By Roy Tan and Jiang Shu

CS 6704, Spring 2002

Extreme Programming
Contents

- What is Extreme Programming (XP)?
- Do we need yet another software methodology?
- When to use XP?
- XP’s rules and practices
- XP’s relation to UML and Design Patterns
- Implementation details, critiques, etc.
What is XP?

- No bungee cords
- Being applied to Credit Swiss Life, First Union National Bank, etc.
- About six years old (originated by Kent Beck and Ward Cunningham)
- Extreme Programming is a deliberate and disciplined approach to software development.
What is XP? (cont. 1)

XP stresses customer satisfaction; “deliver the software your customer needs when it is needed”

XP emphasizes teamwork

XP improves a software project in four essential ways:

- Communication
- Simplicity
- Feedback
- Courage

XP emphasizes teamwork work

XP stresses customer satisfaction; “deliver the software your customer needs when it is needed”
XP enables us to embrace change

- Cut the cost: $100,000 vs. $2,000,000

- Less bugs

- Attitude: XP programmers have towards changing requirements while the code is written, and after the code is written, while tests are created before the code is written.

- What customers will notice:
When to use XP

XP was created in response to problem domains whose requirements change. XP was also set up for small groups of programmers.

XP was created in response to problem domains whose requirements change. XP was also set up for small groups of programmers.

XP requires an extended development team. 30 have reported success.

Between 2 and 12, though larger projects of project risk.

XP is set up for small groups of programmers.

XP was created in response to problem domains whose requirements change.
Do we need yet another software methodology?

XP is an important new methodology for two reasons:

1. It is one of the several new lightweight software development practices that have become standard.
2. First and foremost, it is re-examination of software methodologies.

XP goes one step further and defines a process that is simple and enjoyable.
A software methodology is the set of rules and practices used to create computer programs. A heavyweight methodology has many rules, practices, and documents. It requires discipline and time to follow correctly. A lightweight methodology has only a few rules and practices or ones which are easy to follow. What is a Lightweight Methodology?
In the late 1960s and early 1970s it was common practice for computer programmers to create software any way they could. In 1968 Edsger Dijkstra wrote a letter to CACM entitled "GOTO Statement Considered Harmful." The 1980s were good times. It seemed like if we could just create enough rules to cover the problems we encountered we could create perfect software. Now in the 21st century we find these rules are hard to follow, procedures are complex and not well understood and the amount of documentation written is way out of control.
We don't want to forget what we have learned.

We can choose to keep the rules that help us create quality software and throw away those that hinder our progress.

Extreme Programming is one of several new lightweight methodologies.

We can simplify those rules that seem too complex to follow correctly.

We can simplify those rules that seem too complex.
It is a lot like a jigsaw puzzle. There are many small pieces.

Individually the pieces make no sense, but when combined together a complete picture can be seen.
<table>
<thead>
<tr>
<th>Coding Standards</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-site Customer</td>
<td>Continuous Testing</td>
</tr>
<tr>
<td>40-Hour Work Week</td>
<td>Simple Design</td>
</tr>
<tr>
<td>Continuous Integration</td>
<td>System Metaphor</td>
</tr>
<tr>
<td>Collective Code Ownership</td>
<td>Small Release</td>
</tr>
<tr>
<td>Pair Programming</td>
<td>Planning Game</td>
</tr>
</tbody>
</table>

There are total 12 core practices.

Rules and Practices (cont. 1)
Rules and Practices (cont. 2)
Planning Game

Business and development cooperate to produce the maximum business value as rapidly as possible. The planning game is always the same:

1. Business comes up with a list of desired features for the system. Each feature is written out as a User Story, which gives the feature a name and describes in broad strokes which system each feature is written out as.

2. Development estimates how much effort each story will take, and how much effort the team can produce in a given iteration (the iteration).

3. Business then decides which stories to implement in what order, and when and how often to produce a production release of the system.
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 metaphor to keep the team
Refactoring

We continue to use and reuse code that is no longer maintainable because it still works in some way and we are afraid to modify it. XP argues: When we remove redundancy, long-term maintainability because it still works in no trivial aspect.

Refactoring throughout the entire project life cycle saves time and increases quality. You can do this with confidence that you didn't break anything because you have the tests.
Reference

- [Extreme Programming Roadmap](http://www.jera.com/techinfo/xpfaq.html)
- [O'Reilly Open Source Convention](http://linux.oreillynet.com/pub/a/linux/2001/05/04/xp_int.html)
- John Brewer and Jera Design "Extreme Programming FAQ"
Pair Programming
A pair of programmers work together:

- Two people, 1 keyboard
- Person at keyboard thinks "tactics"
- Person not at keyboard thinks "strategy"
- The programming pairs switch often
Table 1: Percentage of Test Cases Passed on Average

<table>
<thead>
<tr>
<th>Collaborative Teams</th>
<th>Individuals</th>
</tr>
</thead>
<tbody>
<tr>
<td>94.4%</td>
<td>78.1%</td>
</tr>
<tr>
<td>87.1%</td>
<td>70.4%</td>
</tr>
<tr>
<td>88.6%</td>
<td>78.1%</td>
</tr>
<tr>
<td>86.4%</td>
<td>73.4%</td>
</tr>
</tbody>
</table>

Program 1
Program 2
Program 3
Program 4
Pair programming in the classroom works with human nature. Programmers like to "show off" and slack off less when working in pairs than when working alone. 90% say they enjoyed working in pairs better.
Testing is important

- Nobody wants to write tests
- Write only tests for things that might break
- Tests should be:
  - Automatic
  - Isolated
Unit Testing

- Write unit tests first
- All code must have unit tests
- All unit tests must pass before code is released
- When a bug is found, a test is created for it
Who Writes The Tests?

- Programmer – Unit Tests – Must pass 100%
- Customers – Write tests from user stories – Customers write acceptance tests – May be less than 100%
- Dedicated testing person – Write tests from user stories – Customers write acceptance tests – Customers
- Must pass 100%
- Unit Tests – Programmer

Who Writes The Tests?
Putting it together
Implementing XP

1. Pick your worst problem
2. Solve it the XP way
3. When it’s no longer your worst problem, repeat

Testing is a good place to start
XP and UML

- Once "picture" is codified, throw it away.
- Do not think the design is done.
- Do not attempt to be comprehensive.
- Use UML to communicate, do not diagramming notation.
- Kent Beck is uncomfortable with formal.
- XP is centered around code.
Design Patterns provides targets for your refactoring [Gamma]. Form possible IMPLEMENT THE PATTERN IN THE SIMPLEST EARLY WHEN TO APPLY DESIGN PATTERNS: NOT TOO "YOU Arent gonna need it" [Beck].

XP and Design Patterns
References

http://extremeprogramming.org

