Information and Software Technology 51 (2009) 182–194

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Empirical investigation towards the effectiveness of Test First programming

Liang Huang *, Mike Holcombe

Department of Computer Science, University of Sheffield, Sheffield, United Kingdom

ARTICLE INFO

Article history:
Received 6 February 2007
Received in revised form 13 March 2008
Accepted 20 March 2008
Available online 31 March 2008

Keywords:
Agile methods
Empirical software engineering
Software testing
Testing strategies
Software engineering process
Programming paradigms

ABSTRACT

The Test First (TF) programming, which is based on an iterative process of "setting up test cases, implementing the functionality, and having all test cases passed", has been put forward for decades, however knowledge of the evidence of the Test First programming's success is limited. This paper describes a controlled experiment that investigated the distinctions between the effectiveness of Test First and that of Test Last (TL) (the traditional approach). The experimental results showed that Test First teams spent a larger percentage of time on testing. The achievable minimum external quality of delivered software applications increased with the percentage of time spent on testing regardless of the testing strategy (TF or TL) applied, although there does not exist a linear correlation between them. With four years' data, it is also found that a strong linear correlation between the amount of effort spent on testing and coding in Test First teams, while this phenomenon was not observed in Test Last teams.

1. Introduction

The Test First programming is a software development practice which has been proposed for decades [21,35] and mentioned as one of the central components of Test Driven Development (TDD) [6,1] and eXtreme Programming (XP) [5]. It starts with writing unit tests for the proposed functionality using a unit testing framework. Afterwards, the programmers write code to pass these test cases, but are not allowed to write code that does not aim to pass the tests that are already produced. Code for the proposed functionality is considered to be fully implemented if and only if all the existing test cases pass successfully. Moreover, when defects are detected, the corresponding test cases are to be added before fixing the code. In so doing, the test cases are incrementally added and periodically executed, and the code is produced as small implementations after writing tests, continuously. An important rule of Test First approach is: "If you can't write a test for what you are about to code, then you should not even be thinking about coding" [9].

According to Ambler [1], the relationship between Test Driven Development (TDD) and Test First programming (TFP) or Test First development (TFD) can be described with the following simple formula:

$$\text{TDD} = \text{TFD}/\text{TFP} + \text{refactoring}$$

In other words, TDD is a set of processes for developing a whole software system, in which every "small bits of functional code" is written following TFP or TFD. If the programmers refactor every piece of code written with TFP/TFD method to ensure its quality after it works, they are considered to program following TDD.

For the programmers following TDD, they write a number of unit tests for every single story before coding, run those tests after the implementation of the story, rework until all test cases have passed, and then decide which story should be implemented next. All tests should be rerun periodically to ensure that newly developed functionalities do not break tests written previously. That is, a new functionality would never be considered to be successfully implemented unless all tests, including the ones for other functionalities, are passed. Compared with Test First, traditional software development process is different. Software engineers are supposed to design the system at an early stage in the development process, and write code for functionalities one by one, using a set of test cases written after implementation to test.

Test First programming is claimed to help programmers [5,6]:

(1) get continuous feedback during the development process and identify problems with the code early, reduce time/effort used for debugging, thereby enhancing the productivity;
(2) write automatically testable code with test cases, making the software system more reliable; and
(3) prevent new defects from coming into being in debugging and the long term maintenance by continuously running automated test cases.

There exist a number of studies to investigate the distinction between Test First/XP and traditional methods [44,54,41,22,36,18,32,19,29]. However, the results obtained vary from one study referred above to another (discussed in Section 2). This forms the general research question for this research:

Are there any differences between the effectiveness of the Test First programming and the Test Last approach?

* Corresponding author.
E-mail addresses: l.huang@dcs.shef.ac.uk (L. Huang), m.holcombe@dcs.shef.ac.uk (M. Holcombe).

0950-5849/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2008.03.007

To answer the research questions, it is necessary to work out the effectiveness from the perspective of testing, coding, code quality, and productivity, of the Test First and the Test Last programmers, respectively.

Accordingly comparative experiments were conducted in the spring semesters from the academic year 2002 to 2006 in Sheffield Software Engineering Observatory (SSEO), a semi-industry context in which real projects, small sized goal systems that required approximate 1000 person* hours on average, real clients, and student programmers were involved, to expand what is known concerning the effectiveness of Test First programming before software delivery.

The remainder of this paper is organized as follows: Section 2 reviews empirical research related to the Test First. Section 3 presents the goals as well as hypotheses that are set up, and describes the experimental design. Section 4 provides the data analysis and statistical results. Sections 5 presents the threats to validity, followed by Section 6 that provides a discussion for all empirical results in this research. Finally, Section 7 summarizes the findings and future work.

2. Related work

2.1. Previous studies

In this section, we selected only those that focused on the effectiveness of Test First programming in terms of testing, code quality, or productivity for review due to the large volume of literatures pertained to the Test First.

A comparative study [44] with 19 undergraduate students involved in an academic setting showed that Test First method did not help programmers obtain higher productivity or programs of higher reliability but subjects using Test First understood the programs better. A similar result was obtained by Pancur et al. [36] who carried out an experiment with 38 senior undergraduates, showing that Test First group obtained neither higher external quality nor better code coverage. However, Edwards [18] got some different results, which revealed that defect rate decreased and higher code coverage was obtained after TDD's adoption. Erdogmus et al. [19] also obtained similar results in a formal investigation with 24 students who worked independently. It was found that the Test First students wrote more tests, and the minimum external quality increased with the number of tests. Moreover, students who wrote more tests were more productive regardless of testing strategy employed. While Test First programmers in those studies mentioned above [44,36,18,19] did not obtained higher productivity, A result regarding the improvement of productivity was obtained in the experiment of Kaufmann et al. [32]. The amount of code produced by Test First team was 50% more. As Test First team and Test Last team were given the same amount of time and the code produced by two groups was of the same complexity level [42], the result indicated that the Test First team was significantly more productive. And Janzen et al. [29] found that the Test First teams delivered twice as many features as Test Last teams did. Additionally, Test First teams wrote more tests and obtained higher code coverage.

In a case study based on the industrial setting at IBM, the pair programmers using Test First obtained a 40% to 50% improvement of code quality, whereas their productivity did not decrease correspondingly [54,41]. Another structured experiment run by George et al. [22], delivered an alternative view again. In this study, 24 professional pair programmers were divided into two groups, the TDD group of which was less productive but produced code of higher quality because 18% more black-box test cases were passed whereas 16 percent more time were consumed in TDD group. Crispin [13] reported the experience in several companies saying that using TDD has positive effect on the quality of code – the defect rate of delivered software in her company went down 62% after one year and a half since TDD was adopted, and it decreased by another 39% one year later, although programming following TDD was difficult and had no obvious positive effect on the software quality at early stage of its adoption.

2.2. Research motivation

In the studies reviewed above, the effectiveness of Test First programming was observed from the perspective of testing, code quality, productivity, and the correlations between them. And the statistical results that were reported did reveal that the effectiveness of Test First programming was different from that of Test Last. However, the results varied from one experiment to another: In some study Test First programmers obtained higher productivity [29] while in others they did not [44,54,41,22,19]; in some studies Test First programmers delivered software applications of higher quality [54,41,18] whereas in some others they failed to [44,36,19]. Moreover, all the observations reviewed above were based on the development of small sized goal systems with fixed requirements. The reviewed studies were based on small software applications developed by solo programmer or one pair of programmers but this is not always the case in the industry. Besides, in each of the controlled experiments above, there was only one goal system, which possibly resulted in bias [12] because the technical features in one small goal system were quite limited – these limitations had a negative effect on the external validity of those empirical studies and therefore reduced the generalizability of results.

Since the empirical results regarding programmers' productivity and software's external quality varied from one study to another as presented in the last paragraph, this research focuses essentially on the same dependent variables (productivity and external quality) as the reviewed studies [27]. Besides, because one of our colleagues [38] found that Test First programmers and Test Last programmers spent different proportions of time on coding and testing, and the results of some empirical study supported that the more emphasis made on testing, the higher the productivity is [18], we also focus on the effort spent on testing and coding.

However, we apply a different experimental design (presented in Section 3) from other researchers did due to the limitations discussed above – although some of those are quite difficult to be avoided or even inevitable, it is possible to make some changes to the experiment design, to get rid of, or at least alleviate the impact of some limitations, namely, the size of the goal system, the number of the goal system, and the team organization. To do this, experiments are carried out in the SSEO (Sheffield Software Engineering Observatory) environment for further investigation, as described in the following sections.

