

One suite of automated tests: examining the unit/functional divide

Geoff Bache
Carmen Systems AB
Odinsgatan 9
411 03 Göteborg, Sweden
+46 (0) 31 720 8137
geoff@carmen.se

Emily Bache
Astrakan AB
Maskingatan 5
417 64 Göteborg, Sweden
+46 (0) 31 779 35 14
emily.bache@astrakan.se

ABSTRACT

Extreme Programming (XP) as written [1] prescribes doing and automating both unit and functional testing. Our experiences lead us to believe that these two sorts of testing lie at two ends of a more or less continuous scale, and that it can be desirable to instead run a XP project with just one test suite, occupying the middle ground between unit and functional. We believe that this testing approach offers most of the advantages of a standard XP testing approach, in a simpler way. This report explains what we have done, and our theory as to why it works.

Keywords

XP, Automated testing, Functional testing, Unit testing, Test First Development

1. INTRODUCTION

When we introduced XP at Carmen Systems, the worst problem with our development process was not our testing procedures being out of control. We already had automated testing, though not along the lines outlined by Beck, Jeffries et al [1, 2]. Following the advice to “Solve your worst problem first”, we began introducing other aspects of XP, expecting that at some point testing would become our “worst problem” and we would start needing separate unit and functional test suites. That never seemed to happen - we have been doing all the other XP practices in 2 projects for 18 months or so, and our style of automated testing has not only not become a problem, but in fact a great success that seems to fit very well with the rest of XP.

The automated tests we have are perhaps best explained as “pragmatic acceptance tests” - we run the system as closely as possible to the way the customer will run it, while being prepared to break it into subsystems in order to allow fast, easily automatable testing. The overall effect is that the tests are owned by the customer, while being just about fast enough to be run by the developer as part of the minute by minute code-build-test cycle.

2. THE CARMEN TEST SUITE

What we have created is an application independent automatic testing framework written in Bourne shell and Python. The framework allows you to create and store test cases in suites, runs them in parallel over a network, and reports results. For each test case the framework provides stored input data to the tested program via options or standard input redirects. As it runs, the tested program produces output as text or text-convertible files. When it has finished, the testing framework then compares (using UNIX “diff”) this output to version-controlled “standard” results. Any difference at all¹ is treated as a failure. In addition, the framework measures the performance of the test, and if it strays outside pre-set limits, (for example if it takes too long to execute) this is also recorded as failure.

New tests are added by providing new input options and running the system once to record the standard behaviour against which future runs will be measured. This behaviour is carefully checked by the customer, so that s/he has confidence the test is correct. Once verified, the new test case (ie input and expected results) is checked into version control with the others.

Of course, not all differences in system behaviour are undesirable, and it sometimes happens that a test failure is registered even though the new system behaviour seems as good as or better than the old. If this happens, it is up to the developer who made the code change that caused the test to fail to confirm with the customer that the change is desirable, and then check in the new standard results of the test(s). They must also add a comment explaining why the new behaviour is an improvement on the old. In this way the behaviour of the system can evolve in a fully controlled way.

We have been very successful using this technique at Carmen Systems to test the decision making

¹except for run-dependent output such as times and process IDs, which the framework ignores.

middle layer of a larger application - that is the bit between the user interface and the data storage. Since we are not testing the system end to end, we are not really doing Acceptance Testing from the customer's point of view. Since we are not writing tests in the same language as the code, and are not writing tests for individual classes, we are not doing Unit Testing. However, we do get enough of the advantages of both kinds of testing to support XP.

3. STRENGTHS AND WEAKNESSES

The most important ways the testing practices support the rest of XP are by providing developer confidence to refactor and customer confidence in progress being made. The testing we do provides both of those:

✘ Most of the tests can be run in a matter of minutes, (the tests run in parallel across a network), so they can be run at nearly every build, and can provide fast enough feedback to enable merciless refactoring.

✘ Every test corresponds to real input and customer-verified output, so the list of passing tests is an accurate measure the customer can use to assess progress.

This way of testing has other advantages, too. Adding a new test is very straightforward, all it requires is finding suitable input data then having the customer confirm that the output is correct. There is no application- or feature-specific code to maintain and refactor, only the generic testing framework itself. Another useful feature is the ability to run tests in parallel, using 3rd party load balancing software to make maximal use of the computing resources available on the network. This means that the speed of the test suite is only limited by network resources and the time it takes the longest test to run.

One criticism that has been levelled at this style of testing is that without unit tests, Test First Development (TFD) as such is not really possible. Beck describes TFD as a design technique [3], and it has been reported as such by many practitioners of XP [4]. However, despite not doing TFD, we have not had difficulty creating a system composed of objects exhibiting high cohesion and loose coupling. We have also not had difficulty evolving the design via merciless refactoring as new user stories are implemented. In short, our experience suggests that TFD is not the only way to evolve a good design within an XP project.

4. CONCLUSION

In this practitioners report we have outlined our experiences with automated testing in the middle ground on the scale between unit and acceptance

testing. Our main conclusions are that since the customer is far better qualified than the developers to specify tests for the system, they should specify the tests. On the other hand, the power of placing testing in a very tight feedback cycle within development is essential to enable refactoring and agile design, so the tests must run quickly. If we can have one suite of tests that is both customer owned and fast to run, we have a powerful tool to support a simpler process than XP as written - with one type of testing rather than two.

REFERENCES

1. Beck, "Extreme Programming Explained"
2. Jeffries et al, "Extreme Programming Installed"
3. Beck, "Aim, Fire"
<http://www.computer.org/software/homepage/2001/05Design/>
4. Community discussion, for example
<http://www.c2.com/cgi/wiki/TestDrivenProgramming>