Extreme programming and its development practices.

Radmila Juric [1,2]

[1] South Bank University Business School
[2] Radmila Juric now works within the School of Informatics, University of Westminster

Extreme Programming and its Development Practices

Radmila Juric
South Bank University Business School, 103 Borough Road, London SE1 0AA, UK
Tel (+44) (0) 20-7815-7888, Fax (+44) (0) 20-7815-7793, E-mail: juricr@sbu.ac.uk

Abstract: Extreme Programming (XP) has attracted our attention through its fierce denial of many of our well-accepted software engineering practices which we consider today as a sound approach to the development of intensive software systems. XP has been declared to be a new way of software development: a lightweight methodology, which is efficient, low-risk, flexible, predictable, scientific, and distinguishable from any other methodology. In the core of the XP practices are programming activities, with strong emphasis on oral communications, automated tests, pair programming, story-telling culture and collective code-ownership at any time in the XP project.. This paper gives an overview of XP practices and raises some serious concerns regarding their role in conceptual modelling and code generation; which directly affects software architecture solutions. The paper also tackles similarities between Rational Unified Process (RUP) and XP, which have lately been very often juxtaposed by software developers.

Keywords: XP, RUP, OO, Analysis, Design, SE

1. Introduction

A software development process, which guarantees to build a software product or enhance an existing one is expected to offer effective guidelines and capture the best practices of software engineering (SE) disciplines in order to deliver an adequate solution, reduce risks and increase the predictability of software intensive systems. The majority of SE practices are centred today on object oriented (OO) technology and new standards in the format of the Unified Modelling Language (UML) (OMG, 1999), (Booch et al.,1998). It has been designed to provide modelling constructs suitable for any project, any system and any development process. Various methodologies and frameworks have emerged recently (Jacobson et al., 1999) and many of them incorporate the UML elements and have elaborated them towards issues that every SE practice is expected to address (Juric, 1998):

(a) The coverage of a project and system life cycle and its role within a particular workflow,
(b) The set of *rules* that represent the philosophy/practices of the methodology/frameworks,
(c) Fully described deliverables and their employment within different models of the system.

XP came to light within the last two years as an answer to all the problems we still experience when delivering software solutions. XP attracted everybody's attention with its fierce denial of many of our well-accepted SE practices, the courageous approach to them and unusual delivery of all the issues mentioned in (a)-(c) above.

This paper represents an overview of XP philosophy regarding its development practices, their evolution and employment within the XP lifecycle.. At the time of writing there were only a few sources available where one could learn XP (Beck, 1999), (OTUG, 2000), (XP, 2000) and almost non existent examples of XP principles which had been put into practice (Beck et al., 1998). Consequently, the paper represents solely the author's interpretation of XP practices with no attempts to reject them, despite many criticisms and concerns. This helps us to keep an open mind towards these new trends in SE and new answers to our dogmatic approaches to the development of software intensive systems.

The paper is organised as follows: Section 2 describes what XP and its values, principles and practices. Section 3 gives the author's overview of the XP philosophy and raises concerns regarding the core of XP practices and its role in conceptual modelling and code generation. Section 3 gives a

short overview of similarities and contradiction between RUP and XP, which have lately been very often juxtaposed by software developers.

2. What is XP

XP is declared to be a new way of developing software: a *lightweight methodology* which is efficient, low-risk, flexible, predictable, scientific... in brief distinguishable from any other methodology. (Beck, 1999). Furthermore, XP is defined as a *discipline of software development practices*: it strictly prescribes activities you need to complete if you want to claim that you apply XP. It is designed for smaller projects and teams of two to ten programmers, which result in efficient testing and running of given solutions in a fraction of a day. Its incremental planning approach and concrete and continuing feedback from short cycles of the software development allow an evolutionary design process that lasts as long as its system. There is strong emphasis on oral communications and automated tests that monitor progress of the software development, allowing the system to evolve and detect anomalies very early. Above all, XP claims to offer a flexible schedule of the implementation of the system's functionality that actively supports changeable business needs. In short, XP promises to reduce project risks, achieve adoption of constantly changing business requirements/needs and improve productivity throughout the life of the system – all at the same time. This sounds over enthusiastic: are we witnessing a birth of a 'silver bullet' which brings answers to all problems that the SE community has experienced in the last 30 years? We will not answer to the question until XP proves itself in practice, but it is easy to see that almost all XP practices are not new but are re-scheduled within the development process and interwoven to the greatest possible degree.

