Recovery, Redemption, and Extreme Programming

Peter Schuh, *ThoughtWorks*

The author recounts the tale of a development project's rejuvenation through XP: the successes, the shortcomings, and, ultimately, the lessons learned.

In the autumn of the New Economy, my employer, ThoughtWorks, recognized that one of its projects had maneuvered itself into a bit of a jam. In the software development industry it is well understood—and occasionally admitted—that such situations occur. What is less well-known, however, is how to address them. For this particular project, the prospects were not good. On one side (the rock), was a development team both behind schedule and coding to more than a year's worth of requirements—gathered by the previous consultancy—that had outlived their build-by date. On the other side (the hard place), was a client that, acknowledging that it had switched horses midstream, was willing to scale back on functionality but refused to budge on delivery date. In short, ThoughtWorks found itself with a mere two months to both turn the project around and deliver a business-ready application. For the purposes of this article, we'll say that the delivery date was February 2nd, Groundhog Day.

The application under construction was Web-enabled and powered by Enterprise Java Beans. Although the system had a smart *n*-tier architecture, its innards were seriously ill. The business objects served as little more than mappers to the database. The session bean methods—the application's brain trust—were superlong tendrils of procedural code. The written code—its style, semantics, and legibility—varied wildly depending on the area of the application one happened to be nosing about in. Tests, where they existed, were expected to fail. Building and deployment were a near mystery to more than half the development staff.

Possibly because of the problem's sheer size and pervasiveness, ThoughtWorks was slow to realize how fundamentally bad things were. Even when the danger became apparent to the consultants on site, the team lead and project manager had to compete with several other projects for overstretched resources. However, once the situation was clearly stated and understood, ThoughtWorks responded with a swift and sustained effort. The problem areas in the application were identified, and developers with relevant expertise were brought in. After two long months, the project delivered on time. But we didn't stop there.

0740-7459/01/$10.00 © 2001 IEEE

Damage control

I was one of four "ThoughtWorkers" who began flying to the client site in early December—just two months prior to go-live—to assist the six developers already on site. Our mandate was to slap the application into shape no matter the cost. Full of vigor and with little appreciation for the hundred miles of bad road ahead of us, we saw this as an opportunity to test-drive Extreme Programming in a project recovery setting.

By mid-January, conference-room dinners were the norm, all-nighters customary, and weekends worked through. Although we bought, distributed, and even read and quoted *Extreme Programming Explained*,[1] we disregarded practically every XP principle—from the 40-hour workweek, to testing, to continuous integration.

As the severity of the matter quickly became apparent, we were forced to concede that the impending deadline did not allow us the luxury of XP. Instead, we cut up the application and parceled it out by areas of expertise: servlets, documentation generation, business engines, database, and so on. Deemed untenable, the servlet package was all but rewritten. Elsewhere, refactoring became an opportunist's endeavor. If a developer decided that refactoring would take less time than complementing ugly code with more of the same, then that developer refactored. Otherwise, we sucked it up and coded ugly. This was a humbling exercise that engendered such memorable comment tags as "It's 3:30 in the morning. Please forgive me." This was found atop a 150-line switch statement.

Coding standards and test writing were goals we aspired to and occasionally met. In lieu of an object model or static documentation, one developer scrubbed the database clean and reverse-engineered it into a data model. The model was maintained regularly, and it served as the application's most reliable high-level specifications.

The 70-plus-hour workweeks would not have persisted had developer pampering not been administered by both the project manager and the top levels of ThoughtWorks management. Conference-room dinners featured catered sushi, the project manager graciously made 2 a.m. ice cream runs, and the company paid for last-minute, full-fare tickets for hurried trips home. ThoughtWorks' management made it clear, through words, acts, and expenditures, that it understood and genuinely appreciated the sacrifices 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. The team redoubled its efforts. Things improved. Functional tests passed. We performed the last build six hours prior to go-live. Four hours later, 5,000 rows of data were sneaked into the database. After eight weeks, we delivered the system—minus some requirements and plus a few too many bugs—on time. One member of the team best expressed the general consensus: "I'm glad I did that once. I never want to do it again."