3. Experimental design

This section generally describes this study following Basili's Goal-Question-Metric template [3,4]: the goal of this research is presented in Section 3.1. The research questions (R1–R6), which are generated from the discussion in Section 2.2, are raised in the next two sections (Sections 3.2 and 3.3). And the measurements for effort spent on testing, effort spent on coding, productivity, and external quality are presented in Section 3.4. Afterwards, the experiment procedure is described in detail.

3.1. Goal

The experimental goal is to compare the effectiveness of Test First programming with that of Test Last in terms of effort spent

on testing, effort spent on coding, productivity, external quality, and possible correlations between them.

3.2. Research questions

As presented in Sections 2.2 and 3.1, the effort spent on testing, effort spent on coding, productivity, external quality, and possible correlations between them are focused in this research. The research questions are listed as follows.

R1. Can Test First help programmers obtain higher productivity at the team level?
R2. Do programmers using Test First spend more effort on testing when they work as a team?
R3. Which group of subjects is more likely to deliver software of higher quality, as the requirements are changing?
From R1, R2, and R3, we may find that there possibly exist some relationships between the testing effort and some other aspects in the software development process, namely, the coding effort, productivity and software external quality. Thus the fourth, the fifth, and the sixth research question are
R4. Are there any correlations between the testing effort and productivity?
R5. Are there any correlations between the testing effort and software external quality?
R6. Are there any correlations between the testing effort and coding effort?

3.3. Hypotheses and formalized hypotheses

With the research questions outlined in Section 3.2, the hypotheses for R1 to R6 are established as follows.

For **R1**,
Null hypothesis ($\mathbf{N}_1$): Test First teams are of same productivity with Test Last teams overall.
Alternative hypothesis ($\mathbf{A}_1$): Test First teams are more productive overall.
Formalized hypotheses: $\mathbf{N}_1$: $Productivity_{TF} = Productivity_{TL}$
$\mathbf{A}_1$: $Productivity_{TF} > Productivity_{TL}$
For **R2**,
Null hypothesis ($\mathbf{N}_2$): Test First teams spend same amount of effort on testing than Test Last teams do.
Alternative hypothesis ($\mathbf{A}_2$): Test First teams make more effort on testing than Test Last teams do.
Formalized hypotheses: $\mathbf{N}_2$: $TestEffort_{TF} = TestEffort_{TL}$
$\mathbf{A}_2$: $TestEffort_{TF} > TestEffort_{TL}$
For **R3**,
Null hypothesis ($\mathbf{N}_3$): The external quality of software delivered by Test First teams and Test Last teams is of the same level.
Alternative hypothesis ($\mathbf{A}_3$): The quality of software delivered by Test First teams is higher than that of software delivered by Test Last teams.
Formalized hypotheses: $\mathbf{N}_3$: $Quality_{TF} = Quality_{TL}$ $\mathbf{A}_3$: $Quality_{TF} > Quality_{TL}$
For **R4**,
Null hypothesis: The testing effort has nothing to do with the software external quality.
Alternative hypothesis ($\mathbf{N}_4$): The increase of testing effort leads to the improvement of software external quality.
Formalized hypotheses ($\mathbf{A}_4$): $\mathbf{N}_4$: $Quality = c + d * TestEffort$, $d = 0$
$\mathbf{A}_4$: $Quality = c + d * TestEffort$, $d \neq 0$
For **R5**,
Null hypothesis ($\mathbf{N}_5$): The testing effort has nothing to do with the productivity.
Alternative hypothesis ($\mathbf{A}_5$): The increase of testing effort leads to productivity improvement.
Formalized hypotheses: $\mathbf{N}_5$: $Productivity = f + g * TestEffort$, $g = 0$
$\mathbf{A}_5$: $Productivity = f + g * TestEffort$, $g \neq 0$
For **R6**,
Null hypothesis ($\mathbf{N}_6$): The testing effort has nothing to do with the coding effort.
Alternative hypothesis ($\mathbf{A}_6$): The increase of testing effort reduces the effort on coding.
Formalized hypotheses: $\mathbf{N}_6$: $CodeEffort = h + i * TestEffort$, $i = 0$
$\mathbf{A}_6$: $CodeEffort = h + i * TestEffort$, $i \neq 0$

Table 1
Formalized hypotheses

Research question	Null hypothesis – N	Alternative hypothesis – A
R1	$Productivity_{TF} = Productivity_{TL}$	$Productivity_{TF} > Productivity_{TL}$
R2	$TestEffort_{TF} = TestEffort_{TL}$	$TestEffort_{TF} > TestEffort_{TL}$
R3	$Quality_{TF} = Quality_{TL}$	$Quality_{TF} > Quality_{TL}$
R4	$Quality = c + d * TestEffort$, $d = 0$	$Quality = c + d * TestEffort$, $d \neq 0$
R5	$Productivity = f + g * TestEffort$, $g = 0$	$Productivity = f + g * TestEffort$, $g \neq 0$
R6	$CodeEffort = h + i * TestEffort$, $i = 0$	$CodeEffort = h + i * TestEffort$, $i \neq 0$

The formalized hypotheses are presented in Table 1 above.

3.4. Subjects

Students in the Software Hut [24,25] module were used as subjects in this research.

The Software Hut module consists of the level 2 undergraduate students from Computer Science and Software Engineering degrees, and the level 3 (3rd year is the final year) undergraduate students from Mathematics and Computer Science degree studying in Department of Computer Science at the University of Sheffield, UK. The students are required to complete all the courses in level 1 and those in the first semester of level 2 before registering for the Software Hut module. The ones that are related to the Software Hut projects are "Java Programming", "Requirements Engineering", "Object Oriented Programming", "System Design and Testing", "Functional Programming", "Systems Analysis and Design", and "Database Technology". In these modules, they have received the training of coding using traditional method with different programming languages, testing at different levels with software tools such as JUnit [30], developing software applications at the teams level (usually consisted of 4–6 members), and they were also introduced the concept and basic knowledge of agile methods.

Software Hut module ran over 12 weeks. Students were expected to spend 120 h of effort per person on the module. In the first 5 sessions (approximately 10 h) all the students are given advanced training of programming using Test First programming and other 11 practices of XP. They were expected to work as a team and data was collected at the team level. There were 39 subjects/10 teams involved in the experiment in 2006. Moreover, data was also extracted from the 2002, 2003, and 2004 archive (There were 60 subjects/12 teams in 2004, 80 subjects/17 teams in 2003, and 96 subjects/20 teams in 2002) when we investigated the correlation between *TestEffort* and *CodingEffort* to increase the number of data points (see Section 4.5).

In this research, software hut students are allocated to two groups: the Test First and Test Last group, for comparing the effectiveness of Test First programming with that of Test Last.

3.5. Group formation and team allocation

In this research, software hut students were allocated to two groups: the Test First and Test Last group. The Test First group was expected to program using Test First programming and other 11 practices of XP, whereas the Test Last group was expected to program using XP practices except Test First.

One session of the module was given to 3–4 external business clients to present the software systems they wanted in front of all students. That is, each project had an external business client. After that, to help the subjects to form into teams consisted of 3–6 members and two groups (Test First group and Test Last group), we gave them an opportunity of telling us their preferred team members, projects, and testing strategies, to ensure that they were happy with the allocation and thus had enough motivation – we do not want any of them to drop out during the experiment. For the group balancing, not all teams were assigned to do the project or program using the approach of their first preference. Those who had no preference were assigned randomly. Since part of the students had no preference, the subjects' academic records obtained in the previous related courses were used for team/group balancing in the process of team/group formation – they were expected to work as a team and data was collected at the team level. The subjects' levels of expertise of Test First group and Test Last group were proved statistically the same by applying a *T*-test, the result of which was insignificant.

Three or four teams worked for one client. And we introduced a competition mechanism to ensure every single team worked separately from other ones. The requirements kept changing naturally in the first five weeks – some new features were raised by the clients and a number of old ones were changed, but after the fifth and sixth week most of the requirements were stable. Subjects were told that the requirements would change before the experiment but did not know which ones of them would change. For each project, a manager was appointed, to ensure that students conform to the practices as they were expected to. The students were asked to upload all the code and artifacts that were produced weekly from the sixth week to the end of the semester (the end of 12th week, Easter vocation excluded).

The group formation and team allocation in academic year 2006 is presented as an example in Table 2.

Table 2
Team allocation in 2006 as an example

Project	External clients	Team No.	Testing strategies	Manager	New/maintenance project
NHS database	K	1, 9 6, 7	Test First Test Last	S	New project
PDA system	A	4 5, 10	Test First Test Last	H	Maintenance project
Ethics system	G	2, 8 3	Test First Test Last	J	New project

3.6. Apparatus

The subjects were allowed to use the Lewin Labarotory located on the first floor of Regent Court (Of course they can also use their own computers in any place, but they must submit all work to the data repository every week). The lab is equipped with far more than 40 personal computers, all of which have an Internet connection, Eclipse IDE [17] with the JUnit [30] plugin and CVS [8] client built in. Each of the students had an account for the management tool [52] while each team has a "private" repository, which can only be accessed by their team members, managers, and researchers.