2.1. XP and Risks Involved in the Software Development Process

XP addresses risks at all levels of the software development process, which requires communication of the XP discipline to programmers, managers and customers. XP project looks at problems/risks of the development process itself and derives solutions that dictate a set of XP activities. Some examples of risks are given in (Beck, 1999) which are addressed by XP through answers that derive solutions immediately. For example, schedule slips are remedied by a set of iterative and short release cycles. Within a release XP uses one-to-four-week iterations, and within one iteration XP plans one-to-three-day tasks. XP also calls for implementing priority features first. All releases are chosen as a smallest set of tasks that makes business sense. A comprehensive suite of test is re-run whenever a system suffers changes which ensures a flexibility and constant support of changeable requirements. However, as XP shortens the release cycle, there is less change during the development of a single release.

2.2. XP Values, Principles, Activities and Practices

XP uses *values* as the main criteria for a successful software solution. In other words, the XP *values* serve as a guarantee, which shows that an XP's set of practices is taking the right direction. XP *values* are:

Communications: XP employs practices that require communications, such as:
- *unit-testing* (programmer-programmer, customer-programmer),
- *pair programming* (programmer-programmer),
- *task estimation* (programmer-manager, programmer-customer).

Simplicity: XP requires choosing the simplest task you could possibly work on.

Feedback: XP requires feedback at different time-scale: minutes/days/months:
- *programmers* give minute-by-minute feedback on the state of the system,
- *customers* are given immediate feedback on the quality of their stories,
- "*person who tracks progress*" gives feedback if the project is running within a predicted time-scale.

Courage: XP encourages the taking of drastic and unexpected measures/actions such as throwing code away or braking tests that have already been running and fixing the flaw.

The XP values are distilled into concrete *principles*, which determine XP *practices*. The XP basic *principles* are

Rapid feedback: any learning on how best to design, implement and test the system must be fed back in seconds/minutes, rather than months/years.
Assume simplicity: treat every problem as though it could be solved the simplest way possible.
Incremental change: solve any problem with a series of small changes that make a difference.
Embracing change: preserve most options while solving the most pressing problem.
Quality work: you should enjoy your work: this is how you produce good software.

There are some other less central *principles* that can be found in (Beck, 1999) and all of them help us decide what to do in a specific situation. The four XP *values* and those derived *principles* are a basis for building a discipline of software development *practices*. However, before *practices* are identified, XP gives a list of *activities*, which are derived from the XP *principles*. The four basic XP *activities* are:

Coding: is XP basic activity: whether you draw diagrams that generate code or you type at the browser, you are coding. Source codes should be used for everything: to communicate solutions, describe algorithms, express tests ...etc.

Testing: is important: automated tests often test functionality, and non-functional requirements can not be avoided (e.g. adherence of code to standards) are also important. Unit tests are written by the programmers, in order to prove that programs work the way they are expected to. Functional tests are written by customers to convince themselves that a system as a whole works as it is expected to.

Listening: XP develops rules that encourage structured communication and discourages communication that does not help: it is not simply enough to say "everybody should listen to each other".

Design: is part of the daily business of all programmers in XP in the midst of their coding. It is based on the concept "that a change of one part of the system does not always require a change in another part of the system!" In good design every piece of logic in the system has only one home, it puts logic near the data it operates on and allows the extension of the system with changes in only one place.

