Extreme Programming

CMPT 475 Spring 2012

Christopher Tse | cta16@sfu.ca | 556001847

Dominic Renaud | dvr@sfu.ca | 301063898

Brandon Mak | bjm7@sfu.ca | 301058096

Darius Yao | dariusy@sfu.ca | 301008959

Table of Contents

1. Introduction

Extreme Programming is a relatively new software development methodology focused on producing software that can be quickly tested and that is as complete as possible at every step, rather than just at the end of the project. It emphasizes teamwork between the developers and the client, and adapts to client feedback on the project[3].

Unlike in the waterfall method of software development, Extreme Programming sets out the initial requirements very loosely. Scheduling of development is done on a regular basis, rather than all at the start, and the features to be developed are determined based on current priorities and customer feedback, rather than the initial plan. This allows for the flexibility to handle unexpected changes to customer requirements, or to manage misinterpretations of the requirements as initially laid out before the product has shipped.

2. Practices

There are many simple rules and practices that make up Extreme Programming. This essay will only cover a handful of them. They are what makes Extreme Programming, and as such should be followed by teams who want to develop in this methodology.

2.1. User Stories & User Acceptance Tests

In XP, requirements elicitation is done entirely though User Stories. In contrast to other methods of requirements engineering, User Stories allow the customer and programmer to overcome problems with understanding. When a customer “tells a story about how the system will be used” ([5], pg. 16), the requirements contain more background context allowing the programmers to understand the problem better. This allows programmers to ask more effective questions about the system to be developed to gain more insight into the problem. After the exchange of stories and questions the customer may feel that the initial story he or

she started with is not satisfactory and will revise it. The continued revision of the story brings initially hidden requirements into light.

If a Story is too large it is split up into smaller segments because programmers have more accurate estimates for smaller problems. As recommended by one of the founders of XP, Ron Jeffries states in his book: “stories should encompass a week or two of programmer time” ([5], pg. 18). For planning purposes this helps the development team stay as close to their estimated schedule as possible.

After the implementation is complete, Acceptance Tests are performed to verify the system indeed satisfies the system described in the user story. Like other development models, the purpose of Acceptance Tests remain the same – to allow the customer to verify the system performs as desired and to raise confidence in all the project stakeholders. However, Acceptance Tests also have another crucial role unique to XP. The XP model reduces the amount of documentation necessary by working in reverse; rather than writing the documentation first and deriving test cases from it, XP instead uses the test cases to derive what documentation is necessary. This model allows programmers to begin developing a system prior to heavy paperwork, and at the final stage only the relevant documentation will surface. Being agile in nature, XP demands that all tests – Acceptance and Unit Tests – are automated.

2.1.1 Analysis

The method of using User Stories is an interesting approach. On first impression it seems the developers may receive unnecessary information. More traditional software approaches required the customer to only specify what the system must do, and programmers were free to implement it with any desired strategy. Asking someone to “make this for me, and I don’t care how you do it” makes sense as we normally would not expect the

client to have much understanding on the technical aspects of development, and those decisions would be better left for the developer to deal with. Although User Stories take more commitment by both parties – customer and developer – due to the amount of communication that must be maintained, the process of writing and revising Stories helps the customer understand the product truly desired and helps expose details that may have initially been omitted. By the same token, when a user story explains in detail the what, where, why, who, and how, it allows the developers to make more informed decisions during the implementation stage.

The role of user Acceptance Tests strongly enforces the paradigm of XP – a model that is adaptive rather than predictive. In waterfall models, development parties invest too many resources in analyzing the system and the runtime environments to discover all potential roadblocks. It is difficult or near impossible to tabulate every possible constraint, and there are likely to be constraints that are irrelevant to the current iteration of the system. By using Acceptance Tests at each stage to uncover the requirements, only currently relevant documentation is kept. This level of abstraction also allows the development team to focus on the current User Story at hand, and this leads to a cohesive development system.

XP has a heavy focus on testing, demanding all tests be automated and running them on the system frequently. Due to XP's nature in welcoming change, it is important to have lower level modules performing at the highest quality. This allows higher level models to be switched with another when requirements change, and developers will have confidence that the lower level system will continue to maintain its stability. That is not to say lower level modules need not be tested when the system as a whole undergoes change (take Ariadne 5 for example). But as software engineers, we generally do not "reinvent the wheel", and we always use previously developed modules as a basis. For example, when we develop in a programming language, pinpointing the location of an error becomes easier when we can

trust our compiler, because we know the error is within our own code rather than underlying systems. This optimism may not be applicable to large scale projects because of the amount of coupling necessary resulting in a more complex system, but XP is not an effective methodology when the project size is large due to face to face communication losing effectiveness.

