Balancing Agility and Discipline with XPrince

Jerzy Nawrocki[1,2,*], Lukasz Olek[1,2], Michal Jasinski[1,2], Bartosz Paliświat[1,2], Bartosz Walter[1,2], Błażej Pietrzak[1], and Piotr Godek[2]

[1] Poznań University of Technology, Institute of Computing Science, ul. Piotrowo 3A, 60-965 Poznań, Poland
[2] PB Polsoft, Poznań, Poland
Jerzy.Nawrocki@put.poznan.pl

Abstract. Most of the contemporary projects require balance between agility and discipline. In the paper a software development and project management methodology called XPrince (eXtreme PRogramming IN Controlled Environments) is presented. It is a combination of XP, PRINCE2 and RUP. Moreover, some experiments and tools are described that create an important basis for the methodology.

1 Introduction

The first reaction to the software crises in the late 60s was call for discipline. In the next 20 years people proposed many standards (IEEE standards, ISO standards etc.). They were followed by maturity models (CMM, ISO 15504 etc.) and discipline-oriented methodologies (e.g. PSP [13], and TSP [14]). Parallel to this process in the 70s first project management methodologies were applied to support software development. Perhaps the first one was PROMPTII created by Sympact Sytems Ltd. and adopted in 1979 by UK's Central Computer and Telecommunications Agency (CCTA). In 1989 CCTA established its own methodology called PRINCE (for PRojects IN Controlled Environments). Seven years later it was modified and since that time it is known as PRINCE2 [18]. It is quite a popular methodology also outside UK. It has got an opinion of rather restrictive but effective project management method.

However, too much discipline kills initiative and flexibility, which are necessary to successfully build complex systems with changing requirements. To help this in the mid 90s so-called agile methodologies arose. They emphasize the need for effective communication between individuals, customer orientation, software-centric thinking and fast responding to changes. Perhaps the most popular agile methodology is Extreme Programming (XP for short) [3]. As usually, there is no silver bullet and both approaches, agile and discipline-oriented, have their advantages and disadvantages. Discipline-oriented methodologies usually suffer from excessive paper work, low flexibility, slow decision processes and inability to accommodate many changes. XP's weakness is relying on on-site customer (in many projects customer representative is too busy and she/he cannot fulfill this

* Corresponding author.

N. Guelfi and A. Savidis (Eds.): RISE 2005, LNCS 3943, pp. 266 – 277, 2006.
© Springer-Verlag Berlin Heidelberg 2006

requirement), lack of written documents (oral communication is fast but when the system is complex and there are many difficult trade-offs after some time it can be hard to remember what was the final solution and why it was chosen), and sometimes too short planning perspective.

As Barry Boehm and Richard Turner have noted, *every successful venture in a changing world requires both agility and discipline* ([5, p. 2]). In the paper an integrated and flexible software development methodology is presented along with accompanying tools which aims at balancing agility and discipline. It is called XPrince (for eXtreme Programming in controlled environments) and it is based on three other methodologies: XP [3], PRINCE2 [18], and RUP [17]. In the next section we describe a two-level approach to team organization which results in a team structure compliant with both XP and PRINCE2. In Section 3 the project lifecycle is discussed. Again our aim was to obtain a lifecycle that would be conformant with both XP and PRINCE2. Then, tools and techniques are presented that aim at providing agility and effectiveness to requirements engineering (Sec. 4) and software construction (Sec. 5). Our aim was to solve the problems associated with XP's weaknesses and preserve agility. To obtain this we have integrated a project management methodology (PRINCE2) with a software development one (XP), and we have elaborated tools that integrate various software engineering techniques. We have integrated a use-case editor with a mock-up generator and an effort estimator (the resulting tool is called UC Workbench). We have also integrated reuse with testing (test-cases are used as a query to find a function or a class).

2 Team Structure

At the thirst glance, PRINCE2 does not fit XP for a number of reasons. One of them is that roles in PRINCE2 are different from those in XP. In PRINCE2 a project is directed by its Project Board which consists of three roles (see Fig. 1a):

- *Executive* – Represents the investor and is responsible for making the project successful from the business point of view. He can cancel the project if necessary.
- *Senior User* – He coordinates end users and focuses on usability aspects.
- *Senior Supplier* – Represents the supplier organization (a senior manager).