3.7. Experiment task

The goal systems are small sized ones, which took the subjects 12 weeks time to develop (Easter vocation excluded). They were NHS Database, the PDA system, and Ethics System in 2006, Fizzilink, Dating, and Debt Collection in 2004, Domestic Violence, Pharmaco, and Control Engineering in 2003, SFEDI (Small Firms Enterprise Development Initiative), Dentistry, UFI, and NCSS (National Cancer Screening Service) in 2002, as described in the Table 3 below.

3.8. Procedure

As mentioned in Section 3.4, all students have experience of developing software systems using traditional approach at both of individual and team level when they completed the prerequisite courses of Software Hut. Additionally, the concept of Test First programming had been introduced to the students and they received training of programming using agile methods before the start of the module. After they were assigned to their teams and projects, 5 advanced training sessions on Test First programming and other 11 XP practices were provided.

The projects were executed following the agile paradigm as described as follows. From the third week to the fifth week, the subjects had meetings with the external business clients for the requirements capture, and were asked to submit the requirement documents by the sixth week. The requirements kept changing in the first five weeks, but after the fifth and sixth week most of the

Table 3
The experiment tasks

Year	Project name	Description for goal systems
2006	NHS database	A tracking system for monitoring research activities in hospitals having thousands of employees to find whether the ongoing research is illegal or not
	PDA system	A database system which is used to keep track on the clients with their loan on equipments, and to record their visit and episodes
	Ethics system	An automating ethics review application system for the applicants, reviewers and administrators to process and review procedure electronically
2004	Fizzilink	A web based booking system for physiotherapy clinics
	Dating	An online dating agency for men and women
	Debt collection	A planning tool for the allocation of bailiffs to visit properties and agencies to carry out their court order
2003	Domestic violence	An information system for a police service which records domestic violence incidents and detail
	Pharmaco	A web based system for GPs to subscribe data or adverse reactions to pharmaceuticals or reactants
	Control engineering	CD or video game to explain the use of control engineering techniques in public activities
2002	SFEDI	A web based system to against small companies in gaining information about Government support schemes
	Dentistry	An online learning system with assessment and automated mailing
	UFI	An analysis tool for cell phone enquiries, sales and related issues for the University for Industry
	NCSS	An archive for managing publications on the treatments for cancer and related diseases

requirements were stable. All the requirement documents were approved and signed by the corresponding clients.

The appointed managers (academic and research staffs) had meetings with their teams once a week for team management, and to remind students to ensure that they conformed to the practices, if necessary, as they were supposed to. The managers assessed the methodology conformance at the team level using the conformance assessment scheme set up by the teaching staff (see Section 3.9.2). The students worked on their projects according to the plan made by themselves. And they were asked to upload all the code and artifacts as soon as they were produced from the first week to the end of the semester. They were given instructions on classifying the documents required to upload files of different categories to directories with different names (in a server) and were told that they would lose marks if they did not (so the documents were classified by the subjects). Moreover, students were required to fill in timesheets for their working time.

Data was collected from the management tool [52] as well as the data archive. One interview was given before the fifth week to assess their knowledge and motivation of doing Test First or Test Last, to ensure that they were happy with the assigned team members and job. During the development process, data was extracted from the management tool and data archive week by week. Additional interviews were also given to validate the data if any problems with the data were found. Students were informed that the interviews and data collection were conducted for research, but the goals and hypotheses of experiments were kept confidential. The detailed data collection process is presented with the experimental variables described in Section 3.9.

In the analysis phase of the experiment, the effectiveness of Test First and traditional approach was assessed from the perspective of external business clients and experimenters, using the measurements presented in Section 3.9.

3.9. Variables and measurements

3.9.1. Context variables

The main independent variables in the experiment were personal data of subjects and team/group affiliations (see Table 4), all of which do not appear in the hypotheses but are indispensable for the results interpretation and generalization [33]. The context variables are independent variables that were set before the experiment.

3.9.2. Intermediate variables

In this experiment, the process conformance of the subjects (variable *Conformance*) and *TestEffort* were dependent on the context variables and experiment design, because the maturity of subjects and the experiment settings had an impact on the process conformance and the testing-related activities [49]. At the same time, the variables closely related to the effectiveness of Test First programming, namely, *CodingEffort*, *Productivity*, and *Quality*, were dependent on *Conformance* and *TestEffort*. The process conformance also had an impact on the testing effort [49]. Therefore, two variables in the experiment, *Conformance* and *TestEffort*, were considered to be intermediate variables in this multi-dependency system, because they were both independent and dependent.

Table 4
Context variables

Name	Description
Subjects	Level 2 undergraduates major in computer science
Programming language	Java and PHP
Environment	Management tool, Eclipse, JUnit, CVS, and PHP 5.0
Course	Software engineering course (Level 2, Spring Semester)

The managers played the role of assessors of methodology conformance, where the measurement of the variable *Conformance* was partially based on the conformance assessment scheme set up by teaching staffs (see Appendix A). Throughout the development process, managers were given privilege of reading documents in the data archive, and asked to assess the development process using 11 criterions (name/weight): requirements/5, timesheets/2, action lists/2, meetings/2, stories and XXMs (eXtreme X-Machines)/7, design elegance/3, Demo and poster/4, Quality Assurance/6, Solution/2, Unit tests/10, and documents quality and content/5 [39]. The weight (importance) of those criterions was decided by the teaching staffs of software hut module.

The subjects were encouraged by a potential reward of up to 50% of the marks being directly related to the development process. That is, the final marks awarded were composed of two parts: the process conformance mark (50%) and the quality mark (50%). As stated above, in the development process assessment scheme, the weight for each of the criterions was decided by teaching staffs of software hut module – it was subject to the aim of teaching, and a number of those were not really related to the methodology conformance. So, in this research, 8 criterions of the 11 were selected. They are (1) timesheets, (2) action lists, (3) meetings, (4) stories and XXMs, (5) quality assurance, (6) solution, (7) unit tests, and (8) documents quality and content, which were set to be of the same weight in the assessment scheme of the extent of methodology conformance. According to the raw data, there were not significant differences (by applying *T*-test) between managers' process marks and the assessment marks of the variable *Conformance*.

The managers had meetings with the subjects at least once in each of the 12 weeks. The aim of the management meetings was to (1) remind the subjects about doing what they were expected to, and (2) make records for the subjects' responses to the managers' record sheet (see Appendix A) as the validation process of timesheets. The subjects were told that they were assessed in terms of process conformance, NOT the progress of software development. So in very few cases the subjects were found not to tell the truth in the management meeting. Data from teams with low conformance level (below 25 out of 50) was considered to be invalid.

The variable *TestEffort* was defined as the amount of effort spent on testing. Theoretically, the amount of effort can be measured by the time spent on testing, the number of tests, and the amount of test code. However, in this research, it was not applicable to use the data indicating the number of tests or the amount of test code as measurements, because different teams worked on different projects. Thus we use the percentage of time spent on testing as the measurement for *TestEffort* [16]. The numbers of hours were collected from timesheets in the management tool. The timesheets were not considered to be valid unless subjects filled them out regularly.

3.9.3. Dependent variables focused in this research

The main dependent variables related to the effectiveness of Test First in this study are *Productivity*, *Quality*, and *CodingEffort*, the ones mentioned in Section 3.3.

The variable *Productivity* is defined as output per hour. The amount of "output" here can be measured by lines of code (LOC) [20]. In this case, the *Productivity* was measured by LOC per person per hour. The LOC was obtained by a parser, which can automatically remove comments and empty lines from the source code of final delivery.

The variable *CodingEffort* was defined as the amount of effort spent on coding. The percentage of time spent on coding was used as the measurement for this variable. The numbers of hours were collected from timesheets in the management tool. The timesheets were not considered to be valid unless (1) subjects filled them out regularly in the management tool (they were supposed to fill out the timesheets every time they logged out), and/or (2) there did

not exist any inconsistencies between the time sheets, the managers' record (see Appendix A), and the files uploaded to the repository. For example, the timesheet would considered to be invalid and an interview would be further given if the timesheet said little time was spent on coding however a number of features were implemented according to managers' record or files in the repository.

