Defect Management in an Agile Development Environment

Don Opperthauser
AgileTek

Agile development practices are sometimes thought of as an undisciplined approach to software development, lacking such things as effective defect management. However, agile development does not hinder the use of formal defect management process- es in any way. On the contrary, agile development does much to reduce the incidence of defects in the first place. This article will paint the picture of defect prevention and management within an agile development environment.

There are a number of methodologies and approaches to agile development. For the sake of this article, the discussion will center on how we at AgileTek handle defect management within the context of a software project that uses our Agile+ methodology[1]. There is sufficient overlap between the various agile methodologies that this discussion should have ample application to any of them

Software defects only exist if, at the end of the day, someone says, "This soft- ware is not accomplishing the purpose for which it was written with the accuracy, effi- ciency, and ease of use that was intended."

This article discusses defect manage- ment in two broad categories: require- ments defects and implementation defects. The term requirements is used in a broad definition to include all types of require- ments, functional specifications, and other means to define what the software is sup- posed to do and, from a functional per- spective, how it is supposed to do it. Implementation defects refer to defects in architecture, design, coding, installation, or any other aspect of the technical imple- mentation of a software development project.

Let me begin with a case in point. More than a decade ago, I and the other future AgileTek co-founders received a function- al specification from a large ($13 billion today) consumer products company. It was not an overly complex system, but the functional specification ran to more than 400 pages. The painstaking detail of the document was impressive; every detail of the user interface, validation rules, and exactly how everything was to work was all spelled out. We got the job.

While the software was intended for use by the field sales force, our customer was the information technology (IT) organization. We suggested that perhaps it would be wise for our development team to sit down with some of the intended users and review the specifications. We were told that the IT folks had already done that and, moreover, the effort had taken up more of the users' time than they

wanted to give; there was no need for any further review. All we needed to do was to build and test the software to spec – what we call *spec conversion* in our business.

What seemed like a straightforward task of turning the specifications into bits and bytes got complicated when we dis- covered that what it said on page 83 con- tradicted what it said on page 183 and so forth. Could we have possibly analyzed, absorbed, and understood those 400 pages

> *"There is no substitute for adequate client involvement. Clients must invest the right amount of time from the right people if they are going to get an effective result."*

well enough to catch such problems before we began? Of course not. More impor- tantly, do you think that anyone in the sales force really analyzed, absorbed, and under- stood those 400 pages even though they approved the specification? Most assured- ly not! Their eyes probably glazed over around page 20, and they had no choice but to approve a specification they neither had the time nor skill to understand.

Eventually the questions were all answered (by the IT folks, not the users), the system was installed in a test environ- ment, and a group of users came to town for user acceptance testing. At the end of the first day of explaining how the system worked with the users and letting them get some hands-on experience, my architect asked the user-group's supervisor what she thought of the software. "This isn't the software we need!" was her disheart- ening reply.

I relate this story to underline the importance of *building the right software.*

Requirements defects of any nature are the most disastrous and costly. How are requirements and requirements defects managed in Agile+, a software develop- ment methodology[1]? Several of the com- ponents of this methodology speak direct- ly to this issue.

Customer at the Center of the Project

One of the problems in the anecdote above was the fact that the people who really needed the system to do their work were isolated from the people who were building the system. Some development organizations try to keep the customer at arm's length. By building a lot of customer involvement into our projects, we ensure that we are getting adequate and frequent feedback to keep the project on target.

In Agile+, the customer is treated as a full-fledged member of the development team with access to all the information to which the rest of team is privy (e.g., defect logs, issue lists, etc.). Once on the team, constant effort is made to ensure that the customer is an integral part of that team. An Agile+ project is steered by a dedicat- ed individual (customer or customer proxy) who is empowered to determine requirements, set priorities, and answer programmers' questions as they arise.

This is one of the most critical issues in managing requirements defects. *There is no substitute for adequate client involvement.* Clients must invest the right amount of time from the right people if they are going to get an effective result.

