# Flexible Software Development with XPrince

Jerzy Nawrocki*
jerzy.nawrocki@put.poznan.pl

**Abstract:** Discipline-oriented methodologies (e.g. PSP/TSP, PRINCE2) and agile ones (the best known is eXtreme Programming) have their advantages and disadvantages. To be successful one needs to balance agility and discipline. Such a balance can be obtained with XPrince (eXtreme PRogramming IN Controlled Environments). In the paper selected aspects of the XPrince methodology have been presented with emphasis on the role of Analyst, Architect and Project Manager.

**Key Words:** Software engineering, project management, agility, XP, PRINCE2, RUP

## 1 Introduction

There are two opposite approaches to software development: plan-driven and change-driven. The former arose as a reaction to software crises discovered in late 60s. Many people attributed problems with software development to chaotic processes and they believed that those problems can be solved by rigorous planning and strict definition of development processes. That way of thinking resulted in advanced effort estimation methods (including Function Points [2], COCOMO [4], PROBE [11]), very strict project management methods (e.g. PRINCE2[18]), and various maturity models (for instance ISO 9001, CMM etc.).

Unfortunately, software crises survived. People blame changes for it. They are everywhere: in specifications – customers usually change their minds; in teams – people join the team and they go away; in software components and tools – there is always a temptation or even pressure to use new technology, but in many cases 'new technology' means 'new troubles'. If there was no change, software development would be much easier. But wishful thinking is not proactive. If we cannot defeat changes let's embrace them. A solution is a change-driven development which is also known as agile software development. Perhaps the best known example of an agile methodology is eXtreme Programming (XP for short) [3].

But agility is not the panacea – there are situations in which plan-driven development has some advantages. What in fact is needed it is a kind of flexibility based on balancing agility and discipline.

In the paper a new methodology is outlined which is called XPrince – eXtreme PRogramming IN Controlled Environments. It aims at balancing plan-driven and change-driven approaches to software development and it is based on practices coming (mainly) form XP, PRINCE2 and RUP [12]. It is a result of our 8-years-long experience with running the Software Development Studio (SDS) at the Poznan University of Technology. In January 2005 companies cooperating with SDS decided to launch the XPrince Consortium which is responsible for maintenance and further development of the methodology. First the team organization is presented and next the three main roles are described, i.e. the analyst, the architect, and the project manager.

* Institute of Computing Sci., Poznan University of Technology, Piotrowo 2, PL-60-965 Poznan, Poland

## 2 Team Organization

In XP the team structure is rather flat: there is a *customer*, a few *developers* (i.e. programmers), a *coach* (that role resembles a project manager), and a *tracker* (a person responsible for collecting basic metrics concerning time and development velocity). A customer makes business decisions (what to do now and what can be postponed) and he is also a source of domain knowledge for the developers.

Another approach to team organization is presented by PRINCE2. Here the focus is on management issues. A team consists of three layers: *developers*, above them there is a *project manager* that can be assisted by a *project support*, and on the highest level there is a *project board* that includes an *executive*, a *senior supplier*, and a *senior user*.

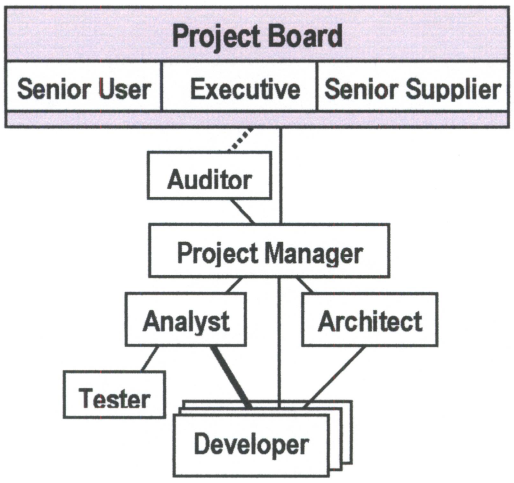


**Figure 1. Team structure in XPrince.**

The team structure used in XPrince is depicted in Figure 1. It is a combination of roles appearing in PRINCE2, XP, and RUP. One can distinguish two layers:

- a project management layer and
- a software development layer.