Since software quality is a subjective issue [14,15], defect rate is not the only measurement for external quality. In this research, the measure of *Quality* was based on external clients' assessment, due to the heavy work load of doing acceptance testing for 10 implementations for 3 different goal systems. The external clients used the delivered software for one month's time and were asked to assess the quality of delivered software using 10 criteria, which haven been approved by our business clients and this assessment scheme has been used for over 15 years [39,53]: (1) the presentation (to provide a brief outline for the delivered software), (2) user manual, (3) installation guide, (4) ease of use, (5) error handling, (6) understandability, (7) base functionality, (8) innovation, (9) robustness, and (10) happiness (overall satisfactory). Each of the ten criterions is of the same weight [39].

Similar to the assessment scheme of *Conformance*, there were some items that were not closely related to the code quality in the assessment scheme of software external quality, for instance, the presentation (a brief outline for the delivered software), user manual, installation guide, and innovation. However, those were items showing the clients' satisfactory levels, which are contributed by a number of factors, including technical and non-technical. Thus all the items on the clients' assessment scheme were taken into account in the measurement of variable *Quality* in this research. With the clients' assessment forms as the feedback, it is possible to select some specific sets of items from the existing scheme, and examine the software quality in different ways, as we did in the measurement of variable *Conformance*.

All the intermediate and dependent variables for the quantitative research and the corresponding measurements are listed in Table 5.

4. Experiment results

In this section, the effectiveness of Test First programming is compared with that of Test Last from the perspective of productivity, testing effort, external quality, and coding effort, where 2 prediction models (M1 and M2) were built, and the methodology conformance is used in the data validation.

Since the testing strategies applied were not the only differences between two groups of subjects in software hut 2002, 2003, and 2004, only data collected from software hut 2006 was used in the data analysis for the Test First and Test Last effectiveness comparison (for research question 1–5 raised in Section 3.2). For research question 6, as the result of Pearson Correlation for Test First teams is very close to the significance value and Test First teams in those 4 years were of no difference, data for Test First teams of software hut 2002, 2003, 2004, and 2006, which was exacted from the data archive, was also put into statistics.

Table 5
Intermediate and dependent variables with their measurements for the quantitative research

Dependent variables	Measurements
Productivity	LOC per person per hour
TestEffort	Time spent on testing as a percentage
Quality	10 criteria approved by business clients (in Section 5.3.2)
CodingEffort	Time spent on coding as a percentage
Conformance	8 criterions selected form approved by teaching staffs

4.1. Productivity

Where the variable *Productivity* was measured by LOC/(person*hour), the effect was in the predicted direction – the median of TF teams' productivity was higher than that of TL teams (see Fig. 1). According to the results of the Kolmogorov–Smirnov Test [34,48], the data points of *Productivity* were normally distributed. Thus Students' one-tailed *T*-test [23] and Mann–Whitney *U*-test [40] were applied to test whether the difference between two groups was significant or not. In our experiment, we collected data at the team level – 5 Test First teams and 5 Test Last teams. The difference was considered to be significant if the result of *T*-test/Mann–Whitney test equalled to or was smaller than 0.1 due to the small number of data points [40]. Thus, the power of this test was 0.57, which was poor [11]. However, we had to live with this drawback, as it is quite difficult and expensive to acquire more teams for experiments of larger scale [3].

The result of *T*-test ($P = 0.128$) and the result of the Mann–Whitney *U*-test ($P = 0.289$) revealed that the difference was not remarkable enough (see Table 6). Therefore, we can **accept the null hypothesis $\mathbf{N}_1$**.

The results for productivity obtained by Test First teams and Test Last teams are presented below in a boxplot graph [20]. The boxes contain 50% of the data points, the lower/upper border of the box stand for the 25/75 percentile. Solid horizontal line in the middle of a box shows the median. The position of the lower "arm" is computed by the formula: 25 percentile – 1.5 * (75 percentile – 25 percentile), the position of the upper "arm" is computed as 75 percentile + 1.5 * (75 percentile – 25 percentile). Possible outliers are marked by small circles.

According to the results, it is safe to claim that Test First programming positively helped programmers obtained higher productivity (As discussed in Section 3, the researchers managed to ensure different projects were comparable). However in this research, the improvement was not statistically significant. It possibly resulted from the small scale of the experiment, or human factors like subjects' motivation and personality [50,31].

Although the null hypothesis $\mathbf{N}_1$ was accepted, the result of *T*-test was close to the significance level. It was calculated that the

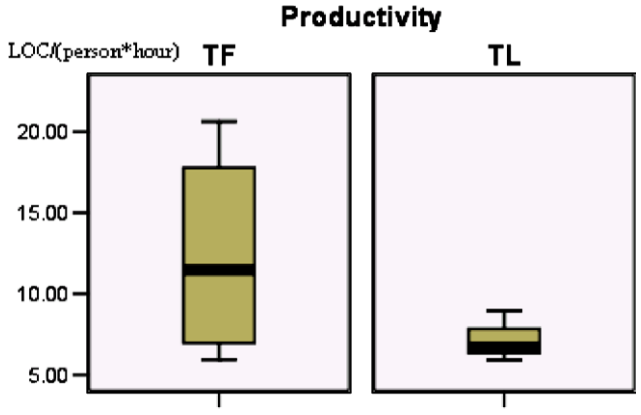

Fig. 1. Productivity, TF versus TL.

Table 6
Statistical results of R1

	Mean	Std. deviation	*T*-test significance	Mann–Whitney *U*-test, *p* value
LOC per person per hour	TF: 12.38 TL: 7.21	TF: 6.71 TL: 1.56	0.128	0.289

mean value of productivity of Test First teams was 70% higher than that of Test Last teams. It was also observed that the standard deviation of productivity of Test First teams was much greater (6.71 versus 1.56). One explanation to the results is the Test First programming did help programmers obtain higher productivity; however, the extent of productivity improvement in different teams varied, because the involved subjects were novices who did not have experience of programming following XP and Test First. In contrast, subjects had had experience of doing Test Last, and thus the productivity in different Test Last teams was close to each other.

4.2. Testing effort

The variable *TestEffort* was measured by the amount of time spent on testing as a percentage. The effect was in the predicted direction (see Fig. 2). As shown in Table 7 – the median and mean of Test First groups' data is significantly higher than that of Test Last groups', and the difference between the two groups of data were normally distributed and significant according to the result of the *T*-test ($P = 0.077$) and Mann–Whitney *U*-test ($P = 0.086$). The power of this test was 0.41.

In this case, we were able to **reject the null hypothesis N_2**. The result pertaining to the effort made on testing was reasonable. Test First programmers should had created more unit tests and run those tests more frequently, however according to our managers' feedback, Test Last teams seldom wrote tests at the early stage of software development, or even left testing to be the last thing to do before the software delivery which is also reported in [54,41], although all programmers, regardless of the testing method they used, are supposed to do an incremental development and regression testing.

A second explanation was that subjects had little experience of doing Test First, so they spent time on learning how to use the method new to them. Thus, it appeared that Test First programmer spent more time on testing – they spent more time on learning doing Test First actually.

A third explanation was that subjects doing Test First wrote tests in accordance with the requirements at the early stage of development process; however, the requirements were changing in the first five weeks. Therefore, subjects in the Test First group had to rewrite some tests correspondingly [6] and thus made more effort on testing.

Nevertheless, of those three explanations outlined above, it was not studied which one or which ones were our case in this research.

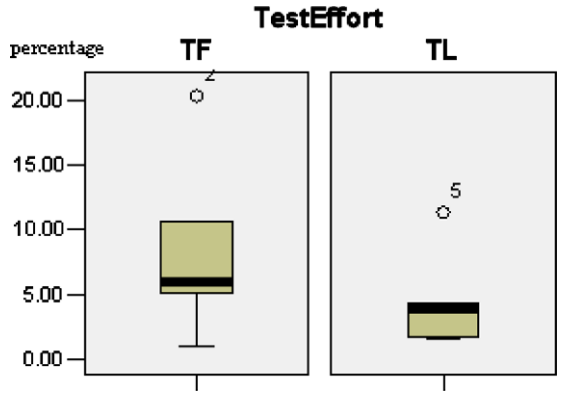

Fig. 2. Time spent on testing as a percentage, TF versus TL.

Table 7
Statistical results of R2

	Mean	Std. deviation	*T*-test significance	Mann–Whitney *U*-test, *p* value
Percentage of time for testing	TF: 8.56 TL: 4.58	TF: 7.41 TL: 3.98	0.077	0.086

4.3. External quality

Each of the delivered software was given a mark by our external clients according to the assessment scheme presented in Section 3.9. In this case, we used the clients' assessment as the measurement for software external quality.

As shown in Table 8 and Fig. 3, the result obtained was not as expected – the mean value of Test Last teams' external quality is even a little bit higher. The data points for *Quality* were normally distributed. The Students' *T*-test significance ($P = 0.375$) and the result of the Mann–Whitney *U*-test ($P = 0.341$) showed that the difference was not statistical significance. So we can safely **accept the null hypothesis N_3** because both the results of the *T*-test and the Mann–Whitney *U*-test were greater than 0.10. The power of this test was 0.18.