The nuts and bolts of code reform

On 3 February, the system was live, and we were all well aware of its shortcomings. The client had a long list of functionality for us to add, not to mention the requirements it had 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 project manager negotiated with the client for time to apply "ease of maintenance" to the system (a senior developer proposed the term in response to the suggestion that "refactoring" not appear on timesheets). With the worst behind us and an easier road ahead, several team members (but by no means all) 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 adopting XP. Another third was impartial, and, as might be expected, the final third was quietly but vehemently wishing that all this extreme nonsense would just go away. What began as lone developers and pairs taking some first steps toward agile processes—with patience and determination—became a teamwide push to adopt XP.

Testing and building both benefited from single-handed accomplishments. The few existing tests were unreliable, largely because they depended on nonrestorable data that had either been altered or dropped from the database. One developer set out to base the exist-

We attacked the buggiest sections of the application first.

ing tests on data that could be replenished, then bundled those tests into a JUnit-based test suite (see the JUnit Web site, www.junit.org). Another developer streamlined the build process, reducing it to a few simple steps that could be quickly learned, letting every developer on the team perform a build before checking in—our first step toward continuous integration.

Some developers paired up to take on more traditional development tasks. We attacked the buggiest sections of the application 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 teamwide perspective, the senior developers advocated a unified coding standard, JavaDoc-based commenting (see the JavaDoc Tool homepage, http://java.sun.com/j2se/javadoc/index.html), 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 was neither smooth nor painless. The general consensus was that refactoring was necessary and that the pattern was solid for both current use and extensibility. The team agreed, however, that we needed better communication for future refactoring. The result was an increase in email "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 analysts handed the first batch of cards 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. One developer made the build process portable, so everyone could build locally prior to checking in. The build machine was moved from an occupied developer's space to an otherwise empty cube. The number of builds per day increased, and the number of broken builds plummeted.

The one insurmountable impediment we encountered after Groundhog Day was the client's lack of cooperation in XP. Without objecting to the shift, the client nonetheless refused to participate in the process in any meaningful way. The team's project manager and analyst (both ThoughtWorkers) assumed the role of the XP customer by writing story cards, providing domain knowledge on demand, and communicating decisions. However, the client's absence in these activities certainly affected us. Without a decision maker to sit in the same room with the team, we could never fully implement the planning game, nor could we convince the customer to make official iterations more frequent than every three months. The irony of all this was that ThoughtWorks was doing development on site. The client was paying to fly multiple ThoughtWorkers out every week, only to house them several office buildings down the road from any potential users.

Making it up as you go along

By April, nearly all the functionality we had originally promised the client had been coded into the application and had passed 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. Innovations in these areas furthered our adoption of XP.

The first crucial change was the mandate of a nightly greenbar. (A *greenbar* is the graphically displayed result of a successful test run in JUnit; conversely, a *redbar* is the result of any test that fails.) It took weeks to whittle the error log down and see our first greenbar. When this occurred, we tacked a calendar to the wall alongside the build machine and recorded the result of each day's last build with a red or green sticky note. With this highly visible performance measure serving as a reminder, developers began to work toward nightly greenbars. Then, about a month after the calendar was posted, it veered dangerously into the red, just as a major delivery date was approaching. After five days of consistent redbars, one developer sounded the alarm, emailing a teamwide plea for a greenbar. Having been alerted to the situation, the analysts and project manager be-

gan pressuring development to promote only builds that greenbarred, and the calendar moved back into the green.

In the end, the calendar served two purposes. 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 the developers—in a heads-down coding frenzy—failed to follow their own rules, the other team members could push them back in line.

Once again, we improved the build process, this time simplifying it to a push-of-a-button procedure that eased the adoption of continuous integration.[2] Possibly as important, we automated build promotion—from the development environment, through internal testing, to user acceptance testing (UAT). Automation started with code checkout and ended with the emailing of unit test results. This meant that even analysts could promote builds, and they did. Guided by the automated test results, which ran on every build, an analyst could promote the latest greenbar build into a testing environment. This saved development the hassle of being on call to perform the task and resulted in quicker feedback on new functionality and bug fixes.