PRINCE2 assumes that Project Board members are too busy to look after the project on the day-to-day basis. Therefore, in PRINCE2 there is another role called *Project Manager* who is responsible for tactical level of management. Among others, he prepares plans which are later on accepted by the Project Board and writes progress reports. To balance "intrinsic optimism" of the Project Manager in PRINCE2 there is an optional role of *Project Assurance* whose mission is to check if the reports send by the Project Manager meet the reality. If a project team is small the developers are immediately coordinated by the Project Manager.

XP team has no particular structure diagram. However, from the description given by Kent Beck [3] one can derive a team structure diagram presented in Fig. 1b. That diagram can be "refactored" into the diagram of Fig. 1c. This refactoring allows to see

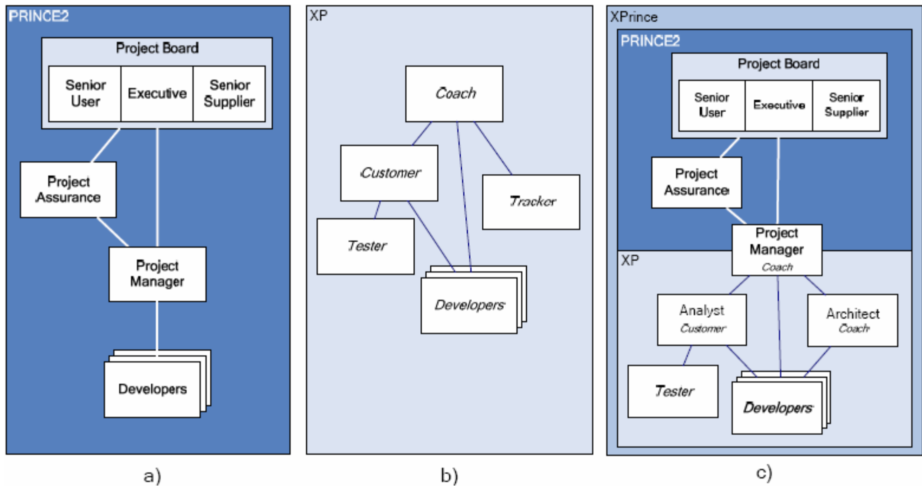

Fig. 1. Minimal project team structure in PRINCE2 (a). A team structure diagram for XP (b). XPrince team structure (c).

XP team as PRINCE-compatible. From the PRINCE2 point of view the Project Manager controls a team of developers. PRINCE2 does not impose any constraints on how the developers should be organized, so from that point of view Analyst or Architect is just a developer. On the other hand, the developers can be unaware of existence of the Project Board. From their point of view the Analyst (a role coming from RUP) is the customer and they have two coaches: Project Manager and Architect. In XP there is only one coach, but its mission is twofold: removing organizational obstacles (e.g. lack of paper) and intervene when developers encounter technical difficulties (e.g. unit tests take too long). Taking this into account we have decided to split the coach role into two roles. *Project Manager* is responsible for right organizational environment (including good contacts with the Project Board), she/he solves interpersonal problems and is rather motivating than directing [4]. A good Project Manager should build his team around character ethic proposed by Stephen Covey [10]. *Architect* is much more technically oriented. It is an additional role not present in XP nor in PRINCE2 (it also comes from RUP). From the developers point of view, Architect is a senior-designer (corresponding to Brook's Chief Surgeon [7]) who is well-experienced and can provide merit (i.e. technical) advices to the developers. The Architect is responsible for establishing and maintaining the architecture and the developers are responsible for "filling in" the architecture with the functionality. A very good description of architect's role is given by Kroll and Kruchten [17].

3 Project Lifecycle

Project lifecycle is a basis for planning. In PRINCE2 (Fig. 2a) a project begins with *Starting-up* (deciding about the management team, preparing a Project Brief and a Project Approach, planning the initiation stage). It can be very short (sometimes a few hours are enough). It is followed by *Initiating a Project* (planning a project, refining the business

case and risks, setting up project controls and project files, assembling the Project Initiation Document). Then a number of *stages* appear (each stage is assigned a list of expected products and it is controlled using its own plan). A project ends with *Closing* (obtaining customer's acceptance, identifying follow-on actions, evaluating the project).