2.2. Iterative Development [10]

A core component of Extreme Programming is Iterative Development. Instead of development being scheduled across an entire release plan, it is scheduled into 1-3 week iterations, which are organized immediately before being worked on. It allows for unforeseen feature requests or requirements issues to be handled when they come up, rather than waiting until the end of the project, or requiring clairvoyant planning teams.

The structure of Iteration is fairly simple. Take the highest priority features than can be done in the iteration length ─ usually one week, but could be as long as three ─ and put them together as the iteration plan. These features are exclusively what will be done in the next iteration. If the features needed to be done seem to be taking too long, re-plan in the middle of the iteration to determine which features can be pushed back. Once the iteration is done, the current version of the product should be tested with the customer, to ensure that the project remains on track.

It is crucial that the iteration be taken as a hard deadline. While extreme programming is much more conducive to floating deadlines for the entire project, the iterations need to stay on track.

2.2.1. Analysis

The primary benefit to iterative development is that the project is being continuously revised to respond and adapt to the customer's needs. There is less concern about requirements as well, as these can be reviewed more easily once parts of the system are in place and thus there is a more concrete perspective on the system.

An element that is somewhat ambiguous is that iterations are the level at which flexibility and adaptability essentially stop. This is good because it provides structure to the project, but may be problematic if the time per iteration is too large to adapt effectively, or too small to allow for the long-term planning of a project to be useful, such as in a large team or with a military or government project, where the requirements may not be as likely to change with use. How this affects development varies greatly from project to project, and is a key aspect of extreme programming that needs to be carefully considered during the initial project planning, when determining the development methodology.

The downside to iterative development is that it can be short-sighted. While this can be mitigated for really large features by breaking them down, or simply making them be built across iterations, this may not always work. Depending on how set in stone the requirements are, constant revisions and changes may cause large or complex features to be put off continuously. It is important to ensure that features are being added to iterations based on priority as well as feasibility within the time frame.

2.3. Iteration Planning [11]

Planning in XP is primarily handled on a per-iteration basis, treating the iteration as a more solid unit than the plan for the whole release. This is the point where the flexibility of XP goes away and the plan becomes set. While there are times where features can end up rolling over to the next iteration, the iteration end date is meant to be a hard deadline.

For planning the release, there should be enough features chosen for the iteration such that the features can be completed in the time given, provided that nothing goes wrong. This should be determined by figuring out how many days it would take to do the feature if there were no distractions, referred to as ideal programming days. If it takes less than 1 day, group them together. If it takes more than 3 days, break them apart. Once you have a good idea how long each feature will take then the whole iteration can be evaluated. At this stage, if there are too many features, put off some of the lower priority ones until the following iteration. If those that exist would likely be completed within the time of the iteration with time to spare, then add more features to the iteration.

When planning iterations, it is also very important to take a look back and see how many features got completed during the previous iteration, and use that to determine how quickly features can realistically be completed, and thus how many features should be on the next iteration. It is worth keeping track of how much time was taken to build a feature compared to how much time was estimated, and also how much time was taken up by fixing bugs. Bug fixing time is less predictable, and doesn't fit into the idea of the ideal programming day. However, if a feature took longer because its completion time was poorly estimated, then that is something which needs to be considered more carefully going forward. It takes several iterations before it is possible to have an accurate estimate of how quickly features can be completed, and thus how many features can be put into a single iteration. So long as there is a critical eye to the amount of time features take, then the time taken to accurately measure how quickly work can be done is minimized.

2.4. Open work spaces

Communication is the most important aspect in any team. Teams often consist of many unique individuals with different levels of experience and strengths. If teammates do not

communicate, then they cannot utilize the full potential of the team. Open Work Spaces addresses the problem by creating an environment where everyone is comfortable to communicate [1].

An example of an Open Work Space is the Japan Airlines office. The CEO of JAL sits at a desk that is on the same floor as many other employees without any walls surrounding him. Many of his employees share desks and don't have a wall dividing them either. By working in the same environment, the gap between different levels and management are not as apparent [1]. This allows employees from all levels to comfortably communicate. Other examples of Open Work Spaces include offices of Facebook and Google.

