Engineering of Unstable Requirements Using Agile Methods

James E. Tomayko
Carnegie Mellon University
jet@cs.cmu.edu

Abstract

Some think that the early absence of requirements specified later are defects. We consider them merely "undiscovered." Finding these new requirements could nearly be done adequately using early prototypes. Lately, Agile Methods are used to refine requirements throughout development.

In a recent cartoon, a manager is seen walking through an area filled with programmers. He says, "You guys start coding and I'll go see what they want." This pretty well characterizes the situation described in the Call for Papers of this workshop. There seems to be a common belief that agile development methods reinforce this unfortunate attitude toward requirements. Conversely, my research has shown that an "agile attitude" toward requirements is a very effective means of acquiring them [6].

I have had 12 teams of from four to six engineers participate in eXtreme Programming (XP)[1, 3] experiments in the past year. The experiments were primarily aimed at defect reduction and not requirements acquisition [9], but observing how these teams go about their work can make several insights to the requirements acquisition process, such as were made in [10]. These observations are further described in this paper. This workshop will hopefully suggest some ideas for formal experiments that deal with the requirements elicitation process in agile methods.

The (Old) Method of Up-Front Requirements Elicitation

Current requirements elicitation methods reinforce improper beliefs. Many projects try to follow the waterfall software life cycle and other obsolete software development life cycle models. The requirements are gathered first in one big effort [4]. Frequently, these requirements turn out to be rife with omissions and misconceptions. Correcting them costs time and money. The result has been a movement toward more iterative models [8].

At first, these took the form of prototypes, both to find missing requirements and to examine the feasibility of solutions to others [5]. The history of iterative requirements gathering has itself gone through several cycles, of which agile methods are the latest. The trouble is that with each new iterative method the requirements elicitation process appears to become less defined.

I feel that the attitude towards requirements gathered early in the process is incorrect in that ones missed at that stage are considered defects when added later. My observations and the entire point of iterative processes is that requirements are seldom omitted, they are just unknown. There is simply no way that the requirements of even well understood problems could be known. Therefore, why even try? Requirements are elicited by agile methods in a more practical way.

Requirements Elicitation in Agile Methods

The "user stories" or the like are just the beginning points of both the requirements gathering and development processes in agile methods. Early requirements are simply a place to start. It is expected to add more requirements as more is known about the product. Conversely, the method of trying to gather all the requirements before starting development is almost certainly rife with errors and surely takes too long. In such a development process, the client is prompted by the product to "remember" some things and by the marketplace to want others changed. "I'll know it when I see it (IKIWISI)" [2] has become a well known requirements method. In effect, the early version of the product becomes a prototype. Agile

methods are designed to appeal to clients that insist on IKIWISI.

Prototyping in agile methods is even more rapid and produces smaller amounts of code than traditional prototypes [1, 9]. Developers that use prototype-based life cycle models are familiar with the case of the client falling in love with the prototype so that they want to take it away as the product. Avoiding a situation where bad code and poor documentation characterizes the product makes the developer produce a less robust prototype. As a result, they are not as useful. Agile Methods, which have the concept of a "spike"—a rapid development of a prototype that answers a single question about requirements content—avoid this problem of showing the client too much.

In this way, the client is kept from the responsibility of "getting the requirements right." There are no wrong requirements. There are simply some waiting to be discovered.

Difficulties Caused by Agile Methods of Gathering Requirements

This attitude toward requirements makes estimation and software architecture development more difficult, and verification easier, than traditional methods. Without knowing the final form of the product, or marketplace demands, estimation is going to be impossible to get correct [7]. It is little comfort that requirements omissions and changes caused by reacting to the competition make most estimates incorrect right now. Agile methods are likely to be right about the costs involved in the current cycle, but estimating is poorly understood for the unknown requirements of the next cycle.

One thing that can be done is to fund the project one cycle at a time, which is equivalent to funding an entire project using an older development method now. However, there will be time when knowing the total cost is necessary, as in contract work. In these cases, the requirements are expressed by the customer as well as they can, and the estimate adjusted by the probable cost of later changes. For instance, if a project is estimated at $1M, and prior projects of roughly those same characteristics have had the cost of "changed" requirements at around 20 per cent, then the estimate is $1.2M. Of course, as with all estimations, this can not be used without considerable historical data.

As for the architecture, that chosen by the team during the early cycles may become just plain wrong, as later requirements become known. Rework of the architecture matches the refactoring principle of eXtreme Programming. Most of my XP teams embraced refactoring, claiming that they would do it anyway, even if the requirements were stable. One student identified refactoring as rework, with it attendant negative properties, notably increased cost. Either way, significant refactoring is to be expected in an atmosphere where requirements are relatively unknown. Confidence in the requirements translates to confidence in the architecture.

Advantages of Agile Methods for Correctness

Aside from refactoring and effective prototyping, agile methods have other advantages for a situaion in which requirements are unstable. Reliance on test-first programming, a principle of XP, means early detection of most minor errors, more certain detection of defects at integration, and early thinking-through of tests for a Graphical User Interface [10]. This is an advantage to any system. For this reason alone, requirements engineering is advanced by the developer knowing right away if a requirement can be tested.

[1] Kent Beck, *eXtreme Programming*, Addison-Wesley, Boston, 2000.

[2] Barry Boehm, "Requirements that Handle IKIWISI, COTS, and Rapid Change," *Computer*, IEEE, July 2000, pp. 99-102.

[3] Alistair Cockburn, *Agile Software Development*, Addison-Wesley, Boston, 2002.

[4] Alan Davis, *Software Requirements*, Prentice-Hall, Englewood Cliffs, N. J., 1990.

[5] D. Leffingwell and D. Widrig, *Managing Software Requirements: A Unified Approach*, Addison-Wesley, Boston, 2000.

[6] John Smith, *A Comparison of RUP and XP*, Rational Software White Paper, 2001.

[7] Richard Thayer and Merlin Dorfman, eds., *Software Requirements Engineering*, IEEE Computer Society Press, Los Alamitos, Ca.,1997.

[8] James E. Tomayko, "An Historian's View of Software Engineering," in the *Proceedings of the IEEE Conference on Software Engineering Education and Training*, IEEE Computer Society Press, Los Alamitos, Ca.,, 2000.

[9] James E. Tomayko "A Comparison of Pair Programming to Inspections for Software Defect Reduction" in *Computer Science Education*, forthcoming, 2002.

[10] James E. Tomayko, "Using Extreme Programming to Develop Software Requirements," *Soft-Ware 2002*. Sprnger-Verlag, 2002, pp. 315-331.