What Makes Testing Work: Nine Case Studies of Software Development Teams

Christopher D Thomson*
Business School
University of Hull
Hull, UK
c.thomson@hull.ac.uk

Mike Holcombe, Anthony J H Simons
Department of Computer Science
University of Sheffield
Sheffield, UK
{m.holcombe,a.simons}@dcs.shef.ac.uk

Abstract— **Recently there has been a focus on test first and test driven development; several empirical studies have tried to assess the advantage that these methods give over testing after development. The results have been mixed. In this paper we investigate nine teams who tested during coding to examine the effect it had on the external quality of their code. Of the top three performing teams two used a documented testing strategy and the other an ad-hoc approach to testing. We conclude that their success appears to be related to a testing culture where the teams proactively test rather than carry out only what is required in a mechanical fashion.**

Testing; test first; test driven development; extreme programming;emprical; qualitative; testing culture.

I. Introduction

Extreme programming (XP) [1] presents what is, on the surface, a simple but effective testing practice known as *test first*, or *test driven development*. The idea is reassuringly simple, that unit tests should be defined and run before any implementation is present. Several studies have attempted to measure the effect of the *test first* practice on the quality of the software produced and time taken, but the results presented are inconclusive.

The testing practice of XP encompassed more than simply *test first*. System testing is automated, incremental, regular, and early. User acceptance testing is similar although often less or not at all automated [1]. But these features can be used independently of *test first* and perhaps with some success. This raises our research question:

How does the practice of testing effect a team following XP – or if test first is not followed then do the other practices of XP still influence the way testing is performed and the external quality?

We collected data from nine teams, which we present as case studies. In the case studies the teams were novice users of XP, who we provided with training. They were given the option of using *test first* and whilst three teams expressed enthusiasm they ultimately did not follow the practice accurately. We found that the teams had a high degree of variation in external quality that cannot be easily explained by the teams' testing practice alone or the practices of XP alone. Instead we find that testing must become part of the culture of the team, and can do so in at least two different ways.

II. Literature Review

Test first (TF) is an established development technique which is essentially the same as *test driven development* (TDD). In both cases the aim is to write tests before writing the functional code. This should in theory aid development as the tests form the basis of the specification, design and functional tests, whereas testing after the code is written is regarded as only a testing technique [1; 2]. Whilst TF and TDD are well defined, the traditional or *test last* technique is interpreted differently by the studies in the literature investigating this phenomenon. The studies' definitions of *test last* can be divided into roughly four categories, which we will refer to as TL 1 to 4:

TL-1: Unspecified traditional method. Most experiments provided at least a few hints as to the method that the non-TF teams should follow, however some did not [3-5]. The following statement was typical of the broad definitions that these papers used: "the control group which followed the traditional process" ([5], p132). As no further discussion was made about the process followed we can't make generalizations about what affected their performance.

TL-2: Specified traditional method. This method is typified by manual or ad-hoc testing and was used in three studies [6-8]. The method was defined thus: "no automated tests were written and the project was developed in traditional mode with a large up-front design and manual testing after the software was implemented." ([7], p71) and "a conventional design-develop-test (similar to waterfall)" ([6], p339). Lastly this study also noted that whilst the teams had been asked to use unit tests, only one team wrote any [6].

TL-3: Unit tests written at the end of the development cycle. Two studies identified a method that used unit testing at the end of the development cycle [7; 9]. These studies were able to provide some metrics on coverage to define the amount of testing undertaken, but assumed that all tests were only written and used at the end of the project. One study

*Author contributed whilst at the University of Sheffield.

also noted that in this method there was late integration of the code [9].

TL-4: Unit tests written just after the code in an iterative development cycle. In this final group the test method was analogous to *test first* in all regards except that: "automated tests were written shortly after code in an iterative test-last fashion" ([7], p87). Of the five studies that used this definition, three had iteration periods the same as test first [7; 10; 11], and two had longer iteration periods [12; 13].

With these definitions in mind we can explore the results of the previous experiments, which at first glance appear contradictory. Two studies have targeted the design-enhancing claims of TF. One group of studies showed that the quality of the design produced as result of TF is better, in that the units are less complex and smaller than those developed using TL-2 and TL-3 methods [7]. However other recent empirical evidence suggests that TF in some cases degrades the quality of the design when compared to TL-4 [11], this was also apparent in one of the earlier individual studies which used TL-4 [7].

The early studies that compared TF to TL-1 in terms of performance were conducted by Müller in an academic situation [5]. He found that TF teams were no quicker but that there were marginal benefits to reliability and that functions were reused more frequently. The other studies in an academic setting had conflicting results. In the first TL-4 was compared with TF [10], where it was found that TDD was marginally less efficient, with marginally higher quality (not significant) [10]. The second study, again in an academic setting, altered the work cycle so that the TL-4 group constructed larger chunks of code before testing, than the TF group. The authors concluded that quality between the teams was the same, but that TF wrote more tests, and their productivity was higher but not significantly so [12].

