Applying automated testing in an existing client-server game
A pursuit for fault localization in Quake 3
Abstract

This paper addresses the question formulation “Is it possible to implement automated testing in an existing client-server game in order to pinpoint faults and achieve credibility to tests?” The gaming industry’s goal, in most cases, is to release games that appeal to both their financial goals and the enjoyment factor of the players. In order to fulfill these goals, the game will need to function properly and the process to assure this is testing the game to find possible faults.

This process is time and cost consuming in an exponential rate in accordance to game extensiveness, which makes this problem a very important decision in the process of development. The problem is most commonly tackled by using massive manual testing session, called alpha or beta sessions. In these session the game is at an early stage of development and gets released to a set player base to test and report issues encountered.

We believe that the process of testing games could be more effective by utilizing automated testing. This thesis will investigate the possibilities to our claim. The result is a visual representation of the tests we managed to apply, while focusing on the client-server connectivity of Quake 3 and a graph of measurements for our improvised fault localization. This paper describes a solution in form of automated tests within a existing client-server game and a start to what could be early stages of a pattern obtained throughout this project.

Keywords: automated testing, client-server, network, fault localization, game, software complexity, Quake 3 Arena
Preface

We love games and all there is to it, which makes it complete wonder to be able to take time and study the essence of how games are made and other surrounding factors. We focused mainly on the network portion of the game Quake 3 here because we believe a great weight lies on it to work flawlessly in today’s games. We have gained great knowledge from writing this thesis, in areas such as the structure of a scientific paper and writing techniques. It has been a tough period of time and we were very reluctant to write text that is not code at first and still are somewhat.

Special thanks goes out to supervisor, Tobias Ohlsson and also Daniel Toll, both of which have been a great support along the way. We would also like to thank Fabien Sanglard for letting us use his images on the Quake 3 architecture, it was most welcome.
1 Introduction

In this chapter we are giving an introduction to automated testing in games. We start with briefly explaining the emergence of general testing, how it was used historically and how it later was embraced by software industries. We illuminate research on the topic, what has been done and is left uncovered. Last but not least we demonstrate a problem definition to our subject and provide a theory to, whether it is possible to apply automated testing in an existing client-server game.

1.1 Introduction / Background

Games today are becoming all the more sophisticated, and the process to develop them are seeing significant increases in both complexity and funding [1]. The gaming industry is one of the world’s largest and fastest growing industries [2], with a turnover around $63 billion worldwide [3]. David Greenspan et al. [3] states that "many major video game titles have budgets comparable to Hollywood movies and the success or failure of a major title can make or break a studio". One of the primarily and broadly reasons to a game not being successful is the cause of bugs, e.g. the game is not working as intended.

An increasing number of today’s modern games are having issues and defects already at first day of launch, which introduced “day one patches”: a required update with bug fixes via Internet. The three biggest franchises launching this year all have problems, ranging from problematic frame rate drops\(^1\), glitches\(^2\) to rendering games outright unplayable, i.e. broken online multiplayer. These are all multimillion dollar, worldwide franchise. [R3] With improvements in complexity of software development process and the extensive budgets involved, Claudio Redavid et al. [1] claims that testing is getting more and more complex and important.

Riaz Ahamed [4] explains testing as a critical element of software quality assurance and represents it as the best review of specification, design and code. The author also implies software testing as the process of testing the functionality and correctness of software by running it. Automated testing is an important technique that is used to quality assure software applications. It involves the developer implementing code that is dynamically testing the source code with the intent of discovering faults and defects. The test code is structured in such a way that the developer declares a predefined state which yields a result when the test is being executed. The result is then verified by matching it towards a predefined expected result [5].

The process of testing software did not arise until 1956 where testing and debugging was considered the same. In the year 1988 the “prevention oriented period” [6] was introduced and lay the ground to how most applications are developed today. Testing has spread widely through the years and is used in various orientations of software development, mainly in web but also game development. One approach that almost always seem to find its way into the life cycle of a game is the use massive manual testing sessions, more often referred to as alpha and/or beta testing. In these sessions the end-users are provided with alpha or beta version of the game in order play the game in a restricted period of time and report defects to the developers [7][1]. The concept of testing is wide and therefore we have decided to specialize our report in one particular area of testing

---

\(^1\)The rate at which the game is presenting images on the screen (creating illusion of a moving environment) and suddenly is dropping drastically.

\(^2\)A short-term fault that corrects itself, and is therefore difficult to troubleshoot.
which is automated testing within network games.

When it comes to testing network games, several other aspects are going to affect the testing process as well as way the game is played for the involved participants. Aspects such as: synchronization and timing [8]. Synchronization in network games is mainly referred to as the idea of keeping multiple copies of a data set in coherence with one another, or to maintain data integrity. This is very important in for example first person shooter (FPS) games where the player’s instance of the game and the game host computer both need to have the same representation of the game world in order to achieve fair play. Timing in network games can be described as the need to ensure that everything occurs at the same time for all participants.

1.2 Previous research

During the forging of our empirical question, searching after earlier studies on the subject of testing distributed computing and even more specific, testing game networks we found only few decent articles on the topic [9][10][11]. We used a selected few sites [R1] in order to search for distributed computing, such as client-server and peer-to-peer, that utilize automated testing.

The scarce results we got were either very different solutions as to what we are attempting or not applicable to our project. The common approach stated in these articles was to develop a tool in order to feed input to the application and observe the output, so called black box testing. Our aim is to perform tests using the actual code of the application as we see fit to achieve a result and thereby pursue the process of white box testing.

In the end, if the application was not developed along with tests from the ground up, the work to create and perform testing becomes very hard. Leaving the developer with one of two choices: to not focus on testing during development at all, and let customers perform a wider testing of the application, or later realize that it would be very beneficial to construct the application in such a way that its main functionality can easily be tested. In choosing the latter, the weight of the decision shifts towards keeping the code or either retracting/removing it in its entirety, and build it anew. The result from our research tells us that testing of network games is barely studied which also applies in the weight and interest to our empirical question.

Last but not least we decided to use Quake 3 which is a fully fledged networked based game, which gives us much weight to our research as it is a real world problem in the industry, applying automated testing in an existing game based network.

1.3 Problem definition

The problem we want to assess, roots in the way that games are tested. According to Alfredo Nantes et al [7], one of the most common ways to test a game is through the alpha and/or beta version of the game. The reason as to why companies perform these large tests is mainly diversity. By letting players from different parts of the world that use different Internet providers and varying technological equipment play the game, you can discover bugs that you would otherwise not be able to [11]. This has proven to be a very beneficial way of finalizing the game, but often leads to it being the only reliable way of capturing the various different situations that may occur. So why thorough testing of the code often is disregarded during development or before the beta release?

We believe that the main reason that testing thoroughly is often disregarded is in direct connection to the extensiveness of the game itself. Applications in general grow larger at
an exponential rate and games are not in any way different regarding that aspect. In larger applications more time and resources will be needed to implement these tests. Other factors that we have discovered in our research that contribute to this problem are issues such as synchronization and timing [8], as stated in the introduction. The time and effort needed to implement tests in an already developed game to ensure that the entire structure is being tested is very extensive.

W.Eric Wong et al. [12] states that regardless of effort and time spent on developing a software program, it may still contain errors and faults, e.g. bugs. He also explains, to debug programmers must first be able to identify exactly where the bugs are, in order words use fault localization. Only after that can programmers find a way to fix them, known as fault fixing [12]. Software fault localization is one of the most expensive activities in program debugging, more than 50% of the total cost may be expended in a software life cycle [12][13]. Without any doubt there are room for huge improvements and therefore exists a high demand for automatic fault localization techniques that can assist and pinpoint to the locations of faults with minimal human intervention [12]. There are already a few techniques developed to localize fault, yet they all share the same goals which are effectiveness, precision and in formativeness [12].

Our focus will lie on testing and fault localizing the client-server connectivity portion of Quake 3. The decision to apply testing to only this area is in direct connection to the amount of time we have to complete this thesis, our lack of knowledge when it comes to the language C and the structure of Quake 3. Which leads up to our empirical question.

### 1.4 Purpose and research question / hypothesis

The purpose of this project is to attempt implementing automated testing in an existing client-server game to facilitate the implementation process of testing while developing games. We also want to raise awareness of complications that can occur while implementing these tests and how to bypass them.

As a result the ultimate goal would be to find and extract early stages of a pattern which might be applied on any client-server game, and hopefully grow towards setting a standard or simple guidance for other researchers as well for developers in the future.

- Is it possible to implement automated testing in an existing client-server game in order to pinpoint faults and achieve credibility to tests?

### 1.5 Scope / Limitation

Our study is limited to one commercial game that use the client-server architecture. This architecture is a common choice for games. A broader scope would include peer-to-peer and hybrid architectures. The game is implemented using C with small parts of C++ and Assembly. The game is of the First-Person-Shooter (FPS) genre. A well-known genre that have high requirements on network implementation.

Our choice to focus on a single game has the consequence that it can be hard to generalize our results to other games. However, we believe that a commercial game gives a more realistic result compared to working with smaller prototypes given the time constraints of the project.

### 1.6 Target group

This study addresses foremost the game industry, even more specifically game developers which may be trying to avoid implementing automated tests in their games because vari-
ous complications. We hope the study will provide some guidance and knowledge to help developers with decisions regarding testing in different stages of the development.

Our study is also an attempt to contribute with knowledge to the subject of automated testing within a networked game which would make it interesting to the academic domain as well. This could be a pioneer step to extract early stages of a pattern which might be applied on any client-server game, which will hopefully grow towards setting a standard or simple guidance for other researchers as well for developers in the future.

1.7 Outline

In chapter 2 we will describe our theory on how to try solving the problem, as well as background and terminology. This leads up to the presentation of our method, explanation of our work process and our use of it to record relevant information for our result. In chapter 4 we present the result of using applied method and in chapter 5 we analyze the result we produced. Lastly, in chapter 6, we discuss various problems we have had along the way.
2 Background / Theory

In order to provide a better understanding of the thesis results and how they were achieved, this chapter will explain a basic review of automated testing within existing networked games. We will also clarify concepts that are considered essential for the thesis such as Testing, Distributed computing, Network games and Quake 3 Arena. Last but not least we give an explanation to pinpointing failure within tests.

2.1 Testing

The simplest way to describe the term Software Testing would be: "A process of executing a program with the goal of finding errors” [10]. The process involves a set of test cases that consist of inputs and preconditions. The program then uses these to identify the behavior of the software and also verify the compliance of specific requirements documented by the stakeholders.

Testing is not easily performed, many aspects such as operating environments, inputs and order of execution increase the amount of test cases needed to make sure that the application is working properly. In regards to the complexity and extensiveness of the application, the amount of time needed to assess all environments, input values and orders of execution in all their respective orders and variances is simply not physically possible.

2.1.1 Automated Testing

Automated testing refers to "the development and usage of tools to determine the success or failure of prespecified test cases, against the Application Under Test (AUT), without human input" [5]. Advantages to using automated testing is mainly that it does not require the aid of humans in order to run. While disadvantages will be increased time consumption and cost in maintaining these test cases and the tools that use them. A great example of automated testing can be used is to observe web applications. There are several tools on the market which use different solutions in order to emulate a browser. Examples of such tools [R2] are: Selenium, Test Studio and Test Complete.

2.2 Test suite evaluation

As explained earlier, the purpose of software testing is to provide quality assurance to software in order to detect faults. Unfortunately, the problem of finding all faults in a program, at least for any meaningful software program is essentially unrealistic. Software testing is therefore always a trade-off between cost of implementing further testing and the risk for cost of undiscovered faults in a program [14].

Rahul et al. [14] explains, to help developers make better and more intelligent decisions about testing, ways of evaluating their testing efforts is needed. The most popular method is the use of code coverage criteria [15]. Code coverage describes structural aspects of the executions of a system under tests performed by a test suite. Code coverage includes three types of coverage, statement coverage indicates which statements were executed, branch coverage indicates which branches were taken, and path coverage which is the one more complex of the three indicates which paths are covered in if-statements and loops. In software testing research, the ultimate standard for suite evaluation is most often considered to be actual faults detected by the tests. The second most successive measure of quality is usually by utilizing mutation testing [16], which measures the ability of a test
suite to detect small changes to the source code. Unfortunately, mutation testing is both difficult to implement and computational expensive and therefore often disregarded.

2.3 Distributed computing

Distributed computing, also known as distributed systems is a software system in which multiple components located on networked computers communicate and coordinate their actions by passing messages between each other. The goal is to achieve a network as such to operate and work as one united system [17]. There are different goals of distributed systems, they either may have a common task such as solving a larger computational problem. Alternatively, each computer may have its own task with individual needs and the purpose may be to provide communication services or coordinate use of shared resources. Coulouris et al. [18] interpret distributed systems as hardware- or software components which within networks coordinate through small messages. Coulouris et al. [18] also states that distributed systems today are everywhere because of the easy access of networks and the Internet, which results in applications giving a bigger opportunity to use resources from another geographical location in order to update through messages as concluded earlier.

One type of distributed systems with high demand on the system itself are networked games. These games utilize multiple components which all have different roles and in real time sending requests and receiving responses from multiple users, putting high demand on the distributed system to provide a united result of a game world for all participants [18][19]. As distributed systems get more complex, ensuring that a system meets its prescribed specification is a growing challenge that confront software developers [20]. To counter this and also provide quality assurance, software developers implement tests in their distributed systems but encounter issues due to several reasons, e.g. synchronization and timing. [8].

2.4 Network games

There is a variety of different network architectures used in distributed games, but the most commonly used are Client-server and Peer-to-Peer. Both architectures are explained further in this chapter.

2.4.1 Client-server

Client-server architecture is a network model that utilizes a server (process) on a networked computer that accepts requests from programs running on other computers to perform a service and responds appropriately. The requesting processes are referred to as clients.

2.4.2 Peer-to-Peer

A Peer-to-Peer architecture is a network of nodes linked together which doesn’t communicate according to the client-server model. This mean that a computer doesn’t get a specific role in the communication and no nodes has any privileges over another, i.e. all nodes in the network can act as any role.
2.4.3 TCP and UDP

In order to connect servers and clients with each other, games and applications in general utilize transport layer protocols such as user datagram protocol (UDP), transport connection protocol (TCP) and reliable user datagram protocol (RUDP). The main difference between UDP and TCP is that UDP have no grasp over connection, while on the other hand when using TCP you establish a connection between two machines indefinitely.

2.4.4 First-Person-Shooter

First-Person shooter games is based on the common action pattern that is shooting. The genre is played as if you are perceiving the world from the eyes of the wielder of said shooting weapon. Although many of these games do not have the single objective of just shooting various weapons. In some games there are sequences where the player must run and escape, or otherwise perform non-combat actions across space [21]. In the early days of games in general where peer-to-peer lockstep in local area network (LAN) was exclusive, you can still see this model alive today in real time strategy (RTS) games. Interestingly for some reason, perhaps because it was the first way – it is still how most people think that game networking works. [18].

However, utilizing peer-to-peer lead to several limitations, such as: ensuring that the game is entirely deterministic, ensuring that the game is played out identically on all machines and last but not least, the machines needs to be at an identical state at beginning of the process. therefore the standard way to initiate an RTS is to join a lobby of some kind and from there start the actual game, to uphold the identical states. This almost eliminates the choice of constructing a way to re-join the game if you were to lose connection. This was an apparent problem to FPS games back then when they tried to bring the experience past LAN and onto the Internet.

In the year 1996, John Carmack at id Software released the game Quake, using client/server instead of peer-to-peer to remedy the need to exchange all the information between the various clients. However, he had one more thing he needed to solve before this technique were to be used on a global scale throughout the future. “The problem was of course latency” (Latency is the time it takes for a packet to travel from the client to server or client to server and back to the client again). They solved the problem in two parts. The first part was a client-size prediction of movement that Carmack developed. The second part was latency compensation on the behalf of Yahn Bernier at Valve for the game Counterstrike. This lay a perfect base for games to grow and lead up to the development of Quake 3 which is the game of choice for us to study in this paper [18].

2.5 Quake 3 Arena architecture

Quake 3 Arena (or Quake 3) [22] is a game of the First Person Shooter (FPS) genre, released in year 1999 and is still a very popular game today. The game is based on the id Tech 3 engine developed by id Software [23]. As the engine was initially developed specifically for Quake 3 Arena, it is often referred to as the The Quake 3 Engine. Additionally to Quake 3 Arena, many of today’s modern FPS games uses the engine including Call of Duty, Return to Castle Wolfenstein, Quake Live and many more [23].

Quake 3 Arena takes advantage of many optimization techniques that deal with the limited computing power and network access technologies which were available back in 1999, when the game was released. Many network optimizations were implemented, including the smart use of deltas, which was used to transmit only the changed game
information, compression techniques for reducing packet size and use of client side prediction to improve the apparent responsiveness of the game during network play [24]. The game is divided into 6 general layers where the game itself is based on a client-server architecture as communication model. The message exchange processes between client and server occur through a layer named qcommon which main responsibility is network transmission between client- and server layer. Messages passed between in the qcommon are primarily delta states - game updates based on changes in the game from the last acknowledged state, as explained earlier in this chapter [23].

Due to the chosen network architecture, an important aspect is the use of two separate game states. The server maintains its own global representation of the game state. The client also maintains a local representation of the game, cgame. Communications are over the network link and the two game states are kept synchronized. Branching out from the cgame are the various functions local to the user – functionality specific to the operating system, e.g. (Linux, Windows and Mac), graphics rendering, user interface etc. [22][23].

Figure 2.1: An overview of Quake 3 Arena Architecture.

2.6 Fault injection

Fault injection is a technique where the developer deliberately produces faults in the source code to observe how the application behaves when encountered with said faults. It can be used in both electronic hardware and software systems to measure the tolerance to faults. Fault injection is most beneficial to use on a system that may not experience faults frequently enough using normal testing [25].
2.7 Pinpointing fault with automated tests

As for this study we will try to implement automated tests within an existing client-server game named Quake 3 Arena. In order to gain a result which can be comparable and analyzed we decided to use software metrics where we chose to measure the accuracy, precision and effectiveness of our tests by applying fault localization.

The localization technique we will be using for this study is partly “Program State-based Methods”. A program state consists of variables and their values at a particular point during program execution which can be a good indicator for locating program bugs. A general approach for using program states in fault localization is to modify the values of some variables to determine which one is the cause of erroneous program execution [13].
3 Method

This chapter will provide a detailed explanation of the methods used in this study. The method can be divided into two procedures; a practical part were we are going to focus on implementing certain tests and use fault localization techniques to locate faults in the code. A theoretical part were we explaining the various techniques we have used, how and why we have implement certain tests, how to locate faults in the code and the reliability of our result.

3.1 Scientific approach

In our study we have chosen to proceed with a qualitative scientific approach. We begin with extracting a few but fundamental features which already exists in the Quake 3. Next step is a practical part were we try to implement automated tests in an existing client-server game for chosen features. We then proceed with injecting simulated faults (Fault injection) in the code and observe how our tests handle the situation. Utilizing an iterative work method we measure our tests with different fault localization techniques to provide numerical data which in the end can help prove our work. We believe that our result can further the knowledge around testing larger applications, help with decisions regarding testability of the reader’s applications and also provide a summary of faults that may occur when attempting to implement automated testing in an existing application.

3.1.1 Techniques

The techniques that have been used in this study is Visual Studio 2013 Community Edition running on Windows 8.1 when performing the automated tests. The system under test consists of the Quake 3 source code, which is built using C, C++ and can be found on GitHub as an open-source project. The tests which were implemented was separated into an own project named q3_test. A framework named Unity was used to implement the different test cases. Unity is a unit test framework written entirely in C language and is scaled well from small to large projects.

3.2 Selection of tests

During the forging of our automated tests, the main focus has been on deriving tests which heavily involve the network part of Quake 3 Arena. We began with analyzing which features were already implemented in the game as Quake 3 Arena is an already existing game. The features chosen were a few commonly and fundamental functionalities which are used in the game. To give our tests a good structure we wrote scenario based test cases and used them while implementing the automated tests. The test cases which was chosen were the following:

TC 1 - Player tries to connect to an online game.

- **Description** - A player tries to connect to an online game session and set connective state to CONNECTED.
- **Prerequisites** - Both client and server portions of the test environment needs to be initialized.
- **Steps** - Start the test environment executable, select the first option by typing 1, then press enter and observe test results.

- **Input** - The input can be found in `NetworkTests.c` within function `PlayerConnectsToGame`. Inputs are stored in a `appConfig` object.

- **Input** - The input can be found in `NetworkTests.c` within function `Expected result`
  - The connective state should be changed to `CONNECTED` at the end of the test execution and thus present a green message that says **PASSED**.

**TC 2 - Player tries to disconnect from an online game.**

- **Description** - A player tries to disconnect from an current online game session and set connective state to `DISCONNECTED`.

- **Prerequisites** - In order to execute this test case, TC 1 must be fulfilled which means a player must participate in a current online game session with a connective state of `CONNECTED`.

- **Steps** - Start the test environment executable, select the first operation by typing 2, then press enter and observe test results.

- **Input** - The input can be found in `NetworkTests.c` within function `PlayerDisconnectsFromGame`. Inputs are stored in a `appConfig` object.

- **Expected result** - The connective state should be changed to `DISCONNECTED` at the end of the test execution and thus present a green message that says **PASSED**.

**TC 3 - Connected player gets kicked by host.**

- **Description** - A game host tries to kick a connected player from the current online game session and set connective state to `DISCONNECTED`.

- **Prerequisites** - In order to execute this test case, TC 1 must be fulfilled which means a player must participate in a current online game session with a connective state of `CONNECTED`.

- **Steps** - Start the test environment executable, select the first operation by typing 3, then press enter and observe test results.

- **Input** - The input can be found in `NetworkTests.c` within function `PlayerDisconnectsFromGame`. Inputs are stored in a `appConfig` object.

- **Expected result** - The connective state should be changed to `DISCONNECTED` at the end of the test execution and thus present a green message that says **PASSED**.
3.3 Method realization

In order to implement the chosen tests with the end-goal of successfully do so and to pinpointing faults, we first had to forge a strategy in which way to proceed. To get a good understanding in how the feature under test worked we first executed each test manually by running the game without any modifications.

Before implementing any automated tests we spent time on researching which way was the easiest and fastest to inject our commands to affect the game. Soon we found out that a dedicated server setup was the easiest way for us. A dedicated server in Quake 3 Arena provided us with a Quake console which we could execute predefined game commands by inputting key words. We took advantage of this option and let our tests inject commands to affect the game.

Figure 3.1: An overview of Quake 3 console.

Our approach to implement the automated tests started with setting up a configuration object named appConfig in which instructions were applied before each test execution. The configuration object held various information, all from current simulated error, current connection state, and a stack trace print to information as which commands should be executed on the dedicated server console to perform various actions.

Each test could either have a single configuration or multiple configurations which then were chained together using Linked-List structure [26]. The last option was used
when the test case had a state prerequisite before executing the actual feature under test. In figure 3.2 below we show an example of a test case setup with multiple configurations. This particular test requires a specific state where the player must be connected to a game before executing the second configuration, in which the player is trying to disconnect from a game.

```c
// AppConfig PlayerDisconnectsFromGame() {
    // Assign
    ac = allocConfig(2);
    ac->testName = "Arranging game state.\n";
    ac->errorType = CONNECT_FAIL;
    ac->ptr = &threadFunc;
    ac->execString = "connect 127.0.0.1";
    ac->finished = FALSE;
    ac->reset = FALSE;
    ac->connstate = (int)CA_DISCONNECTED;
    ac->server = TRUE;
    ac->breakDown = FALSE;

    ac->next->testName = "TEST # 2 - Player tries to disconnect from an online game.\n";
    ac->next->errorType = CONNECT_FAIL;
    ac->next->ptr = &threadFunc;
    ac->next->execString = "disconnect";
    ac->next->finished = FALSE;
    ac->next->breakDown = FALSE;
    ac->next->reset = FALSE;

    ac->next->next = allocTearDownConfig();
    ac->next->first = ac;

    return *ac;
}
```

**Figure 3.2:** A overview of typical test case setup.

```c
int configCount = 1;
if (activeConfig.finished) {
    if (activeConfig.next && countMsec > (100 * configCount)) {
        activeConfig = *activeConfig.next;
        void(*testing)(appConfig config) = activeConfig.ptr;
        testing(activeConfig);
        activeConfig.finished = 1;
    }
}
else {
    void(*testing)(appConfig config) = activeConfig.ptr;
    testing(activeConfig);
    activeConfig.finished = 1;
    configCount++;
}
```

**Figure 3.3:** An overview of the configuration(s) being executed in the game loop.
The next step in our automated test implementation was to provide a user interface where we listed all test cases available, giving the user the option of running a single test case or a whole suite. The user interfaces task was also to present information about the test result for each finished test case. It was solved by using a separate console shown in figure 3.6.

Last but not least a reset feature was implemented to reset the game state after each test case execution which helped getting a fresh game setup for each test when running a whole test suite.

3.3.1 Fault localization

In order to pinpoint faults in *Quake 3 Arena* we implemented a feature to observe the stack trace of execution when encountered with a fault. This feature was named fault localization and was built using *DbgHelp.h* which is found in library *DbgHelp.lib* and is part of *Windows API* in *Visual Studio*. The fault localization is configured to provide a list of methods called during test execution when a test case fails. It helped us to achieve down to actual method of execution when encountered with a handled fault.
3.3.2 Remove dependencies

As we focused on testing the connection parts between client and server of the Quake 3 Engine, we excluded a large portion of the client that is focused on visual representation. We first executed the tests with no dependencies removed and then removed component by component until it wasn’t possible to remove further without the game breaking. Figure 3.7 illustrates which dependencies were excluded from the automated testing.

![Figure 3.7: An overview of removed dependencies represented with red overlay.](image)

3.4 Analysis

Our approach towards analyzing the out coming result started with finding and evaluating metrics which could give us weight and prove success to our implementation of automated tests. We did some research and found five steps that defined a good automated test.

1. Test does what it is supposed to do - sometimes test is designed for one thing but actually checks something else.

2. Test operates with valid data - when we design our tests we should make sure that we use proper input and expectations for the output.

3. Test fails if the functionality under test is inaccessible or changed at all - obviously if system under test doesn't work at all the test interacting with this system should fail.

4. Test is independent - test runs the same way both separately or in any combination of other tests.
5. Test runs the same way multiple times with the same result - each test should be predictable and reliable.

Beyond analyzing the data by previous given metric we decided to give a review on the code which has been implemented for each feature under test.

3.4.1 Fault localization

To analyze the fault localization, also known as pinpointing faults we had to use a different technique. The technique was to make each test case fail when executed, in order to get a list of methods where the fault laid. From these methods we then counted number of method calls until were the fault laid. The total number of calls would be used as result for defining precision and accuracy of the fault localization.

Secondly we summarized number of total method calls called by the automated test, which also was provided by the fault localization and then compared it to total number of method calls while doing manual testing.
4 Results / Empirical data

In this chapter we will present the result we have achieved using the applied theory and method. It will contain graphical information regarding method calls for our test cases, presentation of the test environment, and lastly the contents of our test cases. All above parts will be accompanied by a brief explanation.

4.1 Measurable data

The measurable data that has have been achieved from our method is represented below (Figure 4.1). It illustrates the amount of method calls for the total amount of methods in the scope for the test. The diagram also shows the amount observed by the test and the amount observed when performing the actual step by step approach in Visual Studio.

The breakpoint runtime amount is a more exact representation of calls made to achieve the state required for the test to pass. The amount is combined from all of the game loop iterations for each of the `appConfig` setups, apart from the other two columns that was solely on one iteration.

![Method count](image)

**Figure 4.1**: Diagram showing amount of method calls within each test.

4.2 The test environment

The test environment we have developed contains of a console application that sets up necessary configurations and thread handling to run both itself and the Quake engine. The game engine runs and utilizes the specified configuration to initialize the state required to be in for each test to achieve credibility. These configurations are also used to imitate faults in the application and thus determine if the outcome of the tests change. The state that gives the tests credibility is a connective state on the client portion of Quake engine.
This connective state can hold the representation that the client is for example DISCONNECTED or CONNECTED, that is later used by the tests use in order to compare towards the state we want the game.

The first screen (see figure 4.2) that is presented after starting the test environment is a list of available test cases. There is the option to run each of them separately or all of them after each other automatically. This is cleared after a choice is made and replaced with the appropriate output for selected test.

![Figure 4.2](image)

**Figure 4.2** Visual representation of available test cases.

### 4.3 Overview of test cases

Each of the three tests in the study are explained in detail in this section. Which order they were executed and with provided graphical content to each test.

#### 4.3.1 Player tries to connect to an online game

This test consists of one major command made towards the Quake 3 engine, which is the `connect 127.0.0.1`, where initialized client portion of the test environment is making a request to join the server portion. Figure 4.3 is a representation of where this test is failing after the configuration had been set to instruct the server portion to reject any incoming clients. If a test fails the actual, expected and stack trace analysis is presented. From the stack trace analysis information such as method calls up to the point of the imitated fault creation and up to where the test failed because of the fault. There is an also a user friendly representation of the line number and function names for both of the stacks. See figure 4.3.

By evaluating the test case with the metrics explain earlier in the method it can be stated that the test has passed in all five metric requirements.
4.3.2 Player tries to disconnect from an online game

This test contains of two major commands made towards the Quake 3 engine. First off, it is the very same command that was made in the first test. The second command is the `disconnect` command, that when called will relinquish any connection that the client portion has established to any server portion. When a test passes the expected, actual, method calls and stack trace analysis are not displayed. See figure 4.4.

By evaluating the test case with the metrics explain earlier in the method it can be stated that the test has passed in all five metric requirements.
4.3.3 Connected player gets kicked by host

This test contains of the same connect command from the first and second test and the command `kick PlayerName` where `PlayerName` is the name chosen by the client portion of the test environment. When the server portion receives this command it iterates the list of connected clients and removes the assigned client if found. It then sends a response to the client portion instructing it to perform the same action as it took when the disconnect command was made in test two. See figure 4.5. By evaluating the test case with the metrics explain earlier in the method it can be stated that the test has passed in all five metric requirements.

![Figure 4.5](image_url)

Figure 4.5 Visual representation of feedback from a passing test three on the test console.
5 Analysis

In this chapter we will analyze the result we have achieved and give a more in-depth explaining to why this result was output.

5.1 Diagram analysis

By analyzing the diagram (Figure 10) it is obvious that automated tests are considerably dominant in effectiveness. The fault localization technique used in the method showed a reduction of 93% in amount of methods used when executing the tests. It is with no doubt to conclude that finding the cause of failure decreases significantly, which without fault localization would be around 100 methods to investigate and is now down to 7 methods instead. The number 100 is an approximate amount of method calls that take part in the same section of the main game loop. therefore all method calls for each of the tests are all part of that amount.

5.2 Implemented test analysis

By evaluating the implementation of automated tests which are shown in the result (section 4.3.1, 4.3.2, 4.3.3) it is possible to conclude that testing is something important to have in mind from an early stage of game development and through the whole process. With an existing implementation of Quake Console which already was a feature in the game, it became a granted tool and entrance towards successfully implementing the tests. To feed the values into something that mimics a command prompt and where mocking (Mocking is a technique to create fake object and/or functions that imitate the excepted behaviour of specific parts of the application) of dependencies is not possible, isn’t an optimal environment for automated tests. The approach with feeding values into Quake Console constrained us towards implementing only three features within the client-server network.

However, the three automated tests implemented are solving their intent fully and also pinpointing the user towards a more accurate and precise area where the problem lies. While evaluating the success criteria by the metrics provided earlier it is clear that the tests are proven as successful, good tests.
6 Discussion

In this chapter we will discuss issues we have encountered and how we solved some of them, as well as reflection regarding our method of choice.

6.1 Reflection

Reflecting upon our thesis project and the results gained as a whole we consider this study to be very successful. Our goals of implementing automated tests together with pinpointing faults were fulfilled. On the other hand our tests were only few and similar to each other.

With that said, we are uncertain whether our method and results would be useful in the industry.

6.2 Result discussion

Our diagram has some peculiar values that are very alike from test to test, namely the total amount stack. This stack contains the combined amount of an approximate number of method available for execution in the common part of Quake 3 during the game loop. As such the values are the same for each of the test since they all run the same initial path towards the particular execution of predefined methods. The same can be said for the automated test stack of the graph. Since all our tests root in the same functionality surrounding the network connectivity portion of client-server, we placed the simulated fault that the tests find in single spot where they all arrive at once in their method cycle. The reason as to why the third stack of each test differs from the automated stack is because of an error in our comprehension of the stack trace in accordance to the game loop. For each different action the test takes, e.g. connection and disconnect, the game loop runs an additional time. The stack traces are only recorded for one of these loops and such the difference. This was of course a very major reveal at the time we found out, however we still managed to get the tests to actually stop at these simulated faults. For a more technical explanation of this problem, see 6.2.5.

Regarding our choice of visual representation of tests, there was always the possibility to create a more appealing presentation, by using a windows project with DirectX or OpenGL. However we stuck with a console window as we think that it was the more practical choice time wise. In essence our results points towards the possibility of implementing automated testing in an existing client-server game. However we encountered many issues in doing so and time was not forgiving. We believe that the best practice possible towards testability in games is to include the design of tests as early as possible in the process of creating a game.

6.3 Problem discussion

In this section we will present a discussion regarding each of the problems that we encountered in detail.

6.3.1 Pre-compile issues

We have encountered many various problems along the process of achieving this result. First of all we had to get an integrated development environment up and running that actu-
ally compiled the source code. Which lead to the need of downloading and applying SDL (Simple DirectMedia Layer) to our solution. SDL is designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D, which Quake 3 uses for their input handling.

6.3.2 Source code issues

After the source compiled and the game ran properly we started creating our test environment. The first problem encountered when trying to apply our tests to the game were whether or not to run the entire solution or just parts of it. It proved not possible to constrain the components needed in the amount of time we had. One major factor as well was the complex dependencies across the code. We did however manage to remove some components that was not need in order to run our tests (see 3.4.3). We sought to have our test as the start-up project in Visual Studio, so we compiled the Quake 3 project as a static library which we directly linked to our test project. By doing this we could access the functions needed to run Quake 3 under our demands.

6.3.3 Test issues

One of the major problems we encountered was the separation of client and server for our tests. We tried to start both client and server simultaneously to no avail. Forcing either of them to start while the other was running would cause both portions to become unresponsive. We solved this problem by running the minimal amount of components needed to assure the connection between server and client for our tests (see 3.4.3). As such there is no way to actively perform actions inside the game since the client is not able to relay inputs. Another problem was that we could not disconnect at first after using the connect command. It turns out that the server needs to set a map before the players are actually given the state of CONNECTED. This was a major headache that turned out to have a simple solution.

Later on we also discovered that the server did not recognize itself as dedicated even though we stated that when starting it. Which lead to us being unable to kick ourselves from the server as part of our test. It turns out that the server which thought it was hosted by a player (the client portion of the game), was not allowed to kick its host player, what a shocker. Since we knew that the server was indeed dedicated that solution was to let it kick its imaginary host player and so our test worked perfectly. Another curious thing about this issue was that when the client disconnected, it would bring the server down with it. The same logic was also applied to this case of the code. This time we had to remove the portion of the code which shutdown the server when the disconnect command was executed.

6.3.4 Stacktrace issues

Regarding the stack trace analysis produced by the tests, the amount of methods presented was solely based on our simulated faults. There is also the issue with the game loop of the Quake code that for each iteration of the loop a new AppConfig structure is assigned. Our tests present the stack trace at the end of all those iterations, to be entirely sure that the state is set. The problem with this is that the presented amount of method calls is limited to only one iteration. If time had been more forgiving, a more thorough analysis of the method calls would likely to have been made. We also tried to apply this same
stack trace analysis to fatal errors, that when occur would cause the application to shut down completely. We were however unable to prevent the shutdown of the application.

6.4 Method reflection

In our search for other papers resolving the same problem we are trying to answer, we did not find any method of which to approach the problem. This led us to forging our own strategy based on evaluating what we needed to do and if there was any techniques and tools already developed for it.

Our approach in the method feels both prevailing and well structured. What is questionable is if our approach is sustainable thought the whole process of implementing further tests. We have been forced to choose only three tests to implement and these tests where very similar to one another which resulted in a very small and limited area in types of tests. Another critical issue is that we never had the time and the opportunity to implement the same method in any other existing game which effects the validity. It is also easy to conclude that our approach is heavily dependent on some kind of command prompt which can be used to feed input and execute commands, something that not every game has. However, the usage of Quake 3 as our testbed brings forth a plausible usage of our method in games that have been developed from the Quake 3 engine (See 2.5).

Our chosen problem area was explored and evaluated in a laboratory environment during the implementation process, were we as developer where free to modify and manipulate both the game and tests towards our desire. This might have resulted in a reduction of validity and reliability as this wouldn’t often be the case in the industry. It is also important to have in mind that we had many different roles, such as developers, innovators and testers which may have affected the end result. We want to underline that although we actually succeeded with implementing automated tests and also to a large extent pinpointing faults it does not naturally mean it will work in a real world scenario. As the method is not tested enough in an industrial environment, it creates a significant validation problem.

Last but not least our approach on the method was to a large part forged by ourselves. It particular isn’t widely tested and therefore may give us a wrong result or further more isn’t appropriate to our scientific question.
7 Conclusion

This chapter will present a final conclusion regarding the thesis as a whole followed by a short discussion regarding possible future research based on this thesis.

7.1 Conclusions

Despite all the issues we have had, the scarce amount of tests we actually implemented and the validity of our method, we did manage to implement automated tests in an existing client-server game. We also did manage to perform down to method accuracy of simulated fault localization. We believe that our method needs more testing to prove even higher validity, however we believe that the result can be of assistance for other developers as they take major decision regarding the testability of their games and applications in general.

7.2 Further research

As stated in the method reflection, we have utilized very specific features of this client-server game in order to maintain our test environment. In order to reach a more general result in the future, we believe that games without these specific features and more client-server based games in general will have to apply this theory and method to establish a baseline.

Another interesting limitation that we worked with was the use of Windows as our platform of choice, where perhaps Linux would have been a better choice. If similar studies would emerge and thus increase the attention towards this problem, we believe that this could lead to presenting and standardized way to implement automated testing in not only client-server games, but games in general.
References