Finally, several developers teamed up to devise and code a test-data generator, dubbed ObjectMother.[3] Through a handful of simple method calls, this utility provided a complete, valid, and customizable structure of business objects (think of an invoice: its lines, all related charges, remit to, bill to). ObjectMother yielded numerous benefits. Because it made the test suite database-independent, we could swap UAT and even production databases in and out of the development environment, letting developers code and debug against real data. Moreover, ObjectMother drastically simplified the creation of test data within code, resulting in three major benefits. First, developers were much less likely to cheat and base their tests on supposedly persistent data in the test database. Second, it became much easier to convert existing tests that did rely on persistent data. Finally, developers began writing more tests.

Past the finish line and still running

To reduce travel costs, the project began rolling off some of its more experienced personnel in late March, little more than three months after the cavalry's arrival. Junior developers who had proven their mettle now took greater responsibility, and fresh, impressionable recruits came on board. Good practices were passed along, and XP-based development gained more momentum. Within six months, the project had metamorphosed its well-deserved infamy within ThoughtWorks into high repute.

If we had to do it all over again

Notwithstanding everything I've said, 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 we could adjust the course of development only by degrees, not by turning it on its head. The developers new to agile processes had no time to adopt a coding standard or collective ownership, build a continuous integration process, or learn to program in pairs. More often than not, we didn't have time to refactor bad code, so the hack frequently won out over the simplest thing that could possibly work. The irony of it, however, was that Groundhog Day took the code live, and XPers prefer to work with live code.

Although XP wasn't the team's immediate salvation, it did, in the end, make the application sustainable beyond Groundhog Day. Our gradual adoption of XP recovered, retooled, and rejuvenated the code base. Because of the project's nature, I believe we were correct to put many agile processes on hold during the first months of rehabilitation. However, we could have introduced some principles—such as improving the build and test processes—much earlier. This section distills our experiences with XP into a list of steps that might have best served our team (or any project in similar or less dire straits). Table 1 details our overall experience with each XP practice.

What you should do right away

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?

Within six months, the project had metamorphosed its well-deserved infamy within ThoughtWorks into high repute.

Table 1

Adoption of XP practices

Practice	Adoption status	The experience
Planning game	Partially adopted	The client never really got involved in the planning game (similar to the absence of an on-site customer; see below). Nor did the entire development team participate.
Small releases	Not adopted	We could never get client buy-in.
Metaphor	Not adopted	The one XP practice we overlooked. We did not attempt to implement metaphor.
Simple design	Fully adopted	This went hand-in-hand with refactoring. It became a serious endeavor and permeated its way through the application.
Testing	Fully adopted	Acceptance tests were written and used. There was, however, some kicking and screaming regarding unit tests. Some developers were behind them, some neutral, and some had to be shamed into writing them. In the end, even unit-testing gained team-wide support.
Refactoring	Fully adopted	At least half the team had refactoring in mind moments after they landed on the project, but serious projects were put off until after Groundhog Day. There was no difficulty mustering team-wide support after that.
Pair programming	Partially adopted	We began pairing seriously, but slacked off and settled in at perhaps 30 percent of all developer time.
Collective ownership	Fully adopted	Easily accepted by the team. Developers still had their favored areas, but moved about frequently, and anyone could change any portion of the code.
Continuous integration	Fully adopted	The process arose from our automated build scripts; the team quickly embraced it.
40-hour week	Not adopted	After Groundhog Day, some weeks may have been 40, but some were 60. This depended on where we were in the iteration cycle.
On-site customer	Partially adopted	Although we were on site, the client never provided an XP-like customer. The project manager and team analyst stood in, with some success, in the role of the customer.
Coding standards	Partially adopted	A single standard was proposed and influenced some team members. Peer pressure had an effect at times. JavaDoc was incorporated into the build process.

Before considering XP or any best practices. For starters, I cannot stress enough how essential ThoughtWorks' support was to the project's initial success. First, I do not believe the project would have ever met its original goal without serious moral and financial commitments. Employees simply will not give up their lives for two months and deliver the near impossible, gift-wrapped, if they don't have constant reminders of how important the matter is and how valuable they are. Second, ThoughtWorks contributed to our initial success by making intelligent staffing decisions. When a project is in danger and requires extra developers, additional resources must target the project's specific needs. Finally, reputation matters. ThoughtWorks has long asserted that its employees are its greatest asset, and it has continuously backed those words with deeds. (Forgive me if this comes off as a shameless corporate pitch—it is not.)

