The ITB Journal

Volume 5 | Issue 1 Article 16

2004

Strengthening the Practices of an Agile Methodology?

Jimmy Doody
Department of Computing, Institute of Technology Tallaght, Dublin 24.

Amanda O'Farrell
Department of Computing, Institute of Technology Tallaght, Dublin 24., Amanda.oFarrell@it-tallaght.ie

Strengthening the Practices of an Agile Methodology?

Jimmy Doody[1], *Amanda O'Farrell*[2]
[1] *Department of Computing, Institute of Technology Tallaght, Dublin 24*
[2]*Department of Computing, Institute of Technology Tallaght, Dublin 24*
Contact email: Amanda.oFarrell@it-tallaght.ie

Abstract

An investigation into how the software development process in an agile environment (Extreme Programming) can be aided by intelligent software, leading to the development of a tool that will automate the process of standardising and clarifying source code. The tool will also speed up and aid the testing process, by producing test objects based on the source code, and by providing full test tracking, without having a negative impact on the development process. By making the coding and testing processes more automated, the research aims to evaluate the following hypotheses:

1.That the programmer's productivity can be increased by the use of such a tool

1.That the standardised and clarified source code can reinforce the practices of Collective Code Ownership, Refactoring and Pair Programming

1.That automated testing can reinforce the practices of Code Ownership, Refactoring, Small Releases and Continuous Integration, as stated by Beck (2000).

Keywords: Agile Methodologies, Extreme Programming, Programmer Productivity, and XP Practices

1 Introduction

The needs of software development projects have changed radically over the years, and as a result of this many different software development methodologies have been developed to try to address these changing needs, whilst improving the quality of the software and the predictability of the project.

The first well-known and widely used methodology, the traditional software development life cycle (SDLC), based its approach on the more traditional engineering disciplines (Burch, 1992). This approach has a strong emphasis on planning and is often referred to as being 'document driven' (Glass, 2001). Following on from this a number of methodologies have emerged which take different views on how software should be developed. Rapid application development (RAD) was one such methodology, which based its process around the production of prototypes. However it is only suited to systems that need to be developed quickly, for example, in response to a perceived opportunity to gain competitive advantage in a new market or industry (Hoffer et al., 2002). As a compromise between speed and quality the evolutionary methodologies such as incremental and spiral were developed. These methodologies combined aspects of the more traditional SDLC with aspects of RAD. Then during the 1990s came the object-oriented life cycle, this was developed to support the object-oriented programming paradigm (Martin et al., 1992). Design Patterns quickly followed; however, they try to solve a general design problem in a particular context, as opposed to defining a complete development methodology (Gamma et al., 1995).

Although these methodologies are quite different, they all (although to different levels) try to impose some disciplined process on software development in order to make it more efficient and predictable (Riehle, in Succi, et al., 2001, pp. 35). They do this by developing a detailed process with strong emphasis on planning. Fowler (2003) refers to them as engineering methodologies because they are inspired by other engineering disciplines. However, these methodologies are frequently criticised for being too bureaucratic, with so much additional work to do to follow the methodology that the development process is completely slowed down (Fowler, 2003). Indeed, Fowler (2003) refers to them as the heavy methodologies, because of the huge bureaucratic overhead associated with them.

Over the last few years, a number of new methodologies have been developed in order to address some of the problem issues that exist with the more traditional life cycles. These were originally known as lightweight methodologies, because they do not have the same administrative overhead as the more traditional methodologies (Riehle, in Succi, et al., 2001, pp. 37). However, the preferred term is agile methodologies, because although the methodology is light in weight, the Agile Alliance[1] did not want the methodologies to be referred to as 'lightweight' (Cockburn, in AgileAlliance, 2001).

