Rapid and Proactive Approach on Exploration of Bugs in Web based Software Development

A.Ramachandran[1]
Assistant Professor
Dept.of . CSE
University College of Engineering panruti
Tamil Nadu,India

T.C.Sankar[2]
Final year M.E CSE
University College of Engineering Tiruchirapplli
Tamil Nadu, India

S.Ramachandran[3]
M.E CSE
University College of Engineering Tiruchirapplli
Tamil Nadu, India

ABSTRACT

One of the most difficult and important software quality assurance/testing tasks is to estimate the expected number of bugs in a given software module or a project. Good estimation methods are important for evaluating the project perform once in an optimized manner and deciding which quality assurance strategies are most appropriate and effective. The core objective of this paper is to define a combined testing strategy to identify the more defects and to validate the testing among code inspection and test driven development (TDD) using open source testing tools. Based on the test rules, evaluating the functionalities and the testing will be done automatically upon user acceptance. After evaluating the test cases, the results will be populated for further development. Though both the testing seem to give effective result in maximum level of testing when conducted individually, the best output by testing can be achieved through the combination of both, TDD and code sniffer testing. Our approach plays a major role in detecting and managing fault present in the software development. Implementation of this methodology proves to be cost effective and saves analyzing time. As a result this shows the improvement in the quality of the product of the software test finally satisfying the customers**.**

General Term

Software testing and bugs optimization using code inspection and Test Driven development (TDD), Code sniffer, Code Review, Code reading, Unit Test , Integration Test .

Keywords

Software Engineering, Software Assessment, Testing Ontology, Code walkthrough, Software Testing, Test Driven Development (TDD), automated Code sniffer, Refractor.

1. INTRODUCTION

With the rapid growth of more complex systems, the chance of introduction of bugs, faults and failures increases in many stages of software development life-cycle [2].

Testing is a scheduled process carried out by the software development team to capture all the possible errors, missing operations and also for a complete verification to verify objectives and satisfy user requirement. The design of tests for software and other engineering products can be as challenging as the initial design to the product itself. Test Driven Development (TDD) is a software development process to perform a task to find in which unit test cases are incrementally written prior to code implementation. In general Extreme Programming developer practice Test Driven Development (TDD) [3]. They initiate developing code by writing a failing executable unit test that demonstrates the existing code base that does not currently possess some ability. Once they have a fault in unit test, they then write the production code to make the test pass. When the test is passing, they clean up the code, refactoring out duplication, making the source code more readable, and improving the design. Although results have been mixed, some research has shown that TDD can reduce software defects by between 18 and 50 percent [11], [12], with one study showing a reduction of up to 91 percent [13], with the added benefit of eliminating defects at an earlier stage of development than code inspection.

Numerous automated code sniffers help you easily to detect different inconsistencies. In code sniffer we can find that all inspections are grouped by their goals and sense. Every inspection has an appropriate description. The main contribution of this paper is the software fault classification scheme for the end to end software system. Our classification scheme allows us to categorize distinctly each fault according to the specified criteria. It gives a detailed view of the risks involved in the software. Based on these bug priorities, an appropriate mitigation process can be implemented to ensure a quality of the system. The results show that our approach could effectively optimize the time and cost involved when compared to the existing systems. The exposure inherent in the computing system should be addressed so they can be eliminated before exploited in production system. This ontological software assessment process is implemented against a real world system and it successfully identifies major loop hole present in the system.

Prior research has clearly defined the key factors involved with successfully implementing code inspection, such as optimal software review rates [10-13] and inspector training requirements [11], whereas TDD is not as clearly defined due to its lack of maturity.

TDD is a software engineering development strategy that requires that computerized tests be written prior to writing functional test code in small, quick iterations [3][5].

1. Writing a (extremely) small number of computerized unit test case(s)

2. Running the innovative unit test case(s) to ensure they do not pass

3. Implementing code which should allow the innovative test cases to pass

4. Re-running the unit test cases to ensure software system passing with the new code

5. Refractor the code to be simple and fit with the overall design.

6. Periodically re-running all the test cases in the source code base to ensure that the new source code does not break any previously-running test cases.

Test driven development forces the developer to think about the acceptance criteria for a module, allowing the intelligence to even consider boundary cases and validations. Test driven development also allows faking inputs/outputs, letting the developer focus on the real business logic and most importantly, writing test cases beforehand reduces bugs in the system.

