

Subclassing XP: Breaking its rules the right way

Greg Luck
ThoughtWorks, Inc.
gluck@thoughtworks.com

Abstract

Extreme Programming encourages adoption of all of its practices [1]. In practice many projects drop practices. What remains can be an incomplete methodology, which is dangerous. This problem can be overcome by replacing each removed dropped practice with a compensating practice tailored to the circumstances of the project – effectively subclassing XP.

This experience report recounts the experiences of subclassing of XP at Wotif.com, where Pair Programming was replaced with “Pairing” and refactoring [2] was replaced with “Team Refactoring.”

1. Introduction

This experience report is about two XP practices that were replaced on Wotif.com’s Project Jaguar in September 2003.

Wotif.com is a last minute hotel reservations provider. Their existing system was built using ASP technology[3]. Users search for accommodation by geographical region. The system responds with a complete view of the accommodation market for the next two weeks. Response sizes as high as 1.4MB of textual information can be generated for a single region. At busy times many such requests need to be processed concurrently. The single market view is both a compelling feature and a performance nightmare. The single market view is very expensive to produce.

The application consisted of ASP pages that called stored procedures on a SQL Server [4] database. Wotif.com grew rapidly. As they grew they added additional web servers to cope with demand. This moved the load to the database. They responded by upgrading the database machine. Ultimately all hardware upgrade paths were exhausted and system stability and performance became severely degraded.

Our project is to reengineer the application for performance, based on the J2EE [5] platform. The reengineering is on a phased basis, where the existing system stays in place and is gradually replaced. This adds much complexity to the project.

A team of three developers started the reengineering project using the XP methodology. XP was selected because:

1. Wotif.com had until that time outsourced all development, and did not have an incumbent methodology.
2. XP is often used on ThoughtWorks projects.

All XP practices were followed, with the exception of pair programming. This was primarily because ThoughtWorks had provided estimates in a competitive bid process that did not factor in pair programming. Moreover, in spite of research to the contrary, we didn’t have past experience that lead us to believe pair programming would be faster.

The first release was to be after four two-week iterations. As it happened the existing system became severely unavailable. Two iterations into the project, Wotif.com was experiencing outages for more than three hours per day. When not down, the system was sometimes taking minutes to return region market views. As a result Wotif.com decided to drop non-essential user stories from the last two iterations to make one iteration. The date for the first release was brought forward by two weeks.

Under XP, we were easily able to adapt to the changed requirements. However, technical work for the performance architecture that had been spread out amongst the iterations was required to be solved more quickly.

2 Issues

Nevertheless development proceeded smoothly and quickly and appeared to be a ringing endorsement of XP. After three iterations we were looking forward to the first release. We then went through a week of performance, stability and user acceptance testing (“UAT”), finishing on a Friday. The release was due the following Wednesday. At that point, with all bugs

closed and with a stable and performant system, one of the developers left for a skiing holiday.

The remaining team came in Monday to find the weekend stability test had not been run, so it was initiated on Monday night.

Tuesday morning the remaining team walked into hell. The stability test on both servers had failed during the night. We reran the stability test, which failed after 10 seconds.

The high performance design of the system relied on threading code. Threading allows processing to be split over multiple processors, thus speeding it up. A deadlock had somehow been introduced into the code. This can happen when two or more execution threads are waiting for the same resource. This resulted in an unrecoverable error.

The release was deferred and we went into a mad panic. A quick look at the code produced no answers. All efforts were directed at finding the cause. The project manager called twice-daily crisis meetings with the team. We spent several days fruitlessly searching through the code. The unit tests could not help us, because they were all passing. With time ticking away things were looking grim.

Morale was at an all-time low. Developers were snapping at each other. A new theory as to a possible cause would lead to a manic flurry of activity followed by morose depression when the theory was proved wrong.

It was then discovered that the absent developer had made a massive source code commit late the previous Friday. The commit included fixes for a few trivial bugs together with many different refactors, touching most of the source code in the system.

The deadlocking issue was found not to occur in CruiseControl. CruiseControl is a tool for continuous integration, where the system is built and unit tests are run. Continuous integration enables coding errors to be discovered shortly after the source code is committed.

With Wotif.com on the verge of canceling the project, and with no solution in sight, we decided to remove the code that had been committed the previous Friday. The stability problem disappeared and the first release then went on successfully.

With the release out and the immediate pressure off the business, Wotif.com and the development team started to reflect on what had gone so wrong as to have almost sunk the project. Wotif.com had lost confidence in the methodology, and refactoring in particular, which was now seen as the programming equivalent of Russian Roulette.