2 Agile Methodologies Explained

Agile methodologies have developed as a reaction to the heavy methodologies, in an attempt to establish a compromise between too much process and no process at all (Fowler, 2003). As a result of this, agile methodologies have some differences from other methodologies. The first big difference that is usually noticed is the smaller amount of documentation. According to Fowler (2003), the agile methodologies are code-oriented, and follow a path, which says that the source code is the key part of the documentation. However, Fowler (2003), states that the lack of documentation is as a result of two much deeper differences that exist in all agile methods and these are:

1"*Agile methods are adaptive rather than predictive*" - other methodologies try to plan out all aspects of the projects in advance, and often spend a huge amount of man hours in doing this. This works very well until things change, thus these methodologies have a tendency to resist change. Agile methodologies try to be processes that adapt, even to the point of changing themselves

1"*Agile methods are people-oriented rather then process-oriented*" - other methodologies define a process that works no matter who is involved in using the process. The view of agile methodologies is that the skill of the development team is the most important factor and thus the role of the process should be to support the development team

[1] A consortium of experts that promote the use of agile methodologies

Taking the first point, that agile methodologies are adaptive rather then predictive, Fowler (in Succi, 2001, pp. 5-6) discussed the differences between a civil or mechanical engineering project and a software development project. According to Fowler (in Succi, 2001, pp. 5) a civil engineering project has two fundamentally different activities, performed by two very different groups of people. The first activity is design, which requires expensive and creative people. The other is construction, which requires less expensive people. Another issue is that with other forms of engineering, the models that are produced as a result of the design phase are based not only on many years of practice, but can also be subjected to mathematical analysis, and therefore schedule predictability is a given (Fowler, 2003).

So based on this, software methodologies should have a predictable schedule that can utilise people with a lower skill base, therefore we must separate design from construction – however, two problems exist with this approach. Firstly in software development, programmers must be skilled enough to question the designer's designs, especially when the designer is unfamiliar with the technologies used in the project (Fowler, in Succi, 2001, p. 5). Secondly, the design must be done in such a way that the construction is as straightforward as possible. The more traditional methodologies try to achieve this by using design notations such as UML, but Fowler (2003) questions the ability of any software designer to produce a design that can make the coding process a predictable activity. Indeed one of the main problems with any software design notation is that you will not realise that it is flawed until you have turned it into code.

He also questions the costs involved in producing extensive software design documents. In civil engineering, the cost of the design effort is about 10% of the job, with the rest of the effort going into construction. With non-agile methodologies, software design can take up to 50% of the effort, with construction taking roughly 15% of the effort. It seems that non-agile methodologies try and emulate engineering disciplines in terms of planning and predictability, and yet when it comes applying these processes to software engineering, it's apparent that the same rules are just not relevant (Fowler, 2003).

This is not a new idea. In a C++ journal published in 1992, Reeves (1992) also discussed what he calls "false parallels" with other engineering disciplines. He discusses the relationship between programming and software design and claims that the software industry as a whole has missed a subtle point about the difference between developing a software design, and what a software design really is. He claims that programming is not about building software, but about designing software, and hence source code is your design document. He also suggested that in the software development life cycle, the construction phase is actually the use of the

complier or the interpreter. Combining Reeves's thinking with his own on the subject, Fowler (2003) has concluded that:

- In software, construction can be free
- In software all the effort seems to be on design, which requires not only talented but creative people
- If design is a creative process then it cannot be easily planned, and so it cannot be easily predicted
- We should be wary of methodologies that use the traditional engineering metaphor for building software, because building software is a very different kind of activity to building a bridge

Another important issue with heavy software development methodologies is that they do not deal well with change; however, changing requirements is an unpleasant fact when it comes to developing software. If planning is based around the software requirements, and they are constantly changing, how can we get a predictable plan? According to Fowler (2003) in most cases you cannot, but that does not necessarily mean that you have no control, it just means that you should use a process that can give you control over unpredictability. This is what is meant by an adaptive process, which agile methodologies claim to be. So how exactly do agile methodologies achieve this adaptability? Fowler (2003) states that the most important factor in controlling a project in an unpredictable environment, is to know accurately where the project currently stands, and so we need some feedback mechanisms that can accurately tell us what the situation is at frequent time intervals. According to Beck (2000) the key to this feedback is iterative development, whereby you frequently produce working versions of the final system which have a subset of the required features. Lippert (2002) discusses incremental development, and states that is it important that increments are fully integrated and as carefully tested as any final delivery, because there is nothing like a series of tested, working systems to show exactly how a project is progressing. Iterative development leads to a style of planning whereby the only stable plans are for the next increment, and because of this, long-term plans are more adaptable (Fowler, 2003). However, it is worth noting that not all types of systems will be suited to iterative development, for example, an Air Traffic Control System or a Payroll system (Jeffries et al., 2001).

