Code Redemption - XP as a Means of Project Recovery

Peter Schuh
ThoughtWorks
Suite 600
651 W. Washington
Chicago, IL 60661
+1 312 373 8433
peschuh@thoughtworks.com

ABSTRACT
The paper discusses an actual project that had fallen into developmental disarray, details the steps taken to turn it around, and analyzes how the gradual institution of XP facilitated the project's ultimate recovery. In its introduction, the paper briefly discusses the project's history and its state when the diagnosis was critical. The second section details the two-month-long death march that initially saved the project, and why and how it was undertaken. With the patient stabilized, the paper then explains how principles of XP were gradually administered to reform the codebase, retool the build and testing process, and restore the project to good health. The fourth section distills the lessons learned from this project into a set of general guidelines that may be applied to other projects in similar or less-dire straits. The paper concludes that XP, given in small and gradually-increasing doses, can recover a project; however, depending on the severity of the project's distress, XP alone may not be enough.

Keywords
eXtreme Programming, code recovery, project turnaround, phasing in XP, build process, JUnit, ObjectMother

1 INTRODUCTION

Near the close of 1999, my current employer, ThoughtWorks, had run into a bit of a jam with one of its newer projects. The development process was dearly behind schedule. The consultancy that had preceded us had spent a year defining and architecting to requirements that were already out-dated. Acknowledging that they had switched horses midstream, the client was willing to scale back on functionality; however, they refused to budge on the delivery date. ThoughtWorks had two months to turn the project around *and* to deliver a business-ready application. For the purposes of this paper, we will say that due date was February 2nd, Groundhog Day.

The application under construction was web-enabled and ejb-powered, with a code base that was consistently inflexible and temperamental. While the system was architected in a smart n-tier fashion, its innards were seriously ill. The business objects served as little more than mappers to the database. The session bean methods, the brain-trust of the application, were super-long tendrils of procedural code. The servlets were little more that pages of html stuffed into StringBuffers and wrapped in Java. Tests, where they existed, were expected to fail. Commenting was sparse and undependable. Building and deployment were a near-mystery to over half the development staff.

Possibly due to the sheer size and pervasiveness of the problem, ThoughtWorks had been slow to realize how fundamentally bad things were. Even when the danger was apparent to the consultants on site, the team lead and project manager had to compete with several other projects for overstretched resources. As is the case with many a dire emergency, the situation had to become one in order to receive the attention it needed.

2 DAMAGE CONTROL

I was one of four "resources" that began flying to the client site in December, just two months prior to the application's immutable go-live date. We joined six developers who were already on site. Our mandate was to slap the application into shape, no matter the cost. By the middle of January, conference-room dinners were the norm, all-nighters customary and weekends worked through. Although *Extreme Programming* Explained[1] books were ordered and distributed and even read, practically every principle (from the 40-hour work week to testing to simplifying the build process) was disregarded.

We felt that the impending deadline did not allow us the luxury of XP. Instead, the application was cut up and parceled out by areas of expertise: servlets, doc gen, business engines, database, and so on. Because it was deemed untenable, the servlet package was all but rewritten. Elsewhere, refactoring was an opportunist's endeavor. If it was thought that refactoring would take less time than complimenting ugly code with more of the same, then refactoring was undertaken. Otherwise, developers sucked it up and coded ugly.[1] Commenting and test-writing were goals one aspired to, and occasionally met. In lieu of an object model or static documentation, the database was scrubbed clean and reverse-engineered into a data model. This was regularly maintained and served as the most reliable high-level

[1] Engendering such memorable comment tags as: "It's 3:30 in the morning. Please forgive me." This was found atop a 150 line switch statement.

specs for the application.

The 70-plus hour work-weeks would likely have never continued if developer-pampering had not been administered by both he PM (Project Manager) and the top levels of ThoughtWorks management. Sushi was catered in for conference-room dinners. The PM graciously made two a.m. ice-cream runs. Last minute, full-fare tickets were bought for hurried trips home. It was made clear, through words, acts and expenditures, that ThoughtWorks understood and genuinely appreciated the sacrifices that the team was making.

