Extreme Programming: Strengths and Weaknesses

Ahmad dalalah
Prep. Year Deanship
University of Hail, SA
a.dalalah@uoh.edu.sa

Abstract: *Extreme Programming (XP) is an agile software development methodology. It is a lightweight methodology combining a set of existing software development practices [5]. This paper aims to discuss the strengths and weaknesses of the Extreme Programming methodology through examining the characteristics of the twelve software development practices of the XP methodology.*

Keywords: *Extreme programming, Release,* Exploration Phase, System Metaphor.

1. Introduction – What is Extreme Programming?

Extreme Programming (XP) is an agile software development methodology. It is a lightweight methodology combining a set of existing software development practices [5]. XP tends to rapidly develop high-quality software that provides the highest value for the customers in the fastest way possible.

Extreme Programming is based on values of simplicity, communication, feedback, courage, and respect, which was newly added. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.

Dr. Dalalah is originally with Jordan University of science and Technology, Jordan. He is in leave of absence and now with University of Hail, Saudi Arabia.

Extreme Programming consists of four main phases: Planning, Designing, Coding and Testing. Each of these phases includes a number of rules and practices. There are 12 practices: On-site Customers, planning game, small releases, simple design, system metaphor, re-factoring, coding standards, pair programming, 40-hours work week, continuous integration, collective code ownership, and testing. Testing include both unite testing and acceptance testing. These are well-known common software development practices but XP takes these practices to their extremes.

In addition to these 12 practices XP has a number of supporting practices. These supporting practices include: do the simplest thing that could possibly work, develop what is immediately required to meet customers need, coaching to help the team keep in track.

This paper is organized as follows: section two discusses the practices in the planning phase; section 3 discusses the practices in the designing phase; section 4 discusses the practices in the coding phase which is the main phase of the XP methodology; section 5 discusses the testing phase, this phase includes the unit testing and the acceptance tests; and section 6 concludes the report and provide some suggestions for future work.

2. Planning

The planning phase begins by writing users stories. User stories serve the same purpose as the use-cases, but are not the same [9]. User stories are written by the customer and are used to create time estimates for release plan. The release plan is then used to create iteration plans for each of the iterations in the product life cycle.

The development team needs to release **small releases,** iterative versions, to the customers often. After the first release, the project velocity is calculated. The project velocity is a measure of how much work is getting done on your project [9]. The velocity is used to decide the number of iterations and the time estimates for each of the iterations. Iterative Development adds agility to the development process. Iterative Development adds agility to the development process.

One of the XP principles is to move people around. This is done to avoid any knowledge loss that might cause a coding bottleneck.

The next subsections introduces three practices of the XP methodology used in the planning phase, these are the on-site customers, the planning game, and small releases. For each of these practices, each sub-section explains the main idea of the practice and discusses the strengths and weaknesses of the practice.

2.1. On-site Customers

On-site customer means to include real life customers in the development process. The customers will be always available to answer questions, provide the requirements, set the priorities, and steer the project.

As a result, this will ensure the customers' satisfaction by including them in and will avoid frustration caused by negative feedback caused by misunderstanding the requirements.

On the other hand, it is not realistic to assume that the customers will be available all the time. The *on-site customer* is an ideal situation. Having on-site customers can sometimes be difficult since customers do not fully understandable the benefits of regular developer-customer interactions [5], and they do not want to be bothered by giving feedback to all team members all the time.

2.2. The Planning Game

There are two key planning steps in XP: release planning and iteration planning. The planning game tends to create a time estimate for the release plan. The release plan is then used to create iteration plans for each of the iterations. **Release Planning** is a practice where the developers and the customers decide on which features will be included in which release and when it will be delivered. The programmer gives a cost for each of the stories given by the customer **-- Exploration Phase**. The cost is an estimate of the story difficulty and the time required to develop the story. Using the cost estimates, and with knowledge of the features importance, a plan for the project is laid out and a commitment is done to deliver the features in the date agreed upon **–Commitment Phase**. The plan is not precise as the cost and priorities are not solid. However, the release plan is revised frequently when required **–Steering Phase**.

Iteration Planning during Iteration Planning, the programmers' break down the features provided by the customers into tasks, and estimates their cost. Based on the amount of work accomplished in the previous iteration, the team signs up for what will be undertaken in the current iteration [1]. This gives directions to the team every couple of weeks.