The second point that Fowler (2003) makes is that agile methodologies are "people-oriented rather then process-oriented", whereas with other methodologies people are viewed as replaceable resources which play some role, and thus only the role is important, not the resource that fills it. He describes people working in this type of process as being viewed as "plug compatible programming units", with no room for creativity or individuality, and that when you have good people, this attitude leads to lower morale – and hence lower productivity. Agile methodologies completely reject this process-oriented notion; Fowler (2003) describes

programmers as being "responsible professionals." To back up this point he references a paper written by Alistair Cockburn, which describes people as being the most important factor in software development (Cockburn, 1999). Fowler (2003) states that if you want to retain good people you must recognise them as bright, competent, motivated professionals, and as such they themselves are best suited to decide how they conduct their own technical work. In agile methodologies, there is a clear distinction between the role of the customer and the developer, and although they are on the same team, they are considered to have different decisions to make. According to Wake (2001) for example, when practicing Extreme Programming, the developer must be able to make all the technical decisions and all time estimates, whereas the customer makes decisions on issues such as scope and priority.

3 Research Areas

3.1 Extreme Programming (XP)

To date a number of different agile methodologies have been developed, and while they all share many characteristics, there are also some significant differences, and so with this in mind the research will focus around one particular methodology. Extreme Programming (XP) has been chosen because it is strongest in its view that source code is the driving force in any development methodology. XP is the methodology which has attracted most attention; firstly because one of the main proponents of XP is Kent Beck, and secondly because some of its main practices go against what the more traditional methodologies teach. It is seen to be extreme in its views of software development and as such is considered controversial. However, it is the most widely documented of the agile methodologies and although it is extreme in some of its ideas it also claims to be open to interpretation and adaptive in nature.

Jeffries (2001) describes XP as software development based on the four values of simplicity, communication, feedback and trust. These values are quite vague however, and so they are expanded into concrete principles which are used to guide the development team as to which of the 12 XP practices[2] will best satisfy the four values (Beck, 2000). According to Beck (2000), these 12 practices reinforce each other – he states that none of them stand well on their own, but that each requires the others to keep them in balance. (However, this seems to mean that by ignoring one of the 12 practices, there will be a lessening in the strength of the others that it reinforces. This would indicate inflexibility when it comes to selecting which practices to use for a particular project, even though agile methodologies claim to be highly adaptive.)

[2] Planning Game, Metaphors, Pair Programming, Refactoring, Collective Code Ownership, On-site Customer, Small Releases, Simple Design, Testing, Continuous Integration, Sustainable Pace, Coding Standards

XP, like the other agile methodologies, believes that the key part of documentation is the source code. Indeed Beck (2000) goes as far as saying that source code is the only artefact that the development process absolutely cannot live without. As a follow-on from this statement, if we must have source code then we should use it for as many of the purposes of software engineering as possible (Beck, 2000). Thus, the first major objective for the research is to look at how the source code from a project can be used for other software engineering purposes, such as generating API documentation or creating unit tests.

3.2 Coding Standards and Simplifying Code

Lippert (2002) states that the source code should follow some coding standards as agreed upon by the programmers, so that it is impossible to tell that the code has been written by many different people. This standardisation of code is one of the main practices of extreme programming and according to Lippert (2002), can reinforce some of the other practices, such as:

- Collective code ownership - by allowing any developer to take some code and add value to it, without having to worry about different coding standards
- Refactoring - as the programmers will not have to re-format after making some changes
- Pair Programming – there will be no disagreements over different formatting and standards

We view project quality in two distinct ways: external quality, which is measured by the customer, and internal quality, which is something that is measured by the programmers (Beck, 2000). Sometimes internal quality is sacrificed in order to meet project deadlines, and ignoring coding standards is one common example of this. However, because coding standards reinforce other practices in an XP environment, ignoring them is just not an option. Following coding standards, however, is not enough when working in an extreme environment. You must agree upon and adhere to general good programming practices. According to Beck (2000), in an XP environment, good programming is programming that follows the simplest design possible. Therefore, you should only code what you need when you need it. Hence, the notion of coding for the future is considered a waste of valuable development and testing time, since the future is uncertain. His definition of simple is as follows:

- The system (code and tests) must communicate everything you want it to communicate
- The system must contain no duplicate code (Once and Once Only Rule)
- The system should have the fewest possible classes
- The system should have the fewest possible methods

Obviously taking the time to apply standards and follow the practice of keeping things simple could slow the development process. So with these issues in mind, the next objective of the

research is to look at how team-defined standardisation can be applied automatically and how the values of simplicity as defined by Beck (2000) can be upheld.