In the end, it all came together the last week of January. The application did its part by bursting nearly every seam on the 30th, forcing a 24 hour go\no-go decision. Efforts were redoubled. Things improved. Functional tests passed. The last build was performed six hours prior to go live. Five thousand rows of data were snuck into the database four hours later. After eight weeks the system was delivered—minus some requirements and plus a few too many bugs—on time. There was one member of the team who best expressed the general consensus: "I'm glad I did that once. I never want to do that again."

3 THE NUTS AND BOLTS OF CODE REFORM

On January 3rd the system was live and we were all well aware of its shortcomings. The client had a long list of functionality to be added, not to mention the requirements that had been dropped to meet go-live. Meanwhile, the team was determined to refactor the application into something we weren't embarrassed to leave our names on. The team lead and PM negotiated with the client for time to apply "ease of maintenance" to the system.[2] The team—or some members of it, at least—decided that this refactoring phase would be an opportune time to begin adopting XP.

Walking Out of the Starting Gate

The switch to XP was a slow, unsteady process. Not only was the current code base a reluctant conspirator, but only perhaps a third of the team really supported the adoption of XP. Another third was impartial and, as one might expect, the final third was quietly but vehemently wishing that all this extreme nonsense would just go away.

Each XP-proponent on the team began to advocate one or two of the main principles. Steps toward XP were sometimes made by lone developers, sometimes advocated by pairs, and eventually pushed by the team as a whole. Single-handed accomplishments were made in testing and building. As mentioned earlier, the tests that did exist were unreliable, largely because they were based on non-restorable data that had either been altered or dropped from the database. One developer set out to base the existing tests on data that could be replenished, then bundled those tests into a JUnit-based test suite[4]. Another developer streamlined the build process, reducing it to a few simple steps that could be quickly learned, allowing every developer on the team to perform his own build. Some developers paired up to take on more traditional development tasks. The buggiest sections of the application were attacked first. Because we knew new functional requests were not a long away off, refactoring, for the most part, was pursued gently. Mediocre code was improved upon when convenient, while truly untenable code was gutted and rewritten. From a team-wide perspective, senior developers were advocating JavaDoc[3] comments and unit tests for all refactored or new code.

Learning from Successes and Building on Momentum

Shortly after Groundhog Day, two developers began applying a constants pattern to the application. Because the constants, as they are wont to be, were used throughout the application, the switch-over was neither smooth nor painless. The general consensus was that the refactoring job was necessary and the pattern was solid both for current use and extensibility. It was agreed, however, that better communication was needed for future refactoring,. The result was an increase in e-mail "advisories", pick-up development discussions and regularly scheduled code reviews.

The team's analysts had readily accepted the story card as their new document, both as a way to distribute functionality-requests to the developers and as a basis on which to negotiate with the client. When the first batch of cards was handed to development, some pairs and some individuals began cleaning up the portions of the application associated with their cards. Tests began to appear in the newly-refactored areas of code, and these were added to the main suite. The build process was made portable, so developers could build locally, prior to checking in. The team's build machine was moved from an occupied developers space to an otherwise empty cube. The number of builds per day increased. The number of broken builds plummeted.

Making It Up As You Go Along

By April nearly all of the functionality originally promised to the client had been coded into the application and passed UAT (user acceptance testing). Once it was clear the project was no longer in imminent danger, a few team members turned their attention to the more fundamental aspects of development. As a result, innovations helped to further our adoption of XP.

When the test-suite was initially linked up to the build process, and the JUnit results e-mailed out, developers were spammed with redbars[3]. It took weeks to whittle the

[2] The term was proposed by a senior developer, and future team lead, in response to the suggestion that the word "refactoring" not be put on timesheets.