The former consists of Project Manager, Auditor, and Project Board and it is borrowed from PRINCE2. The latter includes Analyst accompanied by Tester, Architect, and Developers. The Analyst role is responsible for requirements elicitation, documentation, their analysis, and acceptance tests. That role can be supported by a tester who knows very well various testing tools and is an expert with testing non-functional requirements (they are usually most difficult to verify). XP's customer is here decomposed into Senior User representing end-users, Executive making the most important decisions about the project, Auditor checking is reports provided by Project Manager meet reality, and Analyst who serves the Developers with the domain knowledge. In XP all that roles are performed by one person who is called *on-site customer* (i.e. customer who works with the Developers in the same room and is ready to answer their questions). In XPrince the customer role can be distributed between a few people what provides more flexibility (in many case Executive is too busy to answer Developer's questions on time).

In XP the development team is a team of peers: there is no one who has more decision 'power' than the others. In XPrince a team organization resembles 'surgery team' discussed by Frederick Brooks [5]. The Architect is the surgeon. His main duty, as proposed in RUP, is to manage technical risk. He is responsible not only for proposing architecture (e.g. sketching UML diagrams), but also for inventing experiments that quickly check if the proposed solution is technically feasible. And last but not least, he has to have very good communication skills to be able to explain his ideas and to listen to doubts other people can have.

The three roles, Analyst, Architect, and Project Manager, are a backbone of XPrince and they will be described in more detail in the subsequent sections of the paper.

## 3 Analyst and Requirements Engineering

### 3.1 Project Vision

It is extremely important to be aware of the problem one wants to solve by acquiring a computer-based system. Understanding the problem gives freedom to choose another approach that can be simpler, faster, and cheaper than the first one envisioned by the customer. Project Vision in XPrince is a slight modification of that one used in RUP. It is a very short document (not longer than 1 A4 page) and it should answer the following questions:

- Who is the customer?
- What is the problem?
- What are implications of the problem?
- What is general idea how to solve the problem?

Project Vision should be ready before the project is launched, and any change to it should be considered a big change that brings a lot of risk. Analyst is to maintain the Project Vision and make every project stakeholder aware of it

### 3.2 Business Process Description

According to Curtis' law, "*good designs require deep application domain knowledge*" [8]. Many contemporary IT projects concern e-Commerce, e-Government etc. In this context domain knowledge is usually expressed in a form of business processes and it is extremely important to describe those processes in a clear way and to pass this knowledge from end users to the development team.

There are two most popular approaches to describing business processes: *diagrams* (e.g. UML or BPMN diagrams) and *text*. The latter can have a form of *use cases* [6, 1]. In Fig. 2 an example of a use case is presented that describes how to run a project according to the PRINCE2 methodology [18]. The first line contains use-case identifier (*UC-1*) and its name ("*Running a project according to PRINCE2*"). Then the actors are introduced (main and supporting). Most important part of each use-case is the main scenario that is a sequence of steps expressed with a natural language. That scenario can have a number of extensions. Each extension consists of an event (e.g. "*One more stage is needed*") and corresponding steps.

In 2005 an experiment have been conducted at the Poznan University of Technology aiming at comparison of BPMN diagrams and use cases. The participants were 4th year students working on their master degrees in Software Engineering (SE) or Business Administration (BA). There were 30 SE students and 11 BA students. We have split them into two groups: A and B. Group A worked with BPMN diagrams and consisted of 18 SE and 6 BA students. Group B was given description of the same business processes but expressed with use cases.

It consisted of 12 SE and 5 BA students. We have seeded defects into the descriptions and we have observed average value of the defect detection ratio for each group (i.e. the number of detected defects to the number of all the defects in a document). It has proved that defect detection ratio for use cases was greater than for BPMN diagrams and that result was statistically significant with significance level 0.01. That led us to the conclusion that

*use cases are more readable than BPMN diagrams*

(more details on the subject can be found in [17]). Thus, we have decided that in XPrince both description of business processes and requirements specification will have a form of use cases.