The set of guiding values and principles described above guide us as we choose the strategy for each of these activities. The main XP job is to structure the activities in the light of the long list of XP *practices* briefly listed below (no practice stands well on its own. It requires other practices to keep it in balance):

Planning game: determines the scope and timing of the next release.
Small releases: puts a simple system into production quickly.
Metaphor: guides all development through a simple shared story of the overall system.
Simple design: the system should be designed as simply as possible at any moment.
Testing: is constantly undertaken for development to continue.
Refactoring: restructures the system without changing its behaviour.
Pair programming: all production code is written by two programmers at one machine.
Collective ownership: anyone can change any code anywhere at any time.
Continuous integration: integrate and build the system many times a day.
40-hour week: work no more than 40 hours a week as a rule.
On-site customer: must be available full-time.
Coding standards: emphasises communication throughout the code.

2.3. XP Strategies and Lifecycle

The implementation of XP should be based on the *strategies* of the XP project management where almost all XP *values, principles* and *practices* are employed in order to deal with a particular strategy. For example, the *Management strategy* that emerges from evaluation/use of *Accepted responsibility, Quality work, Incremental change, Local adoption, Travel light and Honest measurement* principles

will guide us towards decentralised decision making and leave managers with *Planning game* practice, using metrics as the basic XP management tool. The *Development strategy* is a radically transformed format of the traditional view of the development process. Its motto is that in XP all activities are centred around programming, i.e. "everything you do in XP looks like you are doing programming". You start with *Iteration planning* (from the *Planning strategy*) and include practices such as *Continuos integration* of written code, *Collective* (code) *ownership* and *Pair programming*, which ties the development process together. All four XP values work towards the XP *Design strategy*, which requires the simplest design that runs the current test suite. You also use principles such as *Small initial investment*, *Assume simplicity*, *Incremental change* and *Travel light*, which will result in gradual design change, no extra design results, working on the simplest design we can imagine and removing all unnecessary functionality

XP strategies are put into practice through a lifecycle of an "ideal" XP project. It consists of a short initial development phase followed by a long-term simultaneous production support and refinement.

Exploration: prepare production, practice writing user stories, estimate and experiment with technology and programming tasks.
Planning: run the *planning game* practice; agree the smallest set of stories to be completed.
Iterations to First Release: produce a set of functional test cases that should run at the end of iterations.
Productionizing: you need one-week iterations, certify that the software is ready for production; implement a new test and tune the system.
Maintenance: simultaneously produce new functionality and keep existing system running, refactor if necessary or migrate to a new technology; experiment with new architectural ideas.
Death: XP project ceases to exist; new functionality do not have to be added or can not be delivered.

3. XP overview

XP approach to the software development process is polarised around the following issues:

- Application code in certain environments is so easy to change that we could abandon careful up-front analysis and design activities.
- Requirements that are highly changeable should be split into the simplest test cases within a task of a minimum complexity, which still has some business sense.
- The XP work is centred upon simple unit tests that are written before the production code is written and run every time the new production code changes. Unit tests are done at the granularity of the single method, i.e. at the smallest granularity possible, ensuring very small and fast iterative cycles with immediate feedback. They also contribute towards overall functionality tests.
- Code is written in programming-pairs: not a single line of the code can be written without the attention of two programmers.
- The code is written for particular written unit tests to be verified **now** and without anticipating any potential changes. You work at any moment on implementing what you know is needed **now** and not in the short-term future.
- The code can not be duplicated: if the same code is found at two different places, they must be combined. Refactoring will ensure that every method does not try to do more than it should.

