Agile processes for the maintenance cycle

A smarter work cycle for a Smarter Planet

Moacyr Cardoso de Mello Filho

March 06, 2012

Although everyone wants agile processes, most people are still discussing agile without really pursuing it, some people have already set out on the path that leads to a smarter work cycle. This will transform the way IT supports your business and will improve your performance in software development.

This article was originally published in Portuguese on the developerWorks Brazil site.

A recurring topic when discussing software projects is the issue of system maintenance, which is always seen as the poor cousin of development methodologies or processes. Despite being an important topic, given that it represents most of the work volume developed at organizations today, maintenance is often treated as a particular case of the general development problem. In organizations where software development is not a business objective, an estimated 75% to 85% of the amount of software development is system maintenance. The same techniques and methods applied to new projects can certainly be applied to maintenance as well, but there are specific aspects that can be better explored if we understand the nature of maintenance. This is the field in which the adoption of agile methods has proven the most rewarding.

Despite the huge diversity hidden within the term "maintenance," which corresponds to the variety of applications, maintenance types are usually classified as perfective, adaptive, or corrective:

Perfective
  New requirements and functions

Adaptive
  Changes to comply with laws or for business reasons

Corrective
  Elimination of an error, defect. or poor performance

The debate about classification, frequency, and dimension or size is endless. This debate justifies specific solutions and customizations, but, on principle, adaptive or corrective maintenance alone do not lead to large amounts of work. Consequently, they generate less change in code and...
design. This observation is interesting because, if the software architecture already exists, we are maintaining it, so the amount of work involved in the requirements is minor. A simpler process that doesn't necessarily abandon the design yet removes the burden of its responsibility would be both possible and desirable, because it would speed things up and make maintenance easier.

On the other hand, perfective maintenance theoretically has the potential to worry us, because it can mean major changes in the systems in which it is applied, sometimes even altering the application's original purpose. Considering these scenarios, we could divide maintenance into two groups, adaptive-corrective and perfective, with each group requiring a different process approach. We foresee a more formal and, therefore, heavier approach for perfective maintenance, where attention is still paid to architecture and design alike.

So it seems that we need two processes. But a problem comes up when faced with reality: We are not usually able, in practice, to separate the effort into each maintenance category. When that perfective maintenance measure that has been awaiting the conclusion of emergency corrective maintenance for several weeks is finally ready to be implemented, a critical adaptive maintenance issue comes up because, for example, the Central Bank has changed a parameter in the Brazilian Payment System! This is how reality insists on combining situations. Thus the planner, looking for the most cost-effective option, must plan maintenance that involves, for instance, new requirements (which would be in the perfective line), legal or business changes (an adaptation line), and some correction. Batch maintenance is a fact of life, but maybe we can take advantage of it.

It is hard to describe the size of a generic software maintenance cycle like the one imagined here, but for argument's sake, let's simplify things arbitrarily. Let's imagine a typical maintenance that would be characterized as:

- Requested through one or more orders, usually called demands, where a text contains the intention of business personnel, but it is not a big project
- These demands pile up and can be handled in a batch
- The development team itself, or a designated analyst, evaluates the demand, classifies it, and expresses it in terms of impact on the application and on the daily work
- The team is small, two or three people at most
- The application architecture is well-established (the application already exists)
- Development is done internally, so there is no subcontracting
- Implementations are conducted with little formality

In this kind of maintenance, the main process driver is the client's or business team's priority. There are no engineering considerations like those involved in a system built from scratch. Over time, demands show up that reflect new needs, and the obvious challenge for the planner is to put together a package that is cost-effective for implementation and, at the same time, meets both content and time priorities.

**Wait queue, backlog**

The first smart element is the creation of a backlog (that wait queue for requested demands, which is seen with shame and guilt in IT). I don't mean a spontaneous backlog due to a slow pace
of addressing demands, but rather an intentional backlog that enables you to work the batch-planning element. The smaller the volume of work for a requirement -- meaning the more timely the maintenance -- the higher the costs of process setup, work, and rework. In the long run, if you don't establish some form of demand composition, you will be constantly losing money and wasting valuable resources. You will go back many times to change the code in a certain part of a program, thereby lowering maintenance efficiency and redoing the work. In the worst-case scenario, in which there is no package (batch) and each demand is implemented on an individual basis, the planner is not able to combine requests and boost resources in programming. This is a typical emergency fix situation. Living that way for a long time creates a perpetual emergency environment, in which the organization is used to always reacting hastily.

