Piotr Zadora

University of Economics in Katowice

AGILE APPROACH TO USER-CENTRED SYSTEM DESIGN FOR IMPROVING SOFTWARE QUALITY

Introduction

Agile methods of software development have matured and now are capable to provide several benefits for the enterprises. That benefits comprises of reducing the cost within schedule and increasing responsiveness to the needs of the users. This could be a huge factor in today‘s world, where the companies need to get results faster. Given this backdrop, the research question is: can Agile methods produce a better software product in terms of reducing bugs and improving design? In this article the author argues that Agile approach and frequent feedback from the user are complementary for building high quality software. The relevance of the article is in conjunction of the Agile approach with UCSD. The aim of both methodologies is in fulfilling the users needs by engaging them in the development process.

Agile Strategies for Improving Quality

The vast majority of software projects suffer from a steady degradation of design quality and it becomes more and more difficult to maintain the software with the same level of quality. In some cases it becomes too expensive to maintain and in consequence it is put to rest and rewritten. In others, the software is released with a steadily increasing number of defects. Both of these common situations are deeply unsatisfying, but many of the practices from the Agile world stop the degradation of software quality and turn the trend around.

The rationale behind the Agile approach is to shift the overall focus of software development to a more "lightweight" perspective. This shift can be seen as a contrast to more formal commercial processes. Agile is not a single, well defined process, instead, it is a generic name for several different processes or methods, sharing a set of core ideas, values and principles of software development. The principles are defined in the Agile Manifesto [AgAl01]. The values for the Agile developers are known as well [AgDe12].

The Agile community has been fertile ground for quality improvements in software development. There are four major strategies that can help you improve the quality of your software:

1. Reduction of Defects − it is the first thing that comes to mind when examining the quality of software. A low defect count is often synonymous with high quality software. Defects are also the most visible sign of quality problems.
2. Design Improvement − design is the model that a development team builds and maintains. High quality design makes for an application that is easy to understand and change as new requirements are discovered. Traditionally, the team has one attempt to get the design right and then it degrades over time as it is patched on and on. Agile practices, however, give an alternative; using practices like test driven development and refactoring teams are now able to continuously improve the design of their system.
3. Theory Building − one way to look at software development is "theory building". That is, programs are theories − models of the world mapped onto software − in the head of the individuals of the development team. Great teams have a shared understanding of how the software system represents the world. Therefore they know where to modify the code when a requirement change occurs, they know exactly where to go hunting for a bug that has been found, and they communicate well with each other about the world and the software. Conversely, a team that does not have a shared "theory" makes communication mistakes all the time. The customer may say something that the business analyst misunderstands because he has a different worldview. He may, in turn, have a different understanding than the developers, so the software ends up addressing a different problem or, after several trials, errors and frustrations, the right problem but very awkwardly. Software where the theory of the team does not match, or even worse, the theory is now lost because the original software team is long-gone, degrades in quality as design changes are made that don't fit with the theory, or even just as bad, cut-and-paste work is done because the theory is not understood. Building a shared theory of the world-to-software-mapping is a human process that is best done face-to-face by trial and error with ample time.
4. Build Less − it has been proved that people tend to build many more features than are actually used. In fact, most functionality built is never used. So, one

very effective way to improve the quality of our software is to build less of it. It makes it easier to understand, gives us more time to focus on the important parts that are actually used, and almost always has fewer defects.

Only about 20% of functionality we build is used often or always. More than 60% of all functionality built in software is rarely or never used. One way to improve the quality of software is to write less code which makes it easier to understand and maintain. There are several Agile practices that help you get to that point.

The four strategies above: maintain the theory of the code, build less, building less and improving the design are not independent. Maintaining the theory of the code makes it easier to modify the design because of a greater understanding of the existing design and also directly affects the number of defects and the difficulty in fixing those defects once found. Improving the design also makes it easier address defects by being inherently easier to change. And building less code makes it easier to understand and communicate the theory of the code and is directly related to the number of defects in the system.