**UC-1**: Running a project according to PRINCE2

**Main actors**: Customer, Supplier

**Supporting actors**: Executive, Project Manager, Project Management Team

**Main scenario**

1. Customer and Supplier appoint Executive and Project Manager.
2. Executive and Project Manager complete Project Management Team.
3. Project Management Team continues starting up a project.
4. Project Management Team initiates a project.
5. Project Management Team controls execution of a stage.
6. Project Management Team closes a project.

**Extensions**

5a. One more stage is needed.

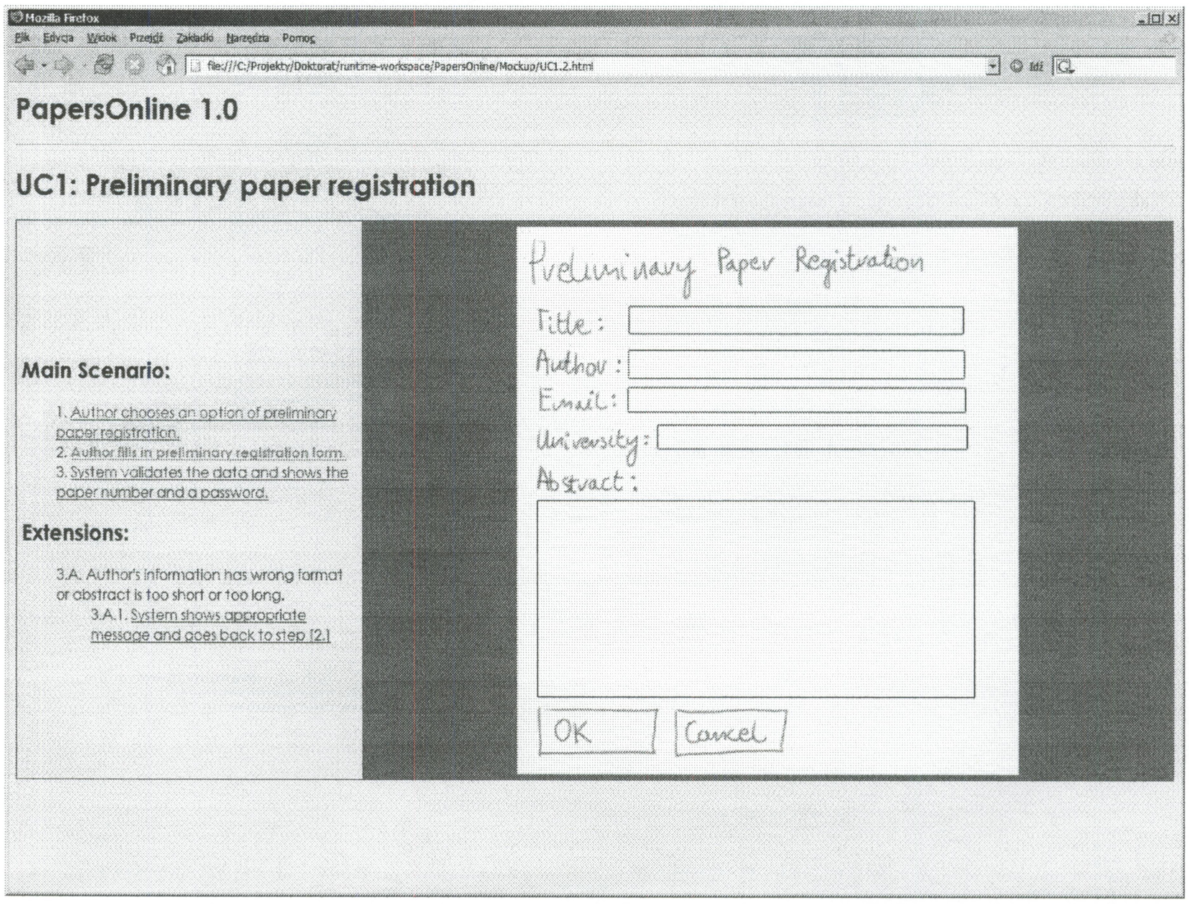
  5a1. Step 5 is repeated.