Three TL-2 studies in academia found similar effects: productivity was increased (but this was not significant, perhaps due to a high degree of variability in the data [3]) and that quality was about the same [3; 7; 8]. In one study it was noted that the increase in productivity was due to an increase in testing time, with a decrease in coding time [3]. Another study identified that TF developers generated more tests [7]. In contrast a TL-1 study found that the TF team were 50% faster in development but this result was compromised by the fact that the TF group had more experience in software development [4].

To date, the results most favorable toward TF have come from industrial studies [6; 7; 9; 13; 14]. In the case studies the first found that the introduction of TDD gave a 40% reduction of defects with the same overall productivity against TL-3 [9]. The second case study found a reduction in defects of 62% in TDD compared to TL-1 [14]. A final study looked at several development projects and concluded that when using TF mature developers delivered less complex code, in smaller units compared to TL-2 and TL-3 [7]. In the first comparative study the TDD group passed 18% more tests than the TL-2 group but took 16% more time to do so [6]. In the second, no significant differences were found between TF and TL-4 and the researcher concluded that the time permitted (2 hours) was not enough for differences to emerge [7]. The final study found that the TF teams wrote more tests and ran them more frequently than TL-4, although this might have been influenced by the experimental procedure which required TL-4 teams only to generate tests towards the end of a story implementation [13].

The previous literature gives contrasting results for similar experiments, in summary the results for TL-1 show that TF is either as good as or better than TL-1, against TL-2 and TL-3 TF is better, and against TL-4 the results are more mixed. The studies using TL-4 suggest that the issue of iteration may be important. Three of the five studies that included a concept of TL-4 concluded that TL-4 maybe better that TF [7; 10; 11] the exception was based on a version of TL-4 where the TL practice was iterated at the end of a story, whereas TF was at the level of a sub-story [12; 13]. It could be that the effect of the testing frequency is more important than when the tests are written. Thus deep exploratory studies on testing methods are required in order to identify confounding factors [15].

The literature therefore supports the argument that the practice of testing takes many years for practitioners to develop; nonetheless, novice developers often deliver good quality software – although quality can be varied. This basic aptitude or naïve method can confound the results of studies that attempt to evaluate empirically the effectiveness of well founded methods. In this paper we will use a qualitative analysis to uncover the naïve methods used by three successful teams. To do this we observed the testing practices and their effect in nine case studies of student teams following the XP method.

III. Study Context

In order to gain some further insight into the effects of the test techniques TL1-4 as used by novices, data was collected about the testing process followed by nine teams. The teams selected their own members, each consisting of between three to five students [16]. The teams were composed of second or third year undergraduate students. The students had no previous instruction on XP but had experience developing software using a plan-driven approach, which was presented in nine hours of lectures and tutorials [19].

The development projects ran over twelve weeks, for fifteen hours a week. There were three industrial clients who each provided a project, two required database driven websites (B and C) and the other an e-learning environment (A). Each client represented a small business and came with a project brief, none had experience in commissioning custom software. Client A had the most expertise in computing, but this was derived from his degree several decades previously having since moved into a different area.

The distribution of teams to projects follows the randomized complete block experimental design [17]; however in this study we treat the teams as a multiple case study [18]. The relationship between projects and teams is shown in Table I, along with the programming language used by the teams.

TABLE I. Projects and Teams

	1	2	3	4	5	6	7	8	9
Project	C	B	B	C	C	B	A	A	A
Language	P	P	CP	P	P	P	J	J	J

P – PHP, J – Java, CP – existing PHP program customized.

We collected data weekly in order to examine the testing process over time. The students were instructed to submit their test logs, test code/document, time spent testing and program code having run the tests on a lab computer. For this project, the authors selected three popular unit test tools to be used by the teams: JUnit (Java); PHPUnit (PHP classes) and Selenium (PHP pages).

IV. Measurement and Evaluation Method

In order to establish how successful the tests were, we recorded the number of test files present for each team and their outcome if run. Each week the number of tests was calculated by counting the tests found in the teams' working directories.

To investigate the value of the testing process we calculated the ratio successful test to unsuccessful test outcomes to against the eventual external quality. To determine the outcome of the test we referred to the logs. We recorded each batch of tests submitted as a single test and if one or more tests failed then a "fail" was recorded. In many cases the tests submitted did not run because they were incorrectly configured, in this case we recorded that they "did not run". It was not always clear if manual tests documented had been run, therefore we recorded if they were successful, if they were changed in the week, or if there was evidence that they had been run, otherwise they were recorded as "not run". These tests were counted per test document which contained one or more tests.

The test coverage was calculated by examining the tests present in the final week of the project. This measurement would reveal if coverage were related to successful projects. This was calculated in terms of class coverage by the tests provided. We define class coverage in a weakly as the number of classes (files in the case of poorly structured PHP) which had one or more test cases. We also identified two additional types of test (which we counted within the manual tests before) which were partially automated by the teams: Abbot tests to test Java user interfaces [23]; and a custom test harness which was produced by team seven.

