Catalog of XP Project 'Smells'

Jennitta Andrea
ClearStream Consulting
250 6th Ave SW
Suite 1200
Calgary, Alberta Canada
1 403 264-5840
jennitta@clrstream.com

Gerard Meszaros
ClearStream Consulting
250 6th Ave SW
Suite 1200
Calgary, Alberta Canada
1 403 264-5840
gerard@clrstream.com

Shaun Smith
ClearStream Consulting
250 6th Ave SW
Suite 1200
Calgary, Alberta Canada
1 403 264-5840
shaun@clrstream.com

Abstract

This paper contributes an initial catalog of XP project 'smells' – indicators of problems with a team's implementation of XP practices. The paper traces the symptoms back to their root causes, and then offers solutions to either fix the underlying problem or customize the process to become a better fit within a particular context. The target audience for this paper is the people who need keep projects on course.

Keywords

eXtreme Programming, XP, Process Improvement, Mentoring

INTRODUCTION

The often-asked question *Are we doing XP?* opens the door to exploring whether the synergistic practices have been adopted appropriately [1]. The next question becomes *Are we doing it well?*

Through our engagements mentoring a variety of clients in their transition to and customization of XP, we have noted significant variation in each experience; concepts that are easy for one team to adopt present a great challenge for another. In the majority of cases, one or more aspects of pure XP are not a good fit for the team or project, so some degree of customization and process adaptation are necessary. We have also found that, if left unchecked, the installed process may degrade over time for a variety of reasons, including: shallow understanding of the practices and concepts, reverting back to old and comfortable ways, and significant changes to team composition.

In this paper we borrow the concept of 'code smell' [2] (an indicator of problems in code) and apply it to the XP project as a whole. This paper contributes an initial catalog of XP project smells – indicators of problems with a team's implementation of XP practices. The paper traces the symptoms back to their root causes, and then offers solutions to either fix the underlying problem or customize the process to become a better fit within a particular context.

The material contained in this paper is distilled from our involvement in seven XP projects over a two-year period (2000 - 2001). The projects ranged in scope from short-term pilot projects to mission critical software projects, ranged in size from two to twelve team members, and ranged in composition from predominately junior team members to predominately senior team members.

XP PROJECT SMELLS

Over Engineering

Symptoms: expressions of frustration and dissatisfaction with the constraints that are placed on highly valued analytical skills, time overruns due to extra work being done that are not part of the assigned task, and estimate padding to facilitate doing 'a little extra'.

Practice Affected: Simplest Possible Thing, Task Estimation.

Root Cause: Big Picture Thinkers: One of the biggest challenges for senior team members who have experience as architects or framework generalists is to strictly do the simplest possible thing for the specific task at hand. They instinctively think several steps beyond the current task and worry about a wide array of details.

Solution 1a Adopt Simplest Possible Thing: For a project that is building a one-off application, it's best to continue to strive to develop the simplest possible thing. The root of the problem lies in the mental shift of team members. Pairing one of these affected team members with a strong mentor will help keep them in check while they are making the transition. Concerns are generally alleviated over time as practice is gained in refactoring, and if they have access to excellent refactoring tools.

Solution 1b Adapt Simplest Possible Thing: For a project that is building application frameworks and generalized component-ware, doing the simplest possible thing in an incrementally evolving manner is not necessarily a good fit. The process must be customized to accommodate more big-picture thinking and synthesis of a number of customer stories into framework stories [3]. This raises the question of whether developing frameworks using XP is fundamentally different than when developing end user systems. This is something that we have been grappling with recently in the development of our own frameworks.

Overly Complex Integration

Symptoms: the integration of small and/or localized changes is more complex and time consuming than normal.

Practice Affected: Continuous Integration

Root Cause 1 Too Long Off Baseline: Delaying task integration for extended periods of time increases the difficulty of the job because the baseline has significantly changed. Large tasks are particularly susceptible to these issues because they take longer to complete and tend to have a wider impact.