[3] For the uninitiated, JUnit is most commonly run in its GUI (Graphical User Interface) incarnation on the developers machine. While running its tests, the tool graphically represents its progress as a status bar. As each test passes, this green bar increments ever-closer to completion. If a test fails, the entire bar goes read (under the assumption that no test has truly passed

error log down and see out first greenbar. When this occurred, a calendar was tacked to the wall alongside the build machine, and the result of the last build of the day was recorded with a red or green sticky note.[4] With this highly visible measure of performance to serve as a reminder, developers began to work toward nightly greenbars. Then, about a month after its posting, the calendar veered dangerously into the red just as a major delivery date was approaching. After five days of consistent redbars, one developer pulled the alarm by e-mailing a team-wide plea for a greenbar. Once aware of the situation, the analysts and PM put pressure on development to promote only those builds that greenbarred, and the days on the calendar moved back into the green. In the end, the calendar served two benefits. First, by providing a simple, straightforward metric, it gave development a clear and attainable performance goal. Second, because it was viewable and easily understood by the rest of the team, it served as a failsafe mechanism for development. When development—albeit, in a heads-down coding frenzy—failed to follow their own rules, the remainder of the team was able to push them back in line.

The build process, itself, was again improved upon, and push of a button builds finally became a reality.[5] Possibly more important, build promotion from the development environment to analyst-testing to UAT was automated. Push-of-a-button started with code-checkout and ended with the e-mailing of unit-test results. It meant that even an analyst could do it, and they did. With the guidance of the automated test results (which were run on every build) an analyst could promote the latest greenbar build into a testing environment. This saved development the hassle of having to be on call to perform the task, and resulted in quicker feedback on new functionality and bug fixes.

Finally, several developers teamed together to devise and code an object-generator, dubbed ObjectMother[5]. This utility could provide a complete and valid structure of business objects (think of an invoice, its lines and all related charges) via a handful of simple method calls. ObjectMother had numerous benefits. First, by drastically simplifying the creation of test data within code, developers were much less likely to "cheat" and base their tests on supposedly persistent data in the test database. Second, the ease with which test data could be created via ObjectMother greatly simplified the task of converting existing tests that did rely on persistent data in the database. Third, by making the test suite database-independent, we gained the ability to swap UAT and even production databases into and out of the development environment, allowing development to code and debug against real data. Finally, because it had become easy to create test data, developers began writing more tests.

Past the Finish Line, and Still Running

In order to reduce travel costs, the project began rolling off some of its more experienced resources in late March (only three months after the cavalry was sent in). The junior developers who had proven their mettle were given greater responsibility. And fresh, impressionable recruits were introduced into the project. Good practices were passed along, and XP-based development gained more momentum. Within six months, the project had risen from well-deserved infamy to one of ThoughtWorks better respected.

4 IF WE HAD TO DO IT ALL OVER AGAIN (GOD HELP US)

Notwithstanding everything said above, what saved the project was not XP. Instead, it was a well-financed and tremendously successful death march. The client's refusal to budge on the delivery date was the single greatest contributing factor to this outcome. Groundhog Day meant that the team could not step back and reassess the situation. It meant that the course of development could only be adjusted by degrees, not turned on its head. There was no time for developers new to XP to learn to program in pairs. More often than not, bad code could not be refactored. Too often, the hack won out over the simplest thing that could possibly work. The irony of it, however, is that Groundhog Day took the code live, and XPers prefer to work with live code.

While extreme programming wasn't the team's immediate salvation, it was, in the end, what made the application sustainable beyond February 2nd. It was the gradual adoption of XP that recovered, retooled and rejuvenated the code base. Due to the nature of the project, there were many aspects of XP that I believe were correctly put on hold during the first months of rehabilitation. But there were other principles, such as improving the build and test processes, that we could have introduced much earlier. In the end, it was important that we all understood that we couldn't change the world in a day. If someone hasn't written it in already, drawing knowledge from previous experience should be included among the principles of XP.

So, it's two months to go-live, the team methodology to-date has been waterfall, the project is a month and a half behind schedule, and Beelzebub is banging at the front door. What do you do? Lock the door. Okay, what next?

What You Should Do Right Away

Non-XP Essentials

For starters, I cannot stress enough how essential

---

until all tests pass). Hence, a suite of tests that completes successfully gives a "greenbar", or "greenbars". Alternatively, a test suite that fails "redbars".