We used the client's mark to judge the external quality [24] of the product produced. Unlike other methods of assessing quality via humans, such as inspection, only one person assesses the quality. As the population of clients for each software system is exactly one, we sampled the whole population. Thus whilst an agreement score cannot be computed in the traditional sense, we can be confident that the score is correct. In addition there is no evidence that other product measurements are correlated to client satisfaction. Another approach would be to collect post deployment metrics (bug reports and so on); however this would not be appropriate here, as three products were developed for each problem and only one of these was selected to be used in production. Lastly, whist users of the software could also be surveyed; this is prone to problems in relation to the requirements specification, where the users and client are not in agreement. So, to measure the success of a project, the only acceptable method is to interview the client.

The measurement of external quality (50 points in total) was split equally between the categories: *demo, manual, install guide, maintenance guide, ease of use, understandability, completeness, innovation, robustness* and *happiness*. They are based on the products required as part of the course (first four items) and secondly on product quality metrics as discussed by Fenton [25]. A detailed description is available in the lab pack [20]. This process selected the three winning teams (3, 5 and 8).

Each developer completed a self assessment at the end of the project. In this paper we consider the responses to the open questions shown in Table II. The complete instrument is in a lab pack [20]. This captured the developers' individual performance, which the developers described in their own words.

Lastly, to evaluate if other project management factors were more important for predicted quality, each member of the team also answered a survey based on the Shodan questionnaire which is used to evaluate XP adherence [21]. We used a shortened version, as questions on the *metaphor* and *changes to release plans* were not relevant for our teams; the full set of modified questions is available in a technical report [22]. To calculate a value of adherence for each team we took the mean value of the responses from the team members.

V. Description of the Cases Studied

The teams were rated by the clients both in summary (as summarized later in Table VI) and an overall grade (Table III). The winning teams 3 and 5 scored the highest, but despite winning for their client, team 8 scored slightly lower than these teams and one of the others (team 6).

Our first observation on test technique was that no team wrote tests before they wrote their code and all used a variant of TL-4. Typically a team member would develop the tests on his own local copy of the system before bringing them into the lab to run and record the tests, but configuration issues at this point meant that many of the tests failed to run correctly – leading to the tests not being run (Fig. 1). For example team 1 mostly found that their tests did not run (although they attempted to run them most weeks) with only two cases where they failed. In contrast team 4 was more successful with the tests running and failing after week 6 and

TABLE II. Self Assessment Survey Questions (extract)

In your own words please comment on your achievements.
In your own words please comment on your failures.
In your own words please comment on your role in the team.

TABLE III. Table III: External Quality for Each Team

Team	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Mark	29	33	39	29	38	35	28	34	34

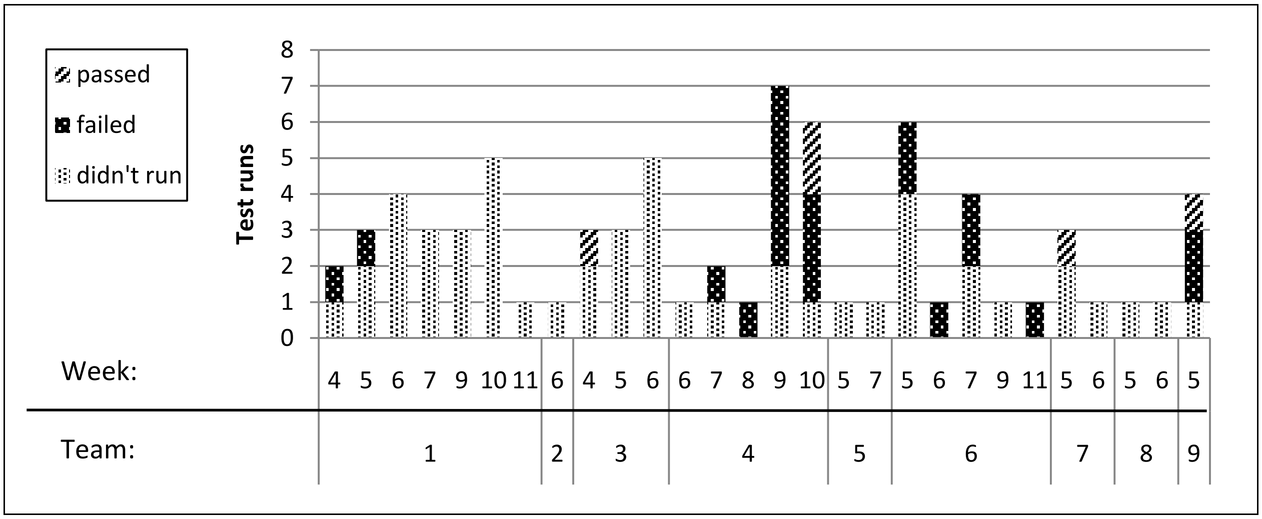

Figure 1. Frequency of automated testing.