**Figure 2. An example of a use-case describing a business process.**

### 3.3 Requirements Specification

Functional requirements can be described with use cases and that approach becomes more and more popular [6, 1]. The problem is that usual customer cannot write good use cases (use-cases resemble novels – everybody can easily read them when they are well written but few people can write them that way). That is the duty of the analyst. But analyst does not have domain knowledge. The solution is to allow customers and end-users to communicate their comments and requests in a natural language. If there is trust between the customer and the analyst the requests can be communicated even orally. The analyst should collect the requests and to present them in a form of use-cases.

To support communication between the analyst and the customer we have developed a tool called UC Workbench (UC stands for Use-Cases). It is a use-case-oriented editor equipped with a bad-smell detection mechanism (e.g. too long use-cases or use-cases without extensions). What is important from the point of view of customer-analyst communication, UC Workbench can generate a simple mockup that combines use-cases with GUI design. In Fig. 3 a screenshot of a mockup is presented. To the left there is a use case and to the right there is a low fidelity design of a screen associated with Step 2 of the use case.


Mozilla Firefox
Plik Edycja Widok Przejdź Zakładki Narzędzia Pomoc
file:///C:/Projekty/Doktorat/runtime-workspace/PapersOnline/Mockup/UC1.2.html
Idź
Preliminary Paper Registration
Title:
Author:
Email:
University:
Abstract:
OK
Cancel

PapersOnline 1.0

UC1: Preliminary paper registration

**Main Scenario:**

1. Author chooses an option of preliminary paper registration.
2. Author fills in preliminary registration form.
3. System validates the data and shows the paper number and a password.

**Extensions:**

3.A. Author's information has wrong format or abstract is too short or too long.
  3.A.1. System shows appropriate message and goes back to step [2.]

**Figure 3. A screenshot of a mockup generated by UC Workbench.**

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. One group was using MS Word and the other UC Workbench. The UC Workbench group saved about 25% of the time at typing-in and about 40% at introducing changes. More information on UC Workbench and the experiment can be found in [14, 15].

## 3.4 Acceptance testing

Analyst is responsible not only for writing requirements specification and its maintenance. His job is also to design acceptance tests. Perhaps many ideas concerning acceptance testing will come from the customer. If advanced tools are to be used and the analyst does not have sufficient knowledge he can be supported by a tester.

Acceptance tests should be assigned to requirements. If use-cases are used for functional requirements specification, the test scenarios can be immediately derived from them. Each scenario can be tested with different input data, so data-driven testing seems quite reasonable in this context [9].

## 4 Architect and Software Construction

### 4.1 Software Design and Modeling

The architect is responsible for managing technical risk from the beginning till the end of the project. He should propose a project approach, a general architecture and appropriate tools. An important issue is software modeling. It is useful but dangerous. The architect must be able to communicate his ideas to the development team and from that point of view UML diagrams can be very useful. The danger is that modeling will take too much time. According to the Agile Alliance's Manifesto, working software is more important than comprehensive documentation. As Fowler has noticed [10], every time when communicating about architecture one must remember that we need descriptions (models) that are correct, consistent, and readable but they do not have to be complete. Completeness is important when one is able to generate software from a description. From the documentation (i.e. communication) point of view completeness is not necessary. If something is obvious then modeling it makes no sense – by omitting obvious things one can save time and can make communication (documentation) more effective.

### 4.2 Reusability and Test-First Coding

Code reuse can reduce software development cost and increase reliability. The main problem with code reuse is finding an appropriate piece of code in a code repository. 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 searches through a code repository looking for a class or a method which potentially satisfies this rough specification [13]. That searching is automatic and quite short. If it fails (i.e. no appropriate piece of code is found) the developer should write the code on his own – there is no risk: the developer can only win.