Different people have different preferences in the way they communicate. In order to encourage interaction even further, Open Work Spaces allows many forms of communications. The various forms may include white boards, sticky notes, charts, and other forms of communication [1]. By providing different ways in which teammates can communicate, all teammates can contribute in a way that they are comfortable with [1]. For example, if a teammate is not comfortable talking in a large group, they can use sticky notes to add to any ideas. Work stations can also be individual or in a cluster to allow teammates to choose how they want to work without excluding themselves from the team [1].

2.4.1. Analysis

Open Work Spaces works in conjunction with other XP components such as Standing Meetings and Pair Programming. The Open Work Space allows everyone to participate in meetings wherever they are in the office. In addition, the location of the Standing Meeting will create a centralized location where everyone can discuss ideas. Open Work Spaces also

optimizes Pair Programming because it allows different groups of Pair Programmers to participate in other Pair Programming discussions [9].

Although Open Work Spaces has many advantages, it is also important to address potential issues that will make the team ineffective. One of these concerns is Privacy [9]. Without privacy, some team members may feel that they cannot work comfortably [9]. However, the lack of privacy will encourage the interaction between team members and remove the idea that a certain team member ‘owns’ a certain part of the project. Distractions are another issue that might affect the performance of the teammates [9]. When many groups are discussing strategies, it may be difficult for team members to concentrate. However, studies have shown that team members will adapt to this by automatically filtering out any unnecessary discussions [9].

2.5. Standing meetings [12]

An important component of any XP project is starting the day with a Standing Meeting. These need to be conducted in a fairly central location, and should be short. The point is to keep everyone up to date on what everyone is doing, and to point out any issues that may need discussion after the standup meeting.

The structure for these meetings is to go around to each of the developers, and have them talk briefly about what they did yesterday, what they plan to do today, and what obstacles they are having in completing their job. This allows other developers to understand where the team is at. It also gives opportunities to bring up quick fixes to small issues, or let someone know if they are a bottleneck on part of the project, so they can re-prioritize appropriately. It is important that these discussions are kept brief. If they require more 1-on-1 or small group discussion they should be done after the Standing Meeting is over.

It is important that these meetings are conducted regularly. If they are not then it is easy for the developers to lose a clear picture of what everyone else is working on, and thus makes future communication with team members more difficult. It also means that more meetings are needed between small groups which can eat more time than necessary if there was the Standing Meeting in the morning. The time constraint on the Standing Meeting encourages quick action and no time wasted, while meetings in smaller groups can get sidetracked by certain ideas or discussions, as they aren't necessarily as constrained.

Another important but non-critical part of Standing Meetings is a whiteboard. With a whiteboard that has the schedule for the team for the iteration cycle, there is a reference point for the activities the developers are doing. When they bring up their activity the previous day and their plans for the day, they can be compared with the whiteboard schedule, which can either cause the developer to change their priorities, or cause the whiteboard schedule to change in response to their activities, which may have been critical bug fixes or simply an increase in the amount of time it is taking to build the feature they are currently working on.

2.5.1. Analysis

The main benefit to Standing Meetings is synchronization with the team. Knowing where everyone else is in the project is good both for knowing how well the scheduling is working for the iteration, and for morale. Constant reassurance that everyone on the team is doing their job, and that they are willing to help with any obstacles that might come up for development, is a great way to keep a team together and motivated. It also ensures that any problems which come up will be resolved by the next morning at the latest.

The downside to Standing Meetings is that they do take some time, and require some discipline in remembering to do them and in keeping track of what work has been done. This isn't a major con, it just means you must maintain disciplined.

2.6. Simplicity

Software development has a phrase called “Keep It Simple Stupid” (KISS), it helps others with understanding your code and would generally make the overall program more maintainable. XP takes this a step further and includes it as a functional requirement instead of a guideline. Simple functions are easier to maintain, create, and modify.

Accompanying the new importance of Simplicity are four subjective qualities: “Testable, Understandable, Browsable, and Explainable (TUBE)”.[2]. The qualities are familiar but are now more important in XP.

Testing is a fundamental part of any software design. It is made to ensure quality and proper functionality in a product. So with every simple function being written a corresponding acceptance and Unit Test must be done. This will help with quick feedback and design changes if necessary.

Browse-ability is a somewhat newer concept, but can be summed up with naming conventions. To make a developer’s life easier it is always good to find useful names for given functions, classes, or variables such that it would help with documentation and ease of use. This also helps with inheritance. For example, if you have a base class person and you want to branch off into employees, managers, and other types it would be easily understood by others analyzing the documents.