1.2 Code Sniffer

A constructive review of a fellow developer's code required a sign-off from another team member before a developer is permitted to check in changes or new code [4].

Mechanics of code reviews

Who: Original developer and reviewer, sometimes together in person, sometimes offline.

What: Reviewer gives suggestions for improvement on a logical and/or structural level, to conform to previously agreed upon set of quality standards.

Feedback leads to refactoring, followed by a second code review. Eventually reviewer approves code.

When: The code author has finished a coherent system change that is otherwise ready for check in the changes should be made, change shouldn't be too large or too small.

Change should be done before committing the code to the repository or incorporating it into the new build.

Code reviews are a very common industry practice.

Inspection: A more formalized code review with:

Several reviewers looking at the same piece of code, A specific checklist of kinds of flaws to look for possibly focusing on flaws that have been seen previously and focusing on high-risk areas such as security, and specific expected outcomes (e.g. report, list of defects)

Walkthrough: Informal discussion of code between author and a single reviewer

Code reading: Reviewers look at code by themselves (possibly with no actual meeting)

Automated Code sniffer is a software engineering development strategy that tokenizes source code (files) to detect violations of a defined coding standard. It is an essential development process that ensures whether the software code remains clean and consistent. It can also help prevent some common semantic errors made by developers.

With the rapid growth of more complex systems, the chance of introduction of bugs, faults and failures increases in many stages of software development life-cycle [2].

Testing is a scheduled process carried out by the software development team to capture all the possible errors, missing operations and also for a complete verification to verify objectives and satisfy user requirement. The design of tests for software and other engineering products can be as challenging as the initial design to the product itself. Test Driven Development (TDD) is a software development process to perform a task to find in which unit test cases are incrementally written prior to code implementation. In general Extreme Programming developer practice Test Driven Development (TDD) [3]. They initiate developing code by writing a failing executable unit test that demonstrates the existing code base that does not currently possess some ability. Once they have a fault in unit test, they then write the production code to make the test pass. When the test is passing, they clean up the code, refactoring out duplication, making the source code more readable, and improving the design. Although results have been mixed, some research has shown that TDD can reduce software defects by between 18 and 50 percent [11], [12], with one study showing a reduction of up to 91 percent [13], with the added benefit of eliminating defects at an earlier stage of development than code inspection.

Numerous automated code sniffers help you easily to detect different inconsistencies. In code sniffer we can find that all inspections are grouped by their goals and sense. Every inspection has an appropriate description. The main contribution of this paper is the software fault classification scheme for the end to end software system. Our classification scheme allows us to categorize distinctly each fault according to the specified criteria. It gives a detailed view of the risks involved in the software. Based on these bug priorities, an appropriate mitigation process can be implemented to ensure a quality of the system. The results show that our approach could effectively optimize the time and cost involved when compared to the existing systems. The exposure inherent in the computing system should be addressed so they can be eliminated before exploited in production system. This ontological software assessment process is implemented against a real world system and it successfully identifies major loop hole present in the system.

Prior research has clearly defined the key factors involved with successfully implementing code inspection, such as optimal software review rates [9-12] and inspector training requirements [8], whereas TDD is not as clearly defined due to its lack of maturity.

TDD is a software engineering development strategy that requires that computerized tests be written prior to writing functional test code in small, quick iterations [3][5].

7. Writing a (extremely) small number of computerized unit test case(s)

8. Running the innovative unit test case(s) to ensure they do not pass

9. Implementing code which should allow the innovative test cases to pass

10. Re-running the unit test cases to ensure software system passing with the new code

11. Refractor the code to be simple and fit with the overall design.

12. Periodically re-running all the test cases in the source code base to ensure that the new source code does not break any previously-running test cases.

Test driven development forces the developer to think about the acceptance criteria for a module, allowing the intelligence to even consider boundary cases and validations. Test driven development also allows faking inputs/outputs, letting the developer focus on the real business logic and most importantly, writing test cases beforehand reduces bugs in the system.

1.2 Code Sniffer

A constructive review of a fellow developer's code required a sign-off from another team member before a developer is permitted to check in changes or new code [4].

Mechanics of code reviews

Who: Original developer and reviewer, sometimes together in person, sometimes offline.

What: Reviewer gives suggestions for improvement on a logical and/or structural level, to conform to previously agreed upon set of quality standards.

Feedback leads to refactoring, followed by a second code review. Eventually reviewer approves code.