The issues above are scattered across various XP principles and practices and are suggested to be employed within an 'ideal' XP project lifecycle. In spite of suggesting some *Exploration* and *Planning* phases, the first release of running tests happens very early within the XP project life cycle, up-front in *Productionizing*, which still allows *Refactoring* and software to evolve but at a slower pace. The emphasis is on the *Maintenance* phase where you exercise everything: from keeping your current release running – however small it is – and incorporating new functionality by using the same tight method of fast iterations of the smallest possible unit testing and coding. The next three sections will assess some of the issues mentioned above

3.1. XP Early Production Coding vs. Up-front Conceptual Modelling

XP encourages code production *before* having an overall investigation of all major system's use cases and domain models. They are all needed in order to generate conceptual design and overall implementation solution. We have learned throughout the last 4 decades that the best SE practices are based on the up-front analysis and clearly specified conceptual models prior to any code production. XP practices directly violate this principle i.e. it almost looks like XP denies all the values of high level analysis and design activities which were prevalent in structured systems development lifecycles as defined in (DeMarco, 1979) and some OO methodologies (Jacobson et al., 1999), (Rumbaugh et al., 1991). However, XP shows some overlapping with the latest OO approaches where the initial analysis and investigation are not limited exclusively to up-front of all activities. In RUP (Jacobson et al., 1999) all activities are happening in parallel throughout short iterative cycles, but an overall understanding of key use cases and domain classes is advocated, before production coding takes place. XP allows us to write simple unit tests and convert them into a production code without the complete picture of the overall business domain problem. This could have the following implications:

(a) It could undermine the usefulness of unit testing before you understand the whole domain problem. It might apply more refactoring in order to remove initial errors within first/early releases.
(b) It ignores software architectural issues, which can not solely be enforced by eliminating duplicated code or replacing it with the *Metaphor* practice as suggested in (Beck, 1999). Software architecture as in (Shaw and Garlan, 1996) exhibits more than XP practices offer. It should exercise for example how interchangeable software components comprise flexible software systems where solutions might be given throughout distributed objects in CORBA environments (Mowbray and Ruh, 1998).

Problem (b) is difficult to remedy without more precise explanations from XP propagators on **what** *Metaphor* practice exactly is and **how** by asking for a metaphor we get an architecture that is easy to communicate and collaborate. The goal of the *Metaphor* practice is to give everyone a coherent story within which to work. However, this story can easily be refactored through the *Continuous integration* practice of newly coded unit tests within the running release. The question is: do we continuously integrate by asking for a metaphor first, or does the integration itself dictate a single overarching metaphor? The only guidelines that we have at the moment are that each XP practice requires another XP practice to keep it in balance. We probably mix and match them according to our experience and needs. Furthermore, XP claims that the final code found in the production is so precisely refactored and readable that it could serve as a software architecture model. However, it is more likely that XP code with extremely small methods using laboriously readable messages and variable names could rather contribute towards a design documentation, i.e. these two might be very similar in XP projects and fall short of our expectations of software architecture. We could assume that software architecture in XP projects could be traced through XP code documentation, coder's individual knowledge of a particular code release and examples of hands-on mentoring during the *Maintenance* phase (or any other) using a clear *Metaphor* and culture of story-telling. Hence, software architecture knowledge of XP projects is rather communicated and socialised than explicitly documented.

Problem (a) resembles very exploited rapid prototyping activities, which have been used within analysis stages of traditional software development in order to prove that a particular assumption or even a particular design works. Regardless of the criticism that the rapid prototyping moves the development/coding into an analysis stage of the system life cycle, it could successfully work with incremental and iterative steps that interchange analysis and overall architectural issues. Whenever new requirements trigger new analysis, changes might be kept within an increment which dictates the revision of the software architecture based upon the revision of major use cases, i.e. functionality of the overall system. If we change the prototyped code itself at this level and update it to accommodate new functionality, we will come very close to the XP *Productionizing* practice, assuming that the simplest business domain has been prototyped initially and has escalated as new requirements arise. Furthermore, the time we spend on analysing high level design and prototyping, XP would have

yielded a partially completed system probably at least equivalent to the prototype. However, rapid prototyping has always been associated as a successful tool for refining user's requirements under the umbrella of system's overall functionality, which XP unit tests simply are not.