Solution 1a Adopt Continuous Integration: Ensure team members are integrating at the completion of each task. If tasks are sized correctly and concurrent tasks have minimal overlap, then the complexity of each integration will be significantly reduced.

Solution 1b Refactor Task: Break a large task into separate phases (e.g. investigation, preparatory refactoring, adding logic, cleanup refactoring). Perform an integration after each phase rather than a single integration at the very end of the task.

Solution 1c Private Integration: Private integration involves synchronizing local code with the current baseline without checking it into the shared repository. During a large task, integrate privately on a regular basis to avoid getting too far off the baseline.

Root Cause 2 Avoiding Bad Tools: In some cases we have discovered that poor configuration management tools deter people from integrating frequently. Configuration management tools that have poor performance, are unreliable, and/or impose complex processes, significantly reduce productivity.

Solution 2 Improve Tools: Developing and publicizing workarounds or "best practices" for using the tool frequently helps to mitigate the problem. In extreme circumstances serious consideration should be given to upgrading or changing the toolset.

Root Cause 3 Refactored Baseline: When tasks involving feature development and major code refactoring occur in parallel, integration is generally more complex and time consuming than normal. This can create a lot of extra work for everyone who is making changes.

Solution 3 Refactoring 'Time-Out': Communication and coordination are key avoiding the mistake of performing major refactoring while others are working on the same areas of code. When planning a major refactoring task, ensure that parallel development is reduced to an absolute minimum. Ensure everyone has a chance to integrate their work before the refactoring task starts, then refactor quickly or outside of prime time. Use powerful tools for comparing and merging code (for example BeyondCompare [4]).

Unrepresentative Acceptance Test

Symptoms: customer crashes the system when they use it manually, even though all of the automated acceptance and unit tests pass. The user is using the system in a 'reasonable manner' and is only using functionality that has been completed.

Practice Affected: On-site Customer, Automated Acceptance Testing

Root Cause 1 Language Mismatch: When acceptance tests are not representative of real world usage, it is often a symptom of problems in the communication between the team and customer. There is a chasm between the customer's informal specification of the acceptance test and how it is ultimately recorded as code. The customer is not able to properly validate the automated acceptance test because they do not understand code.

Solution 1 Demo Script: The customer creates a manual demo script that also acts as a formal specification for the acceptance test. The script is specific in terms of the precondition data setup, the steps to perform, and the expected outcomes. Customer acceptance of a release includes running all automated tests and manually running the demo scripts.

Root Cause 2 Environment Mismatch: Acceptance tests may not match real-world usage because of technical and logistical challenges. For example, automating acceptance tests is challenging for a system that is triggered by system time events. While the unit tests can stub out the system clock to enable fine grained control of the passage of time, the acceptance test must wait for the actual system clock to change and thus may take hours to run.

Solution 2 Realistic Test Environment: Ensure that the acceptance tests operate as closely as is feasible to the way the system will really run. Only use test-stub code in the acceptance tests if test automation would be impossible or impractical otherwise.

Coding Assistant

Symptoms: The partner leaves all the decisions to the task owner and may degenerate into a "spell checker". If the partner takes the keyboard, they often ask, "What do you want me to do?"

Practice Affected: Pair Programming

Root Cause 1 Unbalanced Roles: Pair programming is one of the most foreign concepts that XP introduces. Each team refines the roles and etiquette of pair programming as they gain experience. Unfortunately, some teams develop unbalanced role definitions where the task owner (i.e. the one that signed up for the task and estimated it) is expected to individually own the outcome of the task.

Solution 1 Re-align Roles: Remind the team of the purpose of pair programming, namely: continuous review and knowledge transfer, collective ownership, synergy, etc. Reestablish the role definitions for each party in the pair and clarify acceptable pair programming etiquette. Strategically pair people together to reinforce these concepts through mentoring until they became widely practiced. Pair the mentor with the 'coding assistant' and encourage them to make decisions whether they are at the keyboard or not.