To evaluate the approach a simple experiment has been conducted. Nine programmers (5th year students) were given a natural-language description of 10 rather simple program units. 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 described approach 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. Its main advantage is encouraging the programmers to work on test cases before they start coding.

## 5 Project Manager and Running the Project

### 5.1 Project Lifecycle

A project lifecycle used in XPrince is a combination of lifecycles advocated by PRINCE2, XP, and RUP (see Fig. 4). It conforms to PRINCE2 as the lifecycle is split into Starting-up, Initiating, a number of stages, and Closing. The first stage is Elaborating and it is followed by a number of Releases. The aim of Elaborating is to provide a good architecture an which software development could be based (like in RUP).

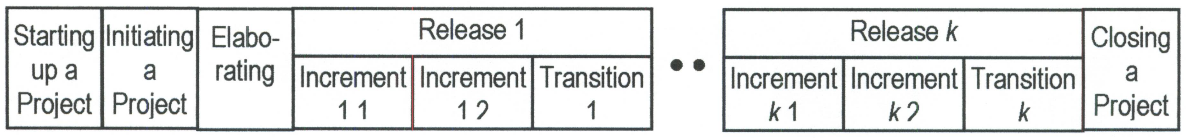


**Figure 4. A project lifecycle according to XPrince.**

The central part of the lifecycle consists of releases – other processes merely support them and, depending on a project, they can be very short or quite long. Each Release is divided into increments (like in XP) and in the end there is a Transition (Fig. 4 suggests two increments per release but obviously there can be more). It is very important not to wait with the Transition till the end of the project – early transitions can reveal some important problems (risk factors) influencing the overall project success.

## 5.2 Project Planning and Progress Tracking

Without planning there is no effective progress tracking and project control. In XPrince planning is done collaboratively but it is managed by the Project Manager. There are three levels of project planning in XPrince: project planning (during Initiating), release planning (at the end of previous stage – that is different from XP), and increment planning (at the beginning of each increment). Release and increment planning is done according to the Planning Game introduced by K. Beck in XP [3]. The difference is that in XPrince use cases are used instead of user stories. First the developers estimate the effort necessary to implement each use case (here the Wideband Delphi method is used). Next the customer (or a person representing him) chooses the use cases to be implement in the nearest release (increment) in such a way that the total effort is not greater than the amount of man-hours available for the release (or increment).

The problem is how to track progress. It depends very much on a stage. Initiating and Elaborating are very difficult to plan and to track progress on-line. During a release stage it is much easier to track progress. As each release (and increment) is to provide working software the best approach is to observe the results of acceptance testing. However, it requires that test cases are available early (in practice they can be created in parallel to the development process) and their execution has been automated (manual tests take to much time and there is a danger that they will be abandoned).

Another problem is test coverage. From the progress tracking point of view it is important to know how requirements (use cases) are covered by acceptance tests. There are two strategies:

- uniform distribution of acceptance tests among use cases (all the use cases assigned to the increment are considered equally important, so each of them should have a number of acceptance test proportional to its cyclomatic complexity);
- importance-base distribution (it is assumed that importance of use cases assigned to a release differs significantly, so a reasonable approach is to check if number of acceptance tests reflects importance of corresponding use cases).

## 5.3 Individual or Pair Programming?

In pair programming, as it is advocated by XP, a pair of programmers is equipped with a single computer and they are assigned a single programming task. While one programmer is writing code, the other is watching (it is so-called continuous review). Cockburn [7] proposed another approach to collaborative programming called Side-by-Side Programming (SbS). In this approach a pair of programmers is given one task but each programmer has his own computer, so they can work on different parts of the task.

There is still an open question which approach is better. Recent experiments performed at the Poznan University of Technology [16] 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. However, the question which approach is better remains open. The answer depends on the context. There are some people who hate other people watching them working – they will never be good pair mates. On the other hand XP-like pair programming is a very good form of mentoring. A flexible methodology should allow for different modes of programming. To make it possible in a controlled way a Project Manager should use Quality Log as required by PRINCE2. That log describes for each artefact which quality control method is assigned to it (it can be a classical review if the artefact was developed by an individual programmer, a continuous review if XP-like pair programming has been use etc.).

