Recognizing and Responding to "Bad Smells" in Extreme Programming

Amr Elssamadisy
ThoughtWorks, Inc.
651 West Washington Blvd., Ste 500
Chicago, IL 60661
312-373-8523
Amr.Elssamadisy@thoughtworks.com

Dr. Gregory Schalliol
ThoughtWorks, Inc.
651 West Washington Blvd., Ste 500
Chicago, IL 60661
312-373-8661
Gregory.Schalliol@thoughtworks.com

ABSTRACT
The agile software development process called Extreme Programming (XP) is a set of best practices which, when used, promises swifter delivery of quality software than one finds with more traditional methodologies. In this paper, we describe a large software development project that used a modified XP approach, identifying several unproductive practices that we detected over its two-year life that threatened the swifter project completion we had grown to expect. We have identified areas of trouble in the entire life cycle, including analysis, design, development, and testing. For each practice we identify, we discuss the solution we implemented to correct it and, more importantly, examine the early symptoms of those poor practices ("bad smells") that project managers, analysts, and developers need to look out for in order to keep an XP project on its swifter track.

1. INTRODUCTION
In his book *Refactoring* [1], Martin Fowler--using the parental experience of Kent Beck—introduces the metaphor of a "bad smell" to describe the identification of early warning signals that something in computer code needs to be rewritten. In this paper, we wish to extend that metaphor to the general examination of early warning signals that certain aspects of an entire software development process need to be "refactored." Given the greater project significance involved in "rewriting" the encompassing process as opposed to a single practice within the process, we propose that bad *process* smells deserve far more attention than they have received previously.

The particular example we will explore is a major J2EE development project that switched to an extreme programming (XP) process after a more traditional approach proved ineffective. The project consisted of a 50-person team, about 30 of which were developers, working over a three year period. The application being built was a comprehensive enterprise resource solution for the leasing industry, including all aspects of accounts receivable, asset management, and contract terminations. The present code base consists of well over 500,000 lines of executable code, with very little of it remaining from the earlier development effort using a different process. We have been members of the project team (Amr as developer; Gregory as analyst) for over two years, and the examples we present in this paper were directly experienced in the day-to-day tasks we performed during that time. Although XP proved to be a more effective and successful software development process on this project, we learned—and wish to show in this paper—that it, too, is susceptible to making "wrong turns" that could prove to thwart a team's ability to deliver quality products in a timely manner.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'02, May 19-25, 2002, Orlando, Florida, USA.
Copyright 2002 ACM 1-58113-472-X/02/0005...$5.00.

2. QUARTERING THE CHICKEN
2.1 Smell
Dividing development into "story cards" the way we did last time (with success) is no longer successful.

2.2 Solution
With each new development iteration, rethink whether the procedures used in the previous iteration are still appropriate.

2.3 Discussion
One of the core trademarks of XP is iterative development cycles during which new functionality is divided into "story cards." These stories become the fundamental units of development for the iteration, and all resource estimation is based on summing the separate estimates assigned to each story card. There are many guides one can follow when deciding how to break down the whole of the application into discreet "stories": what can be done in an iteration, what will result in visible functionality for the customer, etc. In our case, we used one or more of these guides at various times to do the disintegration of the whole into small parts. But as the iterations wore on, it became evident that dividing into stories using a guide from two or three iterations earlier began to yield development tasks that prompted poorer estimates than in earlier, similar cases.

What happened here was our succumbing to a natural human tendency toward laziness. We tended to reuse prior strategies that worked without considering the circumstances in which we reused them. As the application grew in size and complexity, the sort of functional unit that was simple and easily estimated in earlier iterations became more complex and harder to complete within a single iteration. So when we insisted on tackling card x in iteration 6 the same way we did card y in iteration 2 (namely, insisting on having a customer-usable feature finished within a single iteration), we ended up with a very incomplete card at the end of iteration 6. The reason was simple. In order to get the feature fully operational in iteration 6, we had to take into account the various dependencies and other interconnections in the application that were not yet there in iteration 2. Once we honestly admitted to ourselves that this was the case, we began to realize that fully tested, customer-usable functionality in iteration 6 could not be expected. Story card division now would have to be done at a more granular level if we hoped to have the same iteration completion rate we had enjoyed earlier.