4.4. Correlations between testing effort, quality and productivity

Here again, the Pearson Correlation was put into use because the results of the Kolmogorov–Smirnov Test [34,48] indicated that the *TestEffort*, *Quality* and *Productivity* were normally distributed with the significance values 0.597, 0.975, and 0.588. According to the results of Pearson Correlation [10], there were no significant linear correlations between the *TestEffort*, *Quality*, and *Productivity* (see Tables 9 and 10) – the plotted data points were distributed. Therefore, we **cannot reject the null hypothesis N_4** and **N_5**.

As shown in Fig. 4, the insignificance of the expected linear correlation can be attributed to the data points plotted on the top-left of the figure. Those data points implied that some teams that spent small proportions of time on testing delivered software of good

Table 8
Statistical results of R3

	Mean	Std. deviation	*T*-test significance	Mann–Whitney *U*-test, *p* value
The clients' assessment	TF: 38.0 TL: 39.4	TF: 7.77 TL: 4.32	0.375	0.341

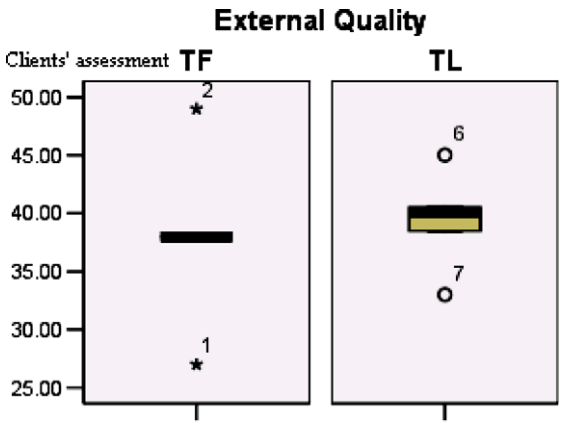

Fig. 3. External quality (approved by clients), TF versus TL.

Table 9
Statistical results of R4

		TestEffort	Quality
TestEffort	Pearson correlation	1	0.405
	Sig. (two-tailed)	N/A	0.245
Quality	Pearson correlation	0.405	1
	Sig. (two-tailed)	0.245	N/A

Table 10
Statistical results of R5

		TestEffort	Productivity
TestEffort	Pearson correlation	1	0.496
	Sig. (two-tailed)	N/A	0.258
Productivity	Pearson correlation	0.496	1
	Sig. (two-tailed)	0.258	N/A

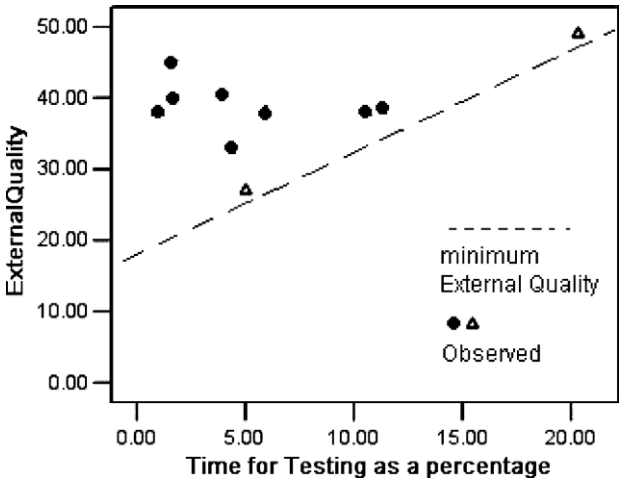

Fig. 4. Model 1, relationship between *TestEffort* and *Quality*.

quality. It is possibly due to the differences of professional skills (for example, programming skills, the ability to test and so on) between teams. However, all the plotted data points lay above the dashed line, which is set by the two triangle data points in Fig. 4, and means the minimum external quality that can be obtained when a certain proportion of time was spent on testing. It showed that the minimum achievable external quality went up with the increase of testing effort, revealing that the minimum software external quality achievable was improved as more effort was spent on testing, although there did not exist a linear correlation between *Quality* and *TestEffort*. We may find that the team plotted on the top-right of Fig. 4 spent 21% of all development time on testing and achieved very high external quality but no one can predict what if a team spends more than 21% of development time on testing. Therefore, we can only conclude that doing more testing increases the minimum quality achievable in our case and this claim is only valid within a certain range, which is not studied in this research.

4.5. The correlation between testing effort and coding effort

The time spent on this module was not restricted because of the differences that existed between different teams and projects. According to the time sheets, the time that subjects spent on the project varied considerably, ranging from approximately 500 to more than 1000 person* hours at the team level. According to the results of Pearson correlation, the linear correlation (see Fig. 5) between *TestEffort* and *CodingEffort* is not significant with the *p*-value 0.11, but there were only 5 data points involved in the statistics and the *p*-value was very close to the significance level.

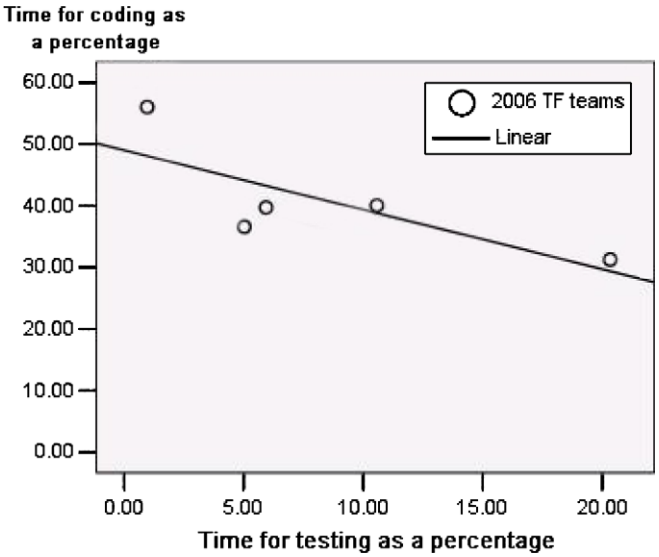

Fig. 5. Correlation between *TestEffort* and *CodingEffort* in 2006 TF teams.

So, besides software hut 2006, data for Test First teams from 2002, 2003, and 2004's software hut module were also extracted for the archive. The reason why we do not use software hut 2002, 2003, and 2004 data for R1–R5 is that the experiments in those years were not for comparing Test First and Test Last, and thus the context of Test Last teams of software hut 2002, 2003, and 2004 varied from that of software hut 2006.

With four years' data, we observed that there existed a strong linear correlation between the variable *TestEffort* and *CodingEffort* (see Fig. 6) in Test First teams, whereas the data points from Test Last teams were scattered (see Table 12). According to the results of Pearson correlation, the regression coefficient is highly statistically significant with the *p*-value approximately 0.001 (see Table 11).

The linear regression results in this model (M2):

$$CodingEffort = 47.85 - 0.778 * TestEffort$$

The model and statistical results showed that less proportions of time were spent on testing as the amount of time spent on coding as a percentage increased. Since there did not exist any linear correlations neither between *TestEffort* and *Quality* nor between *CodingEffort* and *Quality*, the data points in Fig. 6 were not clustered in terms of external quality. Thus the optimum point of "*TestEffort* versus *CodingEffort*" was not found in our experiment, and the only conclusion that we can reach is that testing reduced the amount of time spent on coding as a percentage in our case. Similar to M1, the correlation in M2 is only valid within a certain range, which is not studied in this research.

The reason why there did not exist any significant linear correlation between the *TestEffort* and *CodingEffort* in Test Last teams can be explained. Since the coverage of one test case is limited, doing more testing helps to reveal more anomalies in the code. In other words, the problems with code can be detected and sorted out before they become serious, especially when the programmers begin running tests regularly from the early stage of software development. Testing is not emphasized in the traditional methods thus in some cases Test Last programmers did not write tests until a number of features/functions are implemented, or even test the software at the last minute, as discussed in Section 4.2. In so doing,

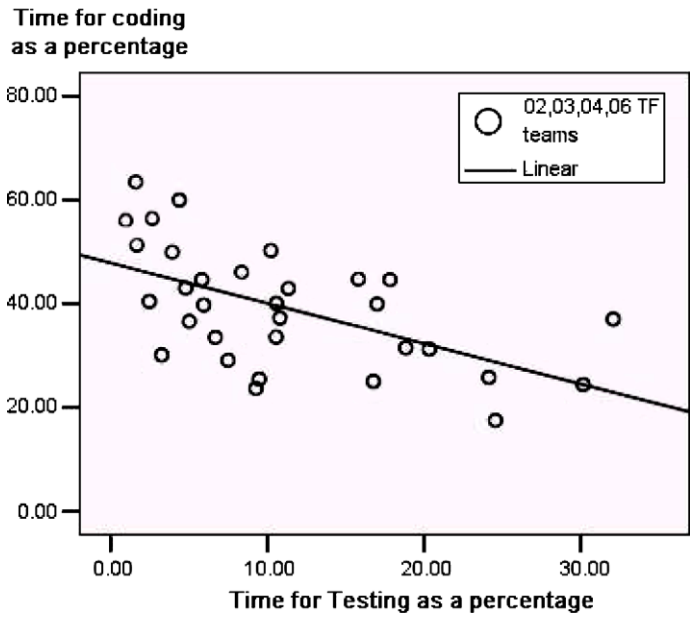

Fig. 6. Model 2, correlation between *TestEffort* and *CodingEffort* in 2002, 2003, 2004, and 2006 TF teams.