## 6 Conclusions

In the paper a flexible software development methodology has been outlined. From the programmers point of view it appears to be XP. They have to focus on programming and the only documentation they have to produce is code and test cases. On the other hand, from the top management point of view XPrince is just an instantiation of PRINCE2. Team organization, lifecycle, and management documentation (which is produced by the Project Manager and Project Support) conform to that project management method.

A methodology is important as it creates a communication framework (one knows what he can expect from his teammates and what is his responsibility). However, it is important to remember that people are the most important assets of each project and a methodology can never be a replacement for good developers.

*This work has been supported by the Poznan University of Technology under grant BW 91-429/06. The author is also grateful to his colleagues for helpful discussions and for developing tools supporting the XPrince methodology: Lukasz Olek for UC Workbench, Bartek Paliswiat for code searching engine, Blazej Pietrzak and Bartek Walter for bad-smell detection plug-in for IBM Eclipse, and Michal Jasinski for running pair programming experiments. Thanks also go to the companies participating in the XPrince Consortium, especially to P.B. Polsoft and his vice-president Grzegorz Leopold for the courage to be the first in adopting the XPrince practices.*

## References

1. Steve Adolph *et al.*: *Patterns for Effective Use Cases*, Addison-Wesley, Boston, USA 2003.
2. A.J. Albrecht and J.E. Gaffney: *Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation*. IEEE Trans. on Software Eng. SE-9 (1983), 739-648.

3. Kent Beck: *Extreme Programming Explained*, Addison-Wesley, Boston, USA 2000.
4. Barry W. Boehm *et al.*: *Software Cost Estimation with COCOMO II*, Prentice Hall PTR, Upper Saddle River, New Jersey 2000.
5. Frederick P. Brooks: *The Mythical Man-Month*, Addison-Wesley, Boston, USA 1995.
6. Alistair Cocburn: *Writing Effective Use Cases*, Addison-Wesley, Boston, USA 2001.
7. Alistair Cocburn: Crystal Clear: *A Human-Powered Methodology for Small Teams*, Addison-Wesley, Boston, USA 2005.
8. Albert Endres and Dieter Rombach: *A Handbook of Software and Systems Engineering*, Pearson Education, Harlow, England 2003.
9. Mark Fewster and Dorothy Graham: *Software Test Automation*, Addison-Wesley, Harlow, England 1999.
10. Martin Fowler and Kendall Scott: *UML Distilled*, Addison-Wesley, Boston, USA 2000.
11. Watts S. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, Reading, Massachusetts 1995.
12. Per Kroll and Philippe Kruchten: *The Rational Unified Process Made Easy*, Addison-Wesley, Boston, USA 2003.
13. Jerzy Nawrocki and Bartosz Paliswiat: *Using Test Scripts for Searching a Code Repository*, in: J. Gorski and A.Wardzinski (eds.): *Software Engineering – New Challenges*, WNT, Poland 2004, 157-168. (in Polish)
14. Jerzy Nawrocki and Lukasz Olek: *UC Workbench: A Tool for Writing Use Cases and Generating Mockups*, Lecture Notes in Computer Science 3556, 230-234.
15. Jerzy Nawrocki and Lukasz Olek: *Use-Cases Engineering with UC Workbench*, in: K. Zielinski and T. Szmuc (eds.): *Software Engineering: Evolution and Emerging Technologies*, IOS Press, 2005, 353-364.
16. Jerzy Nawrocki, Michal Jasinski, Lukasz Olek, Barbara Lange, *Pair Programming vs. Side-by-Side Programming*, Lecture Notes in Computer Science 3792, 28-38.
17. Jerzy Nawrocki, Tomasz Nedza, Miroslaw Ochodek, and Lukasz Olek: *Describing Business Processes with Use Cases*, Proceedings of BIS'06, Lecture Notes in Computer Science, 2006 (in printing)
18. [PRINCE2] OGC: *Managing Successful Projects with PRINCE2*, TSO, London 2002.