A concrete metaphor that illustrates the general principle here is the task of cutting a whole chicken into various pieces. For someone just beginning, the easiest places to cut (the joints) are not readily apparent, but with some experience one can learn where they are. Yet if one insists on having the pieces always cut the same way (for instance, into quarters), then some of the cutting may turn out well (e.g., leg quarters), but other cuts done later, intended to yield the same result, might be sloppy and difficult (e.g., cutting the breast in half through the breastplate). If we had not insisted that cutting the chicken breast into quarters (cards in iteration 6) should be like cutting the chicken legs into quarters (cards in iteration 2), we would have produced much better stories with more reliable estimates. The software development process that purportedly "embraces change" must itself embrace changes to its own specific implementation as needed if it is to succeed.

3. WHEN SHOULD THE CUSTOMER BE HAPPY?

3.1 Smell

After product deliveries following early iterations, the customer has no complaints, but during the late iterations, the customer complains about many things from all iterations.

3.2 Solution

The customer must be "coached" sufficiently to provide honest and substantial feedback from the very beginning of the development process.

3.3 Discussion

In a consumer oriented society, it becomes habitual to think that if you pay money to buy something, you do so partly to avoid the toil or headache of having to produce the product yourself. There remain some instances in consumerism, however, where a customer willingly donates his or her effort to insure proper satisfaction when the sale is complete. Those who buy custom-made clothing, for example, just spend some time with the tailor in order to get the quality product they desire.

Customers of custom-build software need to think more like the buyer of the tailored suit if they wish to receive the quality product they have ordered. For that to be possible, however, they must be integrally involved in the development process, and for an iterative process like XP, that means the *entire* development process. Since new functionality is delivered after every iteration, and since XP permits the customer to re-evaluate his or her priorities after every iteration, an XP customer can no longer drop into the IT "shop" just once, have his or her measurements taken (= traditional requirements gathering), and then expect to receive a fully built system satisfying current needs many months or years later. Without constant and honest feedback after each iterative delivery, the risk of receiving a disappointing product at the end is enormous. Just to think of the tailor example again, if the time to produce the suit lasts months or years, the risk that the suit will not fit the customer in the end is much greater if the customer does not periodically return for new measurements at the tailor's shop. Bodies have a tendency to change with time, just as do software requirements of businesses trying to stay competitive in a rapidly changing environment.

If an XP customer does not find something to complain about or change after early deliveries in the development process, the customer is, more than likely, not as engaged in the process as he or she should be. This turned out to be the case in our project. The development team encouraged our customer to devote their own people full-time to the development process, which they did. However, those customer "partners" were one step removed from the actual end users who would use the new system in the end, and rather than follow our suggestion to write their own functional tests to after every delivery to verify that the system was doing what it should, they refrained from that "toil" and relied instead on the functional tests we wrote ourselves. When the time came near final deployment to train the ultimate users of the system, complaints arose about functions that had long since been "delivered" and "accepted" in previous iterations. Part of this was due to some communication breakdown between our customer "partners" and the actual customer end users, but part was also due to the fact that those people representing the customer should have been much more involved in the "toil" of developing acceptance tests throughout the development process. Without themselves going through the toil of writing their own functional tests, they became lulled into thinking that what we thought they wanted was what they really wanted. But that, of course, was not always right, because their business changed over time, and communication is not always perfect.

Since XP is a relatively new approach to software development, however, the responsibility for integrating the customer into the development process rests primarily on the shoulders of the IT group who actually employs the XP process. We failed to push the customer hard enough early in the process to be an actual partner in the planning and acceptance of the development. We found it more comfortable to let the customer remain comfortable, but we paid a dear price for it in the end. If the customer is to become the true partner in the process, which XP demands, then practitioners of this new methodology must recognize their additional responsibility to successfully educate customers to think more like buyers of tailored suits rather than of generic clothing. The customer will more likely be happy with the product in the end if he or she has had sufficient opportunity and insight to re-evaluate the product rigorously from the beginning.

