Piotr Zadora

University of Economics in Katowice

JOINING AGILE WITH UNIFIED PROCESS IN ORDER TO IMPROVE SOFTWARE QUALITY

Introduction

Business needs for improvement of the software development are increasing. Organizations expect faster results from their investments; they want their improvement projects to adapt to and follow changing business needs. The agile way of working, used more and more in software development, contains several techniques that support these business needs. So the question is: Could an improvement of the software development be performed in an agile way and what about an adoption of this approach by more traditional practices? This article is a quick introduction to Agile practices that can help improve the quality of software creation by reducing defects, improving design, sharing the theory of the code and building less. It includes an introduction of how to choose the practices for your organizational context. It also includes an example how to adopt Agile Modeling in one of the more popular traditional approaches in order to improve software quality.

Improving software quality

The vast majority of software projects suffer from a steady degradation of design quality and it becomes more and more difficult to keep in the same level of quality. As the software ages it becomes harder and harder to maintain because of the lack of skilled personnel capable to deal with it. In some cases it becomes too expensive to maintain and therefore the software 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 there is another way. Many of the practices from the Agile realm inhibit the degradation of software quality and turn the trend around. It is not unknown for teams to have maintained a zero-defect status of their projects for months and years.

Software design and architecture have become elastic and easily influenced by different transformations over time. According to this, Gartner recommends an emergent approach to enterprise architecture. Practices that are helpful in improving the quality of developed software are shown on Figure 1 [Elss08, NeAp09].

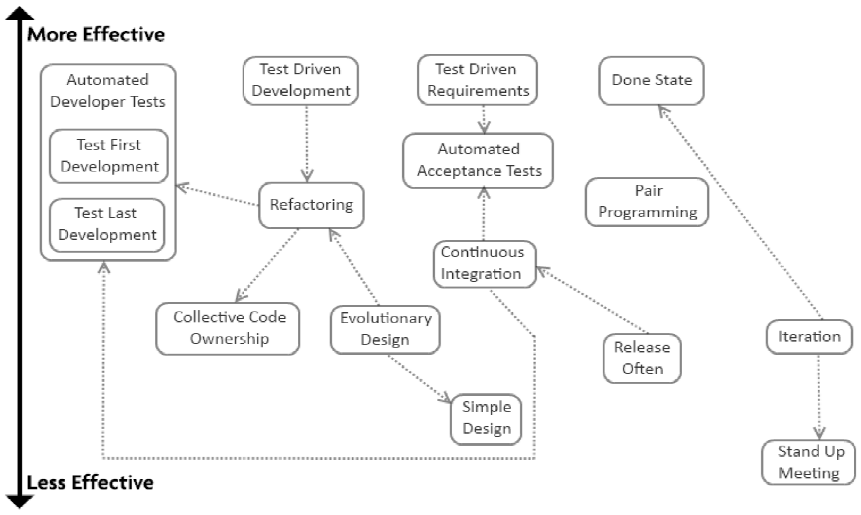

Fig. 1. Effectiveness of Agile practices in terms of quality

There are four major strategies that can help to improve the quality of software:

- The diminution of defects – a low defect count in the code is often synonymous with high quality software. Defects are also the most visible indication of quality problems.
- Design improvement – high quality design makes for an application that is easy to understand and flexible enough to change as new requirements are discovered. In traditional approaches developers usually make very good design in the beginning of the process and then it degrades over time as it is patched many times. Agile practices give an alternative; using practices like test driven development and refactoring developers are now able to continuously improve the design of their system.
- Theory building – programs can be treated as theories, i.e. models of the reality mapped onto software. The proper way of managing these models leads to success. Building a theory of the reality-to-software mapping is a human process that is best done face to face by trial and error with more than enough time. Effective development teams have a shared understanding of how the software system represents the world. In consequence they know how to

modify the code when a requirement change occurs, they know where to go searching for an error that has been reported, and they communicate well with other members of the team about the model and the codebase.

- Building less – only about 20% of functionality which is usually built 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. Additionally smaller codebase almost always has fewer defects.

There are some important dependencies between the aforementioned strategies shown on Figure 2 [RefC11].

