ADVANCED PROGRAMING PRACTICES

Extreme Programming

Extreme programming

- Extreme Programming (XP) is a method or approach to software engineering and a precursor of several agile software development methodologies.
- Formulated by Kent Beck, Ward Cunningham, and Ron Jeffries.
- Kent Beck wrote the first book on the topic, Extreme programming explained: Embrace change, published in 1999.
- The second edition of the book, which appeared in 2005, delves more into the philosophy of Extreme Programming and describes it as being:
  - a mechanism for social change
  - a style of development
  - a path to improvement
  - an attempt to reconcile humanity and productivity
  - a software development discipline

Extreme programming

- The twelve key features of XP, outlined below in Beck's words, are:
  - **The Planning Game:** Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes, update the plan.
  - **Small Releases:** Put a simple system into production quickly, then release new versions on a very short cycle.
  - **System Metaphor:** Guide all development with a simple shared story of how the whole system works.
  - **Simple Design:** The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.
  - **Testing:** Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write user stories for desired features that the system must demonstrate to expose.
  - **Refactoring:** Programmers restructure the system without changing its behaviour to remove duplication, improve communication, simplify, or add flexibility.

Extreme programming

- **Pair Programming:** All production code is written with two programmers at one workstation.
- **Collective Ownership:** Anyone can change code anywhere in the system at any time.
- **Continuous Integration:** Integrate and build the system many times a day, every time a task is completed.
- **Sustainable Pace:** Work no more than 40 hours a week as a rule. Never allow overtime for two weeks in a row.
- **On-site Customer:** Include a real, live customer on the team, available full-time to answer questions.
- **Coding Standards:** Programmers write all code in accordance with rules emphasizing communication throughout the code.

- These ideas are not new. They have been tried before and there have been many reports of failure. The point of XP is that, taken **together**, these techniques do constitute a workable methodology.

Key features of Extreme Programming

Key Feature: The Planning Game

- Description
  - The long term build plan determines the general goals of each successive builds. It is not detailed, and it can be changed as required. It can be changed by either the customer or developers, depending on the situation.
  - The short term detailed plan determines what exactly needs to be done in next few days. It is re-evaluated every day as development evolves.
- Potential drawbacks
  - A rough plan is not a sufficient basis for detailed development. Constantly updating the plan may be inefficient and may confuse customers and developers.
- Why it works in XP – supporting key features
  - The build plan is sufficient to give the clients a vision of what you will achieve.
  - Small releases enables the team to concentrate on immediate goals in the detailed plan. The build plan determines the goals of each successive small release.
- How we do it in the course project
  - Make a build plan that determines what will be the goals of each successive builds.
  - As each build starts, make a detailed plan for the current build, assign tasks.
  - Meet regularly and update the plan according to the latest developments.
  - Make the updated plan available to everybody in the team.

Key Feature: Small Releases

- Description
  - A release is a working version of the software. Between releases, the software may be in an inconsistent state. “Small” releases mean obtaining a working version every week, or every month, rather than every six months, or every year.
- Potential drawbacks
  - Small releases mean that time is spent on getting the releases to work perfectly.
  - May not be necessary if the client does not need/want the intermediate builds to be delivered and used operationally.
- Why it works in XP – supporting key features
  - Planning focuses immediate attention on the most important parts of the system, so even small releases are useful to customers.
  - With continuous integration, assembling a release does not take much effort.
  - Frequent testing reduces the defect rate and release testing time.
  - The design is simple, thus easier/faster to implement but may be elaborated later.
- How we do it in the course project
  - Each predefined build is a small release.

Key Feature: System Metaphor

- Description
  - The system metaphor is a “story” about the system. It provides a framework for discussing the system and deciding whether features are appropriate. A well-known simple example of a metaphor is the Xerox “desktop” metaphor for user-interface design. Another is the “spreadsheet” metaphor for accounting. Games are their own metaphor: knowledge of the game helps to define the program.
- Potential drawbacks
  - A metaphor may not have enough detail. It might be misleading or even become wrong if it is not updated.
- Why it works in XP – supporting key features
  - Small releases provide quick feedback from real code to support the metaphor.
  - Clients and developers know the metaphor and can use it as a basis for discussion.
  - Frequent refactoring are made within the practical implications of the metaphor.
- How we do it in the course project
  - The initial project description is the system metaphor.
  - We constantly refer to it as we discuss the details of the project.