Table 11
Correlation between *TestEffort* and *CodingEffort* in 2002, 2003, 2004, and 2006 TF teams

		TestEffort	CodingEffort
TestEffort	Pearson correlation	1	−0.573
	Sig. (two-tailed)	N/A	0.001
CodingEffort	Pearson correlation	−0.573	1
	Sig. (two-tailed)	0.001	N/A

Table 12
Correlation between *TestEffort* and *CodingEffort* in 2006 TL teams

		TestEffort	CodingEffort
TestEffort	Pearson correlation	1	−0.714
	Sig. (two-tailed)	N/A	0.176
CodingEffort	Pearson correlation	−0.714	1
	Sig. (two-tailed)	0.176	N/A

anomalies in the code are more difficult to be fixed even if they are detected in the software testing.

5. Threats to validity

The threats to validity are discussed in terms of internal validity, external validity and construct validity [12], as follows.

5.1. Internal validity

The process conformance is always a problem in empirical studies. It was possible that subjects in the empirical study did not program using the method they were supposed to. As stated in Section 3.4, most of them had no experience of doing Test First before the experiment, as a result they often reverted to previously studied testing methods when they had difficulties or were working to a tight schedule. To improve their level of conformance, all the subjects were given the freedom to choose their testing strategy, although part of them had no preference. They were asked to conform to the methodology required as this was assessed as 50% of the overall mark. Data from teams with conformance marks lower than 30 (out of 50) were considered to be invalid. Additionally their managers regularly reminded them to conform to the testing strategies at the management meetings, as described in Sections 3.8 and 3.9.

The managers read the files at least once a week throughout this module. In this way the conformance was assessed by checking the files uploaded and the time and date on which the files were uploaded. Admittedly it was not frequent enough. However the projects involved were small sized ones that required 12 weeks time to develop, and the programmers and clients had freedom to decide when to have meetings and when to work. So it is infeasible to carry out close observations throughout the experiment. During this process the data collected from team 9 was removed as the method conformance of team 9 was found to be the lowest. The main problem with team 9's files was that they did not upload their files on a regular base; instead the majorities were submitted only in the final week.

The subjects in one group may consider TF method or TL method to be inferior to another, resulting in the demoralization. Since the practitioners' motivation is reported to have a large impact to the productivity [7,43], in the lectures instructors always emphasized that there was no hard evidence to prove that one method was better than the other. Before the experiment, subjects were given freedom of choosing the testing method, although most of them had no preference. All of the subjects were treated equally.

Besides, subjects in one treatment group may share something, for example, the instrument and experience in their treatment with that in another. To avoid this, all the subjects were assigned to work in the same environment so they did not have any instrument to share with each other. Besides, a competition scheme was introduced in the experiment to avoid the participants from sharing everything. The subjects were told that the top three teams would be given prizes and the best team would be offered the opportunity of joining a contest in IBM.

5.2. External validity

The main threat to external validity of this experiment is that the subjects we used were students. Admittedly, it was students that were used as subjects in this research, thereby possibly resulting in the hindrance of empirical results' generalizability due to differences between the experience, expertise as well as program-

ming environment of students and that of industry professionals [47]. Given the fact that the students are an accept sample of the population of novice developers [33], 87% of experiments are conducted with student subjects [47]. It is less expensive to run experiments with student programmers and thus the same project is able to be run in a number of teams simultaneously.

Porter and Votta [37] ran an experiment, in which 18 professionals and 48 graduate students were asked to validate the software requirements specifications using the Scenario method rather than Ad Hoc method. The requirements specifications were reviewed in two phases: the meeting and individual checking, and the fault detection rates in the two phases were applied as the measurements for the subjects' performance. The experiment results showed that, the performance of students and professionals are statistically the same, although there were indeed some differences between them. Another related work was done by Höst et al. [26] in Lund University. In that experiment, software engineering students were allocated in one treatment group and professionals in another. This time they were not told to write code but to do the lead-time estimation for eight projects using ten factors identified by Wohlin and Ahlgren [55]. Statistics in the later stage revealed that the performance of two groups of subjects was statistically the same – not only the correctness of their estimation but also their concepts.

The two studies mentioned above provide some evidence to prove that students are appropriate subjects for the SE experiments in some cases. However, there are also some experiments demonstrating that the performance of students is not same with that of professionals.

Runeson [45] conducted an experiment in freshman students, graduate students and professionals in industry and measured their performance. The results showed that there existed improvement between the freshman and graduate students, and perhaps more importantly, such improvement also existed between graduate students and the professionals. However, the data were not collected in the same context: Students worked in the context of the Personal Software Process and the professionals in the industrial settings. Difference of the working environment brought some threats to the validity of this experiment. Besides, Arisholm et al. [2] also found some differences between the performance of students and professionals in an experiment, in which all subjects were offered a job of software maintenance. Results of the data analysis showed that the performance of two groups varied and the Arisholm interpreted the result as the consequence of mastery of professional skills.

In summary, whether students can be used as subjects in the SE experiment depends on the situation. Nevertheless, to conduct a comparative study like ours in industry settings is almost infeasible due to the high cost. In SSEO, projects were real ones with complex problems, and the best solutions were put into use after the experiments. Instructors of this module negotiated with external clients to ensure that projects were of the same difficulty level. Besides, in the experiment, teams in the same group followed a similar management routine and development lifecycle. In so doing, data collected from different teams are comparable. With the projects provided by external clients, students as subjects, and data archives, the experiment environment for data collection and analysis was established.

5.3. Construct validity

Since using one task in the study can only reflect the situation of a particular case, to avoid the mono-operation bias, we used three tasks of different types in the experiment in 2006, one of which was a maintenance project. That is, subjects were assigned to projects in different development stages. So the use of the maintenance project also reduced the threats. And they were not told what the hypotheses were so as to reduce the threats of hypotheses guessing.

The threats from experimenter expectations were reduced by the data collection procedure. As stated in Sections 3.8 and 3.9 – the timesheets were filled out and reported by subjects and the management tool, the LOC was calculated by software tools, the requirements capture and team management was provided by external clients and managers who were not involved in the experiment, and the researchers did not participate in the process of development process conformance and external quality assessment. In other words, the data used in data analysis was reported by people other than researchers.

Ideally, it can be concluded that the significant difference revealed in the data analysis is due to the difference of treatments, if all possible threats to validity have been controlled [51]. Unfortunately, it is almost impossible to control all the possible threats to validity in the empirical study – we make randomized allocation when we can well control other threats, we control the allocation when we cannot [46]. No experiment can be perfect.

6. Discussion

Our results (see Table 13) pertaining to software external quality in this research support that were obtained in 3 of the literatures reviewed in Section 2.1 – the Müller's [44], the Pancur's [36], and the Erdogmus' [19], whereas contradict with that of Edwards' [18], Williams' [54,41], and George's [22] work.