Agile Practices

The most important of Agile practices are summarized below [Beck03; AgPr10]:

Test driven development is an effective cluster of practices that brings automated developer tests to the forefront of development and subordinates the design to testability. This form of development produces loosely-coupled designs which are easy to maintain, greatly reduce defect counts, and enable building and maintaining only what's needed. Finally, well-written tests act as a type of executable requirements that help keep the theory of the code from decaying. Practicing automated developer tests, refactoring, and simple design are applicable to all types of development projects.

Test driven requirements call for the customer (end user) to provide requirements in an unambiguous format − usually an acceptance test − at the beginning of the iteration. *Test driven requirements* drive the architecture of the system much like *test driven development* drives the design. They also help developers only build what is needed and maintain the theory of the code as up-to-date executable requirements.

Test driven requirements needs a customer who is willing and able to participate more fully as part of the development team. Your team will also be willing to make difficult changes to the code to accommodate for testing. Finally, you are willing to pay the steep price of the learning curve for this practice (which is well worth it).

The *done state* practice is a definition that a team agrees upon to precisely describe what must take place for a requirement to be considered complete. Defining and adhering to a *done state* directly affects the quality of the software by reducing defects. A properly defined *done state* is as close as possible to deployable software which means that defects are removed to achieve the *done state*. There is no partial credit with *done state*, either you are 100% done or you are not done at all; this mindset is crucial to successfully implementing this practice.

Development team performs iterations; this implies that it needs specific, measurable goals for requirements to be met at the end. Alternatively, the team may not be performing iterations and has a high rate of defects. The team can agree on a *done state* to be met for each and every requirement and still gain the benefits of improved quality.

Automated developer tests are a set of tests that are written and maintained by developers to reduce the cost of finding and fixing defects − thereby improving code quality − and to enable the change of the design as requirements are addressed incrementally. Automated developer create a safety-net of tests that catch bugs early and enable the incremental improvement of design. Beware, however, that *automated developer tests* take time to build and require discipline.

Development team in consolidated form has decided to adopt iterations and simple design and will need to evolve design as new requirements are taken into consideration. Development team in distributed form with the lack of both face--to-face communication and constant feedback should be ready to deal with increased bugs level and a slowdown in development time.

Automated acceptance tests are tests written at the beginning of the iteration that answer the question: "What will this requirement look like when it is done?" This means that developers start with failing tests at the beginning of each iteration and a requirement is only done when that test passes. This practice builds a regression suite of tests in an incremental manner and catches errors, miscommunications, and ambiguities very early on. This, in turn, reduces the amount of work that is thrown away and therefore enables building less. The tests also catch bugs and act as a safety-net during change. Finally, by making the codebase testable, developers are implicitly reducing the coupling which often result in improved design.

The development project group has an onsite customer who is willing and able to participate more fully as part of the development team. The team is also willing to make difficult changes to any existing code. Each member of the team is willing to pay the price of a steep learning curve.

The *refactoring of the code* is Agile practice which changes the structure (i.e. the design) of the code while maintaining its behavior.

Incremental improvement of design is the name of the game with refactoring; continuous refactoring keeps the design from degrading over time, ensuring that the code is easy to understand, maintain, and change.

The reasons of refactoring are: the development team is currently working on a requirement that is not well-supported by the current design or it may have just completed a task (with its automatic tests) and want to change the design for a cleaner solution before checking in the code to the source repository.

The *pair programming* is Agile practice when two developers work together at the same computer to build a feature. One developer is the driver, and the other is the navigator; the driver is at the keyboard building the task-at-hand, and the navigator is thinking forward to design implications and reviewing the work being done. *Pair programming* is sometimes described as a continuous form of peer review. This practice improves the design and reduces the defects because two people working together to solve the same problem almost always do a better job even if they are mismatched in experience and talent.

Continuous integration reduces the defects in a software system by catching errors early and often and enabling a stop-and-fix process. It leverages both automated acceptance tests and automated developer tests to give frequent feedback to the team and prompts removing these defects promptly.

Collective code ownership means that members of a development team have the right and responsibility to modify any part of the code. They get more exposure to the entire code base and are able to remove defects wherever they are found and incrementally modify the design of the system accordingly.