Target the build process. This is one of the first things we should have done. Developers are keen to things that yield the greatest benefit from the least 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 it is that a developer will want to know that new code integrates successfully. Making the build process portable, so it can run on developer machines, further encourages responsible check-ins. Assuming the process is swift and easy, what developer would not check out the latest code, perform a clean compile, and check in with confidence?

The build process doesn't have to be true push-of-a-button at this stage, but it must be streamlined to a handful of steps. The benefit to the team is obvious and measurable: few things in software development are more discouraging than spending an entire day trying to make a clean build.

Organize a test suite. Even if it is the first working test class to go into the code base, write an AllTests class and run it at the end of every build. Add any existing tests that do pass, or can easily be made to pass, to AllTests. Treat a build that redbars no differently than one that fails to compile. (You'll have to sell this idea to the analysts—or the client—as well.) Encourage developers to write tests and add them to the test suite, but don't shove test writing down their throats (at least not yet). Finally, I recommend against including old tests that fail, even if the team intends to get them working some time in the future. Seeing a greenbar has a particular psychological value. A redbar that you consider a virtual

greenbar because "those test never pass anyway" isn't the same.

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. Developers are much more likely to write a test when they can acquire an invoice and all its associated objects from one simple method call; they are much less likely to write the same test when the invoice, its lines and charges, the customer, the bill-to address, and perhaps the associated assets all have to be instantiated and bound together before calling get-total. An ObjectMother also makes it easier to maintain tests when their associated business objects change, because it consolidates the instantiations into one space instead of letting them be spread across the application.

Practices to phase in early

So you've taken care of the most crucial elements. Now you can move on to phase two.

Continuous integration. If a project has a simple build process and a serviceable test suite, the next step is to combine these and tackle continuous integration. There are two ways to approach this. The first is the standard serialized XP practice based on a build machine or build token, where developers queue up one at a time to build, test, and integrate their changes.[1] The second alternative, a variation we use at ThoughtWorks, is for developers to build and test on their own machines before checking in. Backed up by multiple builds per day (to keep everybody honest), this has proved a successful practice.[2] Either way, continuous integration drastically reduces the time developers would otherwise waste converging code. It had a noticeable effect on our team as soon as we implemented it.

Gentle refactoring. Refactoring is good, but at this stage in a project's recovery, it must be tempered for several reasons. First, it may be difficult to get client buy-in. Second, the worse the code base is and the less likely it is to follow the principles of object-oriented abstraction, the more difficult it will be to isolate portions for retooling. Third, at least in the beginning, you probably won't have either a quick build process or dependable test results as success indicators. Nonetheless, do pursue gentle refactoring from the start. Remove insufferable portions of code and undertake any refactoring task that offers low risk and high value.

Simply simplicity. Simple design is straightforward to those in the know and aggravatingly intangible for anyone anywhere else. For our team, as we wended our way along trails of bad code in the first months of project recovery, we could easily pick out issues such as overdesign and logic duplication; however, before Groundhog Day, we had no time to address these issues. After Groundhog Day, as refactoring and new functionality became everyday business, we remembered the squalor we had wallowed through and consistently strove for simplicity. We pushed logic that permeated through the session beans back into the business layer and again refactored the servlets. Weekly code reviews were initiated, and simplicity regularly rolled off nearly every tongue.

Coding standard. This is easy to encourage without spending too much time or effort. In our case, we were fortunate to have two standards to fall back on. First, we had a proposed in-house coding standard that one team member had helped author several months earlier. This document was both a good primer for the greener developers and a good starting point for further discussion and debate. Second, we relied on the JavaDoc standard for code commenting, which we further leveraged by incorporating the generation of API docs directly into our build process. This practice is easiest to encourage when you have an in-house or open standard (such as JavaDoc) that you can simply pass or email about and then occasionally refer to during discussions.

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

Mind the database. Okay, this isn't an XP process, but it's definitely as important as one. The database is an essential component of nearly every business application—neglect it

Refactoring is good, but at this stage in a project's recovery, it must be tempered for several reasons.