When: The code author has finished a coherent system change that is otherwise ready for check in the changes should be made, change shouldn't be too large or too small.

Change should be done before committing the code to the repository or incorporating it into the new build.

Code reviews are a very common industry practice.

Inspection: A more formalized code review with:

Several reviewers looking at the same piece of code, A specific checklist of kinds of flaws to look for possibly focusing on flaws that have been seen previously and focusing on high-risk areas such as security, and specific expected outcomes (e.g. report, list of defects)

Walkthrough: Informal discussion of code between author and a single reviewer

Code reading: Reviewers look at code by themselves (possibly with no actual meeting)

Automated Code sniffer is a software engineering development strategy that tokenizes source code (files) to detect violations of a defined coding standard. It is an essential development process that ensures whether the software code remains clean and consistent. It can also help prevent some common semantic errors made by developers.

2. BACKGROUND AND RELATED WORK

Recently, researchers have started to conduct studies on the effectiveness of the Test-driven Development Practice, Code sniffer and other testing strategies that are often adopted by the software development.

2.1 Unit Testing

We adopt white box testing when using this testing technique. This testing was carried out on individual components of the software that were designed. Each individual module was tested using this technique during the coding phase. Every component was checked to make sure that they adhere strictly to the specifications spelt out in the data flow diagram and ensure that they perform the purpose intended for them [6].

All the names of the variables are scrutinized to make sure that they are truly reflected of the element they represent. All the looping mechanisms were verified to ensure that they were as decided. Beside these, we trace through the code manually to capture syntax errors and logical errors.

2.2 Integration Testing

After finishing the unit testing process, next is the integration testing process. In this testing process we put our focus on identifying the interfaces between components and their functionality as dictated by the DFD diagram. The bottom up incremental approach was adopted during this testing. Low level modules are integrated and combined as a cluster before testing [7]. Porter et al. [8, 9] performed experiments comparing Ad Hoc Reading, Checklist-Based Reading, and Scenario-Based Reading for software requirements inspections using both student and professional inspectors.

The black box testing technique was employed here. The interfaces between the components were tested first. This allowed identifying any wrong linkage or parameters passing early in the development process as it just can be passed in a set of data and checked if the result returned is an accepted one.

2.3 Validation Testing

Software testing and validation is achieved through a series of black box tests that demonstrate conformity with requirements. A test procedure defines specific test cases that will be used to demonstrate conformity with requirements. Both, the plan and the procedure are designed to ensure that all functional requirements are achieved, documentation is correct and other requirements are met. After each validation test case has been conducted, one of the two possible conditions exists. A deviation from specification is uncovered and a deficiency list is created.

The deviation or error discovered at this stage in project can rarely be corrected prior to scheduled completion. It is necessary to negotiate with the customer to establish a method for resolving deficiencies.

2.4 System Testing

System testing is a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all the work should verify that all system elements have been properly integrated and perform allocated functions. System testing also ensures that the project works well in the environment. It traps the errors and allows convenient processing of errors without coming out of the program abruptly. Recovery testing is done in such a way that failure is forced to a software system and checked whether the recovery is proper and accurate. The performance of the system is highly effective.

Software testing is a critical element of software quality assurance and represents ultimate review of specification, design and coding. Test case design focuses on a set of technique for the creation of test cases that meet overall testing objectives. Planning and testing of a programming system involve formulating a set of test cases, which are similar to the real data that the system is intended to manipulate. Test cases consist of input specifications, a description of the system functions exercised by the input and a statement of the extended output. Though testing involves producing cases to ensure that the program responds, as expected, to both valid and invalid inputs, that the program perform to specification and that it does not corrupt other programs or data in the system.

In principle, testing of a program must be extensive. Every statement in the program should be exercised and every possible path combination through the program should be executed at least once. Thus, it is necessary to select a subset of the possible test cases and conjecture that this subset will adequately test the program.

2.5 Component level testing:

To perform testing at the components level with Process server and Performance server which are integrated with work flow engine by checking the clustered environment, staging and production environment.

Load Test Scenarios:

To customize load testing scenarios using different test cases, load levels, load distributions etc and to distribute concurrent users across remote server machines to simulate extreme loads and/or test from different locations.

Application in Clustered Environment:

To check the application in clustered environment if working against the same shared runtime data and same user logins and requests simultaneously from multiple browsers to note the amount of concurrent access to data will lead to potential data corruption.

Methodology Overview

Data-driven testing is the creation of interacting test scripts together with their related data that results in a framework used for the methodology.

Modularity-driven testing is the test script modularity framework that requires the creation of small, independent scripts that represent modules, sections, and functions of the application-under-test.

Keyword-driven testing also known as table-driven testing or action-word testing, is a software testing methodology for automated testing that separates the test creation process into two distinct stages: a Planning Stage, and an Implementation Stage.

The hybrid is the Test_Automation_Framework this is what most frameworks evolve into overtime and multiple projects. The most successful automation frameworks generally accommodate both Keyword-driven testing as well as Data-driven testing.

2.6 General Test automation Expectations

Automation frame work should handle following testing activities:

1) Perform the Load, stress testing with least or no manual intervention.

2) Should be able to perform regression testing.

3) Should have low maintenance cost.

4) Should generate comprehensive Bug analysis report.

5) Should have easy test results interpretation.

6) Should be able to adopt new changes without much effort.

3. IMPLEMENTATION

With the rapid growth of more complex systems, the chance of introduction of faults and failures increases in many stages of software development life-cycle [8].

TDD (Test Driven Development) model is common practice of most of the company which involve with the long time projects. It helps to keep the fixed behavior of functions over the development stage. The functions are implemented after implementing experiment methods for that function. After implementing the function, run the tests and if any test is fault, again modify the function. This process repeats until the all tests are passing.

Test method is a function which checks whether the specific behavior of a function is correct. Any function can have at least two test methods of which one can be a success test and one a failure test. In long time projects like open source projects, the functions are frequently modified by the developers during a long time. If the modified function is not given the previous behavior of that function, the output for other places will be going wrong. It can be cause to failures. By writing tests this problem can be reduced. Before and after modifying the test cases, by running the tests, developer can be verify the function behave correctly. As well by going through the test cases helps even understand what actually function does.

3.1 Steps in Test-Driven Development

The test-driven development process consists of the steps shown in Fig. 1

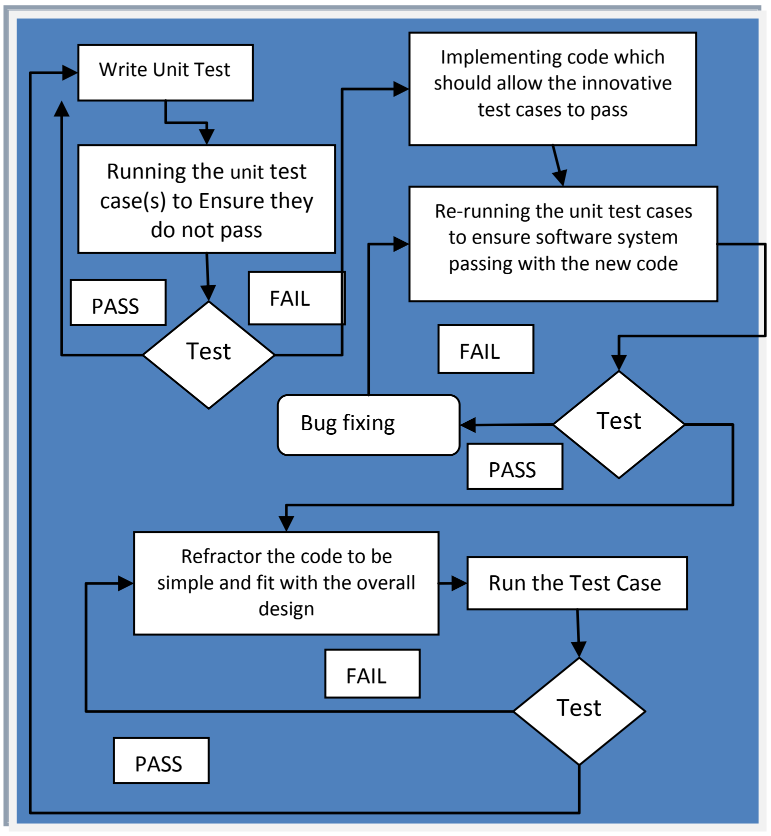

Write Unit Test
Implementing code which should allow the innovative test cases to pass
Running the unit test case(s) to Ensure they do not pass
Re-running the unit test cases to ensure software system passing with the new code
PASS
FAIL
Test
FAIL
Bug fixing
Test
PASS
Refractor the code to be simple and fit with the overall design
Run the Test Case
FAIL
Test
PASS

Fig.1. Test driven development process flow.

The steps can be summarized as follows:

1. **Create the test code.** Use a computerized test framework to generate the test code. The test code drives the development of functionality.

2. **Write/Modify the functional code.** Write the functional code for the application block so that it can pass all test cases from the test code. The first iteration involves developing new functionality, and subsequent iterations involve modifying the functionality based on the failed test cases.

3. **Create additional tests.** Develop additional tests for testing of the functional code.

4. **Test the functional code.** Test the functional code based on the test cases developed in Step 3 and Step 1. Repeat steps 2 through 4 until the code is able to pass all of the test cases.

5. **Refactor the code.** Modify the code so that there is no dead code or duplication. The code should adhere to best practices for maintainability, performance, and security.

3.2 PHP Code Sniffer is a PHP5 script that tokenizes and "sniffs" PHP, JavaScript and CSS files to detect violations of a defined coding standard. It is an essential development tool that ensures code remains clean and consistent. It can also help prevent some common semantic errors made by developers

The most common tasks that are covered by the static code analysis are:

1. Locating dead code.
2. Improving code structure and maintainability.
3. Conforming to coding guidelines and standards.
4. Conforming to specifications.

Code sniffer interface defines two methods that must be implemented; register () and process ().

The register () method allows a sniff to subscribe to one or more token types that it wants to process. Once Code Sniffer encounters one of those tokens, it calls the process () method with the Code Sniffer File object (a representation of the current file being checked) and the position in the stack where the token was found. We are interested in single line comments. The token_get_all () method that Code Sniffer uses to acquire the tokens within a file distinguishes doc comments and normal comments as two separate token types.

4. RESULT AND DISCUSSION

The purpose of this study is to compare the fault rate, identified by both the methodology TDD and Code sniffer. As per our analysis both has advantages and disadvantages. The combined use of code sniffer and TDD is more effective than either method alone.

4.1 Test Driven Development.

A PHPUNIT test is a method of evaluating the Bugs of a computer system by simulating a test case. The process involves an active analysis of the system for any weaknesses, technical flaws.

The study included the Stockpile class lets us safely withdraw any amount of money, returning an error code in case the $ amount withdrawn exceeds our availability.

Phpunit test checks every function of Stockpile class. A test suite is normal PHP class inherited from PHPUnit_TestCase containing test functions, identified by a leading 'test' in the function name.

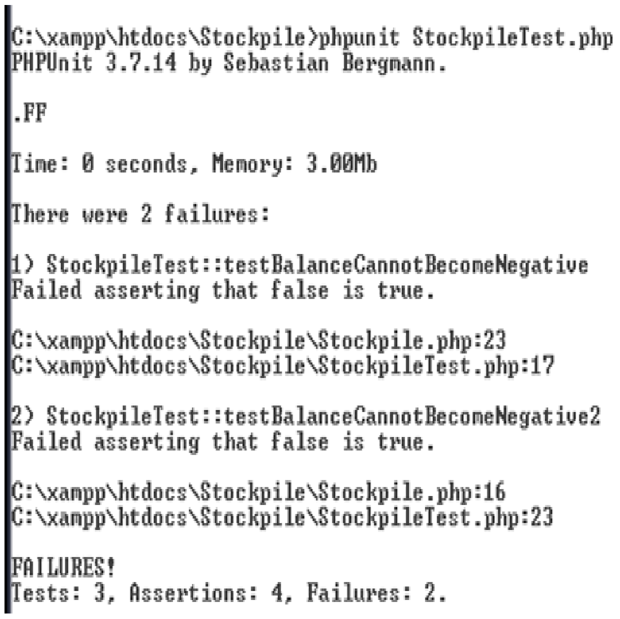
In the test function an expected value has to be compared with the result of the function to test. The result of this compare must delegate to a function of the assert*()-family, which decides if a function passes or fails the test.

The contract for the Stockpile class requires methods to get and set the Stockpile balance, as well as methods to deposit and withdraw money. It also specifies that the following two conditions must be ensured:

The Stockpile initial balance must be zero, the Stockpile balance cannot become negative.

The Stockpile class before we write the code for the class itself. We use the contract conditions as the basis for the tests and name the test methods accordingly, as shown in Fig.2.