Flexibility to Meet Client's Special Needs

If there is any conflict between the prod- ucts produced by our methodology and the customer's needs, the methodology is adapted to serve the customer. For exam- ple, Agile+ takes a minimal approach to documentation – only enough to ensure proper execution and maintenance.

However, in regulated environments such as pharmaceutical research, painstakingly detailed documentation is almost always a required byproduct of any related software development. In such a case, normal documentation procedures are modified to meet project requirements.

Business Process Analysis

A thorough understanding of the business objectives that the software must achieve is crucial to reaching desired results. Too often the development team does not get involved early enough in the process to define the software to be built. The discussions that take place during iteration planning sometimes are not enough to ensure an understanding of the underlying business process to be supported. By the way, I am using the term *business process* in the broadest sense, to include manufacturing processes, military processes, or any process that needs to be carried out to accomplish the goals of an organization.

A formal, facilitated business process analysis (BPA), *with the entire development team present*, should begin any development effort. It is usually not enough for some BPA professionals to work for days and weeks to produce a BPA document, hand it to the development team, and say, "Read this." The discussions and nuances that occur during the BPA sessions cannot all be put into words, and certainly the development team cannot gain the depth of understanding needed to design and build the right application without personal participation in the BPA.

There is a very important point behind all of this. Despite all of our processes and technologies, software development is a rather new industry compared to something like building houses, which we have been doing for thousands of years. There are too many variables in building software, too many nuances, and too many possible user actions and paths through the system. Time to gain a personal understanding is needed, and the BPA is the perfect vehicle.

User Stories and Story Actors

Expressing requirements in terms that everyone can understand goes a long way to ensure you are building the right software. Many approaches to requirements definition produce results that are all but incomprehensible to the customers who really understand what the software needs to be.

In Agile+ and many other approaches to agile development, requirements for the system are gathered through user stories (sometimes referred to as use cases) that are developed through customer interaction. A user story does not fully define a requirement; rather, it defines an underlying business need from which the requirements can be determined. Later during architecture development, these stories inform the scenarios that are used to help validate the architecture.

We have added to the concept of *stories* the concept of story *actors*. Actors are personifications of the various categories of users that the system will encounter. Thinking of the requirements in terms of actors brings the requirements to life, and unmasks nuances that would otherwise remain invisible to both the developers and the customer. It enables the requirements to be written in terms of how the system will be used versus desired func-

> *"The key tenet in all agile software development methods is iterative development and the unforgiving honesty of working code."*

tions. Finally, by associating who is doing what, it helps conceptualize and compartmentalize the functions.

This approach allows high-level requirements to be expressed in terms understandable to users who really know what the system needs to do and to executives who must approve them.

Iterative Development

Short iterations allow the customer to see completed functionality very early on so that feedback is not only meaningful, but also received in time to keep the project on track with respect to the final project goals.

The key tenet in all agile software development methods is iterative development and the unforgiving honesty of working code. The concept of iterative development has been around for a long time and is perhaps best known through the application of spiral development.

Our iterations are kept short, generally no more than three to five weeks. Iterations begin with an iteration planning session during which the customer and project team select the user stories to be implemented during the iteration. Once the user stories are selected, the iteration planning continues with consideration of such elements as screen designs, user workflow, data input/output, etc. This is then input to a period of design *(days of design)* wherein business analysts and developers work together to develop specifications and produce component designs. Tests for these designs are developed prior to writing the code; the code is then exercised using these tests.

At the end of each iteration, we deliver working code for the stories implemented and review it with the customer. This enables our customer and us to evolve our understanding, challenge assumptions, and make informed choices and decisions. Using the information gained during the iteration review, we are in a much better position to plan the next iteration.

A software development effort meeting its requirements is analogous to a projectile hitting its target. In effect, each iteration is an opportunity to provide the development project mid-course guidance. By keeping the time period between iterations to no more than five weeks, the feedback loops are kept short, thus providing frequent guidance and ensuring the project never gets far off-track. *Iterative development is perhaps the single most important vehicle for managing requirements defects.*

In addition to building the right software, i.e. effectively managing requirements and requirements defects, the more traditional concept of defect management, *building the software right*, must also be addressed. Implementation defects are a major source of project trouble in traditional methodologies where late-stage integration brings system modules together near the end of the project. This usually results in an unbelievable number of defects. The development team goes into near paralysis while they try to get their newly integrated, defect-ridden system repaired to the point where meaningful system testing can even begin.