4. FUNCTIONS WORK, BUT JUST NOT TOGETHER

4.1 Smell

When the story cards are written and analyzed, the responsible party for a card feels that he or she cannot be sure all functionality has been accounted for in the functional tests developed for that card.

4.2 Solution

For complex applications, provide the development team with some synoptic "picture" of the whole that promotes a holistic understanding of the place each story has in the total application.

4.3 Discussion

The XP process was initially intended for rather small development projects, with the development team probably not exceeding 10 people, and a product that would not be excessively complex [2]. This limitation of project size is conducive to achieving another fundamental benefit of XP, namely, freedom from the heavy design and requirements gathering phase that typically precedes the first line of written code and often never even gets there. But as the multiple benefits of XP become more and more evident, many development groups (like ours) have dared to employ a modified XP methodology for larger, more complex projects. To do this, however, one must be aware that one has exceeded the recommended sphere of the methodology and be prepared to adapt appropriately.

Our project, for example, tackled the complex details of lease accounting, including governmental regulations, taxes, and professional accounting standards, not to mention managing all possible kinds of assets and all of the possible things someone might do with them. To accomplish this formidable task, we employed several experts with substantial experience in the leasing industry to act as business analysts directing the actual development work at the same time as they coached our customer. But after several iterations, as the application became more and more complex, even these experts found themselves uncertain that new story cards they were implementing were being correctly integrated with all other parts of the system. Following the suggestion of an XP approach, we started coding the most basic building blocks of the system without having produced up-front any great design or model that would show how it should all fit together. Hence, each time a new story card would be played, we relied on our leasing experts to flesh out exactly what other dependencies this new card might produce or might need to heed before we could say it would be done. As the number of implemented story cards began to climb into the thousands, however, even our experts began to miss this or that dependency that should have been addressed in the story card, and we often found ourselves having a new bit of functionality that would work by itself, but it did not work in conjunction with all of its 35 or so other related functions in the system.

What was missing was some type of "picture" or overview that reminded us of the almost endless interconnections in a system that very rapidly became more and more complex. Having a stack of story cards was not enough, because the story cards themselves could not remain static once they were composed. The promise that XP would not require much up-front design lulled us into thinking that we could get away without it, even if we exceeded the recommended project size for the methodology. There was no useable "metaphor" to help us out in this case, simply because leasing is too complex to be made more transparent via a more familiar or accessible image.

Were we to employ XP again on a similarly complex project, we would insist upon including the development and constant updating of a more traditional picture or graphic of the overall application as part of the tasks in each iteration. The aim of such an artifact would be to provide guidance for the writing of story cards and their connected functional tests when the elements of the application under development become very complex and interconnected. To be consistent with the iterative flexibility provided by XP, however, this picture, like the functionality it would represent, would have to be susceptible to modification of some degree at every stage of the development process. Such an integrated picture would provide what a high stack of story cards cannot, namely, a guide for insuring that every new story card can be written and tested completely in its relationships to all other stories.

5. FINISHING VS. "FINISHING"

5.1 Smell

Everyone says that all of the story cards were finished at the proper times, but it still takes 12 more weeks of full-time development effort to deliver a quality application to the customer.

5.2 Solution

Create a precise list of tasks that *must* be completed before the story is considered "finished," and then police it rigorously and honestly.

5.3 Discussion

By practicing test-first coding and forcing deliveries at the end of frequent development iterations, XP tries to avoid the well-known horror that greets so many software development projects at "completion" time, namely, bug-ridden code that requires a very long time to fix. Moreover, more frequent deadlines should make planning more reliable, for new stories should be better estimated with the frequent experience of previous stories. But as we discovered on our project, employing XP (as we did) was little help in this area.