**Steady pace, cadence**

On the other hand, if the planning manager is able to assemble a package of requests to be handled jointly (the so-called batch), assigning priorities according to business needs and placing the packages in the queue for implementation, then it would be possible to create a process that would retain a key factor called cadence.

Let me explain it. For repetitive tasks, even intellectual ones, if the work demand is messy. the team's reaction will vary and its performance will be unpredictable. This situation can be remedied by establishing a steady pace, a work cadence that brings back the desire to implement software maintenance. Such a cadence must start with a fixed-schedule work period. For example, an iteration or sprint (in scrum parlance) must always be equal to 30 days to establish cadence. Therefore, cadence and backlog walk hand-in-hand. The sprint duration is ultimately a matter of learning and adapting.

In systems that have been live for a long time (and consequently, have also been under maintenance for a long time), the team already knows how to evaluate the impact of a change and they have probably already learned, depending on the organization's pace, when is the best time for releases. The implementation of new releases in a mature system follows its own pace in which the ideal timing depends on factors such as team efficiency, user capacity to absorb the new release, business demands, and so forth. Teams determine this timing empirically and adjust their resources and efforts to meet demands in a batch volume that fits that timing. Obviously, this learning process comes with commitments and agreements made by the planning manager with the user areas and the organization's administration.

I saw this happening in real life in a developer team that maintained an ATM (automated teller machine) product. At first, under pressure from marketing, the product manager established that releases would be delivered every two weeks. The system was mature enough, and the backlog could be translated into scheduled maintenance events. But problems with the availability of test labs, availability of machines, and the time for approval always prevented the team from meeting the deadline. After a few bumps, the iteration time was raised to 45 days, which proved more than enough to compensate for the environment issues. In the next step, with the team's approval, the release interval was set to 30 days, and software productivity and quality levels were retained. Is it a coincidence that they ended up with the recommended scrum iteration time?
Scrum

This process of self-organization and empirical discovery of release iteration time is very similar to the way that scrum treats software development projects. (Scrum, short for scrimmage, and sprint are terms from rugby). Scrum was born around 1986 from the study and ideas of Takeuchi and Nonaka about the creation of organizational knowledge through innovation and iteration. Later, in 1995, Ken Schwaber made a formal definition of scrum for management of software processes. This agile development process is based on observation and adaptation cycles, which ended up becoming the iteration and release cycle known as sprint and still used today.

A scrum project is made up of sprints, just like RUP (IBM Rational Unified Process) is made up of iterations. A sprint is a work cycle and, like RUP iterations, it delivers an executable software release which is external, meaning that it can be put in operation at the client's environment. Sprints follow each other without being divided into phases. Without the need to build an architecture, this model fits well into the maintenance problem. Requirements are usually in statement form, but they can also be in another form. User stories and even use cases are common among scrum requirements. When the backlogs have been analyzed and assigned priority, they make up the product backlog, which is the sum of all requirements arranged by priority according to the client's perspective. On the other hand, a sprint backlog is the set of requirements chosen to be implemented during a specific sprint.

Scrum of scrums

Development is accompanied by daily meetings for the purpose of feedback and planning, where the team members can check their progress, what's still left to do, and what is hindering work. These hindrances to work performance come from different sources and are a problem that the scrum master must handle. Otherwise, they are taken first to an intermediate management team and then to the team with the main officer. All management levels in the organization implementing scrum know and are engaged in some way in the implementation effort. But most importantly, a channel is open to the organization direction, which is crucial.

Scrum's true contribution to software development came from its implementation proposal. Because iterations are short and very focused, hindrances appear and must be dealt with. The implementation of the improvement process during the change period leads to the existence of a direct channel with the company direction. Management teams, including the main officer's team, execute shorter scrum iterations, focusing on problem solving. This leads to agility and commitment to the change.

Smart work

Obviously, many obstacles will be shown by the process itself. We usually do not appreciate the bearer of bad news. Ken Schwaber tells us not to "kill the messenger." Scrum is just a messenger, the one showing us the obstacles. Even when the adoption isn't perfect, it still is an achievement for the organization. But the biggest advantage is that the process is self-correcting. The initial implementation period should be 6 to 12 months, and the concentrated effort leads to a more agile and more modern organization -- in other words, a smarter organization in a Smarter Planet™.
Related topic

- Download a free trial version of software.

© Copyright IBM Corporation 2012
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)