3. Subclassing XP is Born

Wotif.com insisted that new quality assurance processes be put in place before the project continued, with particular focus on the practice of refactoring. The team thought that the skipping of pair programming might also have been a contributing cause. This was discussed with Wotif.com but was considered too expensive to implement in its purest form.

The Brisbane Agile SIG had held a heated debate on pair programming in which the presenter held that the practice was unalterable. Indeed, the presenter asserted that all guidance on XP indicated that none of the practices could be dropped. This advice seemed to have been reinforced by the issues we had.

On the other hand we still had a concern that pair programming would be too expensive upfront for Wotif.com, with the result that the project would be cancelled. We were aware of papers that proved the cost effectiveness of pair programming. Each of these offset the additional upfront cost with a reduced long term cost of bug fixing, consequent testing and releasing. We felt that these costs did not characterize the situation at Wotif.com.

The team started thinking about how to stay with an agile approach, preferably based on XP, while still having a methodology that would be safe and that they could live with. How could we break the rules and get away with it?

At this point the idea of subclassing XP to create a derived methodology was born. Instead of dropping practices we would replace some practices with more palatable equivalents. Just like a subclass in programming we would specialize XP to satisfy the requirements of the project. In programming, a subclass is a descendent of another class, with much common behaviour and a few different ones. Like what a bowling ball is to a ball. In so doing we would break the rules the right way. We could claim to be doing XP in accordance with ThoughtWorks orthodoxy, just as a beach ball can claim to be a ball. The new methodology was written up on the Wiki [8], an easily edited, collaborative website used for project documentation, in a page entitled "AgilePracticesForTheJaguarProject". In this way it was communicated to Wotif.com, the team and other interested parties. The team was walked through the methodology and agreed to its practices. Roles and responsibilities were established. The role of Agile Coach was created.

The rest of this paper discusses what the practices are, reasons for their selection, and reflections on how well they have worked.

4. Team Refactoring

4.1 The practice

Team Refactoring is the removal of authority for some refactoring from individual developers or pairs to the team. Two types of refactoring are identified:

1. Refactoring done to fit in code for a new story. These are usually minor and narrow in scope.
2. Refactoring that effects a design change, or affects threading code. These are usually major in impact and wide in scope.

Team Refactoring is applied to the latter type.

Developers suggest refactors to be added to a refactoring list, which is maintained on our JIRA [9] issue tracking system. Attributes such as date, creator and a one-line description of the refactor are identified.

Development is done in iterations, normally a two-week period, after which a release to testing is done. By their nature, refactors are transparent to users, so none are required to complete an iteration. The team estimates the amount of time required to implement the iteration's user stories. In this way we allocate about three quarters of our time. The remaining time is for all the other things a developer needs to do. The ratio of user story allocated time to total time is the team's estimated velocity. At the end of each iteration the actual velocity is calculated. The team velocity is expected to be .75, which usually allows half to 1 day of refactor time per developer in a two-week iteration. Refactoring is always done at the end of the iteration, when a developer has completed the user stories they took on, and provided another developer does not need a hand.

When a developer is available, the team gathers round the list of major refactors and evaluates the suggested refactors and ranks them in priority order. Because they usually impact design, the team considers those implications as well. Finally the Lead considers project risk, the date of the next release, and any other factors, and then either approves the refactor or vetoes it in favour of another. A partner volunteers to pair with the developer, and the refactor is done.

4.2 Rationale

Either uncontrolled refactoring, or refactoring unrestrained by pair programming led to a disaster in the first phase.

Code is generally released to production every two weeks. An iteration is followed by about a week of user acceptance testing ("UAT") and then released. On

other projects a much longer UAT period is normally the case.

Refactoring is normally safe because of the other XP practices, particularly unit tests. Project Jaguar has excellent unit test and integration test coverage. We use Clover [11] to test coverage. It measures the percentage of methods, statement and execution branches which are tested. Our coverage ranges from 80% down, depending on the code. The coverage is less than perfect, and the lack of a long UAT period provides no safety net.

An XP practice is collective code ownership. This can break down when a strong programmer continuously refactors the code without pairing with others. The only person left with a sense of ownership is the late night refactorer. These numerous changes tend to be stylistic and reflective of a single view of the code, rather than a team view. This had been a severe problem on project Jaguar. We had a programmer who monitored commits and then went in and changed the code of the other programmers without pairing or even communicating with them. None of it broke the tests or changed the function of the code, so all of it could be classified as refactoring. The other programmers would go back to the code later and not understand it anymore. Many of the changes were iconoclastic. Moreover, the rest of the team did not accept some of the changes as good ideas.

