XP with Acceptance-Test Driven Development : A rewrite project for a resource optimization system

Johan Andersson, Geoff Bache, and Peter Sutton

Carmen Systems AB, Odinsgatan 9, SE-41103 Göteborg, Sweden
geoff.bache@carmensystems.com

Abstract. In his recent book "Test-Driven Development" [1], Kent Beck describes briefly the concept of "Acceptance-Test Driven Development", and is broadly sceptical to whether it will work. After a successful project that used this technique, we wish to argue in favour of it and the TextTest [7] tool that we have built up around it. We have found that a working XP process can be built based around using only automated acceptance tests, and not doing any unit testing. In this paper we explain and analyse our XP process, its strengths and limitations, and by doing so we hope to inspire others to try and make it work for their projects too.

1 The project

Carmen Systems has been producing planning systems for the airline and railway industries since 1994. By 2000 the C source code for one of the products was showing signs of becoming increasing difficult to maintain. After some consideration the decision was taken in 2001 to rewrite rather than to refactor the code, re-using the code base of another Carmen product written in C++. Therefore a team was created consisting of four software engineers and one product manager (acting as the customer). The team worked for over a year with the challenge of both capturing the behaviour of the legacy system and dramatically improving its run time speed. We used a process that was heavily based on eXtreme Programming.

2 The process

2.1 Description

On most points we have followed XP fairly faithfully, but in certain aspects we have differed. These practices were followed more or less as described in Extreme Programming Explained [2]:

- 2-week iterations following the Planning Game
- Daily stand-up meetings
- Pair programming mandatory for all production code

- Simple Design
- Refactoring
- Integration several times a day
- Coding standards
- Collective ownership of all code
- 40-hour week (in keeping with Swedish working culture!)
- Co-location (in adjacent 2-person offices)

At this point the enthusiasts count up and work out that four of the twelve practices remain. Here our process differed from Extreme Programming as you know it.

On-site customer - as a product company we have many customers, and this role hence became an internal one fulfilled by the product manager. His job as customer was to establish the requirements based on what the old system did, provide a rough priority order so that it could be implemented incrementally and also liaise with the real customers about needed improvements and features that could be dropped.

Metaphor - The legacy system served fairly well as a replacement for a metaphor

Testing - This was based entirely upon acceptance tests written by the "customer" and a story was not regarded as done until he signed off the test as working correctly. Acceptance tests had to run flawlessly before development could continue - 100% or bust, unit-test style. Unit testing itself was not used at all.

Small releases - really became continuous releases because of the way we did testing. The philosophy was that we pretended the system was in production with all features implemented so far at all times. Note that this refers to "internal releases" due to the fact that the customer was an internal role. "External release" only occurred relatively late in the project when the system was as good as the one it was trying to replace.

And then we added some entirely new practices...

Diagnostic debugging - We implemented a logging framework in the spirit of log4j [4], which allowed permanent debugging statements to describe in detail what particular modules were doing. These module diagnostics could be easily enabled independently of each other, and provided the fine-grained analysis we needed in the absence of unit tests.

Usage-First Design - We simulated Test-First Design by insisting that new code was always written "usage-first". This means that if a new class was required, we would first pretend it existed and write the code that would use it. Only when we were happy with this usage code would we attempt to implement the class.

2.2 How the process came about

We evaluated several development methodologies including the Rational Unified Process and XP, and the consensus proved heavily in favour of XP or something like it.

However, a certain amount of process-related activities, chiefly the testing approach and the emphasis on diagnostic debugging, came with the product whose codebase we re-used. As these approaches had been working well there, a decision was also taken to continue with them for the moment and to work on adopting other aspects of XP, with the expectation that we would change over when these aspects became our worst problem.

However, as time went on, the conviction that we had a process that worked really well grew, and these aspects have therefore stayed. We made a brief experiment at working with unit tests and CppUnit [5], but this felt to everyone involved like it was slowing us down rather than speeding us up, so it was abandoned.

By the end of 2002 the project was a success with the software in use at several major airlines with more to follow. The development team has now grown to eight software engineers and has moved on to work with other Carmen software products, using the same process.

2.3 Frequently Asked Questions

The practice of using Acceptance Tests alone was described in our XP2002 Practitioner's Report [6]. Space there did not really allow us to expand fully, and we received many questions at last year's conference on the subject. Here, then, is a description based on the most common questions.