The PHPUnit command-terminal test runner can be invoked through the PHPunit script. The following code shows how to run tests with the PHPUnit command-line test runner.



```
C:\xampp\htdocs\Stockpile>phpunit StockpileTest.php
PHPUnit 3.7.14 by Sebastian Bergmann.

.FF

Time: 0 seconds, Memory: 3.00Mb

There were 2 failures:

1) StockpileTest::testBalanceCannotBecomeNegative
Failed asserting that false is true.

C:\xampp\htdocs\Stockpile\Stockpile.php:23
C:\xampp\htdocs\Stockpile\StockpileTest.php:17

2) StockpileTest::testBalanceCannotBecomeNegative2
Failed asserting that false is true.

C:\xampp\htdocs\Stockpile\Stockpile.php:16
C:\xampp\htdocs\Stockpile\StockpileTest.php:23

FAILURES!
Tests: 3, Assertions: 4, Failures: 2.
```

Fig.2. Tests for the Stockpile class using TDD

For each experiment run, the PHPUnit terminal tool prints one character to indicate augmentation.
. => Printed when the test succeeds.

F => Printed when an assertion fails while running the test method.

E => Printed when an error occurs while running the test method.

S => Printed when the test has been skipped.

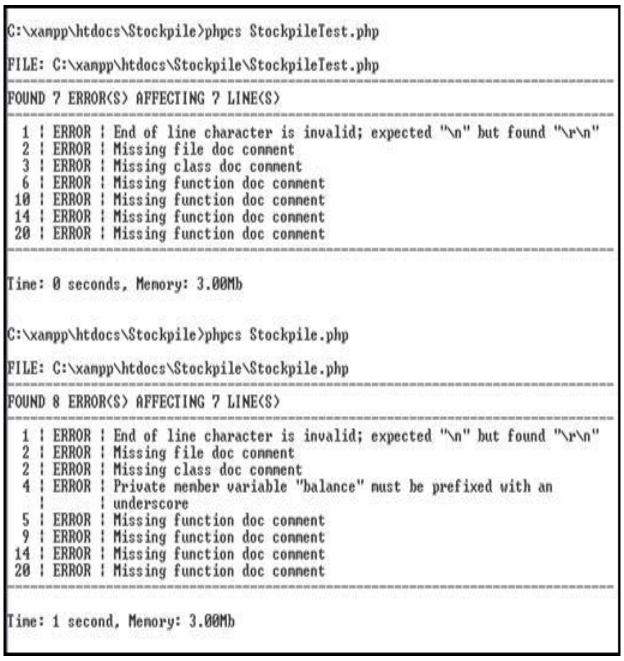
I => Printed when the test is marked as being incomplete or not yet implemented.

Our experiment result shows that there is one success and two assertions are fails and also indicate the line number of bugs. The Stockpiletest.php is the name of the file. 17 is the line number in question, and it is clear that this is an error. The brief message here is quite helpful because it says exactly what was wrong. It is explain about the interface files error, The Stockpile.php is the name of the interface file. 23 is the line number in question, and it is clear that this is an error in the interface.

In the Test-Driven approach, the code needed to make the tests pass. However this methodology is still in its initial stages of development, as it suffers from faults and bugs that prevent the users from trusting it.

4.2 Code Sniffing and Code Review

Although the primary purpose for conducting code reviews throughout the SDLC life cycle is to recognize software defects in the code, the reviews can also be used to enforce coding standards in a uniform manner. Loyalty to a coding standard can only be sufficient when followed throughout the software project from inception to completion. Built-in coding assistance, phpcs provides code style check through integration with the PHP Code and its output is a list of flaws found, with an error message and line number supplied. The study also included the same Stockpile class, as shown in Fig.3.



```
C:\xampp\htdocs\Stockpile>phpcs StockpileTest.php

FILE: C:\xampp\htdocs\Stockpile\StockpileTest.php
--------------------------------------------------------------------------------
FOUND 7 ERROR(S) AFFECTING 7 LINE(S)
--------------------------------------------------------------------------------
  1 | ERROR | End of line character is invalid; expected "\n" but found "\r\n"
  2 | ERROR | Missing file doc comment
  3 | ERROR | Missing class doc comment
  6 | ERROR | Missing function doc comment
 10 | ERROR | Missing function doc comment
 14 | ERROR | Missing function doc comment
 20 | ERROR | Missing function doc comment
--------------------------------------------------------------------------------

Time: 0 seconds, Memory: 3.00Mb

C:\xampp\htdocs\Stockpile>phpcs Stockpile.php

FILE: C:\xampp\htdocs\Stockpile\Stockpile.php
--------------------------------------------------------------------------------
FOUND 8 ERROR(S) AFFECTING 7 LINE(S)
--------------------------------------------------------------------------------
  1 | ERROR | End of line character is invalid; expected "\n" but found "\r\n"
  2 | ERROR | Missing file doc comment
  2 | ERROR | Missing class doc comment
  4 | ERROR | Private member variable "balance" must be prefixed with an
    |       | underscore
  5 | ERROR | Missing function doc comment
  9 | ERROR | Missing function doc comment
 14 | ERROR | Missing function doc comment
 20 | ERROR | Missing function doc comment
--------------------------------------------------------------------------------

Time: 1 second, Memory: 3.00Mb
```