Understandable and explainable can be grouped together. Understandability will come with time spent on a project, whereas the more important focus is the ability to explain the project ideas. For example if a person recently joined the team and has no idea what is going on. It would help if your functions and methods are easily communicated to the confused individual.

To keep things simple and easier it helps to only program functions and methods only when you need them. XP tells us to refrain from creating complex methods that achieve multiple goals. Focus on writing one method for one goal and making it understandable.

2.7. Pair Programming

Pair Programming is an essential part of XP. A team of programmers can use Pair Programming by pairing up programmers of various skill levels. Both programmers share one computer [8]. While one programmer codes, the other looks over the code for any errors as well as conceptualizing the general form of the solution [6]. The programmers should also discuss their strategy at all times to ensure that they both understand the problem specification and their own solution [6]. The two programmers can switch roles at any time.

The first advantage of Pair Programming is the increased productivity of the programmers and the improved quality of the product [6]. Pair programming can improve productivity in three ways. The programmers become more efficient because they waste less time [6]. Instead of surfing the internet or checking their email, they are motivated to work. If one person is stuck on a problem, both programmers can discuss possible solutions or switch roles. This ensures that both programmers are doing something constructive. Also, since one programmer is continuously reviewing the code, there is a greater chance that a bug will be discovered and removed [6]. This will reduce the amount of time spent testing and finding the bug. Lastly, there is a lower number of fragmented code that needs to be merged between groups.

Another advantage of Pair programming is that both programmers become better programmers after each session [6]. Every programmer is different in skill level and use different methods to solve the same problem. When programmers discuss with other

programmers their strategies, they can discover the disadvantages or advantages of solving the problem in another way. By discussing the problems, each programmer can also get a better understanding of the problem and how well their solution solves the problem. Pair programming works well with Open Work Spaces because it allows Pair Programming groups to overhear discussions from other groups [4]. This encourages communication between groups. The result is a more cohesive team and software [4].

2.7.1 Analysis

Some programmers are more experienced than others. It is easy to see how the less experienced programmer will benefit from this form of programming because the programmer with more experience can spot mistakes made by the other programmer. However, the programmer with more experience can also benefit from this form of programming because they can get a fresh perspective on a problem that they had never considered [6].

2.8. Frequent Integration

For ease of integration, developers (in pairs) during the construction phase must get in the habit of submitting code to the repository as frequent as possible. This will ensure that there will be little fragmentation in the development process and minimal code diversion. If this is not done often the inconsistencies would cause a significant increase in time spent integrating when large sections of code and functionality are added.

Frequent Integration helps identify potential conflicts earlier so that projects can be quickly redesigned and adapt to the new needs of the application. Though this may sound tedious it may prove more convenient than to try and integrate weeks of separate functions together causing fragmentation of versions and potential requirement changes done by other team of developers.

Frequent Integration is highly encouraged and a fundamental part of keeping fragmentation to a minimum. It is still good to note that a single pair of coders should be integrating their code with the full project repository per few hours of coding. Ensuring that different pairs of developers submit code at different times will prevent conflicting code testing. During the Standing Meeting there should be some discussion of estimated completion of parts of code for integration. Having a good idea of when new code is about to be integrated will help ensure future commits of code will be current and up to date.

2.8.1. Analysis

With the constant updates it will be easier to gauge how far the progress of the overall project is going. The iterative nature of the integration helps with efficiency of the overall project development. The users will have tested code more frequently and there would be less integration fixes causing delays in the development of other teams of developers. Frequent Integration also promotes communication with other pairs of developers so the entire team can make some assumptions to what is to be expected during integration.

Though the idea of Frequent Integration is simple, it requires a lot of discipline and communication to accomplish. This approach for programming would most likely thrive in small project environment involving few pairs of developers. If this was applied to a larger project involving more people, it could be hard to set times for integration due to the sheer increase in commits.

2.9 Unit Tests

As an agile paradigm, it is imperative for systems developed in XP to be as robust as possible in order to avoid lengthy delays caused by uncaught errors when development on the next iteration begins. The longer an error continues undetected, the costlier it becomes to

fix the further we are in the development cycle. XP's testing principles adhere to minimizing stalls in development and minimizing costs by rectifying errors as early as possible. This is all done during the unit test.