1. *How did the customer manage to write acceptance tests?*
   The process is something like this: he finds appropriate input data for testing the feature in question and informs the developers of broadly what he expects it to do. They go off and implement it. When they have a result that satisfies them, they come back to him and show him what it does. If he isn't happy, the process iterates. If he is, that input data, along with the output and the log file produced by the program, is checked into the version-controlled test suite.
2. *How did you automate the acceptance testing?*
   The program executes in batch mode: i.e. it takes a bunch of input data (typically some flights and some crew), plans for a while, and then produces some output data (a plan). This made automating acceptance testing easy: it is fully text based. The log file and the output solution are compared using a diff tool against what it did when the customer accepted it, and any difference at all (barring process ID, timestamps etc.) is registered as failure. In addition, the performance is compared to the expected performance and failure reported if it differs by more than a preset amount (say 10%).

Over time, our automated test suite has grown from a small, highly product-specific UNIX shell script to become a fairly sophisticated application-independent framework for this kind of testing. It is now called TextTest, is written in Python, and is (or will soon be) available for free download from Carmen's website [7]. See the Appendix (section 5) for more details.

3. *How do you get the test suite to be fast enough to run regularly?*
   Insist that lots of fast tests are added to the test suite, and ask the customer not to provide large data sets (which imply long runtimes) unless they're really needed to prove that the functionality works. The tests are then run in parallel over a network using the third party load-balancing software LSF [3]. Because running all tests at every build would take too long, developers pick a time up to 15 minutes and run all tests that take less than that time, before checking in. This number depends on how radical they believe their changes are, and averages around 3-5 minutes for "normal-risk" changes. All tests that take less than 3 hours are run automatically overnight and a report generated, and any failures are the first thing to be fixed in the morning. Any tests longer than 3 hours are run only at the weekend.
   In this way we maintain very short cycles in our development, and the "time-to-green-bar" is kept very low. Essentially the fastest acceptance tests are treated by the developers in a similar way to unit tests: they are run at every build and failure is treated as a sign to stop work and fix it.
4. *Doesn't it take ages to find and fix bugs without Unit Tests?*
   Fortunately not, because the text-based philosophy extends to debugging, in our practice of "Diagnostic Debugging". When we need to examine some code in more detail, we write diagnostics for that code so that we can see in detail what it is doing, and these diagnostics are kept so that they can be reused in the future. Over time a large amount of these diagnostics are written, and then new diagnostics are only needed for new code.
   When debugging, the first behaviour difference from the log file tells you what was going on when it went wrong, and hence which diagnostics should be enabled (note that diagnostic data, unlike logfiles and solution data, is not version-controlled, there would be way too much of it). It's then a simple matter of running the checked-in code with the diagnostics, then running the new code, and seeing in detail what has changed. Hopefully this will lead to the error. Sometimes it won't, and then we must write new diagnostics, which are checked in when the bug is fixed. To quote the log4j manual: "Debugging statements stay with the program; debugging sessions are transient." [8]
5. *Without tests being isolated from each other, surely an error can break thousands of tests at once?*
   Yes it can, and frequently does. The approach then is to take the simplest, smallest test that failed and fix it. Then re-run the tests and repeat as required. So long as you don't expect that one bug will always produce exactly one test failure, it isn't a problem.
6. *Don't you miss the benefits of Test-Driven Development?*
   The practice of writing unit tests before writing the code (now known as Test-Driven Development [1]) is understood to have five main benefits:

- Verification that code works
- Low-level information about test failures making debugging easier
- Design driven by tests tending to exhibit high cohesion and loose coupling (”Test-first Design”)
- Predictive specification of what code will do, independent of the existence of the code itself.
- Documentation of the design

We have a testing approach that could perhaps best be described as Acceptance-Test Driven Development (ATDD)[1] in contrast to the more standard XP approach of driving development with Unit Tests. We believe that ATDD has covered and in some cases surpassed the benefits described above in our recent project. This is described in more detail below.

2.4 Comparison with Unit-Test Driven XP

Verification

Acceptance tests are written by the customer and utilise the system at a much higher level. They are thus a far stronger verification of system correctness than unit tests. Moving them into the centre of the process will therefore strengthen the verification. Of course, even when development is drived by unit tests, acceptance tests are also meant to be present as verification. However, because they are not central to the ”rhythm of the process”, the verification they provide is somewhat postponed at best, and in practice we believe many practitioners are relying entirely on unit tests for verification.

Refactoring

Acceptance tests are entirely independent of the design, because they do not interact with it. This means that they form a solid rock to lean on when doing refactoring. Interface-changing refactorings, as has been pointed out [9], will require the unit tests themselves to change, rendering them questionable as verification of the correctness of the refactoring, and in large numbers they will therefore exhibit a tendency to act as a brake on the mobility of the design.