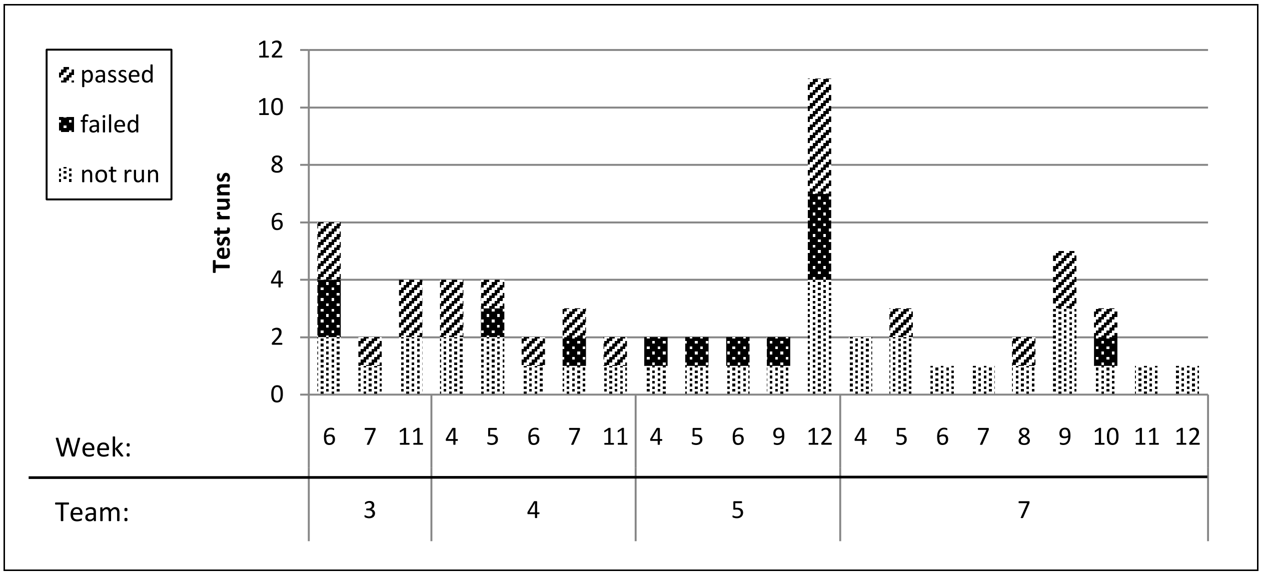

Figure 2. Frequency of formal manual testing.

being successful in week 10. Therefore when we examined the cases, we used these results to indicate when tests were being run by the team, rather than considering the outcome.

Fig. 2 summarizes the data collected on when manual tests were run. Only four of the teams recorded manual tests (3, 4, 5 and 7). The overall testing strategy in terms of code class coverage for each of the teams is summarized in Fig. 3. This shows that only teams 4 and 7 mixed manual and automated tests with team 7 using their own custom test harness and Abbot. Teams 3 and 5 only used manual tests (although they had automated tests, these had insignificant low coverage). Teams 1, 2, 6 and 8 only ran automated tests and team 9 ran both automated tests and Abbot tests.

Fig. 4 shows the time that the teams reported for testing. This mostly correlates with the amount of testing shown in Figs. 1 and 2, with the exception of the manual testing for team 6, where the team did not record any documented manual tests.

We pre-tested all the developers to establish their basic level of ability before the start of the project. Table IV shows the mean pre-test scores for each team, each out of 100. A previous result on a larger data set [16] showed that these tests could account for around 27% of the variance in the mark for external quality as awarded by the client based. For this set of data there does not appear to be any direct relationship between the pre-test score and external quality. Thus ability does not appear to be strongly related to performance for these teams.

Table V shows the responses from the Shodan questionnaire. The data clearly shows that the teams all interpreted and applied the XP practices differently. For example for pair programming team 1 scored 8/10 and teams 3, 5 and 7 scored 3/10. We consider the impact of this effect in the discussion.

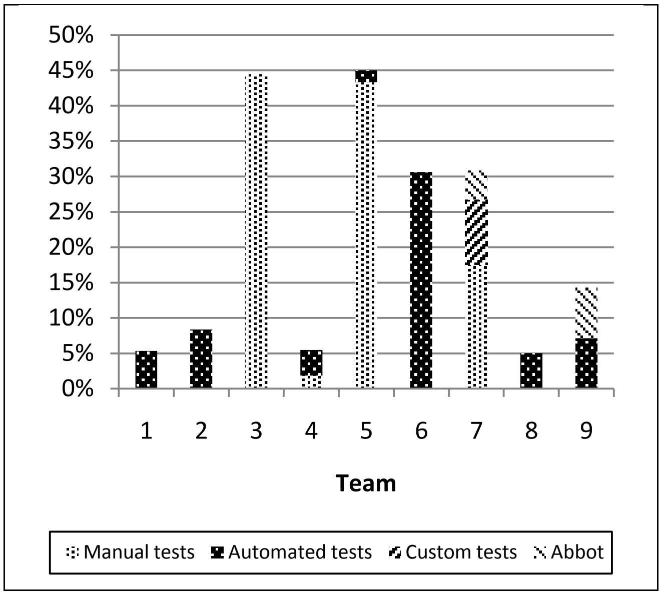

Figure 3. Class coverage at the end of the project.