[4] It should be noted that the calendar idea was actually, and shamelessly, stolen from another ThoughtWorks project, where it had been previously used with great success. This project, and their practice of XP, has been documented in Cutter IT Journal.[6]

[5] For more information on how set up a build process in this manner, and why you would bother to do it, see another ThoughtWorks inspired article, "Continuous Integration."[2]

ThoughtWorks' support was to the initial success of the project. I do not believe the project would have ever met its original goal without serious moral and financial commitments. First, employees simply are not going to give up their lives for two months, and deliver the impossible on a silver platter, if they are not constantly reminded how important the matter is and how valuable they are. Furthermore, ThoughtWorks contributed to our initial success by making intelligent staffing decisions. When it is clear that a project is in danger and extra developers are required, additional resources must be targeted at the specific needs of the project. Finally, it always helps to promise a reward in the event of success.[6]

Target the Build Process

This is one of the absolutely first things we should have done. Developers are keen to those things that get the greatest benefit from the least amount of effort. A long, arduous build process discourages developers from taking responsibility for the code they check in. Conversely, the simpler it is to perform a build, the less time it takes, the more likely a developer will want to know that new code integrates successfully. Making the build process portable, so it can be run on individual's machines, further encourages responsible check-ins.[7] Assuming the process is easy and not time-consuming, what developer wouldn't check out the latest code, perform a clean compile, and check in with confidence. The build process does not have to be true push-of-a-button at this stage, but it must be streamlined to only a handful of steps. The benefit to the team is obvious, and measurable. There are few things that are more discouraging in software development than to spend an entire day trying to make a clean build.

Organize a Test Suite

Even if it's the first working test class to go into the code-base, an AllTests class should be written and run at the end of every build. Any existing tests that do pass, or can easily be made to pass, should be added to AllTests.[8] A build that redbars should be treated no different than one that fails to compile; you will need to sell this to the analysts—or client—as well. Developers should be encouraged to write tests and add them to the test suite, but test-writing shouldn't be shoved down their throats (at least not yet).

Write an ObjectMother

Writing an object-generator is not a trivial task, but it pays for itself by shortening the time spent writing new tests and fixing broken ones. The utility reduces the effort-versus-benefit ratio for test-writing. A developer is much more likely to write a test when an invoice and all its associated objects can be acquired from one simple method call, and much less likely to write the same test when the invoice, its lines and charges, the customer and his billto address and perhaps the associated assets all have to be instantiated and bound together before calling getTotal(). ObjectMother also makes it easier to maintain tests when their associated business objects change, because the instantiations are all centered in one place instead of being spread across the application.

XP Principles to Phase-in Early

Gentle Refacoring

Refactoring is good, but at this stage in a project's recovery, it must be tempered for several reasons. First, on the customer side, it may be difficult to get client buy-in. Second, the worse the code base is, the less likely it is to follow object-oriented ideas of abstraction, the more difficult it will be to isolate portions for retooling. Third, at least in the beginning, you are likely not to have either a quick build process nor dependable test results as an indicator of success. Nonetheless, gentle refactoring must be pursued from the start. Insufferable portions of code should be removed. Any refactoring task that offers low-risk and high-value should also be undertaken.

Code Commenting

This is an easy thing to encourage without spending too much time or effort. It's even better when you have a standard like JavaDoc that you can simply pass (or e-mail) around and occasionally refer to during discussions. If someone is really hot on this topic, they could incorporate the generation of JavaDocs directly into the build process.

Stand-up Meetings

We never introduced these, and I believe it was a major mistake. Quick daily face-to-face meetings keep developers informed as to what others on the team are doing. They help to keep people from stepping on each other's toes. They keep the team lead informed as to who is ahead and who is behind on their tasks. They air new ideas and keep people for duplicating work.

Mind the Database

Okay, this isn't an XP principle, but it's definitely as important as one. The database is an essential component of nearly every business application; neglect it at your peril. Nothing good will come of a database that is architected without thought of conversion or reporting. Similarly, a database where schema is updated for new attributes and entities but not deleted ones, and where test data is allowed to pile up and atrophy, will be cantankerous to develop with, hard to test on, and difficult to alter. Conversely, a well-architected and

[6] In our case, the entire team was flown to Vegas, put up in the Mirage and given stake money. Bonuses, raises, and better future assignments also work well.