Design

We have observed that the benefits described for Test-First Design - high cohesion and loose coupling - have emerged with our practice of Usage-First Design. We also feel that the pressure to isolate everything that comes with TFD is in some sense an artificial pressure, resulting in the creation of many ”mock objects” [10]. Applied to extremes, it seems to lead to a system where every class which is depended on by another class will need an interface, a real version and a mock version, and we do not believe that constitutes a good design.

Isolation of classes from each other, we feel, is something that should be done when the design demands it for some reason. It is not something that should always be done up-front as an end in itself.

Help with time-to-error when debugging

Debugging with easily-disabled log statements has long been advocated in various circles (e.g. [8]) and it has worked well for us. It has the advantage that effort can be spent as it is shown to be necessary rather than up-front, and also that it can be applied easily to any design, not just one of the form discussed above. This makes it much easier to apply to legacy systems, amongst other things.

Predictive specification

A unit test makes a predictive specification about what the code will do. Applied test-first, this prediction is made before the code is written. An acceptance test may or may not: it can do anything from no prediction through vague general predictions to very precise prediction of a test.

Its verification is chiefly regressional, of course: based on the fact that program behaviour can be manually verified as correct by the Customer and then maintained unchanging indefinitely. In practice we have made use of predictive general specifications about what the system will do and not do in all tests - for example enforcing that the text "Internal Error" is never produced, that all solutions are reported as legal in the logfile, etc. We generally do not try to predict what specific tests will do before code exists: partly because our domain does not really allow solutions to the problems to be constructed by hand (which is the whole point of the software).

There is no doubt that there is a psychological aspect of being able to "test-first" (as discussed in [1]). Going in to development knowing that a test is already in place that will say immediately whether or not the behaviour is correct when the code is written gives a powerful feeling of security. We, however, understand predictive specification as a sliding scale: and verifying new features is nearly always a mixture of prediction and reaction. Up-front prediction requires effort spent on it: and we feel there comes a point when that effort is no longer justified in terms of the gain versus reacting to the behaviour when you have it. Prediction that can be applied to all conceivable tests is thus very good in terms of payback versus effort invested.

We feel that the biggest problem in software quality is usually that changes break existing behaviour. If your changes never do that, then the quality can only go up, and then you will have very good quality pretty soon.

Documentation

A unit test is a design statement. An acceptance test is a statement of the correct behaviour of the system. Therefore, it seems logical that unit tests act in some sense as design documentation, whilst acceptance tests document system behaviour.

Both of these things are of course valuable. In the absence of unit tests another way to document the design is needed: in our case we have used "Doxygen"[11], which is a Javadoc-like tool for C++.

2.5 Applicability and limitations

The approach to testing proposed above is focussed on keeping an extremely firm control on the behaviour of the program as it is currently used in practice. For us, this has proven to be an excellent way to ensure the quality of the system stays high, while moving quickly due to not investing lots of developer effort thinking of tests up front. Of course, we cannot know how many defects exist that no customer usage has yet found: our premise is simply that it is hard to control this anyway and hence not worth investing the time trying. The evidence from the few reported defects we have from production bears this out: almost all were crashes on incorrect input to the system (and of course all resulted in a new test, so in future will result in graceful exit-with-error instead)

The fact that this approach works well for us is in some way because it is easy for us to manage and predict how the system will be used. Before it is taken into production with a major airline, a controlled implementation process will take place where we will have access to that airline's data and will know broadly what features they intend to use. This allows our "XP customer" to create tests accordingly. A company that, for example, developed "shrink-wrapped" software would be unable to do this: much more up-front effort on testing is required as many people will be using the software in unpredictable ways.

The Carmen applications and their usage are well suited to writing automated acceptance tests. There are products which are much more dependent on variations in hardware, operating system and network platform. For these, fully automated acceptance tests are not possible without replicating many of those variations, which might well be practically and economically impossible.

There are also products (such as interactive GUIs) whose natural mode of operation is not batch. The challenge of being able to re-use the technique then rests on being able to simulate the interactive part using some sort of script. Tools (such as playback tools) exist to do this, though these have a reputation of being somewhat fragile under code changes. We haven't really tried them ourselves beyond small prototypes.

As a bottom line, it can be done "by hand" by creating a scripting interface that slots in just beneath the GUI layer itself. Carmen's own user-interface team has just completed creating such an interface, though more investigation is clearly needed into the effectiveness of this.

3 Reflections

3.1 Reflections of a new team member (Johan Andersson)

In the autumn of 2002, Johan Andersson joined Carmen Systems and the team. Here are his reflections based on his earlier experience with XP and more traditional unit tests.