In RUP (Fig. 2b) a project consists of four phases: *Inception* (finding out what and how to build), *Elaboration* (working out the architecture, planning the project, mitigating essential risks, putting the development environment in place), *Construction* (preparing a fully functional beta version of the system), *Transition* (preparing deployment site, training users, checking that user expectations are met and deployment is complete).

XP's lifecycle is the simplest one (Fig. 2c). It consists of a sequence of releases and each release is a sequence of increments. Each release and each iteration starts with a planning session (so-called Planning Game). There is no overall project plan – planning perspective is limited to one release. Sometimes people use so-called zero-functionality increment to arrange things and prepare the development environment.

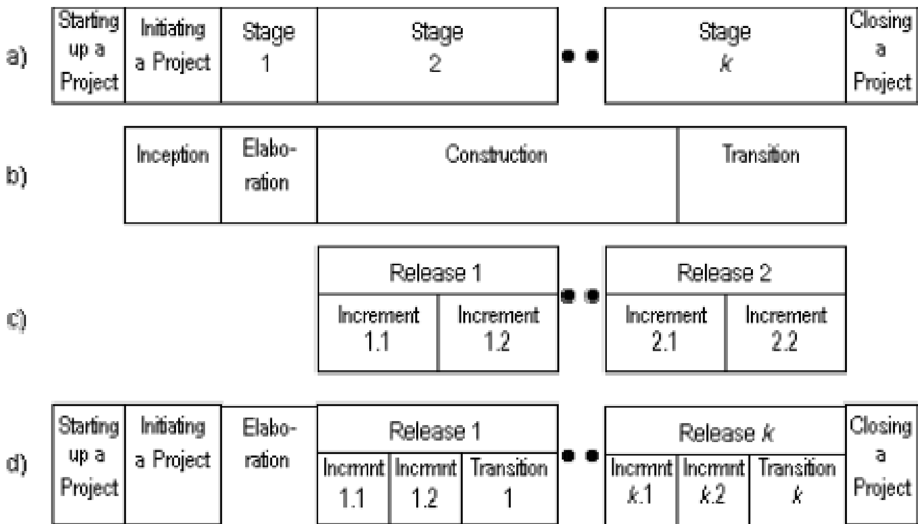

Fig. 2. Project lifecycles proposed by different methodologies: PRINCE 2 (a), RUP (b), XP (c), and XPrince (d)

In XPrince the lifecycle is a combination of all the mentioned approaches. The PRINCE2 concept of starting-closing 'brackets' is quite practical. They are completely non-technical activities, so it makes sense to separate them from the other stages (in RUP it is partially included into Inception and Transition). *Starting up a Project* is usually performed by Project Manager and it has the following objectives:

- Appoint the project management team (see the previous section).
- Produce a vision document (it is a shorter and more concrete version of PRINCE's Project Brief and Project Approach, and it contains an initial version of the business case).
- Plan the initiation stage.

The next stage is *Initiating a Project*. Its aim is to provide a plan and an organizational environment for the project. It is a combination of PRINCE's Initiation and RUP's Inception. It is mainly performed by the Project Manager and the Analyst. Some consultancy with the Architect will also be necessary. The objectives of the stage are the following:

- *Understand what to build*. If necessary, produce a lightweight version of the ConOps document [15] containing a business model based on use cases, list of problems to be solved, and key system functionality required to solve the problems. The key system functionality should be accompanied by a list of quality criteria and work products. The Analyst is responsible for the objective and for updating the risks associated with it.
- *Propose an initial architecture*. It should be a short, high-level description providing information necessary for planning the project. It should also contain a list of the tools that will be used. Nominally the Architect is responsible for the objective and its risks, but if the architecture is pretty obvious the Analyst can do the work.
- *Plan the whole project and refine the business case*. That objective is under supervision of the Project Manager and he is also responsible for maintaining the risks associated with it. A project plan presents a strategic view of the project. To support agility the project plan should be based on the first-things-first principle [10]. It should specify the number of releases and assign features (high-level use cases) to releases. The longer the project the less concrete should be a project plan. Actual planning and contracting should be at the level of releases. In XP there is no project plan – there are only release plans. In XPrince a project plan has been added not only to comply with PRINCE2 but also to provide a wider perspective which can be very useful. It is important to understand that a project plan is a source of valuable information, not an excuse to reject changes. Every change should be welcomed as long as it supports reaching the business objectives stated in the business case.
- *Set-up communication channels and project management environment*. Communication channels include reports (e.g. results of weekly acceptance tests as suggested by XP). The project management environment can be classical, based on files and documents or it can be supported by advanced tools. That objective is the responsibility of the Project Manager.
- *Plan the Elaboration stage*. The *Elaboration* stage is mainly about architecture. The Architect is to propose architectural mechanisms, identify risks associated with the proposed mechanisms, check the risks (e.g. through experiments), and create a framework that will be used by the Developers. The Analyst and the Project Manager use that stage to refine the requirements and the project plan.

Each *Release* stage consists of a number of increments which are followed by transition. At this stage the development process resembles very much XP. The Architect and the Developers produce code and test cases. The Analyst is responsible for requirements and acceptance tests. He also plays the role of on-site customer. An increment is a purely internal checkpoint. Each Release ends with a transition and then a new version of the system is deployed and passed over to end users. As in XP, each increment should have the same duration – that helps the Developers to learn

what an increment is in the sense of time and, as a result, they become better at planning it.

Closing an XPrince project resembles very much its counterpart in PRINCE2. The project is decommissioned, the follow-on actions are identified, and the project is evaluated.

4 Requirements Engineering with UC Workbench

In this section we present a tool supporting requirements engineering based on use-cases. It is called UC Workbench (UC stands for Use Cases). UC Workbench was designed with XPrince's Analyst in mind.

4.1 Text or Diagram?

According to a popular saying, one picture has a value of 1000 words. Unfortunately, it seems it does not hold for requirements engineering. In March and April 2005 we have conducted an experiment at the Poznan University of Technology. The aim of the experiment was to find out which approach, text-based or diagram-based, is better from the understandability point of view. As a representative of the text-based approach we have selected use cases [8, 1]. For diagram-based approach we have chosen BPMN [27] which resembles UML but is specifically designed for business modeling. The participants of the experiment were 4th year students working on their master degrees in Software Engineering (SE) or Business Administration (BA). There were 17 SE students and 11 BA students. The process went through the following steps:

1. A *lecture* presenting an introduction to a given notation (90 minutes).
2. A *rehearsal* session during which the students were given a high level description of PRINCE2 processes expressed in a given notation with a number of seeded defects and their task was to find them. The document was 5 pages long. The session lasted for about 90 minutes.
3. An *experiment* session run in a similar way as the rehearsal. However, this time we have changed the business domain. Instead of PRINCE2 processes the participants were presented business models concerning university regulations (earning university diploma, taking the final exam etc.). They were given one hour to find the defects.

The process was performed twice (each time with different business models) to get more data. Each time there were two groups: one using use cases, and the other working with BPMN diagrams. It could happen that the two groups were not equivalent in terms of their skills. To avoid this, the second time we have switched the groups (the use-case group was given BPMN diagrams and vice versa). Students worked individually. Every detected defect was shortly described on the defect log. As the understandability measure we have assumed the number of defects detected in the document. Defect detection ratio (DDR) for a person p was defined as follows:

$$\mathrm{DDR}(p) = \frac{\text{Number of defects detected by person } p}{\text{Number of all the defects}} \cdot 100\%$$

DDR for use cases was greater than for BPMN and that result was statistically significant (with the significance level 0.05). This justifies the following conjecture: *use cases are easier to understand than BPMN diagrams*. Thus, it is better to express business processes in the form of use cases.

We have also performed another experiment. This time one group was given a business model expressed only in use cases and the other group was given the same use cases accompanied with BPMN diagrams. The latter group detected more defects than the former and the results were statistically significant. Thus, BPMN diagrams are a valuable add-on to the use cases.

