Imposed by the standard document structure.
4.6 Use business concerns to drive requirements elicitation		
XP:	3	Planning Game
MXP:	3	Modified Planning Game
5.1 Define system boundaries		
XP:	2	Conversations + Planning Game
MXP:	3	Planning Game + Functional requirements
5.2 Use checklists for requirements analysis		
XP:	0	No checklists.
MXP:	3	We use checklists.
5.3 Provide software to support negotiations		
XP:	0	Oral communication is used.
MXP:	2	E-mail, web pages, Rational Requisite Pro
5.4 Plan for conflicts and conflict resolution		
XP:	3	Planning Game + conversations
MXP:	3	Planning Game + structured meetings + e-mail
5.5 Prioritize requirements		
XP:	3	Releases & increments + Planning Game
MXP:	3	Releases & increments + Planning Game
6.1 Define standard templates for describing requirements		
XP:	0	No templates
MXP:	3	MS Word templates.
6.2 Use language simply, consistently and concisely		
XP:	2	User stories are usually simple and concise.
MXP:	2	We depend on templates and examples.
6.3 Use diagrams appropriately		
XP:	1	Some people use UML (e.g. Martin Fowler)
MXP:	1	UML diagrams. At discretion of team members
6.4 Supplement natural language with other descriptions		
XP:	1	At discretion of the team.
MXP:	1	At discretion of the team.
7.1 Develop complementary system models		
XP:	1	UML. At discretion of the team.
MXP:	1	UML. At discretion of the team.
7.2 Model the system's environment		
XP:	1	At discretion of the team.
MXP:	1	At discretion of the team.
7.3 Model the system architecture		
XP:	3	Metaphor
MXP:	3	Metaphor
8.1 Check that the requirements document meets your standards		
XP:	0	There is no requirements document.
MXP:	3	Reviews
8.2 Organize formal requirements inspections		
XP:	0	No inspections.
MXP:	3	Compulsory.
8.3 Use multi-disciplinary teams to review requirements		
XP:	0	There are no documents, no reviews.
MXP:	1	Sometimes an expert joins the team.
8.4 Define validation checklists		
XP:	0	No checklists
MXP:	0	Not implemented yet.
9.1 Uniquely identify each requirement		
XP:	2	User stories may have identifiers.
MXP:	3	Each requirement is assigned a unique tag.
9.2 Define policies for requirements management		
XP:	3	The policy is defined in [1].
MXP:	3	That paper + Rational RequisitePro.
9.3 Define traceability policies		
XP:	0	There is no traceability policy.
MXP:	3	The policy is based on Rational RequisitePro.
9.4 Maintain a traceability manual		
XP:	0	There is no traceability manual.
MXP:	3	Rational RequisitePro database.