A

Seminar report

On

Extreme Programming

Submitted in partial fulfillment of the requirement for the award of degree
Of Computer Science

SUBMITTED TO: **SUBMITTED BY:**

www.studymafia.org www.studymafia.org

Preface

I have made this report file on the topic **Extreme Programming**; I have tried my best to elucidate all the relevant detail to the topic to be included in the report. While in the beginning I have tried to give a general view about this topic.

My efforts and wholehearted co-corporation of each and everyone has ended on a successful note. I express my sincere gratitude to …………..who assisting me throughout the preparation of this topic. I thank him for providing me the reinforcement, confidence and most importantly the track for the topic whenever I needed it.

Content

INTRODUCTION

Extreme Programming (XP) is actually a deliberate and disciplined approach to software development. About six years old, it has already been proven at many companies of all different sizes and industries worldwide. XP is successful because it stresses customer satisfaction. The methodology is designed to deliver the software your customer needs when it is needed. XP empowers software developers to confidently respond to changing customer requirements, even late in the life cycle. This methodology also emphasizes teamwork. Managers, customers, and developers are all part of a team dedicated to delivering quality software. XP implements a simple, yet effective way to enable groupware style development.
XP improves a software project in four essential ways; communication, simplicity feedback, and courage. XP programmers communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. They deliver the system to the customers as early as possible and implement changes as suggested. With this foundation XP programmers are able to courageously respond to changing requirements and technology. XP is different. It is a lot like a jig saw puzzle. There are many small pieces. Individually the pieces make no sense, but when combined together a complete picture can be seen. This is a significant departure from traditional software development methods and ushers in a change in the way we program.

If one or two developers have become bottlenecks because they own the core classes in the system and must make all the changes, then try collective code ownership. You will also need unit tests. Let everyone make changes to the core classes whenever they need to. You could continue this way until no problems are left. Then just add the remaining practices as you can. The first practice you add will seem easy. You are solving a large problem with a little extra effort. The second might seem easy too. But at some point between having a few XP rules and all of the XP rules it will take some persistence to make it work. Your problems will have been solved and your project is under control. It might seem good to abandon the new methodology and go back to what is familiar and comfortable, but continuing does pay off in the end. Your development team will become much more efficient than you thought possible. At some point you will find that the XP rules no longer seem like rules at all. There is a synergy between the rules that is hard to understand until you have been fully immersed. This up hill climb is especially true with pair programming, but the pay off of this technique is very large. Also, unit tests will take time to collect, but unit tests are the foundation for many of the other XP practices so the pay off is very great.

XP projects are not quiet; there always seems to be someone talking about problems and solutions. People move about, asking each other questions and trading partners for programming. People spontaneously meet to solve tough problems, and then disperse again. Encourage this interaction, provide a meeting area and set up workspaces such that two people can easily work together. The entire work area should be open space to encourage team communication. The most obvious way to start extreme programming (XP) is with a new project. Start out collecting user stories and conducting spike solutions for things that seem risky. Spend only a few weeks doing this. Then schedule a release planning meeting. Invite customers, developers, and managers to create a schedule that everyone agrees on. Begin your iterative development with an iteration planning meeting. Now you're started.

Usually projects come looking for a new methodology like XP only after the project is in trouble. In this case the best way to start XP is to take a good long look at your current software methodology and figure out what is slowing you down. Add XP to this problem first. For example, if you find that 25% of the way through your development process your requirements specification becomes completely useless, then get together with your customers and write user stories instead.

THE XP PRACTICES

Extreme programming implements several software development practices. But not all of them are exclusive to XP. These practices work well and contribute significantly to the projectâ„¢s overall success. There are 12 practices in XP.

1. Planning Game
2. Small releases
3. Metaphor
4. Simple design
5. Test Driven Development
6. Refactoring
7.Pair programming
8. Collective ownership
9.Continuous development
10.Sustainable pace
11.On-site customer
12. Coding standards

USER STORIES

User stories serve the same purpose as use cases but are not the same. They are used to create time estimates for the release planning meeting. They are also used instead of a large requirements document. User Stories are written by the customers as things that the system needs to do for them. They are similar to usage scenarios, except that they are not limited to describing a user interface. They are in the format of about three sentences of text written by the customer in the customer's terminology without techno-syntax.

User stories also drive the creation of the acceptance tests. One or more automated acceptance tests must be created to verify the user story has been correctly implemented.

One of the biggest misunderstandings with user stories is how they differ from traditional requirements specifications. The biggest difference is in the level of detail. User stories should only provide enough detail to make a reasonably low risk estimate of how long the story will take to implement. When the time comes to implement the story developers will go to the customer and receive a detailed description of the Developers estimate how long the stories might take to implement. Each story will get a 1, 2 or 3-week estimate in "ideal development time". This ideal development time is how long it would take to implement the story in code if there were no distractions, no other assignments, and you knew exactly what to do. Longer than 3 weeks means you need to break the story down further. Less than 1 week and you are at too detailed a level, combine some stories. About 80 user stories plus or minus 20 is a perfect number to create a release plan during release planning.