Fig.3. Tests for the Stockpile class using Code Sniff.

4.3 Comparison of TDD and Code Sniffer testing result

Both the methods are producing the fault report including the Line no. In the above example Stockpile class TDD method explored the logical error in the line no 16, 17 and 23, shown in Fig.2 and while applying the code sniffer method to the same Stockpile class explored the fault in different line no 1, 2, 4, 5, 9, 14 and 20 shown in Fig.3. Developer can identify the line of code where the bug actually occurs; hence developer can fix the bug in the same process. By applying the collaborative approach, Both TDD and Code sniff we obtained the more effective testing result, shown in Table 1.

4.4 Combined analysis report (End to End testing using TDD and Code Sniff)

End-to-end testing is a methodology used to test whether the flow of an application is performing as designed from start to finish. Our approach will be more economical to build and match the requirements mentioned in the scope of the application.

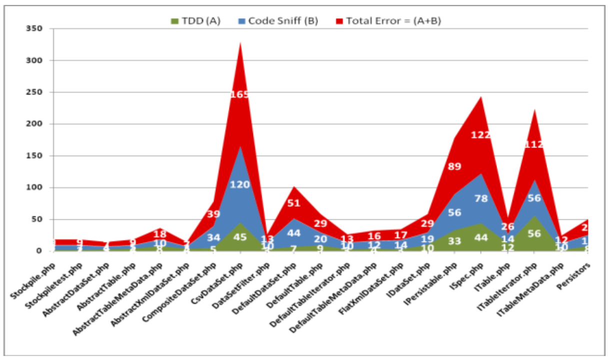


Fig.4. Combined analysis report (End to End testing using TDD and Code Sniff)

We have analyzed the software fault and bugs categories by TDD and Code Sniff and formed the combined analysis report shown in Table 2. Results show that our approach could effectively optimize the time and cost to find more number bugs in a software application shown in Fig 4.

TABLE 1 Comparison of TDD and Code sniffer testing result

Sl.	Project/File Name	LOC	TDD (A)	Code Sniff (B)	Total Error = (A+B)	Time (Sec)
1	Stockpile.php	28	2	7	9	0.01
2	Stockpiletest.php	28	2	7	9	0.01
3	AbstractDataSet.php	20	3	4	7	0.01
4	AbstractTable.php	34	4	5	9	0.01
5	AbstractTableMetaData.php	44	8	10	18	0.01
6	AbstractXmlDataSet.php	66	4	3	7	0.01
7	CompositeDataSet.php	55	5	34	39	0.01
8	CsvDataSet.php	663	45	120	165	2
9	DataSetFilter.php	63	3	10	13	0.01
10	DefaultDataSet.php	10	7	44	51	0.01
11	DefaultTable.php	333	9	20	29	1
12	DefaultTableIterator.php	66	3	10	13	0.01
13	DefaultTableMetaData.php	33	4	12	16	0.01
14	FlatXmlDataSet.php	44	3	14	17	0.01
15	IDataSet.php	41	10	19	29	0.01
16	IPersistable.php	344	33	56	89	1
17	ISpec.php	455	44	78	122	1
18	ITable.php	55	12	14	26	0.01
19	ITableIterator.php	189	56	56	112	0.01
20	ITableMetaData.php	12	2	10	12	0.01
21	Persistors	55	8	17	25	0.01

5. SUMMARY AND CONCLUSIONS

In this research paper we have discussed the characteristics of Testing and compared the software defect rate. Existing methodologies spend more time and cost to find the Software faults. Implementation of automated testing is done with two methods of software defect reduction: Test-drive development (TDD) and code sniffer. Earlier research has indicated that TDD is effective at reducing defects [1]. Yet, their report stands or seems to be incomplete without code sniffer method on detecting the software semantics error.