According to the statistics, the empirical results did not support our expectation in terms of external quality. As stated in Section 4.3, the TF teams failed to deliver software of higher external quality – actually, the mean value of external quality for TF teams was even lower. The first explanation to this result is the emphasis on

Table 13
A summary of the empirical results

Research question	Null hypothesis – N	Alternative hypothesis – A	Empirical results
R1	$Productivity_{TF} = Productivity_{TL}$	$Productivity_{TF} > Productivity_{TL}$	Null hypothesis accepted Productivity in TF team was 70% higher but not statistically significant.
R2	$TestEffort_{TF} = TestEffort_{TL}$	$TestEffort_{TF} > TestEffort_{TL}$	Null hypothesis rejected TF teams spent significant more time on testing as a percentage
R3	$Quality_{TF} = Quality_{TL}$	$Quality_{TF} > Quality_{TL}$	Null hypothesis accepted *Quality* was at the same level in two groups of subjects
R4	$Quality = c + d * TestEffort$, $d = 0$	$Quality = c + d * TestEffort$, $d \neq 0$	Null hypothesis accepted The minimum *Quality* achievable went up with the increase of *TestEffort*
R5	$Productivity = f + g * TestEffort$, $g = 0$	$Productivity = f + g * TestEffort$, $g \neq 0$	Null hypothesis accepted No linear correlation
R6	$CodeEffort = h + i * TestEffort$, $i = 0$	$CodeEffort = h + i * TestEffort$, $i \neq 0$	Null hypothesis rejected Strong linear correlation exists **in TF teams**. Testing reduced the effort spent on coding

unit testing lead to the negligence of higher levels testing. Since TF programmers are expected to write tests before code, they spend relatively more time on writing tests at the unit level, and the importance of unit testing is emphasized when TF is applied. However, writing tests of higher levels was probably to be neglected when they had a tight schedule. Thus the external quality of the final delivery was negatively affected.

A second possible explanation is that programmers in this experiment were not skillful at programming following TF. Although, the concept of agile methods were introduced to the subjects and they had got experience of working on projects at the team level as well as advanced training in software hut module (see Section 3.4), TF is not easy to learn [13]. Thus the ability to test is a possible factor that contributed to the result [28] – this could also be the case in some other experiments, for example, [44,36,19], however it has not been identified rigorously in any of them.

In this research, statistically, the TF programmers in the experiment failed to be more productive, although it was calculated that the productivity of Test First teams was 70% higher than that of Test Last teams on average – the statistical results supports that were presented in [44,54,41,19]. However, the contexts of those studies were different from ours, as discussed in Section 2.2 and described in Section 3. And the standard deviations of data sets were not given. So it is unclear whether the statistically insignificant productivity differences between two groups were due to the same factors. Besides, we find the extent of productivity improvement in different TF teams varied, which is novel.

The standard deviation of productivity of TF teams was much greater. Since the result of one-tail *T*-test was very close to the significance level, we believe that the Test First programming did help programmers obtain higher productivity, and the insignificance was due to the small experiment scale. Another possible explanation is that the standard deviation in two groups of programmers revealed the extent of productivity improvement in different teams varied. Because the involved subjects were novices who did not have experience of programming following XP and Test First, whereas subjects had had experience of doing Test Last, and thus the productivity in Test Last teams was close to each other – TF programmers benefited from doing Test First, however the benefits varied from one team to another, thereby reduced the differences between the average productivity of teams in two groups.

The effectiveness of TF was also measured in terms of testing effort. TF teams spent more effort on testing than TL teams did, which supports the result presented in [38,19,29], while contradicts with none.

Besides, we also find some result that pertained to testing effort and is not reported in literatures reviewed in Section 2.1: in TF teams there exists a strong linear correlation between the effort spent on testing and that spent on coding (the increase of testing effort leads to the decrease of coding effort), whereas this phenomenon was not observed in TL teams. However, the optimum point of "testing effort versus coding effort" in terms of software external quality was not studied. The model is only valid within a certain range. It was believed that Test First programming required programmers to write tests early and run the tests periodically, so Test Last programmers who did not write and run the tests frequently, or even left testing to be the last thing to do, are believed to fail to get rid of the anomalies in code at the early stage in the software development, resulting in the insignificance of linear correlation between testing effort and coding effort in Test Last teams.

As results presented in some of the literatures, neither the productivity nor the software external quality in this research was significantly improved after the adoption of Test First, although some Test First teams did delivered software applications of high quality and obtained relatively higher productivity. In other words, the statistical results did not well support the claims made by Test First programming advocates. However, according to Crispin's experience in industry [13], it is always the case because

(1) programming following Test First is difficult and have no obvious positive effect on the software quality at early stage of its adoption, and
(2) the empirical results in this research were obtained using novice programmers, who were not familiar with Test First prior to the experiments, as subjects.

And her experience in a number of companies did reveal that defect rates significantly decreased as programmers get familiarized of doing Test First. Therefore, researchers and software practitioners in industry should be aware of the fact that programmers, who are getting into TDD mindset are in need of coaching. And, more attention must be paid when the results are generalized in a different setting like fully industry contexts.

7. Conclusions and future work

7.1. Conclusions

In the comparative study that focused on the effectiveness of Test First programming, a group of undergraduate students were involved. The Test First programming, one of the most important components of eXtreme Programming and Test Driven Development, was evaluated from a number of aspects: productivity, coding effort, testing effort, and external quality. In the experiment, subjects were divided into 10 teams (composed of 3–6 people) and assigned to two groups working with external clients on three real small sized projects which required 12 weeks' development time using different testing strategies.

One result of the comparative study is that Test First programmers spent higher percentage of time on testing, and lower percentage of time on coding in turn. This phenomenon was not observed in Test Last teams. Since the testing effort and coding effort in percentages represented relative effort, a model (M2), which shows a strong linear correlation between testing effort and coding effort within a certain range, is build.

A second result pertained to the relationship between the testing effort and software quality. Although the minimum achievable external quality was enhanced with the increase of testing effort, the obtained model (M1) implied that there did not exist significant linear correlations between the testing effort and the software external quality. The results did not well support the expectation that software testing helps the programmers to achieve higher quality on average.

However, statistically, Test First programmers in the experiment neither delivered software of higher quality nor were more productive, although it was calculated that the productivity of Test First teams was 70% higher than that of Test Last teams on average. In the group of Test First programmers, it was also observed that the extent of productivity improvement in different teams varied. In contrast, the productivity of subjects in different Test Last teams, who had had experience of programming following traditional approach, was close to each other.

7.2. Future work

Due to the threats to validity discussed in Section 5, more attention must be paid when the results are generalized to fully industrial contexts. The possible limitations and threats to validity are discussed in detail in Section 5. With this in mind future research

should seek to strengthen and expand these findings to software development projects of different complexity with a larger sample size in various settings, and find out in which circumstances Test First programming is of the highest effectiveness.

Since advocates of Test First programming claim that the test sets produced by Test First programmers throughout the development process positively contribute to the maintenance after the software delivery, not only is it useful to observe the new projects but also helpful to focus on more maintenance projects in future research – there were only 3 teams (1 Test First team and 2 Test Last teams) working on one maintenance project in software hut 2006 thus statistically it was not meaningful to compare the impact of Test First and Test Last on maintenance projects. Moreover, the value of test sets produced by Test First programmers involved in the maintenance projects is worth of being further investigated in future research.

Appendix A. Managers record sheet

Software Hut Managers Record Sheet

Date: **Team:** **TF/L**

Members absent:

a. What has the team changed/done this week?

b. What new/changed requirements are there?

b.1. From the client?

b.2. As the result of bugs?

c. What problems have they encountered this week?

Things to check –

d. Results of testing (per team member):

1. 4.
2. 5.
3. 6.

e. Production of at least one set of minutes:

f. Time Sheets ADEQUATE and FILLED IN, with EVIDENCE: (per team member)

1. 4.
2. 5.
3. 6.

g. Stories to be implemented in the next week: (per team member) (IF A STORY TAKES MORE THAN 2 WEEKS TO IMPLEMENT IT IS TOO LONG)

1. 4.
2. 5.
3. 6.

Please record any problems with the management tool or test server here:

Appendix B. A completed clients assessment form

Software Hut Client Assessment Form

Team:

Category	Clients Comments	Marks	Out of
Demo	*System worked to plan and excellent presentation. CD copy developed. More than two sections incomplete*	*4*	5
User manual	*Very comprehensive guide to using the CD*	*4*	5
Installation Guide	*Good idea to put a brief installation guide within the CD. Brief but helpful installation guide, could have had more tips as to how to deal with problems*	*2*	5
Maintenance Doc	*Again brief, but does not address any potential problems*	*1*	5
Ease of use	*Good to use but no visible links to next section on screen. Some sections did not turn green when completed*	*2*	5
Understandability	*The manual was easy to follow. The only complicated part was trying to create a CD copy of my profile*	*4*	5
Completeness	*Everything included, though oddly no "finish" button on "self assessment" page*	*4*	5
Innovation	*The users page was good with different sections turning green on completion, though not all did information pages very thorough*	*3*	5
Robustness	*Software run OK, though had problems with copying profile to CD, though did eventually work*	*4*	5
Happiness	*Overall a very good program, good to use, though has its weaknesses*	*3*	5
Total	————————	*31*	50

Any comments you would like to make on the team in order to aid their assessment (continue overleaf if necessary):

1. *A progress bar would have been visually more helpful, including what the next section is, though the side panel did some of them.*
2. *The software and program could have been easier to use.*
3. *The details in the information sections are very good, as is the rankings and percentage score for the Holland model in interests.*

Name (Print): _____S_____ Date: _____28/5_____
Sign: _____

References

[1] S.W. Ambler, Introduction to Test Driven Development, in: Agile Database Techniques: Effective Strategies for the Agile Software Developer, John Wiley & Sons, 2003.
[2] E. Arisholm, D.I.K. Sjøberg, Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software, IEEE Transactions on Software Engineering 30 (8) (2004) 521–534.
[3] V.R. Basili, R.W. Selby, D.H. Hutchens, Experimentation in software engineering, IEEE Transactions on Software Engineering SE-12 (7) (1986) 733–743.
[4] V.R. Basili, H.D. Rombach, The TAME project: towards improvement-oriented software environments, IEEE Transactions on Software Engineering 14 (6) (1988) 758–773.
[5] K. Beck, Extreme Programming Explained: Embrace Change, Addison Wesley, Longman, Reading, Mass, USA, 2000.