Before I joined Carmen Systems I was developing a network security product within an 8 person development team. This team had been trying to use XP as its development methodology for two years, and had been fairly successful in doing so. As this team were doing traditional unit tests, I believe I am qualified reflect on the differences in the approach of the Carmen Systems team.

Having done unit tests and Test First Design, I find the process of using acceptance tests only (with "Usage-first Design") to be an application of the 'You Ain't Gonna Need It' principle. My experience is that the vast majority of unit tests never break alone. From an error detection point of view, if a test never breaks alone it is not needed. That is, if two tests always break in unison, you need only one test. Therefore, if a broken unit test is always accompanied by a broken acceptance test, then one of them is not needed.

You could argue that having many smaller tests would help in pinpointing the cause of the error, but then you would be saying that some work now in creating these tests could possibly save some debugging work later on. I do not think that is in the spirit of XP.

In practice this speeds up the development process. The Carmen team gain this speed by sacrificing detailed failure information and checking of borderline cases in the code, as given by unit tests, while still keeping the failure indication and verification of the actually used functionality as given by the acceptance test.

3.2 Reflections from the Customer (Peter Sutton)

The customer accepts full responsibility for the behaviour of the system - even under unusual circumstances. As a result the customer gains full control over time spent developing all aspects of the system. If the customer wants the software to act in a particular way when the input data is incorrect then the customer needs to write a story specifying what that behaviour should be and provide a test case.

The advantage of this approach is that these non-functional requirements are made explicit and are prioritized along with other stories. The disadvantage is that the customer has to think of all possible things that could go wrong - an impossible job. The customer needs then to be able to accept that when something unexpected happens it is his responsibility to determine the significance of the problem and determine whether to create a new story or not. There is thus an attitude of "learn as you go" and an assumption that it will be possible to further update the software even after it has been deployed.

4 Conclusion

Given a business environment where implementation is a controlled process, and an application that runs or can be made to run in batch mode, we believe that Acceptance-Test Driven Development is an effective means of carrying out testing within an XP project. We also believe that it is within the spirit of XP, in that it is simpler and involves less effort invested up-front in the hope of a later pay-off. We hope that our success can encourage others to try out this approach.

5 Appendix - TextTest

TextTest is an application-independent configurable framework for text-based functional testing, primarily regression and performance testing, but also stress testing.

The expectation is that tests are written, run and configured using simple text files only, and that information in the file system itself is used wherever possible. It is not intended that new code should need to be written when a new test is created.

To be tested using TextTest, an application needs to be runnable from the command line using standard options and/or information in standard input. It needs to be runnable in batch mode so that fully automatic tests can be created. It also needs to produce meaningful output that is, or can be converted to, plain text. It will then use this text to monitor and control the behaviour of the application.

Structurally, it is composed of a core framework that is mainly concerned with managing and interpreting the files and directories that constitute a TextTest test suite, and a range of extendable configurations that manage how tests are run and when, which files are compared, how to create reports and so on. This allows users to write their own configurations to take advantage of local circumstances, and also provide platform specific configurations.

It is written in Python. The intention is that it will be available for free download from http://www.carmensystems.com by the time this paper is published: at least a mailing list will be set up for interested people.

References

1. Beck, K.: Test-Driven Development, page 199. Addison-Wesley, 2003.
2. Beck, K.: Extreme Programming Explained : Embrace Change. Addison-Wesley, 1999.
3. LSF is available from Platform Computing at http://www.platform.com
4. log4j can be found at http://jakarta.apache.org/log4j/. A C++ version, log4cpp, exists, but licensing difficulties meant that we were unable to use it.
5. CppUnit can be found at http://sourceforge.net/projects/cppunit/
6. Bache, G. and Bache E.: "One Suite of Automated Tests: examining the Unit/Functional divide" in Proceedings of the 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002). Italy, 2002.
7. TextTest is free and can be found at http://www.carmensystems.com
8. http://jakarta.apache.org/log4j/docs/manual.html. The passage concerned is itself quoting Brian W. Kernigan and Rob Pike's book "The Practice of Programming"
9. van Deursen, A. and Moonen, L.: "The Video Store Revisited - Thoughts on Refactoring and Testing" in Proceedings of the 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002). Italy, 2002.
10. Mackinnon, T., Freeman, S. and Craig, P.: Endo-Testing: Unit Testing with Mock objects, in Extreme Programming Examined. Addison-Wesley, 2001.
11. Doxygen can be found at http://www.doxygen.org