At the end of our last official development iteration, our code was full of bugs and inconsistencies, so we needed several additional months of "cleanup" before we could deliver a workable version to our customer for implementation in its business. There were many causes of this, including an insufficient suite of automated tests (see that "smell" below), but the primary one was our general unwillingness to acknowledge reality. As the iterations wore on in the first half of the project, many team members (developers and analysts alike) found themselves extremely rushed to meet each iteration deadline. Rather than acknowledge that the card "velocity" had been incorrectly estimated, most team members, including the project managers, looked for ways to maintain the velocity, and the only avenues they could find were to cut corners in one way or another. The project managers excused developers from having to write automated functional tests, developers rushed in code directly before iteration end and refrained from completing small refactorings, and testers cut back on the thoroughness of their functional tests. Moreover, certain important dimensions of the application, such as GUI standards and written documentation, were simply ignored. These measures made it possible to continue with the same velocity, but it meant, of course, that the quality of the code was continually degraded until—surprise!—we discovered at the "end" that things did not work, we had no consistent standards across all screens, and our insufficient tests had allowed a swamp of bugs to flourish. The end result was an overall slowdown of the entire project, because just as there is a cost for carrying design [3], there is definitely a cost to carrying poor code that is brittle and hard to understand. Ultimately, the attempt to maintain an accelerated but unrealistic development velocity probably *added* to the overall project completion time.

The primary cause of this mistake was, we believe, the common reluctance of many professionals to admit that their own task estimates for completing stories were inaccurate. XP empowers individual project members to estimate their own tasks, but it also needs to reassure those same individuals that learning to estimate cards well takes time and will seldom be accurate in the beginning. Without sufficient encouragement for learning accurate estimation, team members will most likely try to suppress early, inaccurate estimates by "cutting corners" the way our team did. And once they did this a few times, it became a habit that eventually bogged down the whole project. Such habits are hard to remove once they have developed, so our advice to others facing a similar project using XP would be to begin with very strict and comprehensive guidelines as to what the story card encompasses and what constitutes true completion.

6. FACTORY VS. INSTANCES AND LOOK-AHEAD DESIGN[1]

6.1 Smell

Incrementally producing several instances of like objects since we should *do the simplest thing that could possibly work* [4], even though we know that a subsequent iteration will develop a replacement of the several instances with a flexible tool for creating any instance.

6.2 Solution

Create a factory instead of building different instances. Look ahead – see where you are going and use common sense. Even if you don't end up needing the extra flexibility, the cost of design carry [3] is negligible in this case.

6.3 Example

Picture this... We are starting the invoicing functionality on our application – so the first thing to do of course is do the minimum. Let us start with one specific invoice format. We know – or at least think we know – that we will need several invoice formats for the initial release of our application, and then we will probably need to make it completely customizable so that any user of this application later on can create and modify invoice formats at will.

So we, like good *XPers*, know that we should *do the simplest thing that could possibly work*[4] and we develop a single invoice format. Great – it works well in this iteration, and we have started invoicing. Now our customer provides us the rest of the invoice formats over 3 or 4 iterations. Again, like good *XPers* we only do the minimum. We find ourselves constantly refactoring our code – so that the commonality between invoice formats is never duplicated. So we are happy because we are doing only what needs to be done, doing it fast, and constantly refactoring our code so we have a good code base. Everything is great, right? WRONG.

WRONG??? What do you mean wrong? Well if that was it, we would be fine. We could release a bulletproof package with bulletproof invoicing. The hitch is that we knew that we would need to generalize this over the long run. There is NO WAY that I can give one client another client's invoicing. So, even though we have a good invoicing implementation for the initial release, what we don't have is code that can be refactored easily to have our flexible invoicing for *any* subsequent releases. We have hard-coded all our parameters for invoicing. So were we just ignorant and stupid to do this? Probably to some degree, except that we followed the XP premise although we knew in the back of our minds that we would need to redo a lot of the work.

So what ended up happening was a MAJOR refactoring of our generalized framework (which is another bad smell – next section *Large Refactorings Stink*) with several instances of invoices into an invoice factory that based its information off of a million-and-one parameters in the database. This effort took just as much time as actually putting the invoices together in the first place. There was an inordinate amount of code thrown out because it was just hard-coded – I mean parameters that needed to be moved to the database.