Key Feature: Simple Design

- Description
  - A simple design is an outline for a small portion of the solution to be implemented.
  - Has the smallest number of features that meet the requirements of current phase and does not incorporate solutions to the requirements of the upcoming phases.
  - Overly complicated designs end up having unused features that become hindrance.
- Potential drawbacks
  - A simple design may have faults and omissions.
  - Implementing an overly simple design might turn out to be too simple eventually.
  - Components with simple designs might not integrate correctly into the system.
- Why it works in XP – supporting key features
  - Refactoring allows you to correct design errors and omissions, or adapt an overly simple design to the next build's projected features.
  - The metaphor helps to keep the design process on track with the overall picture.
  - Pair programming helps to avoid mistakes and to anticipate design problems.
- How we do it in the course project
  - Every time a solution is proposed, it should be debated as to whether it is the simplest solution that can meet the required features.
  - Overly complex designs should be avoided as a team principle.

Key Feature: Testing

- Description
  - Write large numbers of simple tests. Provide a fully automated testing process.
  - Unit tests are automated tests that test the functionality of individual methods.
  - Unit tests are written before the eventual code is written. This approach stimulates the programmer to think about conditions in which their code could fail. The programmer is finished with the testing of a certain piece of code when they cannot come up with any further condition in which the code may fail.
- Potential drawbacks
  - Writing tests is time consuming. Time spent on testing must be justified.
  - In larger projects/companies, programmers don't write tests — testing teams do.
- Why it works in XP – supporting key features
  - Simple design implies that the tests should be simple too.
  - With pair programming, one partner can think of tests while the other is coding.
  - Seeing tests successfully run is good for the team's morale.
  - Increases the client's confidence.
  - There are many tests and most of them are run automatically.
- How we do it in the course project
  - Unit tests must be delivered with each build.

Key Feature: Refactoring

- Description
  - Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.
- Potential drawbacks
  - Refactoring takes time, may be hard to control and justify, and may be error-prone.
  - Refactoring is change. Change often introduces new problems.
- Why it works in XP – supporting key features
  - Collective ownership makes refactoring easier as anybody can change any code.
  - Coding standards facilitates the task of refactoring by making code easier to understand.
  - Pair programming makes refactoring less risky and based more on consensus.
  - You have a set of tests that you can run at any time during the refactoring process.
  - Continuous integration gives rapid feedback about refactoring problems.
- How we do it in the course project
  - After each build is delivered, have a meeting and decide what parts need to be cleaned up before development continues on the next build.
  - Make sure that test run successfully before and after each individual refactoring operation.

Key Feature: Pair Programming

- Description
  - Pair Programming means that all code is produced by two people programming on one task as a team. One programmer has control over the programming workstation and is thinking mostly about the coding in detail.
  - The other programmer is more focused on the big picture, continually reviewing the code that is being produced, as well as researching solutions.
  - The pairs are not fixed: it's recommended that programmers try to mix as much as possible, so that everybody can become familiar with the whole system.
- Potential drawbacks
  - Pair programming if not done properly may be inefficient.
- Why it works in XP – supporting key features
  - Coding standards avoids trivial arguments.
  - Simple design, refactoring, and writing tests together helps to avoid misunderstanding and make decisions based on consensus.
  - Both members of the pair are familiar with the metaphor.
  - If one partner knows a lot more than the other, the second person learns quickly.
- How we do it in the course project
  - Work in pairs and make sure both individuals know their responsibilities as part of the pair.

Key Feature: Collective Ownership

- Description
  - Anyone can make changes to any part of the code.
  - This contrasts with traditional processes, in which each piece of code was "owned" by an individual or a small team who has complete control over it and access to it.
  - Speeds up the development process, because if an error occurs in the code, any programmer may fix it rather than wait for it to be fixed and/or have arguments about how/why to fix the code.
- Potential drawbacks
  - Problematic if change are applied without caution.
- Why it works in XP – supporting key features
  - Continuous integration avoids large scale code breakdowns.
  - Continuously writing and running tests warns about breakdowns.
  - Pair programmers are less likely to break code than individual programmers.
  - Coding standards avoid trivial arguments.
  - Knowing that other people are reading your code makes you work better.
  - Complex components are simplified as people understand them better.
- How we do it in the course project
  - Setup a software repository and enforce that it is used as frequently as possible.