[7] This does go against the XP notion of employing a build machine or build token. At ThoughtWorks we tend to forgo this rule and, in its stead, stress local builds and testing prior to check in. This alternative process has been very successful.

[8] I recommend against including old tests that fail, even if it is the team's intention to get them working some time in the future. I believe there is something psychological in seeing or wanting to see a greenbar. A redbar that is a "virtual" greenbar, because "those test never pass anyway," isn't the same.

maintained database, through intelligent and efficient organization of data, can guide good development. Finally, as mentioned earlier, in lieu of an object model or other documentation, a data model can make for an extremely handy overview of the application.

Principles to introduce once the pressure lets up

Step Back and Relax

Once the project has met some of its immediate goals (a major delivery or go-live date) it's a good idea for the development team to step back and get everything into perspective. If the time has come to begin serious refactoring, what parts of the application should be put through the grinder first? How is the adoption of XP coming along—who is resisting and who is welcoming it? How is the mental health of the team? If the last couple months have been a bloodbath, are there exhausted resources who need to be rolled off the project?[9] Would the project benefit from new recruits and some fresh perspective?

Roll in the Rest of XP

Once the pace of the project returns to something akin to normal, the remaining elements of XP should be introduced. When functionality is added in poorly-written areas of the application, as a rule, the code should be refactored. The team should start looking at patterns. What parts of the application might benefit from their use? Pair programming should be strongly encouraged (and changes to the workspace made in order to encourage it if necessary). Story cards need to become the means by which functionality is proposed, deliberated upon and built into the application. Finally, the build process should be made push-of-a-button, and if it is portable then the test suite should be made portable as well, allowing developers to run the full suite on new code before they check in.

Communicate, Communicate, Communicate

If you have so-far managed to avoid instituting stand-up meetings, put them in place now. Whenever possible, XP principles should be propagated from the bottom up, not imposed from the top down. Ideally, this means that the team as a whole should decide what principles of XP it is going to introduce and get serious about first. Involve the entire team in estimation. All of these things foster a sense of collective ownership, not only of the code but of the general well-being of the project.

5 CONCLUSION

Were we to do it all over again, and were the client willing, many members of our team would have razed the code base and started again from step one. But few clients are so giving and few projects so fortunate. Furthermore, who is to say the project will not falter again, for similar or wholly different reasons? In such situations, in the end, little is achieved without a lot of hard work.

The upshot is that a downtrodden project can be turned-around with the gradual institution of XP, and a seriously troubled project can be recovered via XP with sufficient high-level support, encouragement and incentive (but this sort of redemption doesn't come cheap). The guidelines listed above are based on the process that worked for us. Like XP itself, they would have to be tailored to the needs and particulars of any other project. And, ultimately, it is not so much XP that brings the project around, but the efforts of individual developers and the team as a whole.

ACKNOWLEDGEMENTS

Credit must first and foremost be given to the fellow ThoughtWorkers with whom I endured the worst death march I ever wish to be a party to. I am quite grateful to have worked with such a team. Much thanks, also, to Martin Fowler for suggesting the topic of this paper and providing the nagging necessary to ensure that it was written.

REFERENCES

1. Beck, Kent. *Extreme Programming Explained.* Addison-Wesley, 1999.
2. Fowler, Martin and Matthew Foemmel. "Continuous Integration." Online at http://www.martinfowler.com/articles/continuousIntegration.html.
3. JavaDoc Tool Homepage, online at http://java.sun.com/j2se/javadoc/index.html
4. JUnit Website, online at: http://www.junit.org/
5. Schuh, Peter and Stephanie Punke. "ObjectMother:: Easing Test Object Creation in XP." Pending publication.
6. Taber, Cara and Martin Fowler. "An Iteration in the Life of an XP Project." *Cutter IT Journal 13, 11* (November, 2000).

[9] This is not intended to be negative. It is well understood that death marches can take their toll on developers. At the end of one, a change of scenery may be in order.