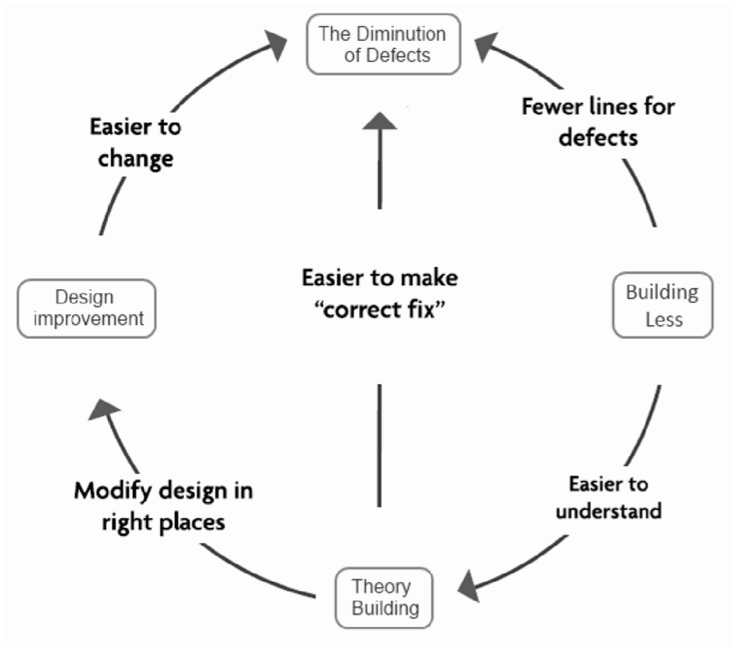

Fig. 2. Strategy dependencies

It can be counted that there are few advantages of given strategies in practice. Maintaining the theory of the code makes it easier to modify the design because of a better understanding of the existing design. Keeping the theory up to date also diminishes the number of defects and the difficulty in fixing those defects once found. Improving the design also makes it easier to remove errors by being inherently easier to change. 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.

Improving software quality in the Agile context

Developers usually start IT projects with general idea of how to choose the first practices of the Agile methodology. Then, once the first practices that best fit of given environment have been chosen, developers need to be aware of the mindset they need to get the most out of the practices they choose. Choosing a practice comes down to finding the highest value practice that will fit into the context of the system to be created. Figure 3 contains dependencies between practices that are capable to improve the quality of the software. There is also a measure which puts these practices in order related to effectiveness shown in that figure.

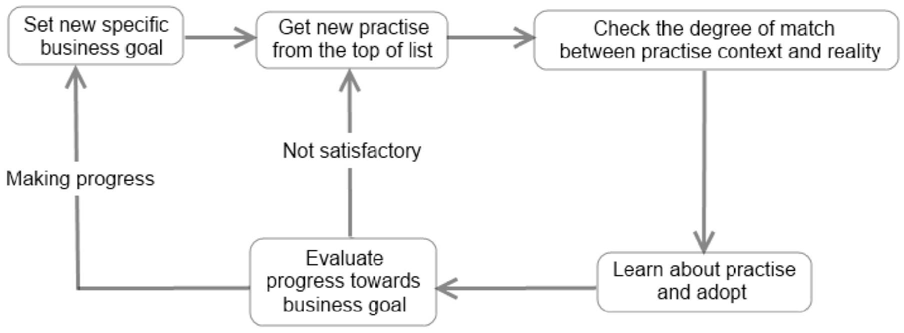
Set new specific business goal
Get new practise from the top of list
Check the degree of match between practise context and reality
Not satisfactory
Making progress
Evaluate progress towards business goal
Learn about practise and adopt

Fig. 3. Steps for choosing and implementing practices

For improving quality, the most valuable set of practices are those nearest the top of Figure 1. Four of the practices are independent: done state, automated developer tests, automated acceptance tests, and pair programming. For example, developers should consider to adopt pair programming as a support practice to the other three practices. Next step to consider should be done state and automated developer tests, and then finally automated acceptance test (probably the most difficult of this set to adopt correctly).

The practices involved in improving the quality of software are some of the most difficult to do from the body of Agile practices. For example pair programming is frequently seen as a waste of resources and uncomfortable to many developers who are used to working alone. Developers should consider to try out this technique by agreeing as a team to practice pair programming for a couple of months before deciding whether it is worth adopting permanently. Most frequently Agile practices show problems that have always been there but have not been discovered. Facing with difficulties is a chance to correct a problem and improve towards the goal of increased quality. A good example of this happens when developer team starts adopting done states for the first time. This is scenario when a team frequently works on multiple features at a time and at the end of the iteration there is a lot of work in the middle and nothing fully completed. This is discouraging and a common response is to stop doing the practice instead of examining the reason of problems and looking for alternatives to correct them in the next iteration.

Another good approach of dealing with these practices are "small steps" because many of them are completely new that may slow down development and cause frustration. It should be taken one practice at a time, performed well, and then used regularly. There is an issue of examining whether the practice is performed well. The estimation may be based on the value that the developers team originally hoped to achieve. If a practice is completely easy and comfortable from the first time, or has not noticeably improved the quality of work done then the team most probably did not performed it well.