Singleton programming

Symptoms: pair programming degrades back to singleton programming.

Practice Affected: Pair Programming

Root Cause 1 Culture Shock: Without having tried it, team members are reluctant to embrace pair programming because it is such a significant departure from their normal work habits. Pairing becomes a challenge when team members have significantly different work hours.

Solution 1 Adapt The Culture: Strategically pair people together to reinforce pair programming concepts through mentoring until they become widely practiced. People are more likely to see the benefit if they are paired with someone who has valuable knowledge or skills that they themselves are lacking. Ensure that there is adequate accommodation for quite, personal time.

Root Cause 2 Office Logistics: Office space logistics are a serious roadblock for adoption of pair programming. It's not common to have a large area available for a team to configure as it wishes. The time and cost to reconfigure cubicles, re-locate people, and purchase larger monitors is often prohibitive. For short projects the upheaval is not considered practical.

Solution 2a Buddy Programming: There are many creative ways to adapt the concept of pair programming – without incurring the cost and disruption associated with creating the ideal workspace. One adaptation that we have seen work well is to introduce a buddy system, whereby people are closely located and collaborate frequently every day. Each person works on a different task and individually writes code. The buddies design their solutions collaboratively and perform small, incremental code reviews each day. Integration is always done as a pair.

Solution 2b Abandon Pair Programming: Of course, this removes the main checks and balances provided by XP and must thus be compensated for; formal design and code reviews must be introduced into the process.

Unmatched Acceptance Test Failure
Symptoms: an acceptance test still fails after the last task for a story is complete, while the entire unit test suite passes.

Practice Affected: Test-first Development

Root Cause 1 Stale Acceptance Test: By definition a set of unit tests will overlap with one or more acceptance tests – each covers the same functionality but at different levels of granularity. An acceptance test validates a path through the end-to-end process at a coarse granularity. A unit test validates an activity belonging to the process at a much finer and focused granularity. Consequently, if an acceptance test fails, one typically expects one or more failing unit tests to pinpoint the problem.

Normally, the first task of a story is to write the acceptance tests, which are initially expected to fail because the supporting software does not yet exist. When the acceptance test does not pass after the last task is integrated, the team typically has one of two reactions: a new task is created which is focused on fixing the failing acceptance test, or the pair integrating the last task assumes this responsibility and has a very long integration step. These are both symptoms of an incomplete and shallow integration process.

Solution 1 Semantic Integration of Acceptance Tests: As part of each task integration, revisit the associated acceptance tests with a view to making them progress farther based on the contributions made by the task. This enforces semantic integration rather than just task-based syntactic integration; merely ensuring the acceptance test still compiles is not enough. Introduce record keeping about the progress of the acceptance tests as part of the integration process; include details such as which acceptance tests fail, and where. This practice facilitates progress monitoring, while acting as a reminder that continuous integration means that the acceptance tests are not allowed to go stale.

Root Cause 2 Missing Unit Tests: Another possibility is that some unit tests are missing. This indicates that either the development and refactoring is not strictly test-driven, or some of the test scenarios were overlooked.

Solution 2 Increase Test Coverage: Employ test coverage analysis tools, like Jester [5] to find areas that lack tests. Unit tests may not exist for components that are reused by the project (e.g. infrastructure and legacy systems), and may need to be developed if problems are found with these components. To improve test development skills in the team, structure the pairings so that at least one experienced tester is involved during the test specification and design stages, and ensure that development and refactoring is consistently performed in a test-first manner.

Obtuse Specification
Symptoms: test learning curve is too long; uncertainty as to where to find a test; test cases are duplicated.