In XP unit tests are written simultaneously with the code. By writing unit tests before the code, you may miss some test cases that become necessary depending on your code implementation. Similarly, writing unit tests after the code is not a practical approach because you have to mentally keep track of what tests are needed at which sections of code, and it is highly likely a test case will be forgotten. When writing test cases simultaneously with the code, test cases are generated when they are immediately necessary (when writing methods for an object class for example). The lists of test cases are run all the time, and whenever a new test is formed, it is added to the list. When tests are performed rapidly, the location of an error is much easier to find – in most cases it would be located only within the most recent changes to the code. This is analogous to the strategy new developers use where they compile their code frequently whenever changes are made. The list of test cases grows extremely quickly, and that leads to why XP demands all tests be automated.

When unit tests consistently pass with 100% success, it is easier to integrate components together. In the long run, it will make newer iterations easier to develop since it has a stable codebase. Occasionally an acceptance test may reveal an error – this indicates that a test case was missing. Uncovering the missing test case may lead to new tests cases also. In XP, a product is not released until all unit and acceptance tests have 100% success rates.

Ron Jeffries frequently used the phrase "Everything That Could Possibly Break" [5] for guidance in deciding what to test. There are some methods that are simply indestructible (such as accessor methods) that don't need to be tested. Deciding what to omit from testing takes experience, but Jeffries claims "There are more things that couldn't possibly break than

you might imagine" ([5], pg. 153) and an adequate amount of time might be saved from not having to write unnecessary test cases.

2.9.1 Analysis

The strategies of unit testing are not unique to XP; they are common testing strategies employed by various methodologies to ensure a high quality product. Even non-iterative methods such as the waterfall model employ the same testing strategies. In order to reduce costs, it is important to resolve an issue as early as possible after it is introduced, and all modern software models have accommodated this into their practices.

XP development differs from iterative development in one aspect however: the error tolerance. XP strives for quality and does not allow a system release until tests pass at 100%, but in iterative development there are situations where a product is released pre-maturely because of deadlines due to marketing strategy. The errors would then be corrected in a future update. This leads me to believe that XP is ideal when the development team is working one-to-one with a client. In contrast, traditional iterative development is more practical when your consumer base is shared among other developers. When the release of a system is not only driven by quality but also competitive edge, some XP practices may have to be broken in order to gain the advantage.

3. Conclusion

Extreme Programming allows for much adaptation and review of a project before it is shipped to the customer. It can easily cut costs, improve customer satisfaction, and keep a team going strong through a project [3]. It does require some work and discipline, the

cooperation of the client, and the right team. It may not be the best methodology for every project either, but when it can be used, it should be.

4. Glossary

Acceptance Test – Allow clients to verify that the product works as intended

Frequent Integration – The habit of submitting to a repository frequently to reduce fragmentation.

Iteration Planning – Organizing the schedule and team responsibilities within each integration

Iterative Development – Scheduling multiple development cycles that consist of implementation and testing instead of spending a large amount of time testing at the end of the project.

Open Work Spaces – A large open area where team members work together.

Pair Programming – Two programmers taking turns working on a computer while discussing possible solutions to a problem.

Standing Meetings – A meeting, typically early in the day, to inform teammates of their progress.

User Stories – A method used by the client to help the programmer understand the purpose of certain features in a program

5. Bibliography

[1] Extreme Programming on Space;
http://www.extremeprogramming.org/rules/space.html

[2] Extreme Programming on Simplicity;

http://www.extremeprogramming.org/rules/simple.html 3rd paragraph.

[3] http://www.extremeprogramming.org/ 2nd and 3rd paragraph

[4] http://www.agilemodeling.com/essays/agileModelingXP.htm

[5] Extreme Programming Installed (By Ron Jeffries, Ann Anderson, Chet Hendrickson; October 16, 2000)

[6] All I Really Needed to Know about Pair Programming I learned in Kindergarten;

http://www2.yk.psu.edu/~sg3/cmpbd205/assign/week01/ACMarticlePairProgramming.pdf

[7] The Art of Agile; http://jamesshore.com/Agile-Book/pair_programming.html

[8] Extreme Programming on Pair Programming;

http://www.extremeprogramming.org/rules/pair.html

[9] Working Together in “War Rooms” Doubles Teams’ Productivity;

www.sciencedaily.com/releases/2000/12/001206144705.htm

[10] Extreme Programming on Iterative Development;

http://www.extremeprogramming.org/rules/iterative.html

[11] Extreme Programming on Iteration Planning;

http://www.extremeprogramming.org/rules/iterationplanning.html

[12] Extreme Programming on Standing Meetings;

http://www.extremeprogramming.org/rules/standupmeeting.html