3.2. Communications of XP Activities and Code Generation

Communications at any level/phase of the XP project seems to be the backbone of all XP practices. You are required to program in pairs with your colleague in order to approach unit tests and code production more dynamically and share someone else's expertise/knowledge. The code is collectively owned, no strict/individual ownership exists, everyone takes responsibilities for the whole system, and everyone can change any code anywhere. An on-site customer is *a must* and a valuable recipient of numerous rapid feedbacks in unit-testing and task-estimations. The XP *Communication* value directly influences *Courage/aggressiveness* value where XP proved to favour both by placing the code production into the heart of any XP project. All these issues above could raise the following questions:

(a) Does this dissemination of knowledge/activities break down the barrier between analyst/designer and coder/programmer?
(b) Do highly communicative XP environments move the management control of the overall software production towards the programmer's level?

The answer to both of these questions can not be given separately: each one influences the other. Question (b) should not represent any problem if the XP practices operate under the stable umbrella of software architectural issues as discussed in the previous section. If this were the case, any decision about overall system functionality taken by a programmer could have been monitored and approved as being within the boundaries of agreed system's functionality at a higher analysis/design level. In other words, we expect that all decisions on changes of system's functionality should be made solely by the analyst/designer and not programmers/coders. However, the XP practices do exactly the opposite: they encourage the programmer's involvement at any stage of the software development process: from requirements capture to code production, hence answers to both questions (a) and (b) above are positive. We used to view this approach as a 'recipe for disaster'. It was claimed that placing the analyst/designer at the front end of software development activities, and isolating them from programmers, is the only way of deriving acceptable software solutions. They are controlled by captured requirements and system's functionality and isolated at the logical level, whose design should not be influenced by any of implementation decisions, i.e. coding solutions.

It seems that such a big division between XP and our old development practices regarding (a) and (b) above is remedied in XP through : (i) highly communicative activities and the culture of pair programming and story-telling at any stage of the XP project, (ii) constant, tight and rapid feedback from any of the XP activities to anyone involved, (iii) customer's involvement into unit tests and tasks estimations which are happening constantly throughout the life of the XP project.

XP claims that analysis and design are part of every programmer's every day work, whose experience, skills and programming decisions directly influence a system's functionality through careful structure of the *Planning game* principle and under a system's Metaphor as a system's model. The XP highly communicative activities with rapid feedback definitively give assurances for successful implementation of changeable system's requirements and question (a) does not represent any threat to XP projects. However, question (b) could generate more obstacles and as long as the XP principles do not address the issue of software architecture more explicitly, this will remain one of the XP project's biggest weaknesses.

3.3. XP and RUP

XP came to light almost immediately after the SE community embraced the UML as a standard in the modelling of software intensive systems and when RUP was gaining popularity as a process, which accommodates all UML modelling constructs. This has triggered fierce discussions among OO

technology practitioners in order to address the applicability of XP and RUP through their similarities and contradictions.

RUP is a generic process framework that can be specialized for any software system including different application domains, types of organisations and project sizes. Its component-based approach to software development is captured in use case driven, architecture centric and iterative/incremental processes. RUP repeats a series of cycles, which contains four phases: inception, collaboration, construction and transition. All phases comprise as many iterations as necessary to deliver various releases. Each workflow: requirements, analysis, design, implementation and testing, is carried out in almost every phase, hence a typical iteration goes through almost all workflow. RUP uses the UML and summarises what we now like to see as the basic success of valuable expertise and skills needed for the successful delivery of software solutions.

The strongest resemblance with XP could be in a highly iterative and incremental RUP approach adopted when delivering releases for any of the phases of its lifecycle. Iterations in RUP refer to steps in workflow and increments to growth of the final product. Each use case is transformed through successive iterations into executable code. Whatever the similarity between XP and RUP incremental approach is, they are undertaken under different circumstances and for slightly diverse purposes. The issue of software architecture again becomes an obstacle in clearly judging how successfully an XP can deliver a solution, as already discussed in the previous two sections. Furthermore, we could extend the problem towards the questions:

(a) Is a "good software architecture" important in the process of delivering software solutions?
(b) Do we need strictly controlled iterative processes under the umbrella of a "stable architecture driven solution"?

