

TRIZ AND EXTREME PROGRAMMING

John W. Stamey, Jr.
Department of Computer Science
Coastal Carolina University
Conway, SC
jwstamey@coastal.edu

ABSTRACT

The problem-solving foundations of Extreme Programming have been found to mirror a number of inventive problem solving principles found in TRIZ. This paper is a first step toward understanding powerful software development methodology as a technique of inventive problem solving.

INTRODUCTION

This paper is a first step in comparing the software development/management methodology Extreme Programming (XP) to Genrich Altshuller's Theory of Inventive Problem Solving (TRIZ). Both methodologies are used to generate a solution to a problem that has never been solved before. Use of the XP methodology is equivalent to saying that some type of new and original software system is being developed. Use of TRIZ is equivalent to saying there is some problem to be solved and the solution is being generated through some sort of inventive problem solving.

In the next section of this paper, we outline the basics of XP, and describe how it differs from the more traditional waterfall model of software development. In the third section, we provide some background about Altshuller and TRIZ. The fourth section provides a comparison between the methods found in XP and a subset of the inventive principles found in TRIZ. The fifth section points the direction of further research in this topic.

EXTREME PROGRAMMING

The waterfall model of software development was first proposed by Royce. (1970) Steps included in this software development methodology included

- Requirements specification
- Design
- Coding
- Integration
- Testing and debugging
- Installation
- Maintenance

Each phase is completed before the next one begins. In a sense, the deliverables of one phase "fall" into the next phase much like the waterfall metaphor used in the name of the methodology. A main objection to the Waterfall Model is that each phase is essentially "set in stone" once it is completed, change management not included as part of the original model.

The waterfall model is frequently called a "heavyweight method," as it provides for a heavily regulated, regimented, and frequently micro-managed strategy for software development. The steps of the waterfall model are frequently as bureaucratic and slow, often contradicting the reality of software development. (Wikipedia, n.d.)

XP was developed by Kent Beck, who understood that any team's software development methodology needs to be customized to the team and their circumstances. (Beck, 2005) There are twelve key practices of XP are almost diametrically opposed to the waterfall model. The twelve XP practices are:

- *The Planning Process:* The XP planning process allows the XP "customer" to define the business value of desired features in units of analysis called User Stories.
- *Small Releases:* XP teams put a simple system into production early, and update it frequently on a very short cycle.
- *Metaphors:* XP teams use a common system of names and descriptions to help guide development and communication.
- *Simple Design:* All programs are designed to meet, but never exceed, the requirements. Code is not written for functionality that is not set forth in the requirements.
- *Testing:* Programmers actually write unit tests before they write the code to be tested (white box testing). Customers direct acceptance tests (black box testing) to make certain that the features needed have been provided.
- *Refactoring:* Duplicated and bad code is removed and/or consolidated as soon as it occurs.
- *Pair Programming:* XP programmers write all code in pairs, with one programmer "drives" while the other "codes." Roles are frequently switched to keep the programmers fresh.
- *Collective Code Ownership:* As all of the programmers own (control) the code in the project, any programmer can change another programmer's code when necessary. This practice definitely speeds up the programming process.
- *Continuous Integration:* XP teams integrate their code into the main system several times per day. This practice helps eliminate integration problems.
- *40-hour Week:* As tired programmers make more mistakes, XP teams are not allowed to work more than forty hours per week. This practice is intended to keep the programmers fresh, healthy, and effective.
- *On-site Customer.* An XP project is steered by a dedicated individual who is empowered to determine requirements, set priorities, and answer questions as the programmers have them. The effect of being there is that communication improves, with less hard-copy documentation - often one of the most expensive parts of a software project.
- *Coding Standard:* For teams to work effectively in pairs, as well as to share ownership (joint responsibility) of the code base, a strict set of coding rules are put in place. All programmers must write the code in the same style.

TRIZ: THE THEORY OF INVENTIVE PROBLEM SOLVING

Examining over 200,000 patents through the years, Genrich Altshuller concluded there was a process by which many inventions were developed. In reviewing the patent applications, he identified contradictions that had existed before the inventions were created. He was then able to identify a set of inventive principles that were used to remove these contradictions.

For Altshuller, a problem to be solved was equivalent to a contradiction between two opposing factors. Given attempted solutions to a problem, one factor would improve while one factor would worsen. As an example, we consider the problem of making a table stronger. If the attempted solution is to add more material to the top of the table, the additional material makes the table stronger (a potential improvement) while it also make the table heavier (a potential worsening).

Years of work resulted in the method of TRIZ, the Russian acronym for Inventive Problem Solving. In total, Altshuller found 39 different contradictions, when properly paired, could be solved with 40 inventive problem-solving principles.(Altshuller, 1998) The forty problem-solving principles have been found to produce many innovative solutions for companies such as Intel, Michelin, HP and Samsung.