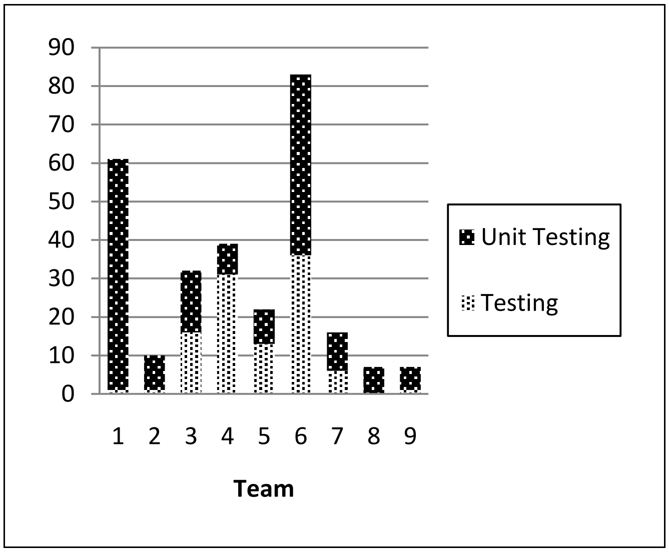

Figure 4. Time in hours spent testing.

Lastly, the data collected from the developer surveys was summarized along with comments received from the client (Table VI). For easy reference each column is labeled by a letter and each row by the team number. Empty cells indicate that either the team or client respectively did not comment on that subject. In some cases the client or team was vague in their descriptions of the problems encountered and this is reflected in evidence presented in Table VI. A brief analysis of the table indicates that the teams were highly variable in their approaches. In the discussion that follows we use the descriptions in Table VI to explore the interpretation of the other data.

TABLE V. SHODAN SCORES FOR EACH TEAM

	Team	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	Automated unit tests	4	4	0	2	1	4	2	3	2
B	Customer acceptance tests	2	2	0	1	1	3	4	0	3
C	Test-first design	5	4	0	2	1	3	0	0	3
D	Pair programming	8	6	3	5	3	5	3	7	4
E	Refactoring	3	5	2	4	3	5	4	4	2
F	Release planning/planning game	4	4	5	3	4	5	2	4	2
G	Short releases	5	7	4	1	5	3	3	3	5
H	Stand-up meeting	1	6	0	8	3	6	0	1	2
I	Continuous integration	7	5	2	6	4	6	5	9	4
J	Coding standards	9	6	7	8	4	6	8	9	4
K	Collective code ownership	7	6	5	8	6	9	6	9	5
L	Sustainable pace	7	6	6	5	8	7	7	5	7
M	Simple design	6	6	5	6	6	7	8	6	7

VI. DISCUSSION

A. Overview

Initial investigations into the quantitative data presented Figs. 1-4 and Tables 3-5 found no significant correlations between external quality and the various measurements. Furthermore the qualitative data in Table VI also gives no clear indication as to why certain teams produced the best systems. Therefore a more detailed analysis was required to compare the teams.

In terms of success, the assessment of the teams' products by the client was conducted using the external quality measurement (Table III) and comments (Table V). The clients' comments highlighted two things that varied between the teams: the completeness of the functionality included and the robustness of the system developed (Table VI: C and D).

B. Factors that Led to Success and Failure

Completeness seems to have been a particular issue for three of the teams. In two cases there were instances of members not being fully involved (Teams 2 and 9, Table VI: C) and in team 2 the willingness of one member take on the majority of the coding and thus being overloaded (Table VI: D). Team 4 was reported both by the team (Table VI: D) and the client as being disorganized (Table VI: E). Robustness was a factor for two of the other teams (Table VI: F): In team 1 this may have been due to the time wasted on automated tests that never functioned as intended (Table VI: A-C), and for team 6 the problems with the tests not keeping pace with the code evolution (Table VI: A). Lastly team 7 was the only team where the client had serious difficulties understanding and installing the software (Table VI: F). In the end he was

TABLE IV. PRETEST SCORES FOR EACH TEAM [16]

Team	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Pre Test 2	66	64	72	74	65	71	69	58	76
Pre Test 3	67	58	62	77	72	69	53	65	69

TABLE VI. TABLE VI: QUALITATIVE OVERVIEW OF THE TEAMS

	A	B	C	D	E	F
Team	**Automated testing**	**Manual testing**	**Team involvement in testing**	**Team Communication**	**Client impression**	**Client assessment**
1	Errors in the test script were difficult to resolve: "Our major failure was [the lack of] a runable test script". After the first two weeks they abandoned the automated methods as they felt they were falling behind.	Everyone, around 60 hours total.	Frequent contact using a variety of online tools. Frequent meetings and pair programming.	Slow to start but then worked well and creative.	Functionally comprehensive product but some parts of the system failed.	
2	The tester "found testing first with PHP time consuming and challenging".	Mostly ad-hoc as the code was written.	One member was did most of the coding and testing.	One member often did not turn up for meetings. They relied on MSN for communication.	Not very innovative.	Brief documentation, some missing functionality, robust.
3	Hard to write as they modified an open source product. This interfered with the tests.	Detailed manual tests in the second half of the project.	Several members.	Used multiple lines of communication and met frequently.		Documentation was clear, but was complex to use. Mostly functionally complete and robust.
4	"The selenium testing took a long time to complete successfully".		One member who was also responsible for the website.	Ad-hoc meetings where "jobs were never really assigned so we didn't know who was doing what"	Laid back and lacked urgency. Had some good ideas.	Not all functionality was delivered and the documentation was not detailed.
5	Did not use PHP classes so could not use PHPUnit.	"We never really did enough documented testing".	Members worked on tests for their own code.	Worked together in the lab but did not pair program or integrate their code frequently.	Started slowly.	Some features did not work as expected, but more robust than team 1. Provided features were comprehensive and creative.
6	Greatest number of tests. But these were reported to be difficult to maintain as the project evolved.	Manual tests were used but undocumented.	One member who spent most of his time testing.	Shared their code regularly, but did not integrate it.		Documentation was comprehensive but not detailed enough. Was not functionally complete and some functions reported errors.
7	Did not understand how to run or write unit tests. Used a custom test harness.	Used both manual and Abbot tests.		Communicated mainly by email with few meetings.		Did not install and run properly and documentation was too complex.
8	11 unit tests: spending time on testing "could jeopardize the completion of the project."	Very little testing recorded.	Two members.	The team frequently integrated their code (up to 3 times a week). Communicated frequently to set tasks and check progress.		Clear documentation, with some unexpected but not erroneous results.
9	JUnit was used.	Focused on ad-hoc testing, Abbot found to be unreliable.	Two members who did most of the work.	Described their team as having poor team-working skills.		Documentation was brief but clear. Was not easy to use, and some options were hidden or disabled. Some features difficult to understand.

forced to mark their system based on a demonstration that the team gave him, with little time for him to explore it by himself. In summary most of the non-winning teams failed in a single area, most likely as an oversight.

This leaves us with the winning teams. Teams 3 and 5 produced extensive manual tests in the second half of the project (Fig. 2), although team 5 left the writing of most until the last week of development (Fig. 3). The comments from the members of team 5 suggest that they did ad-hoc testing prior to this as the whole team highlighted their lack of documented tests (Table VI: B). Team 8 took a different approach and, based on their comments (Table VI: A), were more inclined to deliver functionality. By inspecting team 8's code we found that they developed it in a highly iterative way. When we inspected team 8's directory and we discovered that there were working areas for each of the members as well as a combined solution. The combined solution was copied up to three times a week as the individual elements were integrated; this showed the regular integration of the system. The ad-hoc testing associated with the iterative development appears to have ensured that they kept the project working. The client's comment about unexpected but not erroneous results adds further weight to this argument (Table VI: F), as ad-hoc testing may be less likely to identify such problems if everything seems to be working fine. Given that both iteration and testing could be important factors we can reassess the remaining teams concentrating on these factors.

C. The Effect of the XP Process

The Shodan questionnaire (Table V) measured compliance to XP practices. Team 8 ranked highest on several questions (coding standards, shared code ownership, continuous integration and pair programming). This fits well with the other observations of team 8's method; however other teams (notably team 1) have similar patterns of process on the same questions but did not do so well overall. Furthermore it is surprising, given the iterative nature of the programming style that team 8 adopted, that they should rank so low on the short releases measurement. This indicates that their success was not dependent on client feedback (as they did not give releases to the client) but on internal feedback mechanisms within the team. Lastly the other two winning teams (3 and 5) did not rank highly on these questions with the exception of team 3 which was ranked fourth on the short releases question only.

It is important to note that in the responses to each of the Shodan questions only one of the winning teams (3, 5 or 8) was present in the top four, indeed in several cases a winning team was also at the bottom of the scale. Thus the practices that were related to the questions were not universally important to success. We also investigated if it was the combination of factors that was important. As previously noted, despite ranking highly on similar questions, teams 8 and 1 had different outcomes. The main failure of team 1 as assessed by the client was that several parts of the system did not work (Table VI: F). The Shodan survey showed two differences between these teams: team 1 was ranked fourth for collective code ownership whereas 8 was ranked top (Table V) and team 1 attempted more unit testing but with similar coverage to team 8 (Table V, Fig. 3).

Therefore we can speculate that a team using a highly iterative process, sharing its code and integrating this in regular builds is more likely to achieve greater external quality than one which does not share its code as frequently and relies on automated testing with low coverage.

D. Success Without Documented Testing

The conditions for success, as achieved by team 8, can be further refined by examining the problems the other teams encountered. For example, in team 2 only one member focused on writing the code for the system (Table VI: C) making activities like pair programming less advantageous (as there is no need to disseminate information and the pair observer may have had very little interest in the code being written) (Table V). In team 6 one person focused on testing (Table VI: C) and as a result found it hard to keep pace with rapidly evolving code (Table VI: A) as it was refactored (Table V – ranked top for refactoring).

Thus we can broaden the definition of the advantageous method used by team 8 to include the observation that regular integration (by inspecting their code) that was led by two members (Table VI: C) led to ad-hoc code review and testing (Table VI: D). This was more effective than the other methods because of its regularity. We might then speculate that this led to team 8 being acutely aware of the tests required without the documentation process which teams 3 and 5 used. Furthermore as teams 3 and 5 did not benefit from the iterative approach we might further speculate that these teams required the documented testing method to ensure quality.