One of my colleagues for many years liked to refer to "Larry's two-phase software development methodology – defect creation and defect removal!" Agile+ takes a two-pronged approach to implementation defects. Some of the practices help prevent defects from ever getting into the software, and others facilitate early detection and repair.

Let us examine the practices of Agile+ that impact implementation defects.

Identifying System Components and Interfaces

Clearly defined components and interfaces are key to quality code. Especially for complex systems, it is important to assure conceptual integrity in the final product. Also,

because complex systems can be large, it is important to enable the system to be developed in an environment of distributed ownership.

Architecting a system simply means identifying the constituent components of the system and defining the interrelationship(s) between them. The best architectures are isomorphic (one-to-one) mappings between problem and program space. This ensures that a system's underlying structure and components mirror the problem being solved. This means that for the program to change requires that the problem changes, and as a result, you are *change-proofing* your program. While there may be more efficient ways to solve a problem (e.g., creating one module to perform similar functions by invoking it in a context sensitive way), this efficiency will almost always come at the expense of time spent debugging and later modifying the program if one or more of the functions change.

However, it also means something more. By defining the relationships between the various components, you have gone most of the way toward establishing agreements for the interfaces. The power of interface agreements is that they serve as restrictive liberators. In other words, the individuals working on various system components are free to design the internals of those components without regard for potential untoward effects on the rest of the system – so long as the interface agreements are honored.

As you can see from this discussion, a rigorous approach to identifying components and adherence to well-defined interfaces severely limits the effect that defects can have, thus making it easier to localize and repair defects when they do occur.

Collective Ownership
The team approach leverages the entire team's thinking on critical problems and ensures that no one is working in a vacuum, possibly going off in the wrong direction. The pride of ownership diffuses through the entire team, creating a high degree of motivation to write good, defect-free code. Peer pressure is very effective if there is someone on the team who is creating more than his or her share of defects, thus creating problems for everybody.

Continuous Integration
Software development history is strewn with projects that failed at the critical juncture of integration – bringing all of the components together near the end of the project. Continuous integration uncovers integration issues early. In this manner, integration defects, if any, are introduced one at a time as small pieces are integrated into the system and therefore are resolved more easily.

Relentless Testing/Automated Contract and Regression Testing
Requiring developers to submit virtually defect-free code to start with ensures not only a high quality product, but also consistent quality throughout the project. Agile+ requires that software *contracts* be written and automated tests designed before coding begins[2]. Contracts define the pre-conditions, post-conditions, and class invariants for any function to be written, and the automated tests check for these.

"Too often defect management is so focused on defect repair and getting the software out the door that we fail to learn from what is happening."

Before a developer can check code into the configuration management system, he or she must have a build of the entire system, including new code on either his or her development computer or on a test system designated for that purpose. The developer must then run not only his or her newly written automated test for the new code, but also all automated tests that exist for the entire system. Only when all tests return defect-free results may the developer add his or her new code to the project. In this way, very few defects are introduced into a project build and the system under development is maintained in a relatively defect-free state at any given time.

Refactoring[3]
Designs and architectures are boldly changed when needed to maintain the correct architecture throughout the project. Development that proceeds without fully automated tests on the entire system as described above soon reaches the point where major changes in architecture become too risky. Developers then will use workarounds, *kludges*, and other poor programming practices to avoid doing what they should do – make the major changes necessary to make the system work the way it really should. Refactoring is what enables this architectural and design rework. It keeps the system clean and contributes greatly to minimizing defects and making it easier to identify and repair defects that do occur.

Pair Programming
"Two heads are better than one" (and sometimes cheaper, too). Putting two developers on very complex or high-risk tasks decreases the risk of poor results.