Key Feature: Continuous Integration

- Description
  - The system is assembled very frequently, perhaps several times a day.
  - Not to be confused with short releases, in which a new version with new features is built and delivered.
  - In order to validate integration, newly integrated system can be compiled and tested.
- Potential drawbacks
  - Each integration can be difficult if different programmers are going in different directions or changing existing code without consulting other programmers.
- Why it works in XP – supporting key features
  - Tests are run automatically and quickly, so that errors introduced by integration are detected quickly.
  - Refactoring maintains good structure, reduces the chance of conflicts in integration.
  - Simple designs can be integrated quickly.
- How we do it in the course project
  - Enforce the practice of frequent commits.
  - Activate a continuous integration framework on the repository, e.g. to enforce that any code committed actually compiles and passes all tests.

Key Feature: Sustainable Pace

- Description
  - Many software companies require large amounts of overtime: programmers work late in the evening and during weekends, even more when approaching deadlines.
  - They get over-tired, make silly mistakes, get irritable, and waste time in petty arguments, and eventually are more likely to fall sick or go away.
  - This XP policy ensures that no one works too hard.
- Potential drawbacks
  - Sustainable pace is often not enough to obtain the productivity required for competitive software development.
- Why it works in XP – supporting key features
  - Planning increases the value per hour of the work performed; less wasted time.
  - Planning and testing reduces the frequency of unexpected surprises that lead to complex problems to be solved that requires many hours of work.
  - XP as a whole helps the team to work more rapidly and efficiently.
- How we do it in the course project
  - Distribute work evenly across people and over time. Do not wait until the last few days to work day and night.
  - Implement practices that ensure efficient usage of time.

Key Feature: On-site Customer

- Description
  - A representative of the client's company works at the developer's site all the time.
  - The client is available all the time to consult with developers and monitor the development of the software.
- Potential drawbacks
  - The representative would be more valuable working at the client's company.
- Why it works in XP – supporting key features
  - Clients can contribute, e.g. by writing user stories or commenting on tests.
  - Rapid feedback for programmer questions is valuable.
  - XP focuses on efficiency, which includes efficient of communication with the client.
- How we do it in the course project
  - Discussions about the project during lectures.
  - Contact the instructor any time for clarifications.

Key Feature: Coding Standards

- Description
  - All code written must follow defined conventions for layout, variable names, file structure, documentation, etc. The team agrees to and adopts a group of coding conventions, then ensures that they are followed by everyone.
- Potential drawbacks
  - Programmers can be individualists and refuse to be told how to write their code.
  - Can be overdone and thus be wasteful of time.
- Why it works in XP – supporting key features
  - Coding standards lead to more understandable code, which is required for pair programming, continuous integration, testing, and productivity in general.
  - Refactoring can be used to enforce conformance to coding standards between builds.
- How we do it in the course project
  - Use a predefined set of coding conventions.
  - Keep focused on a simple, reduced set of conventions.
  - Use a documentation generation software (e.g. Javadoc).
  - Use an IDE's automatic code formatting facility.

Values of Extreme Programming

XP value: Communication

- Building software systems requires, for example:
  - communicating system requirements to the developers of the system
  - communicating the software interfaces (APIs) to fellow developers.
- In formal software development methodologies, this task is accomplished through precise and standard documentation.
- Extreme programming techniques can be viewed as methods for efficiently building and disseminating institutional knowledge among members of a development team without relying on heavy documentation.
- The goal is to give all developers a shared view of the system which matches the view held by the users of the system. The developers and users should come to understand and use each other's terms and language.
- To this end, extreme programming favors simple design, common metaphors, collaboration of users and programmers, frequent verbal communication, and feedback.
- During coding, automated code documentation tools (e.g. Doxygen, Javadoc) and coding standards can be used to facilitate communication between developers.
- When discussing the project with the instructor, learn and use correct terminology.

XP value: Simplicity

- Extreme programming encourages implementing the simplest solution. Extra functionality can then be added later when it becomes a need.
- The difference between this approach and more conventional system development methods is the focus on designing and coding for the needs of today instead of those of tomorrow, next week, or next month.
- Coding and designing for uncertain future requirements implies the risk of spending resources on something that might not be needed, while perhaps delaying crucial features that are needed now.
- Often summed up as the "*You aren't gonna need it*" (YAGNI) approach.

- A simple design with very simple and neat code could be easily understood by most programmers in the team, promoting efficiency.
- When many simple short steps are made, the customer and the developers have more control and more frequent and precise feedback over the development process and the system that is being developed.