Taking into account results of those experiments we have decided that in XPrince requirements engineering will be based on use cases and diagrams will be treated as useful adornments.

4.2 UC Workbench

UC Workbench is a tool developed at the Poznan University of Technology [21] to support requirements management and business modeling based on use cases. We were surprised by the fact that there is no good tool for use-case engineering. UC Workbench provides the following functionality:

- *Editing use cases* with automatic renumbering the steps in the main scenario as well as in all the extensions.
- *Automatic reviews* with detecting 'bed smells' (e.g. undefined and unused actors, too short or too long scenarios, an extension with no steps).
- *Generating of mockups* from the collected use cases. An automatically generated mockup is based on a web browser and it consists of two windows (see Fig. 3): the *scenario window* (presents the currently animated use case), and the *screen window* (shows the screen design). In the case of business modeling the screen window would contain a BPMN diagram.
- *Composing the SRS document* based on IEEE Std. 830-1998. UC Workbench generates the SRS document from the use cases.
- *Generating effort calculators based on Use-Case Points* [16] that are to support XP's Planning Game. In XPrince planning comprises three levels: Use-Case Points (the lowest level) provide default effort estimates that can be later on changed by the experts; Wide-Band Delphi Method is used to support effort estimation by the team (experts); Planning Game (the highest level) controls a dialog between the customer (and Analyst) and the Developers led by the Architect about the scope of the next release.

We believe that appropriate tools can be very helpful in balancing agility and discipline. They can provide information available in discipline-oriented methodologies but faster and cheaper. Due to this, changing requirements are not so a big problem as it used to be.

To evaluate UC Workbench we have performed a simple experiment aiming at comparison of UC Workbench with a popular, general purpose text processor (MS Word). There were twelve participants (students of the 4th year working towards their Master degree in Software Engineering). They were split into two equal-size groups.