Accordingly, we are considering both the methods with their advantages and disadvantages to optimize a better solution. So per this analysis, we could sniff out that the combined use of code sniffer and TDD is more effective than either method alone. Hence, we could conclude that by combining the methods, we can optimize time & cost, such that the Bugs/Software fault can be eliminated before exploited in the software deployment in the Go live or production server.

6. ACKNOWLEDGMENTS

The authors are grateful to use of their software testing and code coverage tool. They also thank the study Participants and the inspectors for their time and effort, for inspect to the experiments.

7. REFERENCES

[1] Jerod W. Wilkerson, Jay F. Nunamaker Jr., and Rick Mercer, Comparing the Defect Reduction Benefits of Code Inspection and Test-Driven Development, IEEE transactions on software engineering, Vol. 38, No. 3, 2012.

[2] B. Marick, The craft of software testing, Prentice Hall.1995.

[3] P.M. Johnson, "An Instrumented Approach to Improving Software Quality through Formal Technical Review," Proc. 16th Int'l Conf. Software Eng., pp. 113-22, 1994.

[4] Efficient Object-Oriented Integration and Regression Testing Yves Le Traon, Thierry Jéron, Jean-Marc Jézéquel, and Pierre Morel.

[5] A.A. Porter and L.G. Votta, "An Experiment to Assess Different Defect Detection Methods for Software Requirements Inspections," Proc. 16th Int'l Conf. Software Eng., pp. 103-12, 1994.

[6] A.A. Porter, L.G. Votta Jr, and V.R. Basili, "Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment," IEEE Trans. Software Eng., vol. 21, no. 6, pp. 563-575, June 1995.

[7] M.E. Fagan, "Advances in Software Inspections," IEEE Trans. Software Eng., vol. 12, no. 7, pp. 744-51, July 1986.

[8] T.Gilb and D. Graham, Software Inspection. Addison-Wesley, 1993.

[9] W.S. Humphrey, A Discipline for Software Eng., ser. the SEI Series in Software Engineering. Addison-Wesley Publishing Company, 1995.

[10] W.S. Humphrey, Managing the Software Process, ser. The SEI Series in Software Engineering. Addison-Wesley Publishing Company, 1989.

[11] B. George and L. Williams, "A Structured Experiment of Test-Driven Development," Information and Software Technology, vol. 46, no. 5, pp. 337-342, 2004.

[12] E.M. Maximilien and L. Williams, "Assessing Test-Driven Development at IBM," Proc. 25th Int'l Conf. Software Eng., pp. 564-9, 2003.

[13] N. Nagappan, M. E. Maximilien, T. Bhat, and L. Williams, "Realizing Quality Improvement through Test Driven Development: Results and Experiences of Four Industrial Teams," Empirical Software Eng., vol. 13, no. 3, pp. 289-302, 2008.

8. AUTHORS' PROFILES

First Author: Ramachandran A received the B.E Computer Science and Engineering from Madurai Kamaraj University, M.E Computer Science and Engineering from Annamalai University, M.B.A from Alagappa University and currently pursuing Ph.D. in Computer Science and Engineering from Anna University Chennai. He was a lecturer in Srinivasa Institute of Engineering and Technology and Assistant Professor in JJ College of Engineering and Technology. He was senior Software Engineer in Mainframe technology in Satyam Computer Services and worked several projects for Fortune 500 companies. He is currently an Assistant Professor of Computer Science and Engineering at Anna University of Technology Tiruchirappalli, His current field of interest is elicitation techniques for software requirement specification and database and operating system vulnerabilities. He is a life member of the ISTE.

Second Author: SankarT.C received B.Sc Physics from Ramakrishna Mission Vivekananda College, Madras University, M.C.A form Madras University, M.Phil from Madurai Kamaraj University and currently pursuing M.E. in Computer Science and Engineering from University College of Engineering BIT,Trichy (Anna University Chennai). He was a Assistant Professor and Head of the Department Computer Application in Vel Tech Engineering College Chennai and Assistant Professor in SAMS College of Engineering and Technology. He is currently an Assistant Professor of Computer Science and Engineering at Apollo Engineering College, His current field of interest is bug optimization techniques for software testing and data mining.

Third Author: Ramachandran S received his B.E/ME Computer Science and Engineering from Anna University Chennai. His current interests elicitation techniques for S/W requirement specification, Database Tuning and MANET.