E. Success Using Documented Testing

In terms of manual tests, teams 3 and 5 achieved class coverage of 40% spending 20-30 hours testing mostly in the second half of the development period (Figs. 1, 2, 3 and 4). Two other teams recorded notable amounts of documented manual testing (4 and 7) and team 6 spent around 30 hours manually testing without documented records (Figs. 2 and 4). Team 4 regularly ran their tests from early on in their project (Figs. 1 and 2), but the tests only had low class coverage (Fig. 3), but despite this the client did not address the issue of robustness of the code, commenting instead on missing features and lack of urgency towards the end (Table VI: E and F). Team 7 achieved 30% class coverage for their tests (Fig. 3) but their product was too complex for the client to understand (Table VI: F). Team 6 did not leave evidence of manual tests but recorded time doing them so these must have been ad-hoc (Fig. 4). As with the unit testing for team 6 it seems these tests did not keep up with the final evolution of the code (Table VI: A), allowing the faults into the final version (Table VI: F).

Thus to be successful, the tests must be accurate and up to date. For two of these teams, this seemed to be an issue but for the other team, although their code was robust, it was not complete. Thus we speculate that for testing to be effective the tests should be reviewed to ensure they are current and that this cannot be carried out by a single member effectively.

VII. CONCLUSIONS

A. Summary

The most striking thing about the cases presented is the differences between the teams. All the teams were following much the same development technique with some variability and the evidence supports this. The variability in the process was noticeable in the testing method used (Figs. 1 to 4) and the way the team approached the development task with regards to the practices of XP (Figs. 7 and 8). Analysis of the cases suggested that none of the practices of XP, or testing alone, or XP and testing in conjunction could by themselves guarantee a high level of external quality.

- A balance between testing and programming.
  - Where testers are part of the programming team.
- More than one person responsible for testing.
- Tests kept current.
- Either:
  - Documented testing, increasing towards the end of the project, or
  - Ad-hoc throughout the project combined with regular integration of code.

Figure 5. The characteristics of a testing culture.

The winning teams all carried out testing that could identify errors, some documented and some ad-hoc. To achieve this it was apparent that a review process was required to ensure tests developed kept pace with evolution in the code. This could be achieved by documented tests, or ad-hoc tests that the team was aware of, due to their regular repetition. To achieve an effective review process more than one team member was needed to be involved in testing, so that the changes to the code could be identified.

B. *Testing Culture*

Teams 3 and 5 knew what to test because they had a document test plan; team 8 knew what to test because they had a culture of repeating a familiar set of ad-hoc tests. In both cases a 'testing culture' was present and the quality that client required was delivered. The 'testing culture' for the winning teams has the characteristics listed in Fig. 5.

However for teams to win, well tested code was not enough. They also needed to completely satisfy the requirements of the client. The teams which did not wholly satisfy the requirements were disorganized in some way often due to members not fully taking part in the development process.

C. *Recommendations*

In summary, for teams to achieve the highest levels of external quality, as judged by a client, the team needs to have a testing culture and be organized enough to deliver the client's requirements. Both *test-first* and a highly iterative *test last* method met the requirements for a testing culture which explains the previous positive results for these methods in the literature. A less documented or less iterative *test last* method is perhaps less likely to meet the requirements for a strong testing culture, leading to the negative results found in comparison.

Based on the analysis of data collected, it is recommended that a testing culture is fostered in development teams, particularly if their members are novices. What counts is not so much the chosen testing method, but proactive involvement in testing, either through frequent ad-hoc testing or documented testing that increases towards the end of the project. In both cases the successfully collaborating team should contain at least two testers who are also involved with at least two testers who are also involved in the programming task.

D. *Validity*

We should consider the weight of the evidence in the light of the *theoretical* and *literal replications* of cases presented [18]. *Theoretical replications* represent those where the context or method is significantly different, thus we expect *theoretical replications* to have different outcomes. Literal replications are where the context and method are similar, and similar results are expected.

In terms of *theoretical replication* no *test-first* cases were available although a variety of TF-cases in the TL-4 group were available. Unlike case studies found in the literature *unit*, *manual* and *ad-hoc testing* techniques were also investigated. Given this partitioning we had teams in each of the three replications thus we had some *literal replication*, although some teams fell into multiple partitions. However, the instruction given to the teams to help them use unit tests was clearly not sufficient, given the number of tests that failed on technical grounds. Thus the relationship between the cases studied and unit testing is not *theoretically replicated*.

Given the results of the analysis, the *theoretical replications* as partitioned by testing method was not enough to explain the differences and some consideration to testing culture must be given instead. This suggests that further *theoretical replications* should be planned to investigate the effects of strong testing culture and unit testing, since there were no cases studied thus far with both.