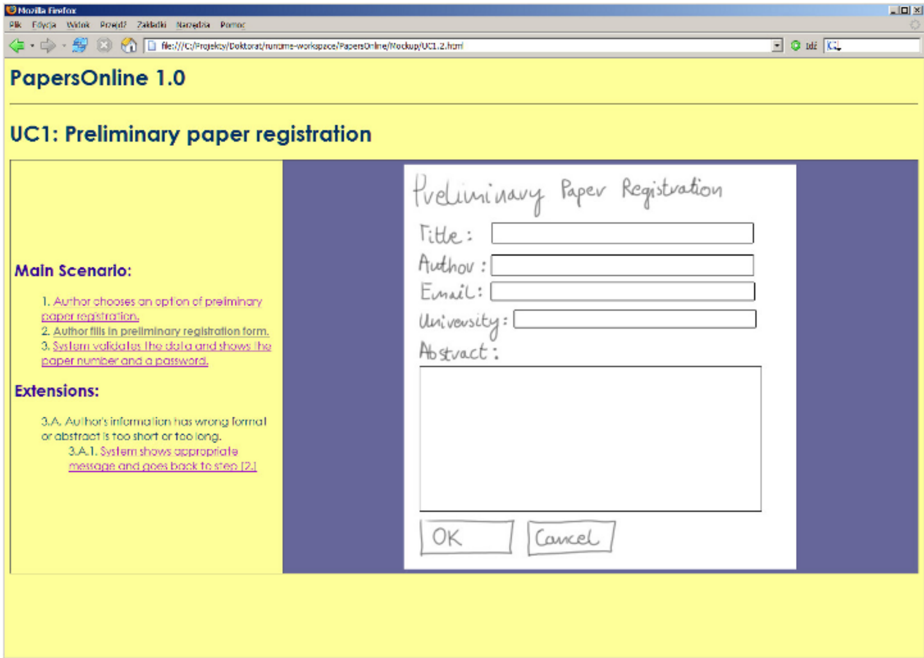

Fig. 3. A screenshot of a mockup generated by UC Workbench

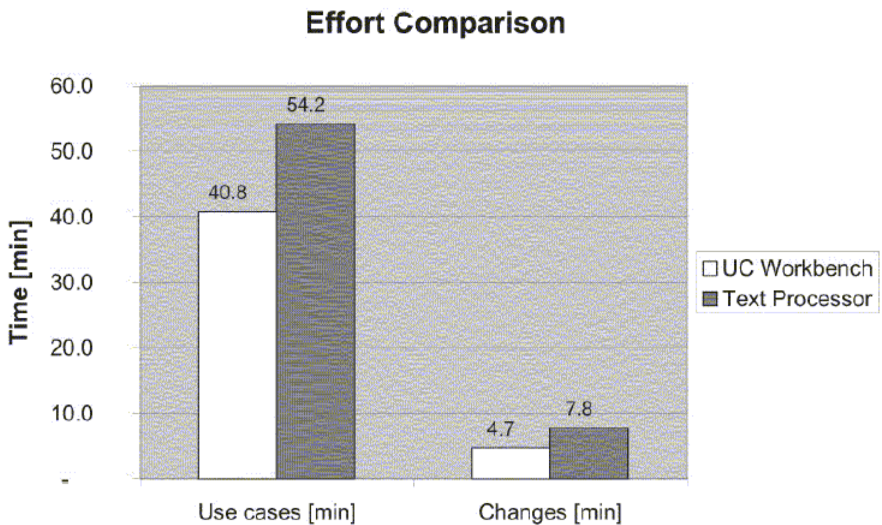

Fig. 4. Effort necessary to type-in (left) and to change (right) use-cases using UC Workbench and MS Word

One group was using MS Word and the other UC Workbench. The students were provided with a draft containing 4 use cases (each 6–9 steps long). There were two steps. First students were asked to type-in the use cases using the assigned tool. Then they were asked to introduce some changes. It turned out (see Fig. 4) that by using UC Workbench one can save about 25% of time at typing-in and about 40% at introducing changes.

5 Developing the Software

5.1 To Do Pair Programming or Not To Do?

Pair Programming is one of key practice of XP. A pair of programmers equipped with a single computer is assigned a programming task. One of the programmers is writing code, while the other is watching, asking questions, and introducing test cases, therefore providing so-called continuous review. Another approach to collaborative programming – called Side-by-Side Programming (SbS) – has been proposed by Cockburn [9]. In this approach a single task is solved by a pair of programmers, each equipped with his own computer.

Results of experimental research on pair programming performance vary from optimistic (speedup at the level of 40% of time and overhead about 20% of the effort compared to individual programming [23, 28, 29]), to quite pessimistic (speedup about 20%, effort overhead about 60% [20]). Unfortunately, up-to-date there are no published experimental data concerning SbS performance.

Recent experiments performed at the Poznan University of Technology [22] indicate that classical Pair Programming is less efficient than the SbS programming. Almost 30 students were working for 6 days in a controlled environment. They were building an Internet application managing conference paper submission and review processes. They were split into three groups: SbS pairs, XP-like pairs, and individuals. It turned out that the SbS pairs were faster by 13% than the XP pairs and by 39% than the individual programmers. Consequently, the SbS effort was by 26% smaller than the effort of XP pairs and only 22% greater than the effort of individuals. This experiment shows that Side-by-Side programming is an interesting alternative to XP-like pair programming and individual programming.

Another experiment run at the Poznan University of Technology in 2005 confirms this observation. 44 volunteer subjects participated in the experiment. They were undergraduate students studying Computer Science (2nd year of study). They had completed various programming courses (including Java and C++) amounting to over 400 hours. Again we have decided that the subjects will work at the university (controlled conditions). Some students worked in XP-like pairs and the others in SbS pairs. They were given two 9-hour long programming assignments and they worked according to predefined process. The results show, that Side-by-Side programming is faster than XP-like Pair Programming by 16%–18% percent.

Interestingly, all the subjects participating in the experiment preferred collaborative programming (55%) over individual problem solving (40%). Moreover, 70% of the subjects preferred Side-by-Side approach and only 30% voted for XP-like pair programming.

In XPrince the Developers can choose between individual, SbS or XP-like pair programming. If they choose individual programming, a reviewer is assigned to check quality of the work (it is another Developer).