[6] K. Beck, Test Driven Development: By Example, Addison Wesley, Reading, MA, 2003.
[7] B. Boehm, Software Engineering Economics, Prenctice Hall, Englewood Cliffs, NJ, 1981.
[8] P. Cederqvist, Version Management with CVS, Network Theory Ltd., Bristol, UK, 1993.
[9] D. Chaplin, Test first Programming, TechZone, 2001.
[10] J. Cohen, P. Cohen, Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences, Erlbaum, Hillsdale, NJ, 1975.
[11] J. Cohen, Statistical Power Analysis for the Behavioral Sciences, second ed., Lawrence Earlbaum and Associates, Hillsdale, NJ, 1988.
[12] T.D. Cook, D.T. Campbell, Quasi-Experimentation: Design and Analysis Issues for Field Settings, Houghton Mifflin Co., Boston, 1979.
[13] L. Crispin, Driving software quality: how test-driven development impacts software quality, IEEE Software 23 (6) (2006) 70–71.
[14] W.E. Deming, Quality, Productivity, and Competitive Position, Center for Advanced Engineering Study, MIT, 1982.
[15] R.F. Delgado, Planning for quality software, SAM Advanced Management Journal 57 (2) (1992) 24–28.
[16] E. Dustin, J. Rashka, J. Paul, Automated software testing: Introduction, management, and performance, Addison Wesley, 1999.
[17] "Eclipse Platform Technical Overview", Object Technology Int'l, 2003.
[18] S.H. Edwards, Using test-driven development in the classroom: providing students with automatic, concrete feedback on performance, in: Proceedings of the International Conference on Education and Information Systems: Technologies and Applications (EISTA'03), August 2003.
[19] H. Erdogmus, M. Morisio, M. Torchiano, On the effectiveness of the Test-First approach to programming, IEEE Transactions on Software Engineering 31 (3) (2005).
[20] N. Fenton, S. Pfleeger, Software Metrics: A Rigorous and Practical Approach, second ed., International Thomson Computer Press, London, 1997.
[21] D. Gelperin, W. Hetzel, Software quality engineering, in: Proceedings of 4th International Conference on Software Testing, Washington DC, June 1987.
[22] B. George, L. Williams, A structured experiment of test-driven development, Information and Software Technology 46 (2004) 337–342.
[23] W.S. Gossett, The probable error of a mean, Biometrika 6 (1) (1908) 1–25.
[24] M. Holcombe, H. Parker, Keeping our clients happy: myths and management issues in 'client-led' student software projects, Computer Science Education 9 (3) (1999) 230–241.
[25] M. Holcombe, M. Gheorghe, F. Macias, Teaching XP for real: Some initial observations and plans, in: Proceedings of 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), Italy, May 2001, pp. 14–17.
[26] M. Höst, B. Regnell, C. Wohlin, Using students as subjects – a comparative study of students and professionals in lead-time impact assessment, Empirical Software Engineering 5 (2000) 201–214.
[27] L. Huang, M. Holcombe, Empirical assessment of Test-First approach, Proceedings of Testing: Academic & Industrial Conference, IEEE Computer Society, 2006. pp. 197–200.
[28] L. Huang, C. Thomson, M. Holcombe, How good are you testers? An assessment of testing ability, Proceedings of Testing: Academic & Industrial Conference, IEEE Computer Society, 2007. pp.82–86.
[29] D. Janzen, Software architecture improvement through test-driven development, in: Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications, ACM, 2005, pp. 222–223.
[30] JUnit.org, Available from: <www.junit.org>.
[31] J. Karn, Empirical Software Engineering: Developing Behavior and Pretence, Ph.D. thesis, the University of Sheffield, April 2006.
[32] R. Kaufmann, D. Janzen, Implications of test-driven development: a pilot study, in: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM, 2003, pp. 298–299.
[33] B.A. Kitchenham, S.L. Pfleeger, L.M. Pickard, P.W. Jones, D.C. Hoaglin, K. El-Emam, J. Rosenberg, Preliminary guidelines for empirical research in software engineering, IEEE Transactions on Software Engineering 28 (8) (2002) 721–734.
[34] A.N. Kolmogorov, Sulla determinazione empirica di una leggi di distribuzione, Giornale dell'Istituta Italiano degli Attuari 4 (1933) 33.
[35] C. Larman, V. Basili, A history of iterative and incremental development, IEEE Computer 36 (2003) 47–56.
[36] M. Pancur, M. Ciglaric, M. Trampus, T. Vidmar, Towards empirical evaluation of test-driven development in a university environment, in: Proceedings of EUROCON 2003, Computer as a Tool, The IEEE Region, 8(2) (2003), pp. 83–86.
[37] A. Porter, L. Votta, Comparing detection methods for software requirements inspection: a replication using professional subjects, Empirical Software Engineering 3 (1998) 355–380.
[38] F. Macias, M. Holcombe, M. Gheorghe, Design-led & design-less: one experiment and two approaches, in: Proceedings of 4th International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2003), 2003, pp. 394–401.
[39] F. Macias, Empirical Assessment of Extreme Programming, Ph.D. thesis, University of Sheffield, UK, 2004.
[40] H.B. Mann, D.R. Whitney, On a test of whether one of two random variables is stochastically larger than the other, Annals of Mathematics and Statistics (1947).
[41] E.M. Maximilien, L. Williams, Assessing test-driven development at IBM, in: Proceedings of International Conference of Software Engineering, Portland, 2003.
[42] T.J. McCabe, A complexity measure, IEEE Transactions on Software Engineering 2 (1976) 308–320.
[43] S. McConnell, Rapid Development, Microsoft Press, Redmond, WA, 1996.
[44] M. Müller, O. Hanger, Experiment about Test-First programming, IEEE Proceedings on Software 149 (5) (2002) 537–544. October.
[45] P. Runeson, Using students as experiment subjects – an analysis on graduate and freshmen student data, in: Proceedings of 7th International Conference on Empirical Assessment and Evaluation in Software Engineering (EASE 03), 2003.
[46] W.R. Shadish, T.D. Cook, D.T. Campbell, Experimental and Quasi-Experimental Designs for General Causal Inference, Houghton Mifflin, Boston, 2002.
[47] D.I.K. Sjøeberg, J.E. Hannay, O. Hansen, V.B. Kampenes, A. Karahasanovic, N.K. Liborg, A.C. Rekdal, A survey of controlled experiments in software engineering, IEEE Transactions on Software Engineering 31 (9) (2005) 733–753.
[48] N.V. Smirnov, Approximation of distribution laws of random variables on the basis of empirical data, Uspekhi Matematicheskikhi Nauka 10 (1944) 179–206.
[49] M. Stephens, D. Rosenberg, Extreme Programming Refactored: The Case Against XP, 2003.
[50] S. Syed-Abdullah, Empirical Study on Extreme Programming, Ph.D. thesis, The University of Sheffield, UK, April 2005.
[51] R. Tate, Experimental design, in: H.E. Mitzel, J.H. Hardin Best, W. Rabinowitz (Eds.), Encyclopaedia of Education Research, Collier Mac Millan, London, 1982, pp. 553–561.
[52] C. Thomson, Defining and Describing Change Events in Software Development Projects, Ph.D. Thesis, the University of Sheffield, UK, 2007.
[53] C. Thomson, M. Holcombe, 20 Years of teaching and 7 years of research: research when you teach, in: Proceedings of 3rd South East European Formal Methods Workshop, 2007, pp. 141–153.
[54] L. Williams, E. Maximilien, M. Vouk, Test driven development as a defect-reduction practice, in: Proceedings of the 14th International Symposium on Software Reliability Engineering, 2004, pp. 34–45.
[55] C. Wohlin, M. Ahlgren, Soft factors and their impact on time to market, Software Quality Journal 4 (1995) 189–205.