Evolutionary design is the simple design practice (below) done continuously. Teams start off with a simple design and change that design only when a new requirement cannot be met by the existing design.

An *iteration* is a time-box where the team builds what is on the backlog and is a potential release and therefore enables building less and forces regularly removing defects to reach the agreed upon done state.

Releasing your software to your end customers as *often* as you can without inconveniencing them forces you to constantly have your software in releasable quality and allows you to build in smaller increments and get feedback before too much of an investment is made.

Simple design − if a decision between coding a design for today's requirements and a general design to accommodate for tomorrow's requirements needs to be made, the former is a simple design. Simple design meets the requirements for the current iteration and no more. In fact, Gartner [Gart12] now recommends an emergent approach to enterprise architecture.

Stand up meetings are daily meetings for the team to sync-up and share progress and impediments daily. This helps keep the entire team aware of what is being done and where in the system.

How to adopt Agile practices successfully?

To successfully adopt Agile practices let's start by answering the question "which ones first?" Once we have a general idea of how to choose the first practices there are other considerations. Then, once you've chosen the first practices that best fit your environment, you and your team(s) will need to be aware of the mindset you'll need to get the most out of the practices you choose.

Choosing a practice comes down to finding the highest value practice that will fit into your context. Figure 1 contains practices that help improve the quality that your software development team(s) builds. This figure will also guide you in determining which practices are most effective in increasing the quality of your software and will also give you an understanding of the dependencies.

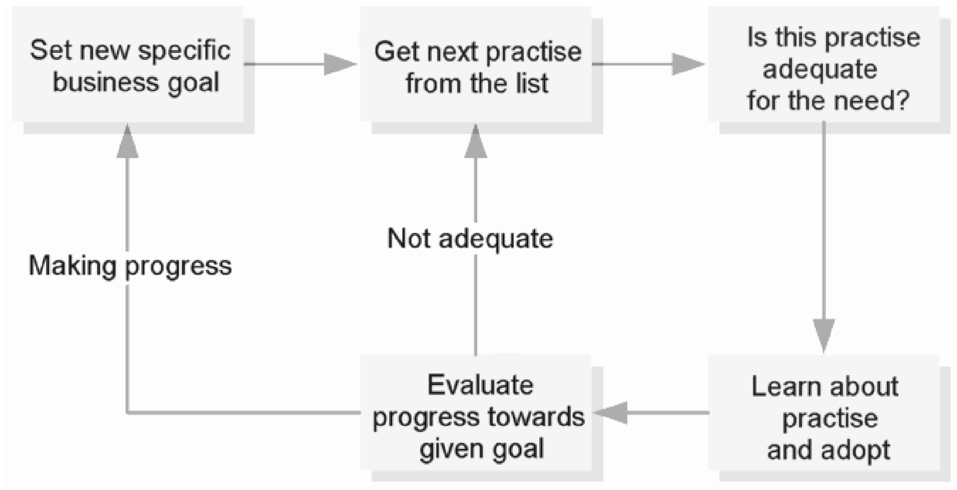
Set new specific business goal
Get next practise from the list
Is this practise adequate for the need?
Not adequate
Making progress
Evaluate progress towards given goal
Learn about practise and adopt

Fig. 1. Steps for Choosing and Implementing Practices

The practices involved in improving the quality to market are some of the most difficult to do from the body of Agile practices. Things will get harder before they get easier. The first rule is to expect the difficulty, be patient, and don't stop the practices just because they uncover significant problems; be disciplined in your practice. Once you start a practice give it a chance because you will slow down and confront frustrations before speeding up. For example, pair programming is frequently seen as a waste of resources and uncomfortable to many developers who are used to (and enjoy) working alone. Consider giving it a chance by agreeing as a team to practice pair programming for a couple of months before deciding whether it is worth adopting permanently.

Confront issues when they come up instead of stopping a practice because it is "too painful". Deal with pain differently than you are used to; instead of discontinuing something painful, examine it and find the source. Often Agile practices will uncover problems that have always been there but have not been felt. Feeling the pain is a chance to correct a problem and improve towards your goal of increased quality. A good example of this happens when teams start adopting done states for the first time. There is no partial credit, either you are 100% done or not done at all. A team that adopts this for the first time frequently works on multiple features at a time and at the end of the iteration they have not fully completed any of the features. Therefore they are 0% done with all of their tasks. This is discouraging and painful and a common response is to stop doing the practice instead of examining the pain and looking for alternatives to correct the problems in the next iteration.

Get Good at Small Steps − small steps are going to save your life with these practices because many are completely new ways of doing things that may slow you down and frustrate you as you are learning them. Take one practice, do it well, and do it regularly. You might consider pair programming along with any and all of the practices to make it easier and keep you on-track. How do you know you are doing a practice well? You get the value that you originally hoped to get – i.e. the quality of your software noticeably increases. You also have confronted pains and learned from them. If a practice is completely easy and comfortable from the get-go, or has not noticeably improved the quality of your work then you probably are not done yet.

Be Prepared to "Suspend Your Disbelief" − much of what you will be doing will not make immediate sense. It will feel that you are doing things that are more trouble than they are worth. For example − writing your tests first, before writing your code in the automated developer tests practice is non-intuitive. What can you possibly gain by doing things backward? Those who have successfully adopted this practice have "suspended their disbelief" and done it anyway. After experientially learning the practice they then made their judgments about its utility and usually kept doing it because they saw the value.

Be Proficient and Cooperate with the Users − members of the development team should realize that they are not competent in everything. Moreover, since everything really does depend on context, and they are not qualified to deal with context as novices or advanced beginners, they had better get access to the people who are experts or at least proficient to help guide them in choosing the right way of progress. Cooperation and feedback between end users and development team are the main itineraries of the so called user-centered systems design (UCSD). One of the definitions of UCSD is as follows: "(…) it is an iterative process whose goal is the development of usable systems, achieved through involvement of potential users of a system in system design" [Kara96].

It may be seen that UCSD is a process which focus is set on usability throughout the entire development process and further throughout the system life cycle (Figure 2) [GGB03].

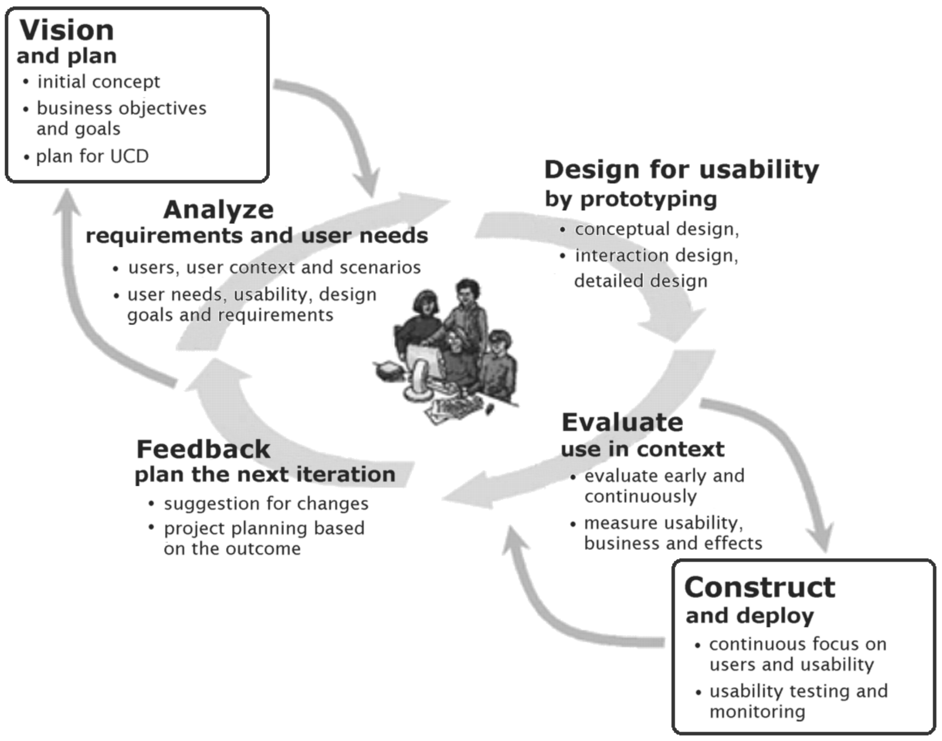
Vision
and plan
• initial concept
• business objectives and goals
• plan for UCD
Analyze
requirements and user needs
• users, user context and scenarios
• user needs, usability, design goals and requirements
Design for usability
by prototyping
• conceptual design,
• interaction design, detailed design
Evaluate
use in context
• evaluate early and continuously
• measure usability, business and effects
Feedback
plan the next iteration
• suggestion for changes
• project planning based on the outcome
Construct
and deploy
• continuous focus on users and usability
• usability testing and monitoring