5.2 Code Refactoring

Refactoring is one of the core XP techniques supporting software maintenance [3]. It depends on changing source code internal structure to improve its readability and adjust it to changing requirements, while preserving its observable behaviour (that allows to reuse the 'old' regression tests) [12].

An agile project most of its life-time stays at maintenance phase because actually it constantly evolves over time. That makes refactoring (and any other software maintenance technique) very important.

Unfortunately, refactoring is also a costly and error-prone technique. Changes are likely to introduce mistakes and unexpected side-effects, which effectively alter the program behaviour. Preventing this requires additional effort, which adds even 80% to overall project cost [6]. However, the investment pays back with subsequent maintenance events: according to some authors [25], refactoring is cost effective after sixth such action. Small-scale experiment performed at the Poznan University of Technology with graduate students in Software Engineering revealed that the refactoring-related overhead in every increment decreased from 75 down to 7 percent in just three development cycles, as compared to a similar incremental process performed without code restructuring [26]. Thus, disciplined refactoring contributes to code quality, whereas its cost is justified for projects with several functional increments or maintenance actions.

As in typical XPrince project there are many increments, refactoring and programming environments supporting it (e.g. IBM Eclipse) are strongly recommended.

5.3 Integration of Code Reuse and Test-First Coding

It is a widely known fact, that code reuse can reduce software development cost and increase reliability. For instance, Toshiba reported decrease in defects by 20–30%, and Hewlett-Packard even by 76% [11]. The main problem with code reuse is finding a piece of code which can be used to accomplish some given goal. That task is definitely not a trivial one, especially when the size of repository and the number of people involved are significantly big (which is actually the situation when the systematic reuse starts paying off). Such a difficult task cannot be approached without a support of both well organized processes and well designed tools.

One of the most interesting approaches to improving the search process is so-called behavioural retrieval [2, 24]. A behaviour of a class or a method is specified by a small program showing the input and expected output. That idea was first proposed by Podgurski [24]. Unfortunately, it was not widely used in practice because of a common belief that specifying a class or a method is not trivial and it will be faster to write a piece of code than to find it in a repository (the technique of behavioural retrieval is oriented towards relatively small pieces of code like functions or classes).

But in XP (and in XPrince as well) coding is preceded by preparing test cases (it is so-called test-first coding). To support code reuse and test-first coding we have developed a tool that takes test cases written in jUnit and using the technique of

behavioural retrieval searches through a code repository looking for a class or a method which potentially satisfies this rough specification. If it succeeds, the Developer can check more precisely if the found piece of code satisfies his requirements and if so, the work is done. If not, he can start programming as he did if he was just doing classical test-first coding.

The proposed technique is not to replace existing methods of searching through repositories. It is rather a complementary solution designed to operate well for small pieces of code, for which commonly used techniques as text-based retrieval or faceted classification may not be sufficient.

To evaluate the proposed tool we have performed a simple test. Nine programmers (5th year students) were given description of 10 relatively simple program units (the descriptions were given in a natural language). They were asked to provide a set of test cases that would allow finding the units in the code repository. In 9 of 10 cases the programmers correctly specified the units. The only problem was with a class representing strings that can be matched with regular expressions (4 of 9 programmers made a wrong assumption).

6 Conclusions

By integrating different methodologies and supporting them with appropriate tools one can obtain a balance between agility and discipline. The solutions presented in the paper follow from our 7-years long experience in running the Software Development Studio at the Poznan University of Technology. The described methodology (XPrince) and the first version of UC Workbench went also through in-field testing – they have been used in a commercial project for a government agency.

Acknowledgements

We are thankful to PB Polsoft, a software company with headquarters in Poznan who provided us with a feedback from a real industrial project run according to XPrince, and personally to Grzegorz Leopold who created that opportunity.

This work has been financially supported by the State Committee for Scientific Research as a research grant 4 T11F 001 23 (years 2002-2005).

References

[1] Adolph, S., Bramble, P., Cockburn, A., Pols, A., *Patterns for Effective Use Cases*, Addison-Wesley, 2002.

[2] Atkinson S., *Examining behavioural retrieval*, WISR8, Ohio State University, 1997.