3.3 Testing

Iterative development is something that is practised by XP teams, and is enabled by having small, frequent releases of running, fully tested software. Lippert (2002) states that a simple system should be put into production quickly and the team should then release new versions as frequently as possible. The XP team should integrate and build the system many times a day, or even every time a task is completed. This in turn means that the developers should test their own code fully before trying to integrate it with the current system. According to Beck (2000) developers must continually write unit tests, which must run flawlessly to allow development to continue. Again, this could slow the development process; according to Lippert (2002) however, as with coding standards, the testing process cannot be ignored, as it reinforces other XP practices such as:

- Small releases – by continually testing, the defect rate is so small that lengthy test cycles are no longer a factor and hence frequent releases are enabled
- Refactoring – by re-running tests, programmers can instantly see if refactoring has caused any problems
- Collective code ownership – by re-running tests, programmers can check to see if a change has caused any problems
- Continuous integration – a programmer can confidently release his code for integration, when it has been fully tested

With these issues in mind, another objective of the research is to try to automate the testing process, to not only produce test classes but also to look for duplicate code, and to identify classes and methods that are no longer used.

4 Future Work

To try to answer the research questions, a prototype tool will be developed which will allow an XP team to fully define their standards and hence apply them automatically to any classes that are produced. This prototype will be written for the java language specifically and so the tool will have the option of automatically inserting javadoc comments. The tool will also automatically produce test classes for the objects defined and will look for duplicate code. It will identify methods and classes that are not used, and classes that have not been tested to date. When a class is changed in any way the tool will re-run tests on all classes, to make sure that the changes have not caused problems with any other classes.

It is hoped that the tool will be used in an industrial setting, in order to test the tool and to review the impact that it can have on the coding and testing practices, and also to discover how these two practices can reinforce other XP practices.

References

The AgileAlliance, 2001, History: *The Agile Manifesto*, Available from http://www.agilemanifesto.org/history.html [Accessed 2 September, 2003]

Beck, K. 2000, *Extreme Programming Explained – Embrace Change,* Addison Wesley Longman, Massachusetts

Burch, J. G. 1992, *Systems Analysis, Design and Implementation*, Boyd & Fraser, Boston

Cockburn, A. 1999, *Characterizing People as Non-Linear, First-Order Components in Software Development,* Crystal Methodologies, Available from http://crystalmethodologies/articles/panic/peopleasnonlinearcomponents.html [Accessed 12 September, 2003]

Fowler, M. 2003, *The New Methodology*, Available from http://www.martinfowler.com/articles/newmethodology.html [Accessed 19 July, 2003]

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Massachusetts

Glass, R. L. 2001, *Agile versus Traditional: Make Love, Not War!*, Cutter IT Journal, 14, 12, Available from http://www.cutter.com/itjournal/2001toc.html [Accessed 25 September, 2003]

Hoffer, J. A., George, J.F. & Valacich, J.S. 2002, *Modern Systems Analysis and Design,* 3rd edition, Prentice Hall International,

Jeffries, R. 2001, *What is Extreme Programming*, XProgramming.com, Available from http://www.xprogramming.com/xpmag/whatisxp.htm [Accessed 20 September, 2003]

Jeffries, R., Anderson, A. & Hendrickson, C. 2001, *Extreme Programming Installed*, Addison-Wesley, United States of America

Lippert, M. & Roock, S. & Wolf, H. 2002, Extreme Programming in Action, Practical Experiences from Real World Projects, Wiley & Sons, England

Martin, J. & Odell, J. 1992, *Object-Oriented Analysis & Design,* Prentice Hall, New Jersey

Reeves, J.W. 1992, *What is Software Design,* Available from http://www.bleading-edge.com/Publications/C++Journal/Cpjour2 [Accessed 29 August, 2003]

Succi, G., & Marchesi, M. 2001, *Extreme Programming Explained,* Addison-Wesley, United States of America

Sun, 2003, *Code Conventions for the Java Programming Language*, Sun Microsystems, Available from http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html [Accessed 9 March, 2004]