Fig. 2. UCSD is a process focusing on usability either during the development or throughout the system life cycle

What is interesting about Agile methods is that they are addressing some of the problems known to be part of traditional attitudes to the development process. With regard to the usability it may be pointed out that Agile methods are "closer" to the end user than older methods. In consequence, Agile methods could create software which meets the users needs much better. Agile processes emphasize the pragmatic use of light, but sufficient rules of project behavior and the use of human and communication oriented principles. Hence, people are more important than processes and tools. Working software is more important than comprehensive documents and model building. Models and artifacts are only means of communication; consequently prototyping and simple design representations are preferred. Agile developers argue that projects should be communication centric, which implies that effective human communication with project members and users are important, e.g., face-to-face is the ideal way of communicating within a project and with users. Usually, there is a direct collaboration with users and customers – preferably, users and developers should sit in the same room during the development.

The problem with the Agile approach is that it does not guarantee by itself better usability of software produced. For example, the user interface of the system created using Agile "style" could be either carefully crafted about user expectations or just generated on used persistance layer. The main aspect of Agile methods is focused on delivering working software. This is of course excellent, as usable software also must be delivered and ready to work. But to get there, the development is focused on making coding effective and there is a risk that usability issues get lost as there could be no explicit user-centered focus. Agile projects include some roles that are supposed to work with user interface design and user requirements, but this is in most cases not enough. The whole project must be committed to the importance of usability. Another problem is that the users involved in the development are not always end users. Sometimes they are customers or domain experts. For the Agile approach it seldom makes a difference.

Summary

The consideration presented in this article support the evidence that practices of Agile development may be seen as a general set of advises for use in IT systems design and development. The effective delivery and implementation of high quality software require to conform to Agile practices and to cooperate with the end users as well. Especially that is true regarding active user participation and the necessity to use the simplest design representation possible.

References

[AgDe12] Agile Development in 30 Seconds. Available at: http:// www.techbookreport.com /tutorials/agile-30-secs.html, 2012.

[Beck03] Beck K.: Test-driven Development by Example. Addison-Wesley, 2003.

[Gart12] Gartner Group. Available at: http://www.gartner.com/it/page.jsp?id=1124 112, 2012.

[Kara96] Karat J.: User Centered Design: Quality or Quackery? The ACM/SIGCHI Magazine, "Interactions" 1996, July-August.

[AgAl01] Manifesto for Agile Software Development. Available at: http://www.agilealliance.org, 2001.

[AgPr10] The Practices of Agile Modeling. Available at: http://www.agilemodeling.com /practices.htm, 2010.

METODY LEKKIE W PROJEKTOWANIU ZORIENTOWANYM NA UŻYTKOWNIKA DLA DOSKONALENIA JAKOŚCI OPROGRAMOWANIA

Streszczenie

W artykule zaprezentowano połączenie dwóch metodologii, których celem jest poprawa jakości tworzonego oprogramowania. Przedstawiono właściwy sposób ich używania. Zdaniem Autora łączenie podejścia Agile z wytycznymi dla systemów tworzonych z udziałem odbiorcy końcowego (UCSD) prowadzi do tworzenia systemów informatycznych wysokiej jakości.