The planning game is very simple, yet it provides very good information about what has been done and what could be accomplished in a two weeks period. It also provides an excellent steering control for the customers. The customers are aware of the progress of the project, and whether the progress is sufficient or not.

On the other hand, progress is so visible, and the ability to decide what will be done next is so complete, that XP projects tend to deliver more of what is needed, with less pressure and stress [1].

2.3. Small Releases

The development team is required to make small frequent releases of working software that customers can evaluate. The first release includes the smallest set of useful features set. Subsequent releases include newly added features.

Small releases are important for both the customers and the development team. The customer can evaluate the software or release to end users which is highly recommended. This evaluation provides necessary feedback to the development team.

On the other hand, it may be impossible to create good releases this often. In addition, it is an overhead for the development team to make a new release every iteration and ensure that this release is reliable and meets the customer requirements. Another thing that should be taken in consideration is that the customers might become overwhelmed with evaluating and commenting the new releases.

3. Designing

XP is an iterative methodology; therefore design is a continuous essential process.

In the designing phase, XP concentrates on keeping things as simple as possible as long as possible -**Simple Design-**. Choosing a **system metaphor** is very important for the development team to keep being organized.

XP encourages using the Use Class, Responsibilities, and Collaboration (CRC) Cards to design the system as a team. XP also encourages the use of Spike solutions to solve technical or design problems. A spike solution is a very simple program to explore potential solutions [10].

In order to keep the design simple and avoid any complexity, Re-factoring is required.

The next sub sections begin by explaining the idea of a simple design and then discuss two other practices that are the System metaphor and the Re-factoring.

3.1. Simple Design

In the designing phase, XP concentrates on keeping things as simple as possible as long as possible. No extra functionality is added early with the assumption that it might be used later on.

A simple design always saves time as it takes less time to finish. Any complex code should be replaced as soon as possible. The earlier the code is replaced the easier it is to replace it.

Simple Design has its disadvantages. As no design techniques are used and no design diagrams are produced, the development team will be missing the "big picture" of the project. This might mislead the team to developing the software in the wrong way leading to excessive re-factoring because inadequate time had been allocated to initial system design [6].

3.2. System Metaphor

System metaphor is a common vision of the project in hand. The metaphor keeps the development team organized by providing a naming convention.

A naming convention is very important as it helps understanding the overall design of the system and reuse code. It saves time as it makes it easier to find the functionality you are looking for and to know where to put certain functionality.

3.3. Re-factoring

Re-factoring is a process of continuous design improvement to keep the design as simple as possible and to avoid needless clutter and complexity.

Symptoms that indicates that re-factoring is required includes; multiple maintenance: functional changes start requiring changes to multiple copies of the same (or similar) code. Another symptom is that changes in one part of the code affect lots of other parts [10].

Re-factoring tends to removing redundancy and duplications and increasing the code cohesion while decreasing its dependences. Re-factoring throughout the entire project saves time, increase quality, and improves understandability.

Re-factoring should be supported by comprehensive testing to ensure that nothing is broken.

4. Coding

In the coding phase, XP concentrates on having **coding standards** to keep the code consistent and easy to read and re-factor.

The coding phase begins by creating test first units. This helps the developers understanding the requirements.

Pair programming is one of the practices that distinguish the XP methodology. Each pair of programmers writes their code and then integrates it together in a serial fashion.

The development team has a collective code ownership. Each team member can change or re-factor any part of the code.

In the next sub sections five practices are discussed. The role of code standards in the XP methodology, the importance of pair programming in XP, the 40-hour work week, the continuous integration, and the collective code ownership.

4.1. Coding Standards

Coding standards keeps the code consistent and easy to read and re-factor, which is very important in XP as it makes the code look as if one developer has written it. This practice supports the collective code ownership practice.

4.2. Pair Programming

Pair programming is one of the practices that distinguish the XP methodology. Each pair of programmers works together to develop certain functionality. This increases software quality.

A pair of programmers working together will have the same productivity as working separately but the outcome will have a higher quality. The better quality saves time later on in the project; therefore pair programming is considered a good investment.

Pair programming has many advantages. In addition to a better code quality, it helps with communicating knowledge and no one developer becomes a bottleneck. It also allows the programmers to share their knowledge, learn, and improve their skills.