Another difference between stories and a requirements document is a focus on user needs. You should try to avoid details of specific technology, data base layout, and algorithms. You should try to keep stories focused on user needs and benefits as opposed to specifying GUI layouts.

ACCEPTANCE TESTS

Acceptance tests are created from user stories. During an iteration the user stories selected during the iteration planning meeting will be translated into acceptance tests. The customer specifies scenarios to test when a user story has been correctly implemented. A story can have one or many acceptance tests, what ever it takes to ensure the functionality works.
Acceptance tests are black box system tests. Each acceptance test represents some expected result from the system. Customers are responsible for verifying the correctness of the acceptance tests and reviewing test scores to decide which failed tests are of highest priority. Acceptance tests are also used as regression tests prior to a production release.
A user story is not considered complete until it has passed its acceptance tests. This means that new acceptance tests must be created each iteration or the development team will report zero progress. Quality assurance (QA) is an essential part of the XP process. On some projects QA is done by a separate group, while on others QA will be integrated into the development team itself. In either case XP requires development to have much closer relationship with QA.
Acceptance tests should be automated so they can be run often. The acceptance test score is published to the team. It is the team's responsibility to schedule time to each iteration to fix any failed tests. The name acceptance test was changed from functional tests. This better reflects the intent, which is to guarantee that customer requirements have been met and the system is acceptable.

UNIT TESTING
Unit testing is a development practice and part of the extreme programming (XP) methodology. But this does not depend on other XP practices; it may be used with other software development process. Unit testing is not done by specialized testers; it is a part of the daily development routine of a programmer. Unit testing means testing a unit of code. A unit of a code is generally a class in an object oriented system. It could also be a component or any other piece of related code. A unit test is an automated test, where one or more methods of one or more classes are invoked to create an observable result.
Automated means that the results are verified automatically. The tests are usually written in the same computer language as the production code. The test verification is therefore code, which compares actual results to the expected results. If a result is unexpected, the test fails. This kind of testing is in contrast to the usual print lines mixed into the production code, where the programmer looks at the output on the command line and tries to figure out whether the code behaves as expected.

Unit tests are written in java using JUnit. JUnit is a testing framework written in Java. JUnit defines how to structure test cases and provides tools to run them. The tests are usually executed in VisualAge.
Principle methods of a class should be tested. Tests should write for the cases that are critical. Writing a test guarantees that if a developer changes the code inn an unanticipated way, the error will be detected immediately when the tests are run. Confidence in the code is enhanced if all tests are still running after a day of coding. Each test should be independent of other tests. This reduces the complexity of the tests and it avoids false alarms because of unexpected side effects. One test method tests one or more methods of production code the developer should write the unit tests while (or before or after) developing the production code.

RELEASE PLANNING

A release planning meeting is used to create a release plan, which lays out the overall project. The release plan is then used to create iteration plans for each individual iteration. It is important for technical people to make the technical decisions and business people to make the business decisions. Release planning has a set of rules that allows everyone involved with the project to make their own decisions. The rules define a method to negotiate a schedule everyone can commit to.
The essence of the release planning meeting is for the development team to estimate each user story in terms of ideal programming weeks. An ideal week is how long you imagine it would take to implement that story if you had absolutely nothing else to do. No dependencies, no extra work, but do include tests. The customer then decides what story is the most important or has the highest priority to be completed.
User stories are printed or written on cards. Together developers and customers move the cards around on a large table to create a set of stories to be implemented as the first (or next) release. A useable, testable system that makes good business sense delivered early is desired. You may plan by time or by scope. The project velocity is used to determine either how many stories can be implemented before a given date (time) or how long a set of stories will take to finish (scope). When planning by time multiply the number of iterations by the project velocity to determine how many user stories can be completed. When planning by scope divide the total weeks of estimated user stories by the project velocity to determine how many iterations till the release is ready.

SYSTEM METAPHOR

Choose a system metaphor to keep the team on the same page by naming classes and methods consistently. What you name your objects is very important for understanding the overall design of the system and code reuse as well. Being able to guess at what something might be named if it already existed and being right is a real time saver. Choose a system of names for your objects that everyone can relate to without specific, hard to earn knowledge about the system. For example the Chrysler payroll system was built as a production line. At another auto manufacturer car sales were structured as a bill of materials. There is also a metaphor known as the naive metaphor which is based on your domain itself. But don't choose the naive metaphor unless it is simple enough.

ITERATION

Iterative Development adds agility to the development process. Divide development schedule into about a dozen iterations of 1 to 3 weeks in length. Keep the iteration length constant through out the project. This is the heartbeat of your project. It is this constant that makes measuring progress and planning simple and reliable in XP

Do not schedule programming tasks in advance. Instead have an iteration planning meeting at the beginning of each iteration to plan out what will be done. Just-in-time planning is an easy way to stay on top of changing user requirements. It is also against the rules to look ahead and try to implement anything that it is not scheduled for this iteration. There will be plenty of time to implement that functionality when it becomes the most important story in the release plan.