XP value: Feedback

- **Feedback from the system**: by writing unit tests, or running tests during continuous integration, the programmers have direct feedback from the state of the system after implementing new code or changes to existing code.

- **Feedback from the customer**: The functional tests are provided by the customer and the testers. They will get concrete feedback about the current state of their system. This review is planned once in every two or three weeks during the delivery of each build so the customer can easily steer the development.

- **Feedback from the team**: When customers come up with new requirements in the planning game the team directly gives an estimation of the time that it will take to implement.
- Feedback is closely related to communication and simplicity.
- Flaws in the system are easily communicated by writing a unit test that proves a certain piece of code will not misbehave in a specific case.
- The direct feedback from the system tells programmers to fix this part.
- A customer is able to test the system periodically according to the functional requirements, simplified as user stories.

XP value: Courage

- Several XP practices require courage.
- Courage to:
  - Change one's habits.
  - Admit one's own mistakes or shortcomings.
  - Have one's work constantly actively questioned.
- One is to always design and code for today and not for tomorrow. This is an effort to avoid getting bogged down in overly complicated design and concentrate on what is required now.
- Courage enables developers to feel comfortable with refactoring their code when necessary. This means reviewing the existing system and modifying it so that future changes can be implemented more easily.
- Continuous integration forces all individuals to confront their own code with the main body of code, which might uncover design flaws or omissions.
- Pair programming forces individuals to uncover their lack of knowledge or erroneous code to their peers as they are working in pairs.
- Courage is required when code needs to be thrown away: courage to remove source code that is obsolete, no matter how much effort was used to write it.

XP value: Respect

- The respect value includes respect for others as well as self-respect.
- Programmers should never commit changes that break compilation, that make existing unit-tests fail, or that otherwise delay the work of their peers.
- Members respect their own work by always striving for high quality and seeking for the best design for the solution at hand through refactoring, and to follow coding standards.
- Adopting good values leads to respect gained from others in the team. Nobody on the team should feel unappreciated or ignored if they embrace the values common to the team.
- This ensures a high level of motivation and encourages loyalty toward the team and toward the goal of the project.
- This value is very dependent upon the other values, and is very much oriented toward people in a team.

XP value: Embracing change

- The principle of embracing change is about not working against changes but embracing them.
- For instance, if at one of the iterative meetings it appears that the customer's requirements have changed dramatically, programmers are to embrace this and plan the new requirements for the next iteration.
- If the design of the system shows significant flaws that are hampering further development, its change should be embraced through redesign and refactoring.
- When encountering unit testing failures or integration problems, one should see this as an opportunity to improve the system.

References

References

- Kent Beck. Extreme programming explained: Embrace change, Addison-Wesley, ISBN 0201616416
- Kent Beck and Martin Fowler. Planning Extreme Programming, Addison-Wesley, ISBN 0201710919
- Martin Fowler. Refactoring: Improving the Design of Existing Code, Addison-Wesley, ISBN 0201485672
- Ken Auer and Roy Miller. Extreme Programming Applied: Playing To Win, Addison-Wesley, ISBN 0201616408
- Ron Jeffries, Ann Anderson and Chet Hendrickson. Extreme Programming Installed, Addison-Wesley, ISBN 0201708426
- Kent Beck. Extreme programming explained: Embrace change, Second Edition, Addison-Wesley, ISBN 0321278658
- Kent Beck, Cynthia Andres. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 2004. ISBN-13: 978-0321278654
- Matt Stephens and Doug Rosenberg. Extreme Programming Refactored: The Case Against XP, Apress, ISBN 1590590961

References

- Beck, Kent. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston. 1999. ISBN-13: 978-0201616415
- Fowler, Martin. *Is Design Dead?* In *Extreme Programming Explained*, G. Succi and M. Marchesi, ed., Addison-Wesley, Boston. 2001. ISBN 0-201-71040-4
- M. Stephens, D. Rosenberg. *Extreme Programming Refactored: The Case Against XP*. Apress L.P., Berkeley, California. 2003.
- McBreen, P. *Questioning Extreme Programming*. Addison-Wesley, Boston. 2003.
- Riehle, Dirk. *A Comparison of the Value Systems of Adaptive Software Development and Extreme Programming: How Methodologies May Learn From Each Other*. Appeared in *Extreme Programming Explained*, G. Succi and M. Marchesi, ed., Addison-Wesley, Boston. 2001. ISBN 0-201-71040-4