Finally, there is quite a bit of threading code in Jaguar. The developer machines and the CruiseControl machine have one and two CPUs respectively. Production has four CPUs. Few production threading problems have been reproducible on the developer machines or the CruiseControl machine. The reason is that the higher the number of processors on a machine, the greater the real contention on synchronized resources. Or, for code that should be synchronized and is not, the greater the chance of data corruption. While the J2EE architecture normally shelters developers from these issues, our multi-tiered cache design went beyond J2EE and required custom threading code.

4.3 What happened

The team was initially very concerned about restrictions being placed on such a cherished XP practice. However, over time many refactors were done, the design of the system changed and these fears were mitigated.

The team is very protective of refactor time. Without sufficient refactoring we incur a technical debt. If an iteration completes without any refactoring having been done, the Iteration Manager, which is a ThoughtWorks term for the sum of the team leader and

agile coach roles, ensures that refactoring is done in the next iteration.

Wotif.com is intolerant of delays between bugs being fixed and released. Maintenance releases are typically done once or twice between new functionality releases. A common problem in software development is merging bug fixes in maintenance releases with the code in new functionality releases. The two diverge over time, making the task harder the more time passes. The team has found that moving refactors to the end of an iteration eases this merging considerably. This is because the structure of the two sets of code remains much the same. New work done is only in the new release code and has no merge problems.

A significant metric from Wotif.com's point of view is the number of release delays and production problems. Since we started the new practice there have been none that Wotif.com or the team has attributed to refactoring.

The whole team knows the nature and timing of each refactor. Consequently they feel comfortable and no one is left behind.

Domination of the code base by a late night refactorer is not possible with Team Refactoring. It works very well in containing this.

We still do have occasional breaches of the rules. Maintaining the practice requires a close watch by the Technical Lead. Everyone is very careful around the threading code. There is a hazy line between refactoring germane to a user story and refactoring which causes a larger design change. There has to be some flexibility.

4.4 Reflections

Reflecting on the practice, the biggest issue seems to be deciding when a refactor is a type two refactor and thus a Team Refactor. The lead developer keeps an ear to the ground and intervenes on occasion to move the refactor to the refactor list, because it is not always clear to a developer that the refactor needs to be a Team Refactor.

It would be helpful if the distinction between the two types could be inculcated into the literature.

The XP literature does not specifically address the special testing problems posed by threading. For example, a popular catalogue of common refactorings [10] does not mention threading at all. The subtlety of threading and its lack of testability on developer machines currently place it outside the reach of naïve refactoring practices. Special practices are required to provide safety for threading code.

5 Pairing

5.1 The practice

Pairing is our term invented for pair programming without the programming part. During the iteration kick-off a developer picks a story and then asks for a partner to pair with. The partner often has knowledge of that part of the code. The pair is jointly responsible for the completion and quality of the story implementation.

All code changes committed to the source code repository are done in the name of the pair. The committer uses the name of the partner using a commit message that starts with "With partner name: ...".

All design work is paired. The design must be agreed by the time the story is declared finished. How and when the design is done depends on the pair.

The pair comes together usually several times per day, with the author updating the partner on progress and discussing problems. This is usually done at the author's computer and involves looking at the code. Sometimes the partner will take the keyboard and navigate through the code.

Sometimes, perhaps four or five times per week, it is decided that it is worth pair programming. The pair stays together while code is written. This can be done where the partner has deep knowledge of an area such that it would be cheaper to do part of the story as a pair. Very subtle or complex code, where there is high risk, can also be paired.

At the time of final code check-in, the partner does either a story and/or code review with the author.

5.2 What happened

The developers seem to really enjoy the flexibility of Pairing.

Initially, it was hard for everyone to work out what was expected of him or her. Many of the developers paid lip service to the practice in discussion with the Iteration Manager. However, anyone observing the team throughout the day would have seen little discussion between developers, and would have noticed the developers sitting mainly at their own computers.

The Agile Coach got involved and cajoled the pairs into interacting. The Lead also started holding both partners in the pair responsible for code quality. Over a period of about a month the practice improved. Now, six months on, no energy at all is required to keep the practice going. A glance around the team area at any time of day will find a pair sitting together engrossed in discussion.

Pair design is often done with the primary author leading the partner through the code on the computer. However, just as often, the pair retreats to the

“Goldfish Bowl”, a glass-walled room with white boards, where they can draw diagrams and vigorously debate design ideas. Because the partner often has previous code experience, he will lead the design session with a suggested iteration. Because the how of design is not dictated, each pair finds a technique that suits them.

Design quality has been high as a result. The lead developer has found few design problems. Because there is a lot of conversation through the day, the lead developer is aware of design decisions as they evolve and can give guidance or direction in a timely way.

The sense of collective ownership is enhanced. At least two people were involved with the design of each piece of code. When a new story is done in a related area, the author has at least two people to choose from to be his pairing partner.

The practice of collective ownership and the sharing of knowledge about the code have been greatly improved.