Coding Standards
The maintainability of code is directly affected by having good coding standards, not the least of which is guidelines for properly commenting code.

In addition to these best practices designed to prevent defects, you will of course need some system for tracking defects, defect repairs, certification of repair after retesting, documentation of items found in system testing that are not really defects but future enhancements, etc. These tracking systems may be more or less sophisticated depending on project complexity and client requirements.

In some regulated environments, such as pharmaceutical or Department of Defense environments, it may be necessary to track defects and repairs back to the original affected requirements. The goal of defect tracking in Agile+ is to have no more tracking than necessary to achieve project goals, legal or client requirements, and metrics desired to analyze effectiveness of the software development effort.

Defect management systems should track a number of basic things, including the following:
- An accurate description of the defect, including detailed steps for reproducing the defect and as much information as possible about the application environment at the time the defect was discovered.
- History of the defect, including who discovered it, who is assigned to repair it, when it was fixed, who is assigned to verify and certify the repair.
- Where the defect originated. In other words, why is this defect here? Is it a mistake in requirements, architecture, design, coding, or perhaps faulty tools such as a compiler defect, etc?

Too often defect management is so focused on defect repair and getting the software out the door that we fail to learn

from what is happening. Analysis should take place to determine why the defect occurred, not just where. Was it bad information, inadequate skills to do the job right, careless execution, or some other cause? Knowing why the defect occurred will help us continuously improve our processes and performance.

In conclusion, Agile+ provides a set of practices that focus on prevention of both requirements and implementation defects while facilitating the effective and efficient identification and repair of defects that do get into the project.◆

Notes
1. Agile+ is a further refinement of Code Science, which is described in "Odyssey and Other Code Science Success Stories," CrossTalk Oct. 2002: 19-21.
2. Bertrand Meyer introduced the idea of Design by Contract. For more on this, see his book Object-Oriented Software Construction. 2nd ed. Prentice Hall, 1997.
3. Martin Fowler defines refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." For more on refactoring, see his book Refactoring: Improving The Design of Existing Code. 1st ed. Addison-Wesley, 1999.

About the Author

Don Opperthauser's software development career has spanned responsibilities from head-down program-mer to senior executive responsible for multi-million dollar projects for Fortune 100 companies. He successfully managed a multi-million dollar software project for the third largest U.S. corporation. The project, which was critical to the client's product development, was completed in a record five-month period. Software developed under his leadership was key in providing the enabling software for the most profitable business unit of another Fortune 100 Company.

Phone: (847) 770-1637
Fax: (847) 813-4903
E-mail: dopperthauser@agiletek.com

WEB SITES

Software Certifications
www.softwarecertifications.com
The goal of Software Certifications is to offer an independent professional certification that carries weight in the information services marketplace, and is considered valuable to all of those professionals who seek and attain one of the certifications. The Quality Assurance Institute Professional Certification division sponsors and administers the software certification programs. Available certifications include the Certified Software Quality Analyst and a newly added Certified Software Project Manager.

Institute of Configuration Management
www.icmhq.com
The Institute of Configuration Management is known for its CMII process, which is an advanced version of configuration management (CM). CM is the process of managing products, facilities, and processes by managing their requirements, including changes, and assuring conformance in each case. CMII is CM plus continuous improvement in these five areas: (1) accommodate change, (2) accommodate the reuse of proven standards and best practices, (3) assure that all requirements remain clear, concise and valid, (4) communicate (1), (2) and (3) promptly and precisely, and (5) assure that the results conform in each case. CMII expands the scope of CM (beyond design definition) to include any information that could impact safety, quality, schedule, cost, profit, or the environment.

International Society of Six Sigma Professionals
www.isssp.com
The International Society of Six Sigma Professionals (ISSSP) exclusively promotes the interests of Six Sigma professionals. It is a global community comprised of individuals seeking to learn how Six Sigma might be introduced – or integrated – into their business processes, deployment and implementation experts, and businesses that are implementing Six Sigma and other change management practices. ISSSP is committed to the advancement of education, research and implementation of the Six Sigma methodology.