It is important to take iteration deadlines seriously. Track progress during an iteration. If it looks like you will not finish all of your tasks then call another iteration planning meeting, re-estimate, and remove some of the tasks. Concentrate your effort on completing the most important tasks as chosen by your customer, instead of having several unfinished tasks chosen by the developers.
It may seem silly if your iterations are only one week long to make a new plan, but it pays off in the end. By planning out each iteration as if it was your last you will be setting yourself up for an on-time delivery of your product.

BUGS

When a bug is found tests are created to guard against it coming back. A bug in production requires an acceptance test be written to guard against it. Creating an acceptance test first before debugging helps customers concisely define the problem and communicate that problem to the programmers. Programmers have a failed test to focus their efforts and know when the problem is fixed.
Given a failed acceptance test, developers can create unit tests to show the defect from a more source code specific point of view. Failing unit tests give immediate feedback to the development effort when the bug has been repaired. When the unit tests run at 100% then the failing acceptance test can be run again to validate the bug is fixed.

PROJECT VELOCITY

The project velocity (or just velocity) is a measure of how much work is getting done on your project. To measure the project velocity you simply add up the estimates of the user stories that were finished during the iteration. It's just that simple. You also total up the estimates for the tasks finished during the iteration. Both of these measurements are used for iteration planning. During the iteration planning meeting customers are allowed to choose the same number of user stories equal to the project velocity measured in the previous iteration. Those stories are broken down into technical tasks and the team is allowed to sign up for the same number of tasks equal to the previous iteration's project velocity.

This simple mechanism allows developers to recover and clean up after a difficult iteration and averages out estimates. Your project velocity goes up by allowing developers to ask the customers for another story when their work is completed early and no clean up tasks remain.

A few ups and downs in project velocity are expected. You should use a release planning meeting to re-estimate and re-negotiate the release plan if your project velocity changes dramatically for more than one iteration. Expect the project velocity to change again when the system is put into production due to maintenance tasks. Project velocity is about as detailed a measure as you can make that will be accurate. Don't bother dividing the project velocity by the length of the iteration or the number of people. This number isn't any good to compare two project's productivity. Each project team will have a different bias to estimating stories and tasks, some estimate high, some estimate low. It doesn't matter in the long run. Tracking the total amount of work done during each iteration is the key to keeping the project moving at a steady predictable pace.

The problem with any project is the initial estimate. Collecting lots of details does not make your initial estimate anything other than a guess. Worry about estimating the overall scope of the project and get that right instead of creating large documents. Consider spending the time you would have invested into creating a detailed specification on actually doing a couple iterations of development. Measure the project velocity during these initial explorations and make a much better guess at the project's total size.

REFACTORING

Refactoring allows us to revise the structure or the design of an existing piece of software so that it becomes easier to add or modify functionality. It is important to be able to revise the software design in this manner because otherwise only additions or modifications that are in some sense consistent with the original design can be made. Individual refactoring steps should be small and each should be checked to avoid bugs. A key feature of refactoring method is that after each refactoring step we test the modified software to ensure that the software continues to work as it did before the refactoring.
Refactorings can be classified into two groups, syntactic and semantic.
A syntactic refactoring step is a change in some essentially syntactic aspect of the piece of the software. An important point to note about such Refactorings is that essentially all of the needed work can be carried out by Refactoring Browsers and if this is done there is no need to carry out a test following the refactoring because the browser would have made sure that all the necessary changes have been properly carried out.
Semantic Refactorings modify the logic of the software and how it functions. Hence the browsers will not be able to make the needed changes. The designer has to do it. Following such a refactoring, the designer will have to test to ensure that the refactoring has been done correctly.

DISTINGUISHING FEATURES OF XP

XP is distinguished from other methodologies by:
Its early and concrete and continuing feedback from short cycles.
Its incremental planning approach which quickly comes up with an overall plan that is expected to evolve through the life of the project.Its ability to flexibly schedule the implementation of functionality, responding to changing business needs.Its reliance on automated tests written by programmers and customers to monitor the progress of development, to allow the system to evolve and to catch defects easily.
Its reliance on oral communication, tests and source code to communicate system structure and intent.Its reliance on an evolutionary design process that lasts as long as the system lasts.
Its reliance on the close collaboration of programmers with ordinary skills
Its reliance on practices that work with both short-term instincts of programmers and the long-term instincts of the project.
Risk management: the basic problems faced by all projects are schedule slips, project canceled, defect rate, business misunderstood, business changes and so on. XP address risks at all levels of development process.

CONCLUSION

Extreme Programming is a software engineering process and philosophy based on well-known practices. Extreme Programming tries to make things happen in ways that people find natural and pleasant. In the case of documentation, this is accomplished by recognizing that the point is communication, not simply documentation, then using the most effective forms of communication, and the most automatic forms, wherever possible. XP is different from established software processes. It also provides new challenges and skills as designers need to learn how to do a simple design, how to use refactoring to keep design clean and how to use patterns in an evolutionary style.

REFERENCES

www.studymafia.org
www.google.com
www.wikipedia.com