Much of what developers are working on at the moment do not provide immediate sense. For example – writing the tests first, before writing working code in the automated developer tests practice is non-intuitive. It causes the issue of what is the real achievement by doing things backward? Those who have successfully adopted this practice have "suspended their disbelief" and done it anyway. After experientially learning the practice such developers then made their judgments about its utility and usually kept doing it because of the value seen.

Merging heavy methods with Agile practices

Every Project Manager can successfully integrate Agile principles and practices with heavyweight approaches to improve software quality, by understanding and applying that Agile is not made for projects as Project Managers define projects but more likely for product management. Product management is concerned with the life of the product; from conception, through development and eventually to discontinuation. Projects are not concerned with the ongoing improvement or enhancement of a product over the entire lifecycle of that product. A project is all about the creation of something but when the something is created the project is done [InfoQ11]. Understanding that subtle difference, it is informative to find out how Agile practices can be merged with heavyweight methods and whether there are some similarities or common elements between them.

One of the most known "heavy" methods of software development is the Rational Unified Process (RUP) created by the Rational Software Corporation, a division of IBM since 2003 [Eweek02]. RUP is not a single concrete prescriptive process, but rather an adaptable and iterative process framework, intended to be tailored by the development organizations and software project teams that will select the elements of the process that are appropriate for their needs [Wiki14].

Many of Agile Modeling principles and practices are a part of the Unified Process (UP) already, although perhaps not as explicitly as it could be. It is relatively straightforward for teams using Unified Process to adopt Agile practices if they choose to do so. This is because the Unified Process is very flexible and let the developers to choose elements that meet their unique needs. Below is the list of Agile practices used in modeling phase which may be adopted within Unified Process framework [AMRUP08]:

- active stakeholder participation,
- applying modeling standards,
- applying patterns gently,
- applying the right artifacts,
- collective ownership,
- creating of several models in parallel,
- depicting models simply,
- discarding temporary models,
- displaying models publicly,
- formalizing of contract models,
- iteration to another artifact,
- modeling in small increments,
- modeling with others,
- proving with code,
- reusing of existing resources,
- single source information,
- updating only when it is needed,
- using of the simplest tools.

Agile principles used in modeling define project stakeholders, users, management, operations staff, and support staff which are compatible with the Unified Process. The UP clearly includes project stakeholders, such as users and customers, throughout most of it disciplines. To be successful project teams should allow project stakeholders to take on modeling roles such as Business Process Designer and Requirements Specifier as appropriate, there is nothing in the RUP preventing this by the way. The more active project stakeholders are the less of a need there will be for reviews, management presentations, and other overhead activities that reduce team's development velocity.

The application of modeling standards, in particular the diagrams of the Unified Modeling Language (UML), is a significant part of the Unified Process. Furthermore the RUP product includes guidelines for the creation of many modeling artifacts, guidelines that developers team should consider adopting and following as appropriate, and explicitly suggests that the guidelines should be tailored or even bend to suit developers needs.

Unified Process teams are free to apply modeling patterns, the RUP product describes many common modeling patterns for any of the modeling disciplines. Practice of applying patterns enhances Unified Process with its advice to ease into the application of a pattern. There is no explicit clarification of this concept within Unified Process framework.

Considering application of the right artifacts it should be stated that one of the advantages of the Unified Process is that it provides advice for when to create each type of model, even for non-UML artifacts such as data models and user interface storyboards (UI flow diagrams).

Agile concept of collective ownership can be used to enhance the efforts on Unified Process projects, assuming that the team behaviour supports the concept of open and honest communication. The UP supports collective ownership with its strong focus on configuration management issues. Developer teams should allow anyone on the project to access and work on any artifact that they wish, including models and documents.

Unified Process clearly includes concept of creation several models in parallel. However, this concept could be communicated better because the near-serial flow in its activity diagrams presented for each major modeling activity doesn't communicate this concept well. There is a larger issue as well when you consider the lifecycle as a whole. Because the UP has organized its modeling efforts into separate disciplines, for very good reasons, it is not as apparent that not only could developer work on several business modeling artifacts in parallel but he could also work on requirements oriented artifacts, analysis-oriented artifacts, architecture artifacts, and design artifacts as well.

The practice of depicting models in straightforward way is a choice made by the modeler, albeit one that must be implicitly supported by the rest of the development team. Unified Process teams will need to adopt modeling guidelines that allow models that are just good enough and the customers of those models (including programmers, project stakeholders, and reviewers) must also be willing to accept simple models. This issue is one of the most difficult for many organizations to adopt.

Modelers on Unified Process teams are free to discard anything that they wish. As with the simplicity practices your organization's culture must accept the concept of developing and maintaining just enough models and documents and no more.

Unified Process teams are free to follow practice of showing models publicly. Developer teams should follow the principle of open and honest communication by making all artifacts available to everyone.

The Unified Process includes the concept of integrating with external systems. These systems are typically identified on use case models and there is suggestion to introduce "boundary classes" as implementation of the interface to these systems. The interaction between systems could be specified with one or more use cases and the corresponding realization of them would be the formalized contract model. Hence, the adoption of this Agile practice can make a positive contribution to strengthening the integration capabilities of enterprise application realized according to Unified Process approach.

The practice of iteration to another artifact can be easily adopted by Unified Process team. Understanding of UP's modeling activities as quasi-serial processes and the division of modeling activities into separate disciplines can hinder the iterative mindset required of Agile developers.

The practice of modeling in small increments is clearly an aspect of the Unified Process. The UP's support for iterations implies that model established at the beginning will be incrementally developing throughout the project. This is obvious that smaller, simpler models may quickly lead to implementation and testing.

Unified Process implicitly includes the practice of modeling with others which clearly defines several roles played by one or more people. The subsequent Agile practice of proving with code is also included in Unified Process framework. At the end of every iteration, except the Inception phase, the UP specifically states that the team should have a working prototype. Furthermore, the UP insists that developers have a working end-to-end prototype at the end of the Elaboration phase that proves given architecture.

Reusing of existing resources is an implicit part of the Unified Process. Furthermore, reuse management is an explicit part of the Enterprise Unified Process. Developer teams should prefer to reuse existing resources instead of building them from scratch, including but not limited to existing models, existing components, open source software, and existing tools.

According to single source of information practice, there is no reason why developers cannot store information in a single place when following Unified Process. Unfortunately, many organizations choose to instantiate the RUP in a documentation-driven manner, and as a result they are proceeding with heavy load and clearly take a multi-source approach.

Considering the practice of updating only when it is needed, many developer teams in reality prove to have a problem with this concept, particularly if they have a strong habit of checking relations between aspects of project artifacts, the support for which is a strong feature of Unified Process as it is an important aspect of its Configuration and Change Management discipline. Furthermore, Rational Unified Process includes tool mentors for working with Rational RequisitePro, a requirements tracing tool, making it appear easy to maintain a traceability matrix between artifacts.

Developer organizations with strong habit of checking relations between project artifacts will often choose to update them regularly, even if it is not necessary at the moment. Such attitude should be replaced with another one which allows to maintain a traceability matrix between artifacts only when there is clear benefit to do so and project stakeholders authorize the effort.

Rational Unified Process product includes tool mentors that make it easier for teams to work with tools sold by Rational Corporation. However, in reality many developer teams are using another development tools which suit their needs and Rational tools are just one of the competitors in this area.

References

[AMRUP08] Agile Modeling and the Rational Unified Process (RUP). Available http://www.agilemodeling.com/essays/agileModelingRUP.htm, 2008.

[Elss08] Elssamadisy A.: Agile Adoption Patterns: A Roadmap to Organizational Success. Addison-Wesley Professional, 2008.

[Eweek02] Taft D.K.: IBM Acquires Rational. Available: http://www.eweek.com/c/a/Desktops-and-Notebooks/IBM-Acquires-Rational, 2002.

[InfoQ11] Flahiff J.: Integrating Agile into a Waterfall World. Available: http://www.infoq.com/articles/agile-in-waterfall-world, 2011.

[NeAp09] Gartner Identifies New Approach for Enterprise Architecture. Available http://www.gartner.com/it/page.jsp?id=1124112, 2009.

[RefC11] Agile Adoption: Improving Software Quality. Available: http://refcardz.com, 2011.

[Wiki14] Rational Unified Process. Available: http://en.wikipedia.org/wiki/ Rational _Unified_Process, 2014.

ŁĄCZENIE PRAKTYK AGILE Z PODEJŚCIEM UNIFIED PROCESS W CELU DOSKONALENIA JAKOŚCI OPROGRAMOWANIA

Streszczenie

Wytwarzanie oprogramowania wysokiej jakości stanowi jedno z największych wyzwań stojących przed deweloperami. W artykule zaprezentowano koncepcję wykorzystywania praktyk Agile, która sprzyja wytwarzaniu oprogramowania wysokiej jakości. Przedstawiono również jedną z możliwości łączenia praktyk Agile z tradycyjnymi „ciężkimi” podejściami. Zdaniem Autora wykorzystanie zalet obu podejść, zamiast przeciwstawiania ich sobie, powinno prowadzić do skutecznego tworzenia oprogramowania wysokiej jakości.