E. *Future Work*

The emergence of a testing culture with novice developers during highly iterative development and regular integration is an alluring concept. In general, software developers are not keen on testing. Extreme programming and *test-first* have gone some way to address this, but writing unit tests, even after the code exists, is still challenging, as can be seen from the case studies. If an iterative method with the right features can lead to the same outcome then this is desirable if the effect can be replicated and controlled.

Therefore we recommend that future research looks more closely at the effect of iteration and how developers cope with testing during integration. Ideally a future research project should observe, in detail, a development team that plans to use an iterative approach with frequent integration and ad-hoc testing. Given the difficulty of analyzing the quantitative data in a meaningful way we recommend the use of ethnography to discover the significant features of this process which are likely to be in the detail of the observations.

ACKNOWLEDGMENT

We thank the reviewers for their invaluable help in improving the presentation of this paper.

EPSRC Grant awarded: £500K over three years (March 2006-2009) to carry out research in the Observatory. EP/D031516 - the Sheffield Software Engineering Observatory.

REFERENCES

[1] K. Beck, and C. Andres, Extreme Programming Explained: Embrace Change, 2004.

[2] B. Vodde, and L. Koskela, "Learning Test-Driven Development by Counting Lines," Software, IEEE, vol. 24, no. 3, 2007, pp. 74-79.

[3] L. Huang, "Analysis and Quantification of Test-first programming," master's thesis, Dept. Computer Science, University of Sheffield, 2007.

[4] R. Kaufmann, and D. Janzen, "Implications of test-driven development: a pilot study," Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003), New York, ACM, 2003, pp. 298-299.

[5] M. Müller, and O. Hagner, "Experiment about test-first programming," Software, IEE Proceedings, vol. 149, no. 5, 2002, pp. 131-136.

[6] B. George, and L. Williams, "A structured experiment of test-driven development," Information and Software Technology, vol. 46, no. 5, 2004, pp. 337-342.

[7] D. Janzen, "An Empirical Evaluation of the Impact of Test-Driven Development on Software Quality," doctoral thesis, University of Kansas, 2006.

[8] A. Gupta, and P. Jalote, "An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development," Proc. First International Symposium on Emprical Software Engineering and Measurement, IEEE, 2007, pp. 285-294.

[9] E. M. Maximilien, and L. Williams, "Assessing test-driven development at IBM," Proc. 25th International Conference on Software Engineering, 2003, pp. 564-569.

[10] M. Pancur, M. Ciglaric, M. Trampus, and T. Vidmar, "Towards empirical evaluation of test-driven development in a university environment," Proc. EUROCON 2003. Computer as a Tool. The IEEE Region 8, IEEE, 2003, pp. 83-86, vol. 2.

[11] M. Siniaalto, and P. Abrahamsson, "Does test-driven development improve program code? Alarming results from a comparative case study," Proc. Second IFIP TC 2 Central and East European Conference on Software Engineering Techniques, (CEE-SET 2007), Poznan, Poland, Springer-Verlag, 2007, pp. 143-156.

[12] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," Software Engineering, IEEE Transactions on, vol. 31, no. 3, 2005, pp. 226-237.

[13] A. Geras, M. Smith, and J. Miller, "A prototype empirical evaluation of test driven development," Proc. 10th International Symposium on Software Metrics, 2004, pp. 405-416.

[14] L. Crispin, "Driving Software Quality: How Test-Driven Development Impacts Software Quality," Software, IEEE, vol. 23, no. 6, 2006, pp. 70-71.

[15] D. Janzen, and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," Computer, vol. 38, no. 9, 2005, pp. 43-50.

[16] C. D. Thomson, and M. Holcombe, The Sheffield Software Engineering Observatory Archive: Six Years of Empirical Data Collected from 74 Complete Projects, tech. report CS-09-01, Dept. Computer Science, Univ. of Sheffield, 2009.

[17] B. Ostle, and L. C. Malone, Statistics in Research: Basic Concepts and Techniques for Research Workers, Iowa State Press, 1987.

[18] R. Yin, Case study research: Design and methods, Sage, 2008.

[19] M. Holcombe, Running an Agile Software Development Project, Wiley, 2008.

[20] C. Thomson, and M. Holcombe, Software Hut Lab Pack, tech. report CS-09-03, Dept. Computer Science, University of Sheffield, 2009.

[21] L. Williams, L. Layman, and W. Krebs, Extreme Programming Evaluation Framework for Object-Oriented Languages--Version 1.4, tech. report TR-2004-18, Department of Computer Science, North Carolina State University, 2004.

[22] J. Karn, C. D. Thomson, S. J. Wood and G. Michaelides, The Shodan Adherence Survey for Extreme Programming: Sheffield Revision 1, tech. report CS-09-05, Dept. Computer Science, University of Sheffield, 2009.

[23] T. Wall, "Abbot framework for automated testing of Java GUI components and programs," 2002-2008; http://abbot.sourceforge.net/doc/overview.shtml.

[24] F. J. Macias, "Empirical assessment of extreme programming," doctoral thesis, Dept. Computer Science, University of Sheffield, 2005.

[25] N. Fenton, and S. Pfleeger, Software Metrics: A Rigorous and Practical Approach, PWS Publishing Co, Boston, 1997.