Looking at XP practices one sees that an "architecture" is simply a foldout of a code-centric approach, i.e. it is something which results from managing code dependencies and extensive code refactoring. In RUP an "architecture" means that all aspects of the software development depend on the "use case view", i.e. the software development is driven by the architecture which is an emergent property of the writing of the deliverable software. These two approaches can not be juxtaposed.

The answer to question (a) above should then probably be negative if we think about the quality of software production in small projects where XP is highly applicable and possibly more efficient than any other approach. These are cases where we do not think about the architectural issues very much and could conform to and accept a very wise wording of *Planning game* practice under a *Metaphor* of a system' model, instead of using the word "architecture", as expected. In RUP "architecture" is also decided by the first few iterations during the inception and elaborating phases, which both involve writing a fair amount of code. We could then say that in RUP coding does drive the "architecture" to a certain extent as in XP. However, this is strictly controlled through iterations under the umbrella of common functionality described in major use cases, which should give us an assurance that the code delivered stays within earlier agreed decisions on system's functionality that guide the development.

Consequently, the answer to question (b) above is probably positive if we talk about complex projects where we could not rely on guidance from *user stories* and *Planning game* practices as advised in XP. This view advocates that programming should extensively be guided by logical and technical system architectures agreed earlier, before delivering any implementation solutions. In other words, the architecture should result from the business domain model (e.g. use cases), it should be defined and maintained by the project and not extracted from the perfectly refactored code.

XP practices and RUP framework could be merged by brave developers into a combined process of RUP phases, where each iteration is conducted under some XP practices. This might ensure that the code delivered is controlled by the architecture centric and use case view approach, but each iteration exploits the highly communicative XP practice of pair programming, rapid feedback, shared code ownership, and the culture of story telling to its maximum. This remains to be seen.

4. Conclusions

This paper has attempted to raise serious concerns regarding XP's courageous approach to the software development process and its impact on software architecture solutions. However, the aim of the paper has not been to dispute any values that XP might contribute towards SE practices due to their relative immaturity, insufficient number of sources where one could learn how to use XP, and few examples where XP is put into practice. Consequently, this paper does not reach any conclusions that explicitly judge XP or state what the future of XP is. However, it is worthwhile mentioning that XP represents the most *avant garde* and the most tempting approach to software development for decades, which primarily draws our attention towards its denial of all sound SE principles that we adopted throughout our last 4 decades of software development experiences.

References

Beck K. (1999) Extreme Programming Explained, Embrace Change, Addison Wesley

Beck K., et all (1998), Crysler Goes to Extreme, Case Study, http://www.DistributedComputing.com

Booch G, J. Rumbaugh, and I. Jacobson and (1998) "The Unified Modelling Language User Guide", Addison Wesley

De Marco, T. (1979) "Structured Analysis and System Specification", Yourdon Press New York, US

Jacobson, I., G. Booch, and J. Rumbaugh, (1999), "The Unified Software Development Process", Addison Wesley

Juric, R. (1998) The UML Rules, in ACM Software Engineering Notes (SEN), Volume 23, Number 1, pp. 92-97

Mowbray T.J. and W. A. Ruh (1998) "Inside CORBA Distributed Object Standards and Applications", Addison Wesley.

OMG (1999) UML version 1.3, http://www.omg.org/uml/

OTUG discussion forum (2000), http://www.rational.com

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen (1991) "Object Oriented Modelling and Design", Prentice Hall International

Shaw M, and D. Garlan (1996) "Software Architecture, Perspectives on an Emerging Discipline", Prentice Hall

XP (2000), http://www.XProgramming.com, http://www.extremeprograming.org/