What we SHOULD have done is realize that it is OK to look ahead and trust our inner-instinct that we are going down the wrong path. We should have continued to do thing incrementally, but after the first instance we should have pulled the work and made it into a factory instead.

What types of problems does this apply to? Anytime you find yourself working on several instances of a solution, you find yourself with the decision of going towards a hierarchical solution with hard-coded parameters written into the class hierarchy or with a factory solution. The first solution is usually easier and hence tends to be what you will lean toward. But if you know or suspect that it will need to be flexible to the user, then creating a factory will only have a little overhead in the beginning. It does not carry a large design/maintenance burden[3]- only a larger startup cost. As always, use common sense and don't ignore it, thinking the process knows best[2].

7. LARGE REFACTORINGS STINK

7.1 Smell

Basically, the premise here is that if you find yourself having to do a large refactoring then you should have done many smaller refactorings earlier and you got lazy.

7.2 Solution

You will still have to do the large refactoring, but you can avoid this in the future by refactoring more often and not putting band-aids on your code.

7.3 Example

One piece of functionality in maintaining a lease is what is called unbooking and rebooking the lease. This is done when a change

[1] We are presenting a concrete example of where we believe look-ahead design is warranted. In general we believe that we should ALWAYS be cognizant of where the application is going, and leverage our experience, and more importantly common sense, in each iteration.

[2] Many people will give testimonials to XP

in the terms of an active lease will change its basic financial nature. In this case, our initial implementation was basically to modify each little detail that needed updating because of the financial change. This was fine initially, but as more and more functionality was added, the developers (who were usually different in different iterations) incrementally updated the code to work. It was slowly dawning on us that this whole approach was wrong – we needed to re-do the whole unbook/rebook functionality differently.

Unfortunately[3] both of the authors were the ones who got stuck with fixing this problem. Also, because one of them is a big whiner, he kept complaining that 'we knew this all along – something really stank for iterations and now I'm stuck with fixing it.' The sad truth is that he was right. Every developer after the initial implementation knew that the code needed to be refactored, but for one reason or another (see the "finishing" smell above), they never made the refactoring.

This type of smell really has no easy solution. The large refactoring had to be made because the inertia of the bad design decision was getting too high[4]. By taking the easy road from the beginning and completely ignoring the signs, we coded ourselves into a corner. So the moral of the story is this: when you find yourself making a large refactoring, stop and realize that it is probably because that you have skipped many smaller refactorings. Don't do it again! Also, look around in the project and you will almost surely find similar circumstances that you might be able to catch early.

8. AUTOMATED FUNCTIONAL TESTS

8.1 Smell

All the unit tests are passing and the system is still broken.

8.2 Solution

Automated functional tests. Just like in unit tests, when you fix a bug you need a test to make sure it doesn't happen again. Many of these tests can be coded and automated right along with the unit tests.

8.3 Example

Let's go back to our unbook/rebook example in the previous "smell." This whole unbook/rebook functionality is really useful to the user of the leasing application. Unfortunately, it is quite a complex business process that entails undoing all financial transactions, allowing the user to make the edits he wants, and then re-creating the financial transactions as needed. So, taking a specific example, changing the commencement date for a lease will cause the lease to unbook and then rebook and recalculate everything. Now, even with our unit tests, this change broke the application after it was implemented. So you might be tempted to say that the testing was incomplete, or the specifications for the functionality of changing the commencement date was incomplete, but this was not the case. The unit tests were complete. The problem was that OTHER business objects were not written with this possibility in mind. So other business objects had incomplete tests that did not test commencement date changes.

So we are making a case for another set of tests, functional tests, to be automated. These tests are specified by the customer and are the acceptance tests. If THESE tests are in place, then changes that affect many objects and cause bugs across these objects have a much higher chance of getting caught instead of just flying under the unit test radar.