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

[4] Blanchard, K., Zigarmi D., Zigarmi P., *Leadership and the One Minute Manager*, 1985.

[5] Boehm, B., Turner, R., *Balancing Agility and Discipline. A Guide for Perplexed*, Addison-Wesley, Boston, 2004.

[6] Bossi P., Repo *Margining System.* http://www.communications.xplabs.com/lab2001-1.html, visited in 2004.

[7] Brooks, F., *A Mythical Man-Month*, Addison-Wesley, Boston 1995.
[8] Cockburn, A., *Writing Effective Use Cases*, Addison-Wesley, Boston, 2000.
[9] Cockburn, A., *Crystal Clear. A Human-Powered Methodology for Small Teams*. Addison-Wesley, Boston, 2005.
[10] Covey, S., *The Seven Habits of Highly Effective People*, Simon and Schuster, London, 1992.
[11] Ezran M., Morisio M., Tully C., Practical Software Reuse, Springer, 2002.
[12] Fowler M., *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, Boston, 1997.
[13] Humphrey, W., *A Discipline for Software Engineering*, Addison-Wesley, Reading MA, 1995.
[14] Humphrey, W., *Introduction to the Team Software Process*, Addison-Wesley, Reading MA, 2000.
[15] *IEEE Guide for Information Technology – System Definition – Concept of Operations (ConOps) Document*, IEEE Std. 1362–1998.
[16] Karner, G., Use Case Points – Resource Estimation for Objectory Projects, Objective Systems SF AB, 1993.
[17] Kroll, P., Kruchten, Ph., *The Rational Unified Process Made Easy*, Addison-Wesley, Boston, 2003.
[18] *Managing Successful Projects with PRINCE2*, TSO, London, 2004.
[19] Nawrocki, J., Jasiński, M., Walter, B., Wojciechowski, A., Extreme Programming Modified: Embrace Requirements Engineering Practices, 10th IEEE Joint International Requirements Engineering Conference, RE'02, Essen (Germany), IEEE Press, Los Alamitos (2002) 303–310.
[20] Nawrocki, J., Wojciechowski, A.: Experimental Evaluation of Pair Programming. In: Maxwell, K., Oligny, S., Kusters, R., van Veenendaal, E. (eds.): Project Control. Satisfying the Customer. Proceedings of the 12th European Software Control and Metrics Conference ESCOM 2001. Shaker Publishing, London (2001) 269–276.
[21] Nawrocki, J., Olek, L., UC Workbench – A Tool for Writing Use Cases and Generating Mockups. **In**: Baumeister, H., Marchesi, M., Holcombe, M., (Eds.) Extreme Programming and Agile Processes in Software Engineering, *Lecture Notes in Computer Science* 3556, (2005), 230–234.
[22] Nawrocki, J., Jasinski, M., Olek, L., Lange, B.: Pair Programming vs. Side-by-Side Programming. Proceedings of the European Software Process Improvement and Innovation Conference, *Lecture Notes in Computer Science* 3792 (2005), 28–38.
[23] Nosek, J. T., The Case for Collaborative Programming. *Communications of the ACM*, Volume 41, No. 3 (1998) 105–108.
[24] Podgurski, A., Pierce, L., *Retrieving reusable software by sampling behavior,* ACM TOSEM, Volume 2 , No. 3 (1993) 286–303.
[25] Stroulia E., Leitch R., K., *Understanding the Economics of Refactoring*. In: Proc. of the Fifth ICSE Workshop on Economics-Driven Software Engineering Research. Portland, 2003.
[26] Walter B., *Analysis of Software Refactorings*. PhD dissertation, Poznań University of Technology, Poznań (Poland), 2004 (in Polish).
[27] White, S., Introduction to BPMN, http://www.bpmn.org/Documents/_Introduction%20to%20BPMN.pdf, visited in 2005.
[28] Williams, L.: *The Collaborative Software Process*. PhD Dissertation at the Department of Computer Science, University of Utah, Salt Lake City (2000).
[29] Williams, L. et al.: Strengthening the Case for Pair Programming. *IEEE Software*, Volume 17, No. 4 (2000) 19–25.