Quick, daily, face-to-face meetings keep developers informed about what others on the team are doing.

at your peril. Nothing good ever comes of a database architected without a mind to conversion or reporting. Similarly, a database that updates schema for new attributes and entities but not deleted ones—a database that lets test data pile up and atrophy—will be cantankerous to develop with, hard to test on, and difficult to alter. Conversely, a well-architected and maintained database, through intelligent and efficient data organization, can guide good development. Finally, as mentioned earlier, in lieu of an object model or other documentation, a data model can provide an extremely handy overview of the application.

What to do when the pressure lets up

Groundhog Day has come and gone, and you've phased in several XP practices. So what's next?

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

Iterations and small releases. I didn't list this practice earlier because the principles in the first two groups weigh in as either more important or easier to accomplish. Why? Because a project plan is already in place, and negotiating for a formal change to that plan expends the scarce resources of both time and political capital. However, even without formal customer consent, a team can institute internal iterations. We did this. They were not the same as true iterations because the customer wasn't bearing down on us, but breaking the work out internally was definitely better than delivering once every three months.

Get an on-site customer. This is also on my wish list of XP practices to phase in early. I fear, however, that any project that falls to the level of depravity that ours did does so partly because there has been no real participation on the customer's part. Therefore, attempting to foster customer input early in the recovery process may well be a Sisyphean endeavor. If your project can actually produce a real customer earlier, then Tyche has shined on you.

Roll in the rest of XP. As the pace of the project returns to something akin to normal, the team should address the remaining elements of XP. As a rule, when you add functionality to poorly written areas of the application, refactor the code. Start looking at patterns. What parts of the application might benefit from their use? Encourage pair programming and make changes to the workspace to facilitate it if necessary. Story cards and the planning game should become the means by which the functionality is proposed, deliberated, and built into the application. We never got serious about metaphor or the 40-hour workweek; they should, however, be given serious consideration.

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 propagate from the bottom up rather than being imposed from the top down. Ideally, this means that the team as a whole decides what XP principles it will introduce and get serious about first. Involve the entire team in estimation. All these things foster a sense of collective ownership, not only of the code but also of the project's general well-being.

Had our client been 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 a project will not falter again—for similar or wholly different reasons? In such situations, little is achieved without a lot of hard work.

XP had no hand in recovering our project but weighed in heavily on its redemption. Before Groundhog Day, although we discussed XP and made occasional, nearly clandestine incursions into agile development, we simply could see no way to rewrite our development process in the time we had. Rather, we stepped in to shore up and compensate for the

current process until we had the time to address the many issues in earnest. After Groundhog Day, we made a sincere and sustained effort to go XP, which was successful on many levels. In the end, we landed somewhere within the realm of agile development without quite having achieved XP—but it was quite a good place to be.

I cannot stress enough how the team composition factored into our success. If we did not have so many stars and suckers—those capable of great things and those willing to put their lives on hold to make great things happen—the project would have never attained the success it did. Ultimately, our project's redemption rested on two strong foundations: the processes drawn from Extreme Programming and the team that applied them.

Acknowledgments

First and foremost, credit goes 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. Many thanks, also, go to Martin Fowler for suggesting the topic for the article and providing the guidance necessary to ensure that it was written. I presented an earlier version of this article at XP2001 in May 2001.

References

1. K. Beck, *Extreme Programming Explained*, Addison-Wesley, Reading, Mass., 1999.
2. M. Fowler and M. Foemmel, "Continuous Integration," www.martinfowler.com/articles/continuousIntegration.html (current 16 Oct. 2001).
3. P. Schuh and S. Punke, "ObjectMother: Easing Test Object Creation in XP," www.thoughtworks.com/cl_library.html (current 24 Oct. 2001).

About the Author

Peter Schuh is a team lead at ThoughtWorks, Inc., a provider of custom e-business application development and advanced system integration services to Global 1000 companies. He has written and spoken about XP, the adoption of agile processes, agile development's impacts upon database administration, and the ObjectMother pattern. With a BA in English literature and an MA in international relations, he is an unapologetic interloper in the software development industry. Contact him at ThoughtWorks, Ste. 600, 651 W. Washington, Chicago, IL 60661; peschuh@thoughtworks.com.