However, pair programming might be a poor practice if done in the wrong environment. If the two programmers have different skill levels, the higher-level skill programmer might dominate and the other programmer becomes idle. Personality differences might also have impact on pair programming.

4.3. 40-Hour Work Week

A 40-hour work-week means that the developers should not work more than 40 hours per week - no overtime. This will give the developers a comfortable working environment with no pressure. In pressure times, up to one week of overtime is acceptable. Multiple weeks of overtime will exhaust the developers and reduce their productivity.

4.4. Continuous Integration

XP team should maintain a fully integrated project. The integration process should be continues and carefully controlled. Developers should integrate tested code at least daily. This should be done serially as parallel integration might lead to serious problems.

Continuous integration often avoids diverging or fragmented development efforts, where developers are not communicating with each other about what can be re-used, or what could be shared [10]. Continues integration ensures that everyone has the latest version of the project. Continuous integration also avoids or detects compatibility problems early.

4.5. Collective Code Ownership

The development team has a collective code ownership. Each team member can change or re-factor any part of the code.

Collective code ownership ensures that no one developer becomes a bottleneck for changes. It allows programmers to reuse any functionality that might be required by multiple user stories.

Collective code ownership might be difficult to implement, as it is hard to make the entire team responsible for the entire project. This practice adds an overhead that all the developers are required to have all the knowledge used in the project.

5. Testing

Test in XP comes in tow types: unit tests and customer tests.

As mentioned before the coding phase begins by creating test first units for each feature to be developed. The developed feature should pass all the test units to be considered as completed. This is called **unit testing**.

Acceptance tests are test done be the customers to ensure that the overall application contains all the required features.

In XP, it is preferable that all test carried are automated. Automated testing results in much better overall quality.

5.1. Unit Testing

Unit Tests are automated tests written by the developers during the coding phase to test features as they are developed. Each unit test typically tests only a single class, or a small cluster of classes [10].

Unit tests are very important as it can save a large amount of effort. But for approaching deadline, unit tests are sometimes skipped as it requires time to develop the unit test and run them.

Often some small changes to the code would also require that some unit tests needed to be changed or rewritten because they were to specifically tighter to the implementation [3].

5.2. Acceptance Testing

Acceptance tests are test done be the customers to ensure that the overall system contains all the required features. Acceptance tests are also used as regression tests prior to a production release [10].

The acceptance tests should be done at each of the iterations of the process to ensure that the new release contains all the features agreed upon. The acceptance test score is published to the team. It is the team's responsibility to schedule a time to fix any failed test, in every iteration [10].

6. Conclusion and Future Work

Extreme Programming (XP) is an agile software development methodology. It is a lightweight methodology combining a set of existing software development practices [5]. Each of these practices has its strengths and weaknesses.

The XP methodology has some excellent practices that have proven useful such as pair programming and unit tests.

The success of the XP methodology as a software development process depends heavily on the context of the project. XP should be implemented with projects that have a very frequent requirement changes.

It would be better if XP is supported with a traditional methodology to provide the required design diagrams and documentations.

7. References

1. Beck, K. "*Extreme Programming Explained: Embrace Change*", Addison-Wesley, Reading, MA, 2000.
2. G. Vanderburg, "*A Simple Model of Agile Software Processes –or- Extreme Programming Annealed*", ACM, 2005, pp. 539-545.
3. G. Hedin, L. Bendix, B. Magnusson, "*Introducing Software Engineering by means of Extreme Programming*" Department of Computer Science, Lund Institution of Technology, Sweden, IEEE, 2003.
4. J. Noble, S. Marshall, S. Marshall, R. Biddle, "*Less Extreme Programming*". Information Group, Victoria University of Wellington, 2004.
5. J. Schneider, L. Johnston, "*Extreme Programming at Universities- An Educational Perspective*", Swinburne University of Technology, IEEE, 2003.
6. N. LeJeune, "*Teaching Softwaer Engineering Practice with Extreme Programming*", Metropolitan State College of Denver, Department of Mathematical and Computer Science, 2005.
7. P. Grishman, D. Perry, "*Customer Relationship and Extreme Programming*", ACM, 2005.
8. R. Paige, H. Chivers, J. McDermid, Z. Stephenson, "*High-Integrety Extreme Programming*", ACM, 2005.
9. http://www.extremeprogramming.org