Verification and Validation of an Open Source Software

Part-I: Automated Static Analysis

Navneet Chamala
9402071477
nach14@students.bth.se

Konduru Krishna Chaitanya
9308126474
krko14@students.bth.se

Eada Priyanudeep
9206088776
prea14@students.bth.se

MVNA Uday
9308017434
vnma14@students.bth.se

Lavanya Pampana Vinod
9301031606
lapa14@students.bth.se

Dilip Somaraju
9307203373
diso14@students.bth.se

***Abstract*— This paper presents the findings of a detailed static and dynamic code analysis of an open source multi player online game, called Spaxe Hilk. In order to enhance the quality, it is important that the game is tested extensively. As a part of this extensive test, we have conducted an automated static analysis using the open source PMD tool. Further, a test plan is presented that focuses on level test planning.**

I. INTRODUCTION

Our document is divided into two parts, part I deal with the Static Automated Testing (SAT) and part II talks about the Dynamic Analysis. In part I we used the PMD tool for testing the given source code and evaluated the outcome of the violations provided by the tool. In part II we have written a test plan for system Spaxe Hilk.

Further, part I is divided in the following way: section II describes the inspection goals, followed by the motivation for selecting the tool to implement the static code analysis in section III. In Section IV we talk about code sampling. The warnings and outcome of the static test are presented and discussed in section V. An overview of how the review is carried out is presented in section VI. Our findings about the use of the static analysis tool are presented towards the end in section VII as conclusions.

II. INSPECTION GOALS

Three primary aspects of code: clarity, correctness and consistency were chosen as our primary goals. Additionally, we decided to find a pattern in the warnings that were encountered during code inspection. The motivation for choosing the above said inspection goals are elicited as follows:

- Clarity: The game being tested, Spaxe Hilk is an open source code. Anybody with an intention to improve the quality must fundamentally be able to understand the code constructs at every part. Hence, clarity of the code is important.
- Correctness: Whether the code is written properly, avoiding code smells is another prime issue that we dealt. Code smells have the potential to produce serious errors during code implementation. Thus, minimizing such code smells would implicitly improve code quality as well as the overall software quality.
- Consistency: When changes are made to the existing code for improving the quality, it must be ensured that the changes made are aligned in line with the code that already exists. This is termed as code consistency, and was chosen as another goal to investigate.
- Further, identifying patterns in the warnings produced would help to categorize the warnings based on severity.

In order to achieve the goals, it is important to choose a static analysis tool that fulfills most of them. The selection of the tool is described in the next section.

III. SELECTION OF THE TOOL

Static code analysis tools try to identify qualitative and quantitative information as possible by checking the source

code completely by finding bugs like bad smells, programming errors[8]. Since the objective of this assignment involved implementing an automated static analysis tool, our initial lookout had been to understand the variety of tools available. As the source code of the game under inspection is written in Java, our search results narrowed to a list of 13 most commonly used tools. This list was a result of a web search using www.google.com. Owing to the limited knowledge and time, analyzing each tool's benefits and limitations was not possible. Therefore, a brief study about the tools by referring the documentation led to the acceptance or rejection of the tools for deeper understanding. The list of tools surveyed and the rationale for choosing a select few and rejecting the others is provided in Table1. From the initial set of tools gathered, we have analyzed the traits of every available tool in order to finalize on the tool suitable for our requirements and which will help in achieving our inspection goals stated above. After careful inspection of the various features of each tool, all of us came to a conclusion that the PMD tool will help us in acknowledging the defined inspection goals [14].

From our initial study on each of the tools, we have straight away rejected the tools like Checkstyle, GrammaTech CodeSonar, JTest, SemmleCode, Soot tools. Checkstyle was rejected as it checks code clarity alone and correctness and consistency was not inspected. GrammaTechCodeSonar is rejected as it is better for inspecting C, C++ code rather than JAVA. JTest being a memory intensive tool, testing would take a lot of time and hence not viable. We decided not to use SemmleCode as it needed basic knowledge about .QL to implement. We eliminated Soot as it doesn't cover a majority of code metrics. After rejecting the above tools, we were reduced to AgileJ Structure Views, FindBugs, PMD and SonarJ. Among the final set of short listed tools, AgileJ Structure would help us in acknowledging the goals but analysis of the data would be a tough task. SonarJ gives a graphical visualization of violations but we have opted it out because none of us were efficient in analyzing the data it produced due to lack of experience. Kalistick was acceptable as it monitors code quality but needed experienced testers. LDRA Testbed provides good test coverage but was rejected as we felt it would take us more time in evaluating the errors it produced. Finally, we were left with the choice of FindBugs and PMD, and when we ran a sample part of our source code in both the tools, we noticed that PMD gave results in more detail than FindBugs. This made us prefer PMD over FindBugs.

PMD tool is an open source static code analyzer for Java programs, which comes with a default set of rules that can be used to detect common programming mistakes in Java such as empty try-catch blocks, unused variables, unnecessary objects, and null pointer dereferencing, God class, law of Demeter etc., [7].

IV. CODE SAMPLING

Taking into account the time available to perform the project, we realized that it was not possible for us to examine the entire code. Hence, we have decided to select part of the code and examine it. In order to select the part of the code, we first examined the code by using perspective based reading (PBR) technique. In PBR, the different reviewers view the document from different perspectives answering a set of questions which are formulated based on each viewpoint. Empirical studies say that PBR has beneficial qualities because it is systematic, focused, goal oriented and transferable via training [4]. So, forming three groups having two in each, we have taken up the roles of an end user, a designer and a tester in order to find answers for the questions framed in the various scenarios (see Appendix-F). PBR yields better results with experienced reviewers but this can be a threat to validity in the current context because none of us have previous experience in testing. After taking up the roles, we have framed a set of questions to be answered by examining the code and features of the game. Based on the answers we could conclude the important modules for testing.

After carefully viewing the system from all the three perspectives, since it is an open source system open to all the people, we found that it would be important to choose those modules which are important to the users. Thus, we have selected properties, multiplayer, world, weapons and states as our modules to be tested.