9. OBJECT MOTHER AND THE SPECIAL INSTANCE OF A FACTORY FOR TEST FIXTURES

9.1 Smell

Extensive setup and teardown functions in unit tests and difficulty setting up complex objects in different parts of their lifetime.

9.2 Solution

Create fixtures that will return different objects in different states, as described in [5]. It is also a different way to approach the idea of *General Fixture* presented in [6] because this solution becomes more useful in larger and more complex business objects.

9.3 Discussion

Testing, testing, testing is the backbone of the XP process. It allows you to code at *extreme* speeds because you rely on tests catching your errors. When writing unit tests for a small scenario within a much larger scenario, you find that many times you need to use an object at a particular point in its lifetime. For example, in a leasing system, to test the effect of changing the commencement date on an active lease, you first need an active lease with at least an asset and a billing schedule and some charges on it. To create a lease and bring it up to this point is a very large task. Of course there is the mock-object solution in [6], but this quickly loses applicability when the objects become very complex and you want to be able to build on your test base.

Now back to the problem: many times in setting up proper test cases you need a business object or group of business objects at a specific point in their lifetime with a specific state to write a correct unit test. This is a very recurrent need in all projects with complex business objects. The smell is when you find yourself writing very large amounts of setup code before you actually do the test, or even worse, not writing tests because the setup is too extensive or not well understood. You find yourself constantly writing this setup code, and so you refactor it into a fixture to create these objects.

[7] was presented at the XP Universe conference in the summer of 2000 in Raleigh, NC to address this problem. During the presentation a member of the audience raised his hand and asked a couple of questions, and as the presenter answered the questioner, he said something to the effect "Congratulations, you have just rediscovered the Factory Pattern!" We tend to disagree with the questioner. We do not think this is a rediscovery of the factory pattern, but, in fact, we think it is a very significant instance of the factory pattern that needs to be addressed separately because of its usefulness to any meaningful testing of an application.

It is true that this solution is nothing new. It is also true that these fixtures are a special case of a factory. But unlike the regular

[3] Or maybe it was fortunate, since they actually learned from this mistake and wrote this paper as a warning to others.

[4] Look ahead design would have been useful here also.

factory methods, where you usually create objects in their initial state, these factory methods can actually create the same objects in different states. The key input is that the entire system is in the correct state – with all the secondary objects and settings that you can easily forget to add or set because these methods actually create the object in its initial state and act upon this object until it reaches the state needed.

10. CONCLUSION

Extreme programming is a software development methodology that promotes several desirable practices for a variety of projects. But as much as it may promise greater success in delivering high quality software in a timely fashion, it, too, depends on conscientious people who think carefully about what they do if it is to succeed. As with any new method that promises improvements over older methods, new users of the process might be easily lulled into thinking that the process will correct or even overcome problems that stem from the people using the process. In our experience, we learned that XP is a valuable and effective approach to software development, so long as one recognizes that 1) it cannot succeed without conscientious participants and 2) it must be adapted as necessary for projects that do not fit the "small team" limits recommended by its founders.

11. REFERENCES

[1] Fowler, M. *Refactoring* (Reading, MA, 1999), Addison-Wesley, Chap. 3.

[2] Beck, K. *Extreme Programming Explained* (Reading, MA, 2000), Addison-Wesley, p. 157.

[3] Wiki Discussion, Cost Of Design Carry, http://www.c2.com/cgi/wiki?CostOfDesignCarry

[4] Wiki Discussion, Do The Simplest Thing That Could Possibly Work, http://www.c2.com/cgi/wiki?DoTheSimplestThingThatCouldPossiblyWork

[5] Deursen, A., Moonen, L., Bergh, L., and Kok, G. Refactoring Test Code, http://www.xp2001.org/xp2001/conference/papers/Chapter22-vanDeursen+alii.pdf

[6] Mackinnon, T. Freeman, S., and Craig P. Endo-Testing: Unit Testing With Mock Objects http://www.connextra.com/about/mockobjects.pdf.

[7] Schuh, P., and Punke, S. ObjectMother: Easing Test Object Creation In XP, http://www.xpuniverse.com/Testing03.pdf.