Practice Affected: Test-first Development

Root Cause 1 Unreadable Test: A way to measure the quality of a test is the length of time it takes someone else to understand it. The test forms the formal specification of the system, thus it is of utmost importance that it is clear, concise and well organized. However, the essential meaning of a test is easily obscured by the code-level details required to make it run automatically.

Solution 1a Improve Test Writing Skills: Mentor team members as they write tests, highlighting intent revealing coding concepts, and single-purpose tests. If time permits, mentor team members in the refactoring of existing tests to improve their standards.

Solution 1b Adopt Testing Standards: Introduce a standard format or template that all tests must conform to. Readability is dramatically improved by developing custom domain-specific test frameworks to encapsulate details behind well-defined method names. For example, instead of repeating ten lines of validation code in each test method, create a custom 'assert' method that contains the ten lines of code. Test set-up and pre-conditions are other prime candidates for custom framework methods.

Hard to Test Software
Symptoms: The software as designed is hard to write automated tests for.

Practice Affected: Design for Testability, Test-first Development

Root Cause 1 Overly Coupled Software: Sometimes, a class or component is too intimate with other classes. This makes it hard to test it without also testing the other classes and this results in tests that are overly complex or difficult to automate. When dependencies are hard-coded, it is virtually impossible to replace a real object/component with a test stub or a mock object [6].

Solution 1 Configuration Manager: Evaluate whether test-first development is being practiced; theoretically, this situation should not arise when tests are used to drive development. Refactor the software to make it more testable. Typically this involves a centralized component factory, which is used to override the real components with test stubs [7].

Root Cause 2 Hard to Test Interface: User interface code is hard to test because of the input comes from manual user interaction.

Solution 2 Layered Architecture: Clearly separate the user interaction from the core business processing. Bypass the user interface and test directly against a façade that exposes the core logic. Test the user interface logic separately.

APPLYING THE CONCEPTS

While children can be forced to bathe, adults do not respond as well to just being told to do something. Fortunately, most adults find value in bathing after having experienced the consequences of not doing so. XP mentors and coaches should let people experience some of these smells for themselves so that they have a deeper learning experience and so they can become accustomed to detecting/fixing problems early. Helping them detect the smells, identify the root causes and choose the solution (bath or deodorant!) is a good way to reinforce the learning experience.

CONCLUSIONS

Early detection of problems can help steer an XP project away from serious trouble and provides direction in the customization of the process to best fit a particular team or project. The "smells" that characterize common challenges and misapplication of XP practices can be used to quickly detect the problems and identify appropriate solutions.

This paper contributes an initial catalog of project 'smells'. There are certainly more root-causes and solutions for the 'smells' described in this paper. Many more 'smells' exist that have not been identified in this paper, for example in areas related to project tracking, and the planning game. It is our hope that the work of cataloging project 'smells' continues within the XP community in a collaborative fashion for the benefit of all.

ACKNOWLEDGEMENTS

The authors would like to thank the clients who's projects gave us the opportunities to gain the experiences described here as well as the colleagues who reviewed and commented on drafts of this paper.

REFERENCES

1. Jefferies, Ron, "Are We Doing XP?", Invited Talk, eXtreme Programming and Flexible Processes in Software Engineering - XP2001, May 2001.
2. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
3. Meszaros, Gerard, et al., "Building Frameworks Using XP", eXtreme Programming and Flexible Processes in Software Engineering - XP2002, May 2002.
4. Beyond Compare is a product of Scooter Software, online at www.scootersoftware.com.
5. Moore, Ivan, "Jester – a Junit Test Tester", eXtreme Programming and Flexible Processes in Software Engineering - XP2000, May 2000.
6. Mackinnon, T., et al. "Endo-Testing: Unit Testing with Mock Objects", eXtreme Programming and Flexible Processes in Software Engineering - XP2000, May 2000.
7. Smith, Shaun, "Test-First Development with Mock J2EE, JMS, and JNDI", eXtreme Programming and Flexible Processes in Software Engineering - XP2002, May 2002.