V. REVIEW PROCESS

The code analysis and review was carried out over a period of 23 days. After forming a group of 6, all of us downloaded the game from http://sourceforge.net/projects/spaxehilk/?source=directory. Since we knew nothing about the game, we sat down to play the game so that we can get an overview of the functioning of the game. The next step in the review was to decide on the goals of inspection. Based on the inspection goals, we have adopted an open source testing tool for checking the errors in the code. Since, there was no source code available freely on the internet; we used an open source JAVA code decompiler tool to extract the code. This could be a threat to validity as usage of a decompiler might not produce the full source code always. Hence, there might be lesser or even more violations than what we have gathered. After decompiling the code, part of the code was picked for review as testing the complete source code was not possible. Then, each reviewer had analyzed the code and the violations it produced when tested. Each of us maintained a log book to note the time taken for review. It is observed from the table that initially the time taken to analyze the errors took more time. As the review process progressed, the time taken to decide on the violations reduced. Then all of us have discussed and analyzed individual reflections on the violations and concluded the final decision on the violation. We have documented all the events that took place in the process of analyzing the code and this is presented in the form of a Gantt chart (see Appendix A).

VI. ANALYSIS OF THE OUTCOMES

While evaluating the outcome of results after performing static automated testing, we have identified several violations by using the PMD tool. Our main objective was to review the violations presented by the tool. Taking into account our experience as reviewers, most of the violations were completely new to us. After individually analyzing the errors, we have identified that though initially it was a time taking process to decide which type of error types they belong to, later on it took lesser time relatively. We could identify three types of violations. (i) True positive: Such a violation is true and can cause potential errors at some stage of the software. (ii) False positive: It is an error but wouldn't cause any harm to the software and can be ignored. (iii)Uncertain: These were violations which we could not decide their impact on the software. It is because of lack of experience and understanding about the particular error or based on the context. Among the violations we primarily noticed violations related to the coding style having an impact on the code clarity of the software process such as wrong variable naming conventions. However, among those indicating incorrect naming conventions some of them turned out to be false positive because following the naming convention rules it would be difficult for others to understand the names. Hence, in a practical context it can be ignored.

Further, we have also recognized bugs which were prone to affect the cohesion and coupling of the code. For example usage of array list instead of an interface.

VII. CONCLUSION

After conducting the automated testing, we could draw several conclusions based on the violations the tool has produced and about the pros and cons of using an automated static tool. We have discussed them here.

A. *Impact of the tool:*

We have selected the PMD tool after carefully analyzing the available open source JAVA testing tools and collectively decided that PMD tool would help us in identifying and acknowledging our inspection goals which are code correctness, consistency and clarity. We identified that the tool had its impact mostly on coding styles like naming conventions. We have also concluded that there were many violations which turned out to be false positive but the tool also indicated a lot of errors which turned out to be true positive and needed serious attention. However, all the analyses we have made were basically with respect to the goals of inspection we have decided. So, there could be a threat to validity wherein we might have judged a violation wrong because it might have satisfied the prescribed inspection goals.

B. *Comparison of individual reviews:*

After individually analyzing the violations, all of us sat down to collectively decide on the category of violation it belongs to. Here, we decided the type based on logical reasoning and not on the basis of majority. In most of the cases all of us had similar line of thought about the violations. There were cases where in, there was one reviewer who had a logical reason and a more convincing evidence than the others, in such a case, the rest of us were in agreement with it. Similarly, there were cases where everyone had different opinions and the final outcome couldn't be decided since none of us had a convincing explanation. There were certain errors which were easy to decide upon such as god class, law of Demeter and certain bad smells in coding. We have also identified that some violations type varied on the basis of context. For example, a violation was considered true positive at a stage of the source code and the same violation at some other context in the code produced a false positive. We also came across violations where none of us could decide what category it belonged to.

C. *Tool performance:*

Our main inspection goals were code clarity, correctness and consistency. The tool we used indeed helped us in examining the code based on the goals we have defined. The tool has been highly efficient in pointing out at incorrect naming conventions. Like we have already mentioned it has also shown a few false positive violations relating to code clarity. It turns out to be better than FindBugs because FindBugs couldn't check for the errors in code clarity. Code consistency was also determined as we have observed outcomes like loose coupling. Code correctness has also been checked. Certain error messages like God class were also identified by the tool which checked for code correctness. Ergo, the tool was indeed productive in finding the bugs related to our inspection goals in spite of a few false positive errors.

D. *Cost Effectiveness*:

We have identified that the time taken to individually analyze the errors varied over time. Initially it took us more time in examining them and later on it gradually reduced (see Table 2). From this, we deduced that static automated testing is initially expensive but over the long run it is more cost effective. At the same time, in spite of our no experience in testing, we could easily examine the violations it produced. This shows that automated testing doesn't require professionals with vast experience or knowledge in the field of testing. Anybody having minimum knowledge about software design or coding principles can perform this testing. Thus, we find Static Automated testing a cost effective process.

Based on our observation, we could say that static analysis is also efficient in implementing regression testing. Since the product we are dealing with is open source software anybody can change it. Whenever there is a change in the functionality of the software such as an enhancement or a patch file, static

automated testing tools help in efficiently unveiling new bugs or violations. From the literature we have studied, we found that static automated analysis is cost efficient and is less expensive when compared to other testing techniques.

Static testing also gave us an insight about the dynamic analysis of the software which falls to be part II of our project.

TABLES USED

S.No	Tool	Accept/Reject	Motivation
1	AgileJ Structure Views	Accepted	Creates class diagrams from source code. Improves visualization.
2	ObjectWeb ASM	Rejected	Designed for Run-time purposes and depends on programming languages
3	Checkstyle	Rejected	Checks only code clarity. Correctness and consistency not inspected.
4	FindBugs	Accepted	Analyzes the code by using detectors. Provides information about errors.
5	GrammaTech CodeSonar	Rejected	Better for inspecting C, C++ code rather than Java.
6	IntelliJ IDEA	Rejected	Needs prerequisite knowledge and understanding
7	Jtest	Rejected	Memory intensive. Testing each module at a time will lead to delays.
8	Kalistick	Accepted	Monitors code quality
9	LDRA Testbed	Accepted	Provides good code coverage analysis
10	PMD	Accepted	Code is analyzed using rule sets that describe a warning.
11	SemmleCode	Rejected	Requires a basic knowledge about .QL to implement.
12	SonarJ	Accepted	Gives a graphical visualization of violations.
13	Soot	Rejected	Doesn't cover majority of code quality metrics

Table 1 List of Static Analysis tools

Reviewer's Name	No. of violations examined			Time Taken		
	Itr.1	Itr.2	Itr.3	Itr.1	Itr.2	Itr.3
Navneet Chamala	107	247	514	210 mins	210 mins	260 mins
Eada Priyanudeep	87	333	448	180 mins	248 mins	372 mins
Lavanya Pampana Vinod	193	168	507	356 mins	169 mins	348 mins
Konduru Krishna Chaitanya	142	281	445	290 mins	310 mins	270 mins
MVNA Uday	117	345	406	185 mins	347 mins	365 mins
Dilip Somaraju	131	267	470	280 mins	230 mins	270 mins

Table 2 Individual time log of defects

Part-II: Dynamic Analysis

***Abstract*—This part of the document presents a test plan for the online open source multi player game, Spaxe Hilk. The document was prepared following the guidelines presented in IEEE 829-2008. The results of the static analysis of the code, presented in part-I, significantly formed the input for the test plan.**

1. INTRODUCTION

Spaxe Hilk game is adapted from a board game that dates back to 1989. Drawn as an inspiration from the online game Space Hulk, it is set against a backdrop of isolated passageway in the graveyard of space. In this game, marines battle out against aliens for survival. Moreover, the user gets to choose between single player and multiplayer game. Multi-player game is enabled only when there are two or more players involved in a session of game. A host of options are available in maps to select from. Certain maps are restricted for multi-player game and single-player game. Chat-box is also provided as a means of communication to pass messages amongst players during a multi-player game.

The test plan is prepared on the basis of system and acceptance testing. As it is essential to produce good test inputs to test software dynamically, we present test cases in Appendix- B. The precondition and post condition for each test case serve as valuable input to the testers for evaluating certain functionalities of the system. As it is preferred to remove as many defects as possible before release of any software product, we present a detailed analysis of dynamic test that was conducted.

This part of the document is organized in two sections: section I presents the introduction covering the scope, level in the overall sequence and overall test conditions. Section II contains details for the level of test plan containing test traceability, features to be tested, approach and test deliverables.

1.1. Document Identifier

Following the guidelines presented in IEEE 829-2008, the document identifier was named as **"TP-TLP-22032014-V5.3"**. The group members involved in drafting this document along with the approver responsible for the course, Dr. Kai Petersen form the authors of this document.

1.2. Scope

The primary objective of this task is to develop a test plan for the game Spaxe Hilk. The game is open source software that is available for everybody [17] to test and analyze. As the game is a host of many operational features such as multi player game hosting, client server interaction and inter player communication via chat box, etc. we present a detailed dynamic testing methodology that helps the practitioner identify defects, correct the defects and have a mitigation policy to tackle it in case it is encountered ever again in the future.

Owing to the fact that the system under test is open source and not developed by any organization, the scope of the test plan is limited to System testing and Acceptance testing for the following reasons:

- Unit testing was not viable as it must be done during development of the software. Since the game under inspection had already been developed by various people, there is no clear definition of a unit. Further, this level of test also requires a Software Requirements Specification (SRS) document in order to test if the design features and requirements are met. However, the SRS document for the game is not available.
- Integration testing was not suitable to implement as a level test because the game undergoes frequent changes in chief functionality quite often. As the game is developed in an open source platform, many users are free to add and modify existing features and functions of the game. Testing each module after combining with the existing modules is not feasible as the entire game lacks proper structure in maintaining modules.
- System testing encompasses the testing of the key functionality of the software features [11]. Although there is no SRS document available for the game under inspection, there is a possibility to create one of our own by considering many design features of the software code. The SRS document that we designed is presented in Appendix-E. This provides as valuable input to identify key functionalities of the game from various perspectives. For this reason, we have tried to identify the prime features of the game by adapting perspective based reading (PBR) during code inspection. Further, the results of the static analysis using PMD tool (as a part of this assignment, included in Part I) formed input to our identification of functionalities. Therefore, we have chosen system testing as one of our level test plans.
- Acceptance testing had been our other level test because any software developed must be accepted by the end user. For any software

development organization, the purpose is totally beaten when a key feature or functionality is understood by the user as an additional feature and merely gold plating. Therefore, as mentioned earlier, we have identified key features assuming various roles as a part of PBR. It is again, using the same strategy that we could come to certain attributes that are accepted by the end user.

The above reasons form the motivation for our choice of level test plan. Although we have mentioned the various points that were considered during the selection process, it is also necessary to mention the possible drawbacks, as validity threats, that our arguments may suffer from. Firstly, as naïve users without any industrial experience, we have conducted PBR though our findings in literature clearly said that this approach is better suited to experienced professionals [6]. Secondly, the choice of key functionalities was done under the light of ISO standards, yet might not well match to industry standards. Lastly, the modules of the game to be tested were chosen by means of prioritization. The limited time constraint had a significant influence while prioritizing these modules. Hence, the final selection may not be the best one possible.

1.3. References

1.3.1 External References

This document is not tailored as a result of any governmental policy or any law regulation, the test plan that we are presenting doesn't require any external references. Hence any references pertaining to laws, governmental regulations, standards and policies are not applicable to our test plan document.

1.3.2 Internal References

The test plan is developed with the help of an SRS document that was prepared by our team. Due to the lack of any specifications from the developers, the only SRS document that was drafted has served as a reference. This SRS was constructed by going through several guidelines presented in many online discussions and blogs by technical practitioners [15][16].

Project Plan

Here we present a summarized project plan that helped us to execute the testing process and further have control over the proceedings. The project plan involves steps related to the project plan as described in PMBOK [11] along with the testing guidelines followed in the V-Model of testing. This plan supports the activities that we carried out in order to conduct dynamic testing. These activities are represented as stages in our test plan.

1.4. Level in the overall sequence

System and Acceptance testing is chosen as levels for testing the code dynamically. The motivation for specifically choosing the two levels is provided under section 1.2. Hence, all test activities illustrated in the previous section are based on these two levels. The project team carried out the relevant test strategy by assuming the roles of user, programmer and tester. The roles are given in the tabular form in Appendix-F.

1.5. Test classes and overall test conditions

1.5.1. Test Classes:

As System and Acceptance testing were chosen as the basis for the level test plan, the test classes are so chosen that the objectives of these levels are fulfilled. System testing focuses on testing the overall system's functional requirements that were developed as a result of an SRS document. Acceptance testing is performed to ensure that key features of the software are tested effectively to avoid defects that lead to reduced performance and reliability.

- **Valid test of input values:** This class encompasses all possible tests in which user can provide valid inputs to the game during interaction. For example, when user chooses a particular map style, the associated game scenario along with the number of players allowed in the map must be shown to the user as pop up. Hence, the selection of the user from a list of map styles forms the valid input values in this instance.
- **Invalid test of input values:** All the remaining values that are invalid as input parameters are covered by this class. Any values chosen from this set of invalid class must make the test condition to fail. This ensures that the user doesn't get undesired results by giving valid inputs.
- **Valid test of output values:** The expected outcome of any user action could either be a successful execution of user request or an error message that is caused due to several functionality issues. This class contains all those valid output values that the user would encounter. Hence this class must have a detailed flow of execution as it is necessary to know the outcome of an input action.
- **Invalid output value analysis:** As it is practically not feasible to understand the cause for malfunctioning due to an invalid input, it is advisable to categorize the encountered errors into groups. This helps to handle such errors efficiently in the future.
- **Testing normal values based on usage profiles:** In order to understand the most commonly used features, we have conducted a role based usage inspection by following PBR technique. Although we, as a test team of six members couldn't represent all user types, yet we endeavored to uncover the most commonly used features in the game by playing it extensively. Using these operational profiles as input, we tested the features by identifying the chief modules.
- **Inter player communication during game play:** As Spaxe Hilk is an online strategy based game, inter communication between players during game

play is essential. Hence various combinations of communication between varied numbers of users are tested to the extent possible.

1.5.2 Test Conditions:

- We have carried out the test process based on the test strategy that was tailored according to system testing and acceptance testing.
- The test process involved conducting desired tests on limited number of modules of the game.
- The choice of the modules was based on the PBR technique that we employed.
- With the limited resources both in terms of code and people, we have conducted the dynamic testing in our personal laptops. The configuration details of every member's computer are provided in Appendix-C.

2. DETAILS FOR LEVEL TEST PLAN

2.1. Test Items and Identifiers

The test cases and the corresponding functional or nonfunctional requirements are presented in Table 2 as a part of the requirements traceability matrix. A description about every test case and about the requirements is provided in Appendix-D.

2.2. Test Traceability Matrix

In this sub section, we present a requirements traceability matrix that is aimed at testing certain functional and nonfunctional features of the game. The requirements presented are a result of the SRS document that was developed by our team. The requirements traceability matrix (RTM) is drawn based on the guidelines provided in IEEE 829-2008.

2.3. Features to be tested

The modules that were selected to analyze as a part of the static analysis in Part-I are retained to be tested for the dynamic analysis as well. Modules namely, Properties, World, Weapons, States and Multiplayer were shortlisted based on the PBR inspection. The module States being very lengthy and hard to handle, only a part of it was tested. The test process conducted is depicted in the form of methodology (mentioned in Section 1.3.2 under 'Project Plan') as follows:

- **STAGE 1 (Choice of test level):** Initially for the test process to take off there needs to be some baseline for developing further strategies for future testing. This baseline is the choice of level of test plan. Care must be taken while choosing at this stage as even a slight slippage of facts related to the testing process would cost heavily in the later stages of the testing process.
- **STAGE 2 (Dividing Roles):** Having identified the level test plan, the next task is to identify and prioritize important modules of the game. For doing so, we have conducted a systematic PBR for identifying the modules that are important to specific stakeholders. In order to carry out PBR for identification of the important code modules of the game, it is necessary that the team members are divided into groups representing users, developers and testers. As a result, all six members of our team have formed three groups of two members (see Appendix-F).
- **STAGE 3 (Requirements Elicitation):** After identifying the relevant modules in the game that are assumed to be important and hold a higher priority than the rest of the modules, the corresponding functional and nonfunctional requirements are mapped in order to generate test cases. Therefore, the functionalities identified in the modules further form basis for the traceability matrix (see Table 4 in Section 3).
- **STAGE 4 (Risk Mitigation):** To carry out the test process, few potential risks are involved. Before initiating the test execution part of the testing process, it is necessary and important to identify these risks and also prepare a mitigation policy so that such risks are addressed easily during any future encounter. For example, procedural delays while testing can cause multiplication of the overall cost of testing. If a test schedule is prepared without keeping in mind such a threat, there is a possibility that the further activities are also disturbed leading poor testing and increased defects that would result in overall poor performance of the system. Therefore, our team has tried to cover as many risks as possible, document these risks along with the mitigation strategy. Other risks caused due to external factors such as system breakdown, data corruption are also analyzed to the extent possible. However, there might be few errors that were not looked into due to the inexperience of our team.
- **STAGE 5 (Test Schedule):** This stage is the start of the actual testing process that was intended to be carried out on the specific test cases. The entire testing effort required keeping in mind the risks that were elicited in the previous stage, is broken down into a schedule. The work break down structure is designed to make every member of the team understand the roles and responsibilities. Although there are no budget constraints applied on this test plan, the time constraint and the threat of not meeting the deadline still apply.

- **STAGE 6 (Test Group identification):** Once the testing process begins, it is important to look for those aspects of code that have similar functionality and group them into a category so that all modules can be handled efficiently. The warnings encountered during static analysis testing helped in identifying those parts of the code where errors are reported. This helped to categorize the defects as well as the code functionalities.
- **STAGE 7 (Prioritizing Tests):** Among the various test processes involved during test plan execution, few tests such as testing for important functionalities take up a higher precedence compared to regression test or integration test. Therefore, even the test strategies used in testing the selected modules are given priorities during testing schedule. This means that tests with high priorities must be answered and require immediate attention.
- **STAGE 8 (Collecting Test Status):** All reports related to the tests being conducted are essential to evaluate the test process and further help in tailoring the test plan for the rest of the project lifetime. Any unexpected outcomes of a test case implementation are reviewed so that such unanticipated behavior can be controlled. Therefore, all the results of the tests are gathered at this stage from all testers and reviewed.
- **STAGE 9 (Test Summary):** Towards the end of the testing process, it is important to reflect the findings of the entire test process as a summary. This is done with an intention of documenting the testing experience which would be essential for further improvement and also cater to process repeatability. Anybody who intends to conduct the same testing must be able to do so successfully.

2.4. Features not to be tested

For the game to run successfully it is important that the system configuration complies with the game requirements. Graphics being a major aspect of the game play is heavily dependent on the system graphic card configuration. As this is one aspect that is entirely related to the end user and not related to either the developer or tester, the graphic compatibility issue is not addressed. Likewise, the goodness of connection between users relates to the bandwidth, the connection speed and the client-server response time. Such aspects are hard to analyze in the limited time constraint. Restricting our test to only certain specific modules of the game as mentioned earlier, might leave behind a lot of important code parts. This is a possible validity threat that our observation might suffer from.

2.5. Approach

Due to the limited time available to carry out both static and dynamic analysis coupled with the inexperience of our team members, inspecting the code in its entirety was not a viable option. Another reason for not going through all aspects of the code was that the quality of the analysis might be compromised. Therefore, in order to produce quality analysis for limited functionalities, we have selected only a few modules of the game. Our objective of testing the important modules of the game under System and Acceptance testing was supplemented by the following four steps:

i) **Test Planning:** This step included deciding which level of test plan to be implemented. After exhaustive discussion and review, our team has decided to choose System and Acceptance testing as the basis for our level test. Accordingly, the relevant test cases pertaining to functional and nonfunctional requirements were evolved. Motivation for the choice of the level test plan is already mentioned in Section 1.2.

ii) **Test Design:** How exactly are the activities involved in the test plan are to be implemented is the basis for the test design stage. The traceability matrix artifact provides all the group members as to which module and further which aspect of the functional requirement is being tested.

iii) **Test Execution:** Testing the game code against the set of test cases in order to reduce defects and improve game performance is the essence of this stage. The pass/fail criteria provide the tester with relevant information to verify the expected outcome against desired outcome. As pointed out in earlier sections, the test cases were so developed such that the implementation of the test process is made repeatable and independent of the tester.

iv) **Test Analysis:** Towards the end of the testing process, many errors are found out under various test conditions. This stage of the testing approach involves exhaustive review by all team members to analyze the type of error, its occurrence during game play and finally the mitigation strategy to be followed in the future. It is during this stage of the analysis that the warnings presented by the use of static analysis tool PMD, have given us valuable inputs in understanding the defects. Analyzing the code that flagged the error and initially deciding if the error was false positive or true positive is the basis for the approach.

The above mentioned steps were implemented as a part of the test approach to implement the test plan. Typical steps involved in a software testing life cycle (STLC) have formed the basis for the test strategy that we implemented [10]. As a part of the STLC, motivation for the choice of tool selection, the measures used in analyzing and the metrics for measurement are included. It is for this reason that the

following two headings speak about tools, measures and metrics:

- **Tools:** For implementing Part-I of the assignment, a Java code inspection tool, PMD was chosen. As this was the case for automated static code analysis, implementing a tool was inevitable. Whereas, for the second half of the assignment where dynamic code analysis had to be carried out, our team has decided to conduct the test process manually without implementing any automated tool. By doing so, we also tried to analyze the significant differences between using an automated tool and conducting manual test. However, finding out which among the two approaches suites better is not our objective.
- **Measures & Metrics:** During code analysis, we have retained *code clarity, correctness and consistency*, the same evaluation criterion that was chosen for static analysis during code inspection as the measure for dynamic analysis. Along with this, the code cohesion and coupling had been additional aspects under study. The outcomes of the static analysis formed valuable input for the dynamic analysis.

2.6. Pass/Fail Criteria

Based on the test case and the functionality of the requirement under test, we have prioritized the various test conditions and accordingly decided the pass and fail criterion. Although we have not conducted an exhaustive regression testing that uncovers defects at a later stage after repeatedly testing the code, we have designed the criteria that work well for a single test too. The criterion that was decided works well irrespective of the tester. Thus, we expect our test plan to be independent of any internal or external factors.

2.7. Suspension and Resumption Criteria

Suspension Criteria:

Suspension criteria relates to all those internal and external reasons for ending a test process. Although we encountered several reasons to shut down our ongoing work, in this section we present only a select few which are illustrated as follows:

- A particular stage in the test process takes much longer time than intended. This leads to unexpected delays and further increases the number of pending tasks while software inspection.
- While carrying out this assignment, both static and dynamic testing had to be conducted and the results were to be reflected. In the process, there were many events that required immediate attention and proper mitigation. Hence, such top priority tasks were to be addressed preempting the current task for certain time.
- Due to the limited time constraint and also due to scarce availability of system resources within the group members, we couldn't carry out the test process continuously.

Resumption Criteria:

For all the above mentioned criteria to suspend the activities, there also exist a motivation to resume the activities. It is important and essential to have some criteria to start the test process again as it is not viable to waste the limited time. Based on such mitigation strategies, the following points formed the resumption criteria for our testing process:

- When a suspended work task is not reviewed for a long time, it is brought back to inspection.
- When system resources are up and back to work, the test process is resumed.
- When the test criterion is tested based on test prioritization, the element lower in priority is reviewed when there are problems with an element higher in priority. Hence, if answering a lower level requirement provides with a possible solution to a higher level requirement, the test process that was suspended earlier is resumed back.

2.8. Test deliverables

The following are the software artifacts that emerged at various stages of the testing process. Few of these artifacts have formed the baseline for activities conducted later on, during further stages of test process.

- SRS document: Developed during initial requirements elicitation phase. Provides input for testing functional and nonfunctional requirements against set of test cases. As this document was not readily available, we have developed it by identifying key features of the game under inspection (see Appendix-E). The task of analyzing important features was done based on PBR.
- Traceability matrix: The test cases that were generated in order to test the game focus on a particular feature of the game. Matching these functional or nonfunctional requirements to the corresponding test cases is performed with the help of traceability matrix (see Table 4). In an industrial environment, this provides with valuable information to the tester as well as end user. In our inspection process, the traceability matrix helped us to handle the requirements and tests in an effective manner.

Test reports: At every stage of the testing process, test reports were generated as a result of carrying out the test process against the set of test cases. The result of the test reports formed input to the subsequent stages in the testing process.

3. TABLES USED

S.No	Stage	Purpose
1	Choice of Test Level	Inform group members about type of test
2	Dividing Roles	Conduct PBR for identifying modules
3	Requirements Elicitation	To prepare SRS document and test cases
4	Risk Mitigation	Make action plan to tackle any risks/threats
5	Test Schedule	The dynamic test plan is given in Appendix-
6	Test Group identification	Helps in categorizing similar program areas
7	Prioritizing Tests	Focus on important modules
8	Collecting Test Status	To review the progress of the test process
9	Test summary	Provides overall observation of test done

Table 3 Test Strategy

		Requirements					
S.No	**Test Cases**	Req-1	Req-2	Req-3	Req-4	Req-5	Req-6
1	TC-001	X					
2	TC-002		X				
3	TC-003			X			
4	TC-004				X		
5	TC-005					X	
6	TC-006						X

Table 4 Test Traceability Matrix

REFERENCES

[1] P. Hsia, D. Kung, and C. Sell, "Software requirements and acceptance testing," *Ann. Softw. Eng.*, vol. 3, no. 1, pp. 291–317, 1997.

[2] P. Hsia, J. Gao, J. Samuel, D. Kung, Y. Toyoshima, and C. Chen, "Behavior-based acceptance testing of software systems: a formal scenario approach," in *Computer Software and Applications Conference, 1994. COMPSAC 94. Proceedings., Eighteenth Annual International*, 1994, pp. 293–298.

[3] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, 2009, pp. 141–150.

[4] A. Aurum, H. akan Petersson, and C. Wohlin, "State-of-the-art: software inspections after 25 years," *Softw. Test. Verification Reliab.*, vol. 12, no. 3, pp. 133–154, 2002.

[5] O. Laitenberger and J.-M. DeBaud, "Perspective-based reading of code documents at Robert Bosch GmbH," *Inf. Softw. Technol.*, vol. 39, no. 11, pp. 781–791, 1997.

[6] M. Ciolkowski, "What do we know about perspective-based reading? An approach for quantitative aggregation in software engineering," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 133–144.

[7] C. Chahar, V. S. Chauhan, and M. L. Das, "Code Analysis for Software and System Security Using Open Source Tools," *Inf. Secur. J. Glob. Perspect.*, vol. 21, no. 6, pp. 346–352, 2012.

[8] R. Plosch, H. Gruber, A. Hentschel, G. Pomberger, and S. Schiffer, "On the relation between external software quality and static code analysis," in *Software Engineering Workshop, 2008. SEW'08. 32nd Annual IEEE*, 2008, pp. 169–174.

[9] R. E. Fairley, "Tutorial: Static analysis and dynamic testing of computer software," *Computer*, vol. 11, no. 4, pp. 14–23, 1978.

[10] J. Tang, "Towards Automation in Software Test Life Cycle Based on Multi-Agent," in *Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on*, 2010, pp. 1–4.

[11] R. S. Pressman and W. S. Jawadekar, "Software engineering," *N. Y. 1992*, 1987.

[12] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *Softw. Eng. IEEE Trans. On*, vol. 32, no. 4, pp. 240–253, 2006.

[13] J. Watkins and S. Mills, *Testing IT: an off-the-shelf software testing process*. Cambridge University Press, 2010.

[14] About PMD http://pmd.sourceforge.net

[15] http://stackoverflow.com

[16] http://msdn.microsoft.com/en-US

[17] About Spaxe Hilk
http://sourceforge.net/projects/spaxehilk/?source=directory

[18]. M. Fewster and D. Graham, *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., 1999.

APPENDIX-A GANTT CHART

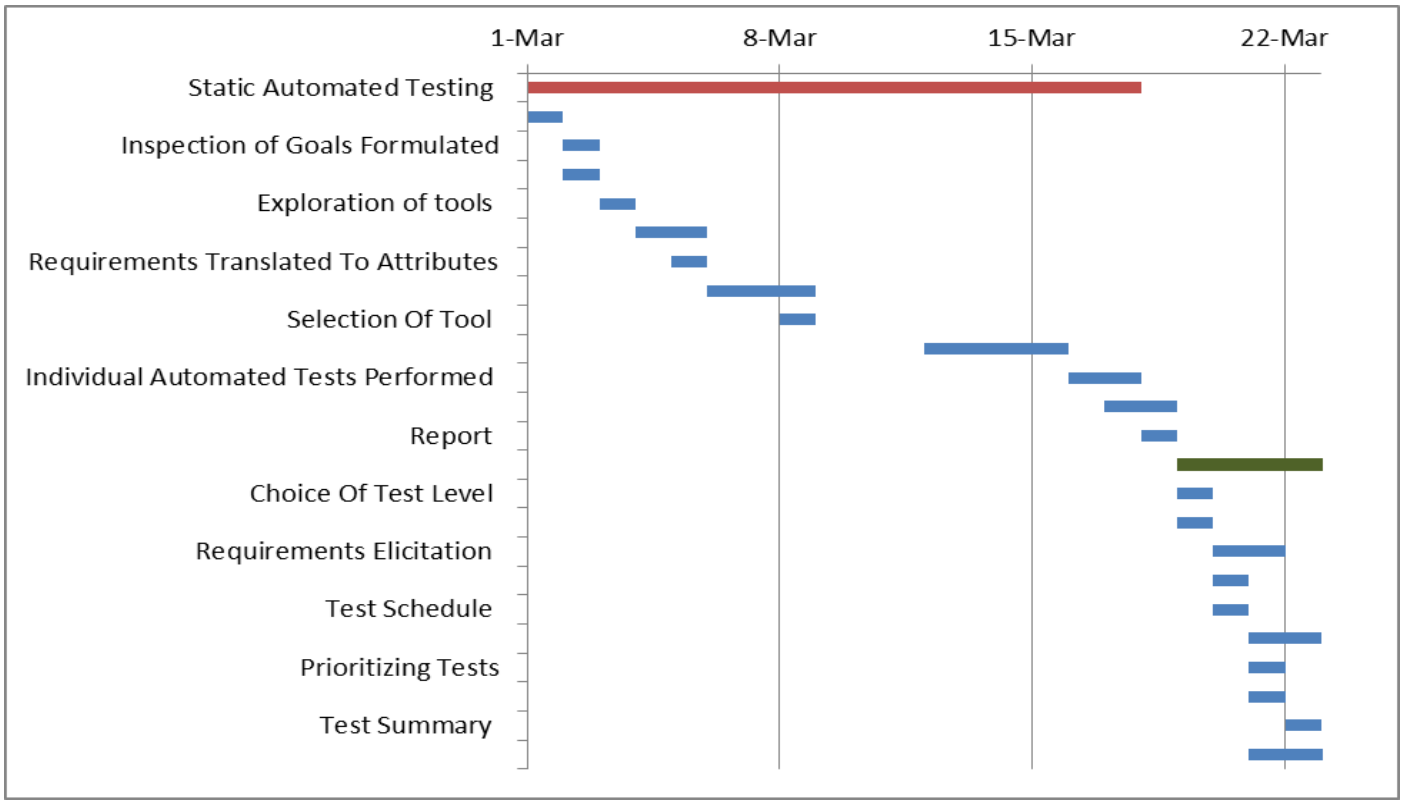

Figure1. Gantt chart

APPENDIX-B TEST CASES

Test Case ID	Generated By	Performed By	
TC-001	Navneet Chamala	*(to be filled by the tester)*	
Test Case Description	1. Any changes made by the clients must be updated after clicking the "Synchronize" button.	**Req. ID**	FRM001
Purpose	1. To test when the changes are being updated.	**Factors**	Functional factors
Pre Requisites	1. Stable Local area Network Connection. 2. Properly functioning game. 3. Basic knowledge about the game.	**Priority**	Medium
Test Process	1. Establish a network connection. 2. One of the group members will host a game and the others connect to the host. 3. One of the clients will update his information. 4. Clicks Synchronize button.	**Any other Test cases involved**	NO

Expected output	Actual Output
1. Host creates a game. 2. Clients connect to the game. 3. One of the clients updates the information and clicks synchronize. 4. The information is updated.	*(to be filled by the tester)*

Remarks	*(to be filled by the tester)*	Repetition No.	*(to be filled by the tester)*

Status (pass/Fail)	Defect Severity	Bug Id
(to be filled by the tester)	*(to be filled by the tester)*	*(to be filled by the tester)*

Test Case ID	Generated By	Performed By	
TC-002	Eada Priyanudeep	*(to be filled by the tester)*	
Test Case Description	The game runs on all the platforms uninterrupted.	**Req. ID**	FRM-002
Purpose	1. To test whether the game runs on different operating systems.	**Factors**	Portability
Pre Requisites	1. LAN connection required.	**Priority**	High
Test Process	1. The game is downloaded. 2. The game is run on the operating system.	**Any other Test cases involved**	NO

Expected output	Actual Output
The game must run on any operating system.	*(to be filled by the tester)*

Remarks	*(to be filled by the tester)*	Repetition No.	*(to be filled by the tester)*

Status (pass/Fail)	Defect Severity	Bug Id
(to be filled by the tester)	*(to be filled by the tester) (HIGH/MEDIUM/LOW)*	*(to be filled by the tester)*

Test Case ID	Generated By	Performed By	
TC-003	MVNA Uday	*(to be filled by the tester)*	
Test Case Description	1. After selecting a map, it shouldn't accept more than the defined number of players.	**Req. ID**	FRM003
Purpose	1. To check whether the game is accepting more than the defined clients for a particular map.	**Factors**	Functional factors
Pre Requisites	1. Stable Local area Network Connection. 2. Properly functioning game. 3. Basic knowledge about the game.	**Priority**	Medium
Test Process	1. Host creates a game and selects a map. 2. Clients join the game.	**Any other Test cases involved**	NO

Expected output	Actual Output
1. Number of clients who will have access to the game must be the same as the defined number of players for that particular map.	*(to be filled by the tester)*

Remarks	*(to be filled by the tester)*	Repetition No.	*(to be filled by the tester)*

Status (pass/Fail)	Defect Severity	Bug Id
(to be filled by the tester)	*(to be filled by the tester)*	*(to be filled by the tester)*

Test Case ID	Generated By	Performed By	
TC-004	Krishna Chaitanya	*(to be filled by the tester)*	
Test Case Description	1. The status of the system restored when a client (player) leaves the game in the middle of a session.	**Req. ID**	FRM-004
Purpose	1. To check the status of the system when a client (player) leaves the game in the mid-session.	**Factors**	Robustness
Pre Requisites	1. Stable LAN connection required. 2. Basic knowledge about the game required. 3. Proper Functioning of game.	**Priority**	High
Test Process	1. Two or more players join the game. 2. One of the players (client) leaves the game abruptly.	**Any other Test cases involved**	NO

Expected output	Actual Output
The game should restore and continue functioning normally in spite of a client leaving the game abruptly.	*(to be filled by the tester)*

Remarks	*(to be filled by the tester)*	Repetition No.	*(to be filled by the tester)*

Status (pass/Fail)	Defect Severity	Bug Id
(to be filled by the tester)	*(to be filled by the tester) (HIGH/MEDIUM/LOW)*	*(to be filled by the tester)*

Test Case ID	Generated By	Performed By	
TC-005	Lavanya Pampana Vinod	*(to be filled by the tester)*	
Test Case Description	1. While playing the game, if a player clicks on "End Turn", he shouldn't be given access to any of the features until his next turn.	**Req. ID**	FRM005
Purpose	1. To check if the game responds properly when a player clicks on "End Turn".	**Factors**	Functional Factors
Pre Requisites	1. Stable Local area Network Connection 2. Properly functioning game 3. Basic knowledge about the game	**Priority**	High
Test Process	1. Host creates a game. 2. Clients connect to the game. 3. All the information is updated. 4. Game is started. 5. A player finishes his moves and clicks end turn.	**Any other Test cases involved**	NO

Expected output	Actual Output
1. The player who clicks end turn shouldn't be given access to rest of the features until he gets his next chance.	*(to be filled by the tester)*

Remarks	*(to be filled by the tester)*	Repetition No.	*(to be filled by the tester)*

Status (pass/Fail)	Defect Severity	Bug Id
1. *(to be filled by the tester)*	*(to be filled by the tester)*	*(to be filled by the tester)*
(to be filled by the tester)	*(to be filled by the tester)*	*(to be filled by the tester)*

Test Case ID	Generated By	Performed By	
TC-006	Dilip Somaraju	*(to be filled by the tester)*	
Test Case Description	1. An intruder invades into a game session.	**Req. ID**	FRM-006
Purpose	1. To ensure security in game. 2. To check whether the intruder can modify game.	**Factors**	Security
Pre Requisites	1. Stable LAN connection required. 2. Game runs properly. 3. Clients do not violate internal security breach.	**Priority**	High
Test Process	1. Two or more players join the game. 2. An intruder intrudes the game.	**Any other Test cases involved**	NO

Expected output	Actual Output
The game does not fall prone to the security attacks by an intruder.	*(to be filled by the tester)*

Remarks	*(to be filled by the tester)*	Repetition No.	*(to be filled by the tester)*

Status (pass/Fail)	Defect Severity	Bug Id
(to be filled by the tester)	*(to be filled by the tester) (HIGH/MEDIUM/LOW)*	*(to be filled by the tester)*

APPENDIX-C SYSTEM SPECIFICATION

Team Member	PC/Laptop	Processor (Type, Speed)	Memory	OS
Navneet Chamala	Laptop	Intel i5-2.4GHz	4GB RAM	64-Bit Windows7
Kondaru Krishna Chaitanya	Laptop	Intel i7-3.3GHz	8GB RAM	64-Bit Windows8
Dilip Somaraju	Laptop	Intel i5-1.8GHz	4GB RAM	64-Bit Windows8.1
MVNA Uday	Laptop	Intel i3-2.2GHz	4GB RAM	64-Bit Windows7
Eada Priyanudeep	Laptop	Intel i7-2.4 GHz	6GB RAM	64-Bit Windows 8
Lavanya Pampana Vinod	Laptop	Intel i7-2.2GHz	8GB RAM	64-Bit Windows8

APPENDIX-D TEST CASE AND REQUIREMENTS DESCRIPTION

Sr.No.	Test Case ID	Type of Requirement	Test Case Description
1.	TC-001	Functional	Functionality of the synchronize button.
2.	TC-002	Non-Functional	The game runs on different platforms.
3.	TC-003	Functional	Constraint on the number of players.
4.	TC-004	Non-Functional	The status of the system restored when a client leaves the session abruptly.
5.	TC-005	Functional	Functionality of the end-turn button.
6.	TC-006	Non-Functional	Intruder invades a game session.

Note: TC= Test Case

Sr. No.	Requirement ID	Type	Attribute
1.	Req1	Functional	Functional factor
2.	Req2	Non-Functional	Portability
3.	Req3	Functional	Functional factor
4.	Req4	Non-Functional	Robustness
5.	Req5	Functional	Functional factor
6.	Req6	Non-Functional	Security

APPENDIX-E SRS DOCUMENT

SRS Document:

- **Purpose:** A Software Requirements Specification document is aimed at gathering the functional and nonfunctional requirements of the game Spaxe Hilk. The functional and non-functional requirements form the base for generating the test cases. It also gives a brief overview about the software product. SRS complies with the traceability matrix. We have gathered the functional and nonfunctional requirements from the game's website http://sourceforge.net/projects/spaxehilk/?source=directory.

- **Scope:** The scope pertains to a game called Spaxe Hilk which is a turn based tactical battle game, where marines battle it out against aliens. It is a purely multiplayer game and can be played with 2 or more people, depending on the mission. The following SRS document provides details about the functional and nonfunctional requirements of the game. From the available information on the website and our insight about the game, we prioritized the top functional and nonfunctional requirements in our SRS document.

- **Specific Requirements:**

  - Functional Requirements:

    1.1: Hosting and joining a game:
    - 1.1.1: The system allows players to host or initiate a game.
    - 1.1.2: The system allows players to join a game.
    - 1.1.3: Players can have their own name and own IP address.

    1.2: Starting a game:
    - 1.2.1: The game allows the host to select a map.
    - 1.2.2: The number of clients depends on the map selected.
    - 1.2.3: The clients update the information regarding the team name and other information.

    1.3: Intercommunication between players:
    - 1.3.1: The system allows players to communicate with each other by using the chat box.
    - 1.3.2: The messages are seen by everyone involved in the game.

    1.4: Game play:
    - 1.4.1: The players belong to either the marine team or the alien team.
    - 1.4.2: Rotate and move are the fundamental features that a player can execute.
    - 1.4.3: Once a player clicks on End Turn, he cannot carry out any operations until the next turn.

  - Non-Functional Requirements:

    2.1: Portability: The game should work on any operating system.

    2.2: Robustness: The game should work even if one of the players doesn't respond due to unexpected reasons.

    2.3: Security: The system must be protected from security breach.

APPENDIX-F ROLES OF INDIVIDUAL TEAM MEMBERS

S.No	Name	Role
1	Chamala Navneet	User
2	Eada Priyanudeep	User
3	Pampana Lavanya	Programmer
4	MVNA Uday	Programmer
5	Krishna Chaitanya	Tester
6	Dilip Somaraj	Tester

Table 5 Roles of Team Members