Where a partner does a code review, it is more than an empty formality. The partner co-designed the code and has been in touch with it from inception. It is easy for them to step through the code and readily understand it. They can easily critique it and improve it, on the fly. The code review is usually done with the aid of the diff tool in IntelliJ [12], our Integrated Development Environment. The Commit Project command shows a list of each file that has changed and allows differences to be explored at the touch of a button.

The code review always involves looking at the tests. A partner will gripe if the test coverage is poor, so the authors tend to write lots of them.

5.3 Reflection

While Pairing has been well accepted by the team, we really do not know how things would have gone from a quality viewpoint had we pair programmed instead.

Wotif.com is very focused on the level, cost and impact of bugs. All bugs are recorded in JIRA and classified according to the step in the development process that caused them. Time spent on each bug is recorded. This could be when the user/business analyst created the story, an architectural mistake by the team affecting many user stories, the user interface design done by the web designer, a developer misunderstanding the story and/or miscommunicating with the user, or a bug introduced when the code was written.

The lead developer has found that a little less than half of the bugs are developer related. Very few of these are caused by poor design. Consequently, many

are quickly fixed. The average developer bug takes three hours of developer time to fix, including running tests and check-in.

The most expensive bugs have been threading issues in third party libraries. Each of these has taken weeks of work to track down and remedy.

The second most expensive has been user/business analyst errors. The whole implementation ends up being wrong because the user/business analyst does not understand what the rest of the business really wants. Developer bugs have been the third most expensive.

The fourth most expensive have been developer misunderstandings of stories. This can usually be tracked to a lack of interaction between the pair and the user/business analyst.

A calculation was recently done to reflect on what the cost benefits of pair programming would be. Assuming that half of the developer bugs could be avoided by pair programming, it was found that pair programming could not take more than 12% longer than Pairing to pay for itself. To give an example, a story done in two days by an author would need to take pair programmers one day and one hour to pay for itself. Our experience on the few stories that have been mostly pair programmed is that it is unlikely that pair programmed stories would be written in a little more than half of the time it took to write the same stories by pairing. Accordingly, we have recently reaffirmed the Pairing practice over pair programming. Our position is somewhat supported by the standard paper on the cost of pair programming [13] which finds that it costs an extra 15%.

The findings on the relative costs of Pairing versus Pair Programming cannot be extended beyond the Jaguar project. Wotif.com has a short UAT cycle and a very low cost of releasing to production. This is not typical for enterprise application development. The complete cost of developers' bugs in an environment with high release costs would be much more significant.

6. Reflections on Subclassing XP

Wotif.com's subclassing of XP was born in a crisis. The team wanted to retain control of the methodology. The team and Wotif.com are happy that the early project problems have been solved, the two changed practices are effective, and the new methodology is safe.

Of the two practices, Pairing is the easier of the two to implement.

After each iteration we do a retrospective, where the team comes together and reviews the iteration. We look back at what went well and what went badly. We

add these observations to the Wiki. Ideas for changes get generated and sometimes implemented.

An example is a recent tweak to try and reduce the cost of pair misunderstandings of user stories. While we have a user, she has not been available sufficiently to help developers understand stories. We now get the developer to initiate a walkthrough with the user of a story when it is started. They then record the walkthrough in the JIRA entry for the story. This has had good results in reducing the bug level.

We see the recording of the exact methodology being practiced as one of the most important things we have done. The practices used will change as circumstances dictate. Having a document that records what it actually being done, not what should be done, enables the team and Wotif.com to understand what the methodology is. It also allows other ThoughtWorkers to assess the efficacy and safety of the methodology.

The whole experience has given the team the confidence to make further changes to XP. Finally, in our situation XP didn't need to be taken wholly as is. That is, it was easy to subclass XP and build a safe

methodology that preserved the value of the practices and the spirit of XP without being dogmatic about following them by the book.

7. References

- [1] Kent Beck, *extreme Programming explained*, Addison Wesley, New York, December 2000
- [2] Martin Fowler et al, *Refactoring*, Addison-Wesley Pub Co; New York, June 1999
- [3] msdn.microsoft.com/asp.net
- [4] www.microsoft.com/sql
- [5] java.sun.com/j2ee
- [6] cruisecontrol.sourceforge.net
- [7] martinfowler.com/articles/continuousIntegration.html
- [8] Bo Leuf and Ward Cunningham, *Wiki Way*, Addison-Wesley Longman, March 2001, ISBN 0-201-71499-X
- [9] <http://www.atlassian.com/software/jira>
- [10] <http://www.refactoring.com/catalog>
- [11] <http://www.thecortex.net/clover>
- [12] <http://www.jetbrains.com>
- [13] <http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF>