Testing and Maintenance in the Video Game Industry Today

Anthony Jarman

A Senior Thesis submitted in partial fulfillment
of the requirements for graduation
in the Honors Program
Liberty University
Spring 2010

Acceptance of Senior Honors Thesis

This Senior Honors Thesis is accepted in partial
fulfillment of the requirements for graduation from the
Honors Program of Liberty University.

____
Robert Tucker, Ph.D.
Thesis Chair

____
Mark Shaneck, Ph.D.
Committee Member

____
Troy Matthews, Ed.D.
Committee Member

____
Marilyn Gadomski, Ph.D.
Assistant Honors Director

____
Date

Abstract

Testing and maintenance are important when designing any type of software, especially video games. Since the gaming industry began, testing and maintenance techniques have evolved and changed. In order to understand how testing and maintenance techniques are practiced in the gaming industry, several key elements must be examined. First, specific testing and maintenance techniques that are most useful for video games must be analyzed to understand their effectiveness. Second, the processes used for testing and maintaining video games at the beginning of the industry must be reviewed in order to see how far testing and maintenance techniques have progressed. Third, the potential negative side effects of new testing and maintenance techniques need to be evaluated to serve as both a warning for future game developers and a way of improving the overall quality of current video games.

Testing and Maintenance in the Video Game Industry Today

Computers are used in almost every field imaginable today. Whether working at the drive-thru for a local fast food chain or holding the position of CEO for a multimillion dollar corporation, many people will most likely use some type of computer to perform or help to perform their jobs. Computers, however, are useless without some sort of instructions to tell the computer what tasks to perform. The instructions are delivered to the computer hardware by special programs referred to as software. The computer industry is therefore dependent on the development of good software and good software is developed by following a software development cycle (Zeitler, 1991). While several different models for software development cycles exist, all the models have a few key stages in common. Every model needs a stage for planning out the software project, then a stage to implement the software via writing code, and finally a stage to test and maintain the new software after its initial release.

Typically, the types of programs that come to the average user's mind when mentioning software are programs for one's personal computer (PC). Programs like Microsoft Word or McAfee AntiVirus Suite are a few of the many examples of software designed for the home computer. However, software is not limited to the realm of task oriented programs for a PC. Software also extends to the realm of entertainment, specifically in the field of video game development.

Unlike PCs, home gaming consoles have only recently implemented the ability to connect to the internet. Therefore, home consoles, the first of which was released in 1972 (Classic Gaming Museum), have spent a large majority of their existence with little to no way of performing typical software maintenance, with patches distributed via networks. Unlike computer software, video games could not release patches and new versions of the game for most of the industry's lifespan, making the process of software testing and maintenance in the gaming industry an interesting and relatively unexplored topic. Game developers had to use what little maintenance methods they had at their disposal, such as sequels that contained many corrections and gameplay additions that should have been included in the original game. Testing has occurred since video games were first made, but true maintenance in the gaming industry is still a new, constantly evolving process. To understand testing and maintenance in the video game industry completely, one must examine current testing and maintenance procedures, how these testing and maintenance practices have evolved since the gaming industry began, and how their evolution has negatively impacted video game design today.

Current Testing Phase

Testing in General

Testing is a necessary step in any software development cycle. To understand how testing is done in the video game industry, one must examine what software testing is and how testing is done in general, then look into how testing video games is best accomplished.

Testing can be defined as "evaluating software by observing its execution" (Amman & Offutt, 2008, p. 13). Large software projects contain highly complex source code that can often give unpredictable results when the code is finally executed. Testing helps developers see if their code works properly and correct any errors that may occur before the product is put into implementation. Testing, however, does not simply involve running an executable file and performing a few random tests to determine the output. A developer must carefully plan out the testing phase to ensure that the maximum number of glitches and potential risks are accounted for and corrected with the tests.

In order to complete a testing phase successfully, a software engineer uses a series of test cases which are fulfilled via the various tests performed on the software. According to Jorgensen (2008), a test case consists of two types of inputs and two types of outputs. The two types of inputs for a test case are preconditions and actual inputs. Preconditions are the conditions that a particular piece of software must meet in order for a test case to occur. If the preconditions are all met, then the actual inputs for the test case are put into the software. The program continues execution with these inputs until the test case is finished and actual outputs are delivered to the tester. The tester will compare these outputs to the expected output for the test case and determine if the postconditions, the conditions that the software must be in after a test case is finished, are correct as well. If both the actual outputs and postconditions match the expected results of the test case, then the software has successfully passed the current test. Jorgensen (2008) noted that the "essence of software testing is to determine a set of test cases for the item to be tested." Test cases are thus vital to testing any piece of software, forcing a software developer to decide how to determine test cases that are appropriate for his or her piece of software.

Functional Testing

Two main methods exist to determine the test cases for a particular project: functional testing and structural testing. With functional testing, the tester cannot see how a particular piece of software is implemented. He or she only knows which inputs map to which outputs. Functional testing is also known as black box testing, which comes from the fact that the tester cannot see the specific implementation of the software. This method of testing is based on the idea that any piece of software can be considered a function that maps input values in the function's domain to output values in the function's range. When a designer is using functional testing to create test cases, the task becomes human centered. The designer must examine the requirements of a program, what tasks the software should perform and when the software should perform these tasks, and develop a system of tests that encompass all the requirements of the software (Jorgensen, 2008). Pezze and Young (2007, p. 162) stated that the "core of functional test case design is partitioning the possible behaviors of the program into a finite number of homogeneous classes," where each of these classes can ultimately be determined to be correct or incorrect.

Functional testing has several strengths and weaknesses. One of the biggest strengths of functional testing is the fact that since the test case exists independently of the implementation of the piece of code being tested, changes and further development of the code can take place along side the execution of the test case. As long as the inputs and outputs remain the same, the implementation can freely change and the test case is still valid (Jorgensen, 2008). Such a strength is particularly helpful in the video game industry. Today, many games are made using the programming language C++ (Harbour, 2007). C++ is an object-oriented language, meaning that coding takes place using various objects which can have unique variables and operations that can be performed on each specific object. With functional testing, test cases can be centered on the various classes in a piece of software, which can very helpful when testing a video game. For example, in a game involving a third-person shooter, the player-controlled character may be a class, the various guns the player use may be separate classes or multiple instances of a single class for all guns in the game, and the enemies may all be separate classes. With functional testing, a designer can write test cases that revolve around how these various classes interact with each other during the game. One test case may involve testing if the player's current weapon injures enemy type A on every area of the enemy. A tester would be told that when he or she is aiming at this enemy and presses the button to shoot, the enemy should take damage or react to being shot. Since the tester is only aware of what the expected enemy output is supposed to be for this particular test case, if any errors are found, a programmer can make adjustments to the code and still keep the test case the same as long as any of the inputs or outputs do not change. Functional testing can thus provide better time management by allowing testers and programmers to work together simultaneously without having to alter test cases (Tamres, 2002).

However, functional testing is not perfect. This is especially true when designing video games. Using C++, video games can be viewed as a collection of objects interacting with one another via player commands in a confined space. Developing test cases for a video game with functional testing then becomes a matter of determining a finite number of ways that objects can interact with each other. The problem with having to go through all possible interactions of objects in any software project is the large number of interactions that are possible. This is especially true since most major video games today are large in scale. Missing a test case or having several test cases overlap are easy mistakes for a designer to make when creating test cases with functional testing. For example, a game designer making a third-person shooter may remember to check whether the player can shoot, if those bullets affect the enemies, and if the enemy's bullets affect the player. He or she may make each of these a test case. In this example, the test case to see if the player can shoot is redundant since the test case to see if player bullets affect the enemies already proves that the player can shoot. Furthermore, the designer failed to make a test case to ensure that player and enemies interacted with the ground or if enemies could react to each other. Functional testing is still needed since the developed test cases are easily proved correct or incorrect, and the overall efficiency of a project can increase since programmers and testers can work together simultaneously.

Functional testing is used in many different areas of the gaming industry, but due to its often high rate of overlapped test cases, the second main method of testing is always necessary. Before examining specific examples of functional testing in the gaming industry, this second method, referred to as structural testing, must be discussed (Jorgensen, 2008).

Structural Testing

Structural testing is often referred to as white box testing. In structural testing, test cases are developed based upon how the code in a program is written. Unlike functional testing, where the implementation of the program is like a black box, structural testing makes the tester aware of how the code is written and the test is thus more like a white box because implementation is known in addition to appropriate mappings of inputs to outputs (Tamres, 2002). One of the biggest strengths structural testing has over functional testing lies in the area of test coverage metrics, which is the ability to measure how much of a specific program is tested. Using structural testing, a designer can look at the source code of a program and write a test case that tests a specific function of the program or maybe a small module of code. For example, a two dimensional platforming game, something similar to Super Mario or Sonic, may contain a function that causes the player controlled character to jump. This function may perform multiple tasks, such as controlling how high the character jumps, making sure the character does not fall through the ground, checking if the character made any collisions with other sprites on the screen, or a myriad of other tasks. Using functional testing, multiple tests would be required to test the jumping function. One test may check if the player can jump when a button is pressed, another may be used to see if collisions between the player and enemies are detected while the player is jumping, and a third test may ensure that the player does not fall through various areas of the ground. These tests would not be able to be combined since the tester is unaware of how jumping is implemented in the game. If a jump function caused the player to jump and collision detection was handled outside the jump function, testing for collision outside of the jumping test would be necessary. In short, the nature of the jump function determines how many test cases are necessary. Using structural testing, the function’s design is known when writing test cases, so a test case can be written with the implementation in mind. After the test cases are finished, the design team can also determine the total test coverage based upon what percentage of code has been tested in the various cases. Structural testing can allow a design team to develop specific test cases with little to no redundancies and be able to specifically measure how much code was tested (Jorgensen, 2008).

Like functional testing, however, structural testing is not perfect. First, changing the structure of a piece of code related to a test case may, unfortunately, change the test case, making programming and testing difficult to run simultaneously. Second, and much more prone to error, structural testing only verifies that the code works the way it was written, taking in little concern for logical correctness. Due to the complexity of most major software projects today, small mistakes that have large consequences are easy to overlook during testing. Sometimes, a mistake may only occur under specific conditions that only happen a few times over the entire project's lifecycle. Only one user out of millions may be affected by such an error, but that error could have serious consequences. These types of errors are found throughout the software industry in general, and are not limited to video games (Tamres, 2002). For example, Marcus Fisher (2007) told the story of a professor who was in charge of a program designed to trigger rockets to carry a launch package into low Earth orbit to assemble a spacecraft when the rockets were 100,000 feet in the air. During the run, however, the altitude reading eventually stopped at the value of 95,990 feet. The mission was a failure and almost ended tragically. The issue turned out to be a switch between an assignment operator and an equality operator in one of the source code's conditional statements. This example demonstrates the fact that one wrong symbol or misplaced semi-colon can cause a program to seem as if it is running smoothly when in reality, a hidden bug exists in the code. When such an error is made in a video game, the consequences are less costly and serious than in Fisher's example, but are still deadly to the overall quality of the game. A transposed equality operator may cause an infinite stream of enemies to flood the screen constantly or a misplaced semicolon may result in random game crashes. Structural testing may have difficulty discovering such a bug since only the implementation of the code is tested, and not the code's logic. If the bug does not affect the function's intended purpose and instead omits a behavior that the program should have contained, a structural test case will fail to discover the bug. Structural test cases also fail to detect if a certain requirement or behavior of the program has not been implemented. Since structural test cases are written by examining the source code, if a programmer forgets to implement an ability to allow a character to jump or fire a weapon, this missing ability will not be detected via structural testing. Structural test cases can only test what has been directly written in the source code (Jorgensen, 2008).

Determining Which Testing Method to Use for Video Games

Since both functional testing and structural testing each have different strengths and weaknesses, a design team needs to decide which method of testing is best suited for their current project. Most experts will agree that the most effective testing method to use, regardless of the project one is working on, is a combination of both functional and structural testing (Pezze & Young, 2007). Functional testing allows a developer to determine if all the project requirements have been met while structural testing allows the developer to ensure that the software does not implement any unintended behaviors. Test cases can be written with fewer redundancies while still covering the full spectrum of the project's requirements (Jorgensen, 2008).

In the area of video game design, the most useful mix of these two testing methods appears to be a heavier focus on functional testing supplemented with structural testing to ensure code specific bugs do not occur (Williams, 2006). For example, examining a third-person shooter, functional test cases would involve testing to make sure the player can move, shoot, damage enemies, and any other activities the developers deem necessary for their game. One requirement for the game may be that a maximum of 20 enemies need to be able to appear on the screen at once. A functional test could be used to ensure that 20 enemies can be present on the screen at one time. However, if a 21st enemy was placed on the screen, functional testing would not deal with this issue without serious implications. Assume the game is written in C++ and a class called "Enemy1" exists containing all the functions and variables required to manipulate and draw that enemy type on the screen. The programmer would most likely create an array of type Enemy1, referred to as enemy1Array. Since the requirement for the game is for a maximum of 20 enemies to appear on the screen, the programmer may set the size of enemy1Array to 20. C++ contains no built in safety protocols to determine if an array is out of bounds. The programmer is in charge of ensuring that array bounds are tracked and used correctly in the code. A second programmer may fail to realize that enemy1Array is of maximum size 20 and attempt to add a 21st enemy to the screen, but only under certain circumstances, such as a group of enemies being eliminated under a certain amount of time. Attempting to access an array value outside the array's bounds can easily cause a fatal error in a C++ program, causing the game to crash or, in much worse cases, allowing the game to continue running while unintentionally altering some other data structure's value, resulting in logic errors and glitches that are difficult to pinpoint. Functional testing could not determine this error exists unless the tester accidentally met the requirements to allow the 21st enemy to be placed on the screen. Structural testing, however, would allow a test case to be developed that specifically tests the second programmer's function that makes the illegal array access. The bug would be caught and eliminated (Cole, 2000). Large games benefit from a functional-structural combination approach to testing more so than smaller games with smaller development teams due to the fact that multiple programmers will be using numerous functions developed by different people. Mistakes like the example described above are more likely to occur with a larger development team since larger teams involve more people, working on bigger projects, with more functions being written by different people. Bigger projects have a higher amount of potential mistakes because a larger number of processes are being performed by the project as opposed to the amount of processes performed by a smaller project (Williams, 2006).

Real-World Example: Beta Tests

Functional-Structural Combination. Many game developers use a functional-structural testing combination in today's high budget, mainstream games. One of the most common and effective testing methods used by game developers today are beta tests. Beta testing is a form of acceptance testing, which is the "process of comparing the end product to the current needs of its end users" (Kit, 1995, p. 103). Beta tests are generally performed either by the public, in the case of an open beta, or a select group of people, who are usually the customers or some other outside group. Beta testing is especially effective for video games today because they encompass all forms of testing most useful for video games in one, big, often final, test.

The recent Star Trek Online beta is a good example of how a beta encompasses all forms of necessary testing. Star Trek Online is a massive multiplayer online game, first released on February 2, 2010. The game ran a beta test from January 8 to January 27 for select, interested customers. Craig Zinkievich (2010), the game's executive producer, discussed his feelings on the outcome of the beta on the game's official website. Zinkievich discussed how the team worked to fix last minute bugs and will continue to remain hard at work to fix bugs that appear after launch. The beta revealed numerous bugs that were able to be fixed before the launch of the game and gave feedback to the developers as to how much enjoyment their customers got from the game. The beta test also provided the development team with a way to test both functional and structural test cases. Functional testing was performed as the customers played the beta. The customers completed tasks that are performed in the actual game which gave the developers a chance to make sure that all the requirements for their game held up in the real world. At the same time, the developers could examine the code that was running and make sure certain parts of the code worked in a real-world environment, such as the network code for connecting users from all across the world with minimal lag time.

Usability Testing. In addition to traditional functional and structural testing, the beta also provides a form of testing called usability testing, which is testing a piece of software to ensure that a customer is able to use and enjoy the software. The participants of the Star Trek Online beta test left feedback for the design team about what they liked about the game and what they did not like about the game. The developers implemented some changes before launch but, due to time constraints, had to wait until after launch to add all the features they wanted. In addition to beta tests, internal testers are used to ensure that all test cases are successfully performed prior to any demos or betas that may be released to the public. Beta tests are useful in determining how the general public views the game and how well the game holds up in a real-world environment. While beta tests are far from the only test method game developers employ, they are certainly one of the most useful due to their combination of being able to test functional and structural test cases simultaneously in addition to providing excellent opportunities for usability testing (Kit, 1995).

Current Maintenance Phase

Maintenance In General

After the testing phase is completed, the software product is released and the longest phase in the software development cycle begins: the maintenance phase. As with testing, in order to understand how maintenance is used in the video game industry, one must first examine how maintenance is used in the computer software industry in general. Maintenance in the software development industry can be defined as “all the actions that are needed to keep software in such a running order that it achieves all its objectives from the beginning until the end of the usage” (Vehvilainen, 2000, p. 1). The maintenance phase is an important part of the software development cycle. On average, two thirds of a product’s total cost is spent on maintenance (Lientz, Swanson & Tompkins, 1978). Maintenance can be a tiresome, thankless job, as the maintenance programmers are left to deal with all the bugs and other issues that arise after a piece of software is released. Most of the time, the rest of the development staff moves on to their next project and leaves the maintenance team alone to deal with all the issues that arise. A maintenance programmer sometimes has very little documentation to work with since by the time a software project is released, all the design specification and requirements documents are either gone or misplaced, and the only hints as to how a program works are the comments in the code itself, which can often be of little to no value if the original programmer did not leave thorough comments in his or her code (Schach, 1996). Maintenance can be a tedious, difficult, and extremely time consuming activity, yet many reasons exist to perform maintenance on a finished piece of software.

Three Forms of Maintenance

Three of the most common forms of maintenance are corrective maintenance, perfective maintenance, and adaptive maintenance (Cote & St-Pierre, 1990). In general, most maintenance is perfective, taking up approximately 60% of all maintenance time while adaptive and corrective maintenance take up about 18% each. The remaining 4% is spent on various forms of other maintenance. Corrective maintenance is the type of maintenance that most people associate with the entire maintenance process even though, in reality, it takes up a surprisingly small amount of overall maintenance resources. Corrective maintenance generally takes place directly after release. It is the process of removing any bugs and errors from the code that slipped through the testing phase. Perfective maintenance is the process of improving and adding new features to a program over the program’s lifecycle. Adaptive maintenance involves updating a product to incorporate changes made to the environment where the product operates. For example, if a software developer has to change the program’s code due to a new service pack being released for Windows, he or she would be performing adaptive maintenance. These three types of maintenance are the most common maintenance practices that occur in the software industry today and each type plays an important role in the video game industry (Schach, 1996).

Information Hiding and Encapsulation

Depending on the piece of software, maintenance can often be a difficult task. Some programming languages do not naturally lend themselves to being easily maintained. Object-oriented languages, however, allow code to be easily maintained due to information hiding and encapsulation (Schach, 1996). Information hiding is a technique used in object-oriented programming languages that allow the implementation of a certain class or function to be hidden from the rest of the program on a syntactic level. The compiler still knows how the code is executed, but the developer of the class or function can choose to place the actual implementation of said class or function in a completely different location. Encapsulation is the means by which information hiding takes place (Perry & Kaizer, 1990). Encapsulation and information hiding aid in maintenance in two ways. First, encapsulation allows a program written in an object-oriented language to be designed in bits and pieces of code that are used by a main program to create the actual application. Separating the code into individual pieces allows a maintenance programmer to easily determine where in the code an error is occurring or what part of the code to edit in order to add new features to the program or, in other words, to perform corrective and perfective maintenance (Schach, 1996). For example, a program that inputs a user's personal information and uploads it to a database may have a problem where the user's name field gets corrupted for certain users who are using an older version of their operating system. To fix the issue, the maintenance programmer can find the function that is in charge of sending the information to the database and edit the function so it supports the older version of the operating system. Information hiding ensures that the change to this function will not affect any other code in the program that may be using that function. Since most video games today are made using C++ or similar object-oriented languages, maintenance programming on video games can be much easier than other software projects on the market (Rahardja, 1994, p. 999).

Networks and Gaming Maintenance

Adaptive Maintenance. Maintenance in the video game industry is a relatively new subject as the use of networks in video game consoles has only recently made maintenance a widespread activity in the gaming industry today. Four major platforms exist today to deliver a video game to the user's home: Microsoft's Xbox360, Sony's PlayStation3, Nintendo's Wii, and the user's personal computer. All four of these platforms have the ability to connect to the internet. Developers can use this ability to distribute patches or extra content for games they have already released. Gaming maintenance differs from maintenance on other pieces of software in regards to adaptive maintenance. Network connections can be used to update the firmware installed on home consoles, and operating systems on PCs are constantly changing, but for the most part, these changes cause few problems for video games, especially on the home consoles.

Some adaptive maintenance is still used in the gaming industry but, for the most part, maintenance programming performed on video games is either corrective or perfective in nature. Corrective maintenance mainly occurs through patches or updates to games while perfective maintenance mainly occurs through downloadable content.

Corrective Maintenance. Corrective maintenance in video games is completed similarly to any general piece of software. When a game is first released, many bugs and glitches are likely to be present. A modern game is generally a large, complicated piece of software. Even the most rigorous of testing techniques will fail to eliminate all the bugs present in the game. Thanks to network connections in all modern home consoles, game developers can release a patch for their game after release. PC games and other computer software projects have been able to perform corrective maintenance for a much longer time than the video game industry due to the fact that personal computers have had internet access, and thus the ability to distribute corrective patches via a network, for the past several years. Home gaming consoles have only recently gained widespread internet access and have wasted no time using this ability, as many new releases get some form of corrective patch near launch.

Corrective patches are also used after perfective maintenance is performed. Adding new features to a video game involves modifying and adding new pieces of code. New code always makes room for new programming errors which lead to corrective maintenance. Corrective maintenance can also lead to more corrective maintenance. For example, several online multiplayer game developers will often encounter the issue of players exploiting one or more features placed in their game by either using the feature to glitch the game or discovering that the feature was unbalanced. Jumping into a certain area of a wall, for example, may cause a variable to be reset and warp the character into a position it was never intended to be able to access, or the network code may cause the network to crash when too many people are playing on the network at the same time (Moore & Novak, 2009). The maintenance programmers will fix whatever issue arises, but correcting code involves changing code which can often lead to more errors. A developer may release a patch to fix the jumping glitch described above, but in doing so may produce a completely different glitch. This type of fault is called a regression fault. In such a situation, more corrective maintenance must occur until all the glitches can be taken care of. Programming games with object-oriented languages certainly makes corrective maintenance easier, but it is still a difficult task that can often lead to a frustrating cycle of fixing errors and creating regression faults in the process (Schach, 1996).

Perfective Maintenance. While corrective maintenance is a large part of maintenance in the video game industry, the most common type of maintenance used on video games is perfective maintenance. The Institute of Electrical and Electronic Engineers (IEEE) defined perfective maintenance as “modification of a software product after delivery to improve performance or maintainability” (IEEE, 1998, p. 5). Perfective maintenance involves a wide range of activities from improving the efficiency of a program to adding new features to that program. In the video game industry today, the most common use of perfective maintenance lies in the area of downloadable content. The concept of downloadable content is mostly isolated to the video game industry, making perfective maintenance in video games more unique than testing or other forms of maintenance, which are performed similarly to any other piece of software. Developers will often release additional content for their game a few months after the game is released. The tactic is used to make sure interest in the game stays high and customers remain happy. The downloadable content is made available for users via the network connections of whatever console the game was released on. Sometimes this content is free, and other times it is so substantial that users are required to purchase the content. One of the best examples of perfective maintenance is a game developed by Criterion Games called “Burnout Paradise.” In the first two years of the game’s release, Criterion released ten different updates and add-on packages for this game. These updates often fixed some bugs in the code, but they always included something extra like new cars or tracks on which to race (Burnout Paradise, 2010). Some of this content was free and some required the user to pay additional money for the content. By charging users for additional content for their game, developers can make more money off the game they already released. This tactic not only provides developers with more funds to make more games, but also makes the maintenance process a little less tedious and thankless. Combining corrective and perfective maintenance by fixing coding errors in the same update that adds new content to a game is a great tactic as well since it limits the number of downloads a user has to make.

Without networks, perfective maintenance in the gaming industry would be much more difficult, with one of the only feasible solutions being the release of sequels, which are, in some cases, barely considered a form of perfective maintenance. Networks have truly revolutionized the way maintenance is handled in the video game industry. Bugs and glitches can be patched, additional profit can be made on a game long after its initial release thanks to downloadable content, and user satisfaction levels can rise due to the ability to update and add new features to a game at no additional cost to the user (Burg, 2008).

Effect on Maintenance Programmers. Downloadable content not only has an effect on the user of a software program, but also on the maintenance programmers. As discussed earlier, maintenance programming can be an unappreciated job. In the earlier days of the computing industry, a maintenance programmer could work for months on a new feature for a program, release the feature, and get no thanks or recognition because the feature was simply included in a new patch with some other minor fixes to bugs or other glitches. The programmer got no additional profit or recognition, which could definitely lower the programmer's motivation. Motivation was especially important for a maintenance programmer since their job was difficult and often looked down upon (Landsbaum, 1992).

Even today, some companies continue to look down on maintenance programmers, though most companies value them, which downloadable content helps to enforce. Downloadable content offers a developer a chance to provide significant, additional content to a game and make profit at the same time. The maintenance programmer in charge of such a task is given recognition since paid content cannot be included in a patch that fixes some errors most users never even knew existed. The user is guaranteed to see the additions to the game since they are paying for the content.

Maintenance programmers need motivation, and downloadable content is a good way of providing such motivation.

Testing Techniques in the Early Gaming Industry

Why Poor Testing Practices Were Used

While the majority of the video game industry today practices testing and maintenance techniques that are considered by software engineering experts to be good practices, when the gaming industry began, most of these techniques were not used. Companies often viewed video games as children's toys. Naturally, a company is not going to place as much quality assurance resources into a product designed to entertain children as they would into an operating system or anti-virus program. In the beginning of the video game industry, the amount of testing that was performed was significantly less than what is performed today, in part due to the fact that more testing is required today due to most games being much larger, more expensive projects, but also because of a low return of investment. A return of investment is a "comparison of the value of doing something versus the cost of doing something" (Everett & McLeod, 2007). A majority of all business-like decisions are made based upon the expected return of investment. If fixing a defect in a product will cost more than what is gained by fixing the defect, the defect will not be repaired. Most game publishers saw thorough testing as having a very small return of investment for two main reasons.

First, most games coming out at the start of the gaming industry were filled with glitches. Most of the hardware for a gaming console was new and not very powerful. Very few people had any experience programming for the particular platform and those that did often found themselves held back by hardware limitations. Bugs would often be present in the final release of a game, not because testing procedures failed to discover the bug, but because the glitch did not break the game or affect the overall enjoyment of the game enough to warrant an attempt to discover how to fix the error.

Second, most children are not going to care about small glitches in their games. If a character can inadvertently pass through a certain wall or one of his or her legs occasionally disappears for a brief moment, the child will most likely not be bothered and continue playing the game. Fewer games focused on immersing the player in the game's world with realistic, believable graphics and set pieces. Most were designed to entertain children, who most likely did not worry about being immersed in a game. Developers could get away with having several minor errors in their games because their intended user base simply ignored most of these errors. Today, many games are intended for an older audience who are not nearly as patient with small glitches as children are (Industry facts, 2008). Testing techniques were not necessarily substandard in the early stages of the video game industry, just underdeveloped and overlooked (Pezze & Young, 2007).

Proficient Early Testing Techniques

Not all testing practices in the early days of the gaming industry were overlooked, however. In fact, some good testing practices were useful even when developing games designed as children's toys. One such tactic is designing test cases that are reusable for future projects. In its simplest form, the video game industry is a business. Businesses exist to maximize profit, a task that is easier if fewer resources are used to make a product. If a developer can design test cases for a game that can be used for future iterations and sequels of that game, less time can be spent developing test cases for said iterations and sequels. Sequels were, and still are, popular in the video game industry, so any reusable test cases gave developers more time to design their game rather than develop test cases. Also, if the test cases were well designed, future sequels to a game could be almost guaranteed to be just as error free as the original game (Dustin, 2002).

Maintenance Techniques in the Early Gaming Industry

Maintenance without Networks

While testing has improved in the gaming industry since its inception, maintenance has improved at a much faster pace. In the video game industry, maintenance has been virtually impossible until quite recently when game consoles became ubiquitously connected to the internet. PC games had the ability to perform extensive maintenance long before console games, but a majority of game industry profit comes from the home console market so a large amount of game developers have had a difficult time maintaining their games after the initial release. Before the use of networks in the gaming industry, the options available for maintaining a game were very slim. Two of the major ways that developers performed maintenance on their games prior to the widespread use of networks in gaming consoles was via sequels and re-releases of a game (Moore & Novak, 2009).

Maintenance via Sequels

Due to a lack of network connections on gaming consoles, corrective maintenance was difficult to perform on video games for a large part of the industry’s lifecycle. Any corrections or error fixes to a game’s code had to be performed prior to the game’s release. As usual, perfective maintenance took up almost all resources for any maintenance projects the developers planned for their game. One way to perform perfective maintenance without the use of the internet is by releasing sequels. Sequels can only loosely be considered perfective maintenance for many reasons. In most cases, a sequel is made by the same team who worked on the original game. Perfective maintenance is generally performed by a separate maintenance team. Sequels can also be so different from the original product that they are not really adding or enhancing any content in the original game. Most users actually expect sequels to implement changes to a game so radical that the sequel feels like a new, stand alone product, but at the same time feels familiar to those who played the original game. Some sequels, however, can definitely be considered perfective maintenance, especially in the beginning of the gaming industry.

One advantageous coding practice is writing code that can be reused. Most early game developers had no trouble writing easy to reuse code because it made sequels easier and faster to produce, which gave developers a faster, easier way to make profit. For example, a development team trying to produce a two dimensional platforming game may spend a large portion of their development time coding functions and classes that perform tasks that are used in any platforming game. One such function may make the player jump, one could spawn enemies, and another may draw the level on the screen. These objects help to create the game, but take up time to write and test. When the developers get ready to work on a sequel to their game, they can reuse all these functions and classes to make the sequel. The developers do not have to spend time writing the code for the objects or testing the objects, they can focus more on improving and adding new mechanics to the new game. A sequel designed using a similar process would be considered a form of perfective maintenance since the developers are essentially modifying already existing code to add new levels and features to their game (IEEE, 1998, p. 5).

Maintenance via Re-Releases

A second method of maintenance that was possible without networks was re-releasing games. In many instances, a developer may have an idea that could have been implemented in the first game but does not warrant creating an entire sequel to give users access to the mechanic. In such an instance, the developer may decide to release a special edition of the game, after the game's initial release, which includes this new idea. This process was, in many ways, a precursor to the concept of downloadable content in the gaming industry today. Developers could add new levels or characters in the re-release of their game much like developers today make these levels available for download. Unfortunately, users have to repurchase all the content they owned before as these new features were not available as standalone products. Corrective maintenance can also be performed through this method since bugs and glitches can also be repaired through a re-released game, or even through a new batch of games that are produced as the game continues to sell. Maintenance was not impossible without networks, it was just much more difficult to distribute updates and new content to all users of a particular game (Moore & Novak, 2009).

Problems with Today's Maintenance Techniques

While the use of networks in game consoles today have improved overall testing and maintenance techniques in the gaming industry, they have also led to a few problems that were not as prevalent before game consoles used networks. One of the major issues networks present to the gaming industry is the temptation for a developer to design sloppy, bug ridden code, release it to the public, and then simply patch it later. For example, if a developer is designing a game that could easily be overlooked by consumers if it is released near a more anticipated game later in the year, the developer will try to release their game at a time when no other major games are being released. The development of the game goes smoothly, but a bug is found near the end of the testing phase that causes about twenty percent of the games to crash about three fourths of the way through the game. The development team attempts to fix the problem, but realizes they will not have enough time to fix the bug and meet their release date. Before networks were used in games, the development team would most likely delay the game to fix the bug, since any user who purchased initial copies of the game containing the glitch would be forced to repurchase a different copy of the game and hope that copy did not glitch. Today, developers can merely patch the game after its release. If the situation described above happened today, the developer would probably release the game and put out a patch a few weeks later. Such a practice is dangerous since it lowers user satisfaction and trust in the developer. If a certain developer builds a reputation for releasing erroneous games at launch, several consumers may wait a month or two to purchase the game at which point they may lose interest in the game and never purchase it at all. Releasing bug filled games and patching them later makes the gaming industry seem lazy and unprofessional, two images that should be avoided by any industry attempting to make a profit in the world today. Developers need to be careful about giving in to the temptation of releasing a flawed game just to meet a release date. Shigeru Miyamoto, a game designer well known for creating games like Super Mario Bros. and the Legend of Zelda, once said that a “delayed game is eventually good, but a bad game is bad forever” (Almaci & Kemps, 2004, p.1). Delaying a game by a month and fixing any bugs makes development easier for the maintenance team and the game itself more enjoyable for the user (Harbour, 2007).

Summary

In conclusion, understanding how testing and maintenance techniques are used in the gaming industry today requires one to examine what techniques are currently being used for testing and maintaining video games and why those techniques are used, how these techniques have evolved from practices originally used in the gaming industry, and the negative side effects these techniques have on the industry as a whole.

Functional testing is the main method of testing in the video game industry today since it lends itself well to determining if all the requirements of a game have been met. Using only functional testing, however, is a poor testing practice. Structural testing is thus used in conjunction with functional testing to ensure that no unintended side effects occur as a result of logic errors in the code itself. Beta tests are a great example of how both functional and structural test cases can be used in a real world project. Beta tests give developers feedback relating to whether the game holds up in a real world setting and also whether the intended users of their game enjoy the experience. Beta tests can be used to correct coding errors and remove or add functionality to make the end user have a more enjoyable experience.

Maintenance has only recently been able to be widely implemented in the gaming industry thanks to the recent universal use of networks on the four major gaming consoles today. Maintenance, for the most part, is practiced via free patches and updates and downloadable content. Patches provide a method for developers to use corrective maintenance techniques to fix any bugs in their games after initial release or the release of a major update or expansion. Perfective maintenance is practiced via downloadable content, which allows developers to enhance their games with new levels or game mechanics and make extra profit on the game after its release. Adaptive maintenance is, for the most part, not as widely used as corrective and perfective maintenance in the gaming industry.

Due to the increased complexity of games today, testing has been given a much larger emphasis in today's gaming industry than when the industry first began. In the early years of the gaming industry, most consumers were used to glitches in their games and these glitches did not affect the overall enjoyment of the game. The return of investment for extensive testing on a video game was much lower than today. Maintenance was much more difficult to practice in the early days of the gaming industry since networks were not used on gaming consoles until recently. Maintenance was mostly practiced through sequels and re-releases of games. Many consumers would not receive any maintenance since they would often have to repurchase content they already owned.

Networks allow developers to distribute enhancements to a game to a much wider audience. Design teams need to be careful to not use networks as a way to release an unfinished game and patch it later. This practice is becoming more common in the industry today and causes consumer dissatisfaction and an unprofessional reputation. Overall, the gaming industry practices proper testing and maintenance techniques that continue to improve on a regular basis.

References

Almaci, H., & Kemps, H. (2004). Interview: Shigeru Miyamoto. The Next Level. Retrieved from http://www.the-nextlevel.com/feature/interview-shigeru-miyamoto/

Ammann, P., & Offutt, J. (2008). *Introduction to software testing*. New York: Cambridge University Press.

Burg, D. (2008). Burnout Paradise Islands DLC will be free. Joystiq. Retrieved from http://xbox.joystiq.com/2008/03/14/burnout-paradise-islands-dlc-will-be-free/

Burnout paradise is always changing. *Criterion games*. Retrieved from http://www.criteriongames.com/packs/index.php

Classic gaming museum. *GameSpy*. Retrieved from http://classicgaming.gamespy.com/View.php?view=ConsoleMuseum.Detail&id=1&game=12

Cole, O. (2000). White-box testing. *Dr. Dobbs*. Retrieved from http://www.drdobbs.com/windows/184404030

Cote, V., & St-Pierre, D. (1990). A model for estimating perfective software maintenance products. *IEEE Xplore*. Retrieved from http://ieeexplore.ieee.org.ezproxy.liberty.edu:2048/stamp/stamp.jsp?tp=&arnumber=131382&isnumber=3638

Dustin, E. (2002). *Effective software testing: 50 specific ways to improve your testing*. New York: Addison-Wesley Professional.

Everett, G., & McLeod, R. (2007). *Software testing. Testing across the entire software development life cycle*. New Jersey: Wiley & Sons, Inc.

Fisher, M. (2007). *Software verification and validation. An engineering and scientific approach*. New York: Springer Science+Business Media.

Harbour, J. (2007). *Beginning game programming (second edition)*. Boston: Thomson Course Technology PTR.

IEEE standard for software maintenance. (1998). *IEEE Xplore*. Retrieved from http://ieeexplore.ieee.org.ezproxy.liberty.edu:2048/stamp/stamp.jsp?tp=&arnumber=720567&isnumber=15562

Industry facts. (2008). *Entertainment Software Association*. Retrieved from http://www.theesa.com/facts/index.asp

Jorgensen, P. C. (2008). *Software testing: A craftsman's approach (third edition)*. Boca Raton: CRC.

Kit, E. (1995). *Software testing in the real world: Improving the process (ACM Press)*. New York: Addison-wesley Professional.

Landsbaum, J., & Glass, R. (1992). *Measuring & motivating maintenance programmers.* New Jersey: Prentice Hall.

Lientz, Swanson, & Tompkins. (1978). Characteristics of application software maintenance. *ACM.* Retrieved from http://delivery.acm.org.ezproxy.liberty.edu:2048/10.1145/360000/359522/p466-lientz.pdf?key1=359522&key2=3392905621&coll=ACM&dl=ACM&CFID=74298616&CFTOKEN=91239334

Moore, M., & Novak, J. (2009). *Game development essentials*. Florence: Delmar Cengage Learning.

Perry, D., & Kaiser, G. (1990). Adequate testing and object-oriented programming. *J. Object-Oriented Programming*. Jan./Feb. pp. 13-19.

Pezze, M., & Young, M. (2007). *Software testing and analysis: Process, principles and techniques*. New York, NY: Wiley.

Rahardja, A.(1994). Information hiding, knowledge clustering approach to software maintenance. *TENCON '94. IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology. Proceedings of 1994*, 2, 997-1001,

Schach, S. (1996). *Classical and object-oriented software engineering.* Chicago: Irwin.

Tamres, L. (2002). *Introducing software testing (ACM Press).* New York: Addison-wesley Professional.

Vehvilainen, R. (2000). What is preventive software maintenance? *IEEE Xplore*. Retrieved from http://ieeexplore.ieee.org.ezproxy.liberty.edu:2048/stamp/stamp.jsp?tp=&arnumber=882971&isnumber=19102

Williams, L. (2006). Testing overview and black-box testing techniques. North Carolina State University. Retrieved from http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf

Zeitler, D. (1991). Realistic assumptions for software reliability models. *Symp. Software Reliability Eng.,* Los Alamitos. pp. 67-74. IEEE CS Press

Zinkievich, C. (2010). State of the game: January 27th 2010. *Star Trek Online.* Retrieved from http://www.startrekonline.com/node/957