TRIZ AND XP

When the twelve principles of XP are compared with the forty problem-solving principles for TRIZ, some clear relationships are found.

- User Stories (The Planning Process) may be described by TRIZ Principle 1, *Segmentation*. *Segmentation* is a process of dividing a system into parts in order to isolate or integrate beneficial properties. The segments are then reassembled, or integrated, to perform the required functionality. Each user stories encapsulates an important requirement. Assembled together, the user stories represent the requirements for a complete software system.
- Small Releases may be described by TRIZ Principle 30, *Flexible Membranes or Thin Films*. Traditional construction is replaced with constructions of thin films or flexible/pliable membranes. The development of programming systems with XP is the continual accumulation of many small releases.
- Metaphors may be described by TRIZ Principle 2, *Extraction*. The important properties of a system may be separated, either physically or virtually a system. Metaphors provide a virtual abstraction of system concepts from a point of reference that is, hopefully, familiar and meaningful to both the development team and the users.
- Simple Design may be described by TRIZ Principle 7, *Nesting*. Nesting is the quality of being made to fit closely, fit together, and/or fit inside each other. As simple design does not allow for additional (unnecessary) functionality, existing releases may be immediately extended, with no worry of nuisance side-effects from unnecessary code.
- Pre-written unit tests may be described by TRIZ Principle 9, *Prior Counteraction*. This principle suggests that we plan, in advance, to reduce or eliminate things that

- might go wrong. Pre-written tests help guide the development process, providing tight coupling between what we intend to test and what code needs to be written in order to accomplish that test.
- Refactoring may be described by TRIZ Principle 5, *Consolidation*. New functionality, that did not previously exist, can be created by combining functions. Consolidation (in both physical systems and in software) can be done in space, time or contiguous operations. Refactoring is a general collection of duplicate code, as well as code that needs to be optimized.
 - Pair Programming may be described by two TRIZ principles: Principle 15, *Dynamicity*, which prescribes that a system adaptive, flexible, or changeable; Principle 17, prescribes translation into a higher dimension. Pair programming is a translation of programming from one person to a community of two. Roles are frequently reversed; hence both programmers must be adaptive, flexible and changeable.
 - Collective Ownership may be described by TRIZ principle 25, *Self Service*. This principle state that an object must service itself, carrying out supplementary and repair operations. The idea of collective ownership and use of code is exactly the idea of self-service described in this principle.
 - Continuous Integration may be describe by TRIZ principle 20, *Continuity of Useful Action*. A continuous flow of action is used to increase efficiency. In system development with XP, small releases are continually integrated into the existing code base. This continuous flow of new code into the existing system will help identify problems the moment they occur.
 - The 40-hour Week may be described as TRIZ principle 16, *Partial Action*. Excessive actions can be harmful. This identifies the ideal of limiting programmers to a 40 hour week. Work (programming) that is full of mistakes from a tired programmer is certainly worse than no work at all.
 - The on-site Customer may be described as TRIZ principle 23, *Feedback*. The principle of feedback is based on system output that is returned back to the system (as input) for the purpose of quality control. An on-site customer provides invaluable communication between the project owner and the project developers. (Beck, 2005, p. 20)
 - Coding Standards may be described as TRIZ principle 33, *Homogeneity*. The interaction of two systems is best when they use the same material or information. The use of similar coding and documentation styles provides a common language through which the programming team may communicate. Fred Brooks, in the *Mythical Man-Month* (Brooks, 1995) underscores the importance of programming team communication in his essay “Why the Tower of Babel Fell.”

CONCLUSIONS

This paper is a first step toward understanding XP, a new but powerful software development methodology, as an instance of inventive problem solving. The next step in this comparative analysis is the examination of the process of XP. We will take a diagram of the process, found at www.ExtremeProgramming.org, and replace the steps with the name of the inventive principle to which they are associated. The directed arrows in and out of the various inventive processes will provide some indication as to the

contradictions (in the software development process) that are solved by the inventive principles.

REFERENCES

Beck, K. & Andres C. (2005) *Extreme Programming Explained, 2nd Edition*. Upper Saddle River, NJ: Addison-Wesley.

Brooks, F. (1995) *The Mythical Man-Month: Essays in Software Engineering, 20th Anniversary Edition*. Upper Saddle River, NJ: Addison-Wesley.

Altshuller, G. (1998) *40 Principles Extended Edition: TRIZ Keys to Technical Innovation*. Worcester, MA: Technical Innovation Center.

W. Royce (1970) "Managing the Development of Large Software Systems," *Proc. Westcon*, IEEE CS Press, pp. 328-339.

Wikipedia, (n.d) Agile Software Development. Retrieved May 4, 2006 from http://en.wikipedia.org/wiki/Agile_software_development/.

Jeffries, Ron (n.d.) "What is Extreme Programming?" Retrieved May 4, 2006 from http://www.xprogramming.com/what_is_xp.htm /.