

Mark Woods

Level 3 Games development

50202088

Version 0.1

January 7, 2016

Unit 77

**Contents**

**Contents1**

Defect Types2 - 5

Function, Assignment2

Checking, Timing3

Build/Package/Merge, Algorithm4

Documentation, Interface5

Defect Triggers6 - 8

Hardware & Software configurations, Exception6

Normal, Restart7

Configuration, Startup, Stress8

Operating regions9 - 10

Pre-Game, Game start9

In-game, Post-game10

Test Phases11 – 14

Preparation11

Alpha12

Beta13

Gold, End of certification, Regression14

Testing Processes15 – 17

Black box, White box15

Smoke, Cleanroom16

Play, Ad Hoc17

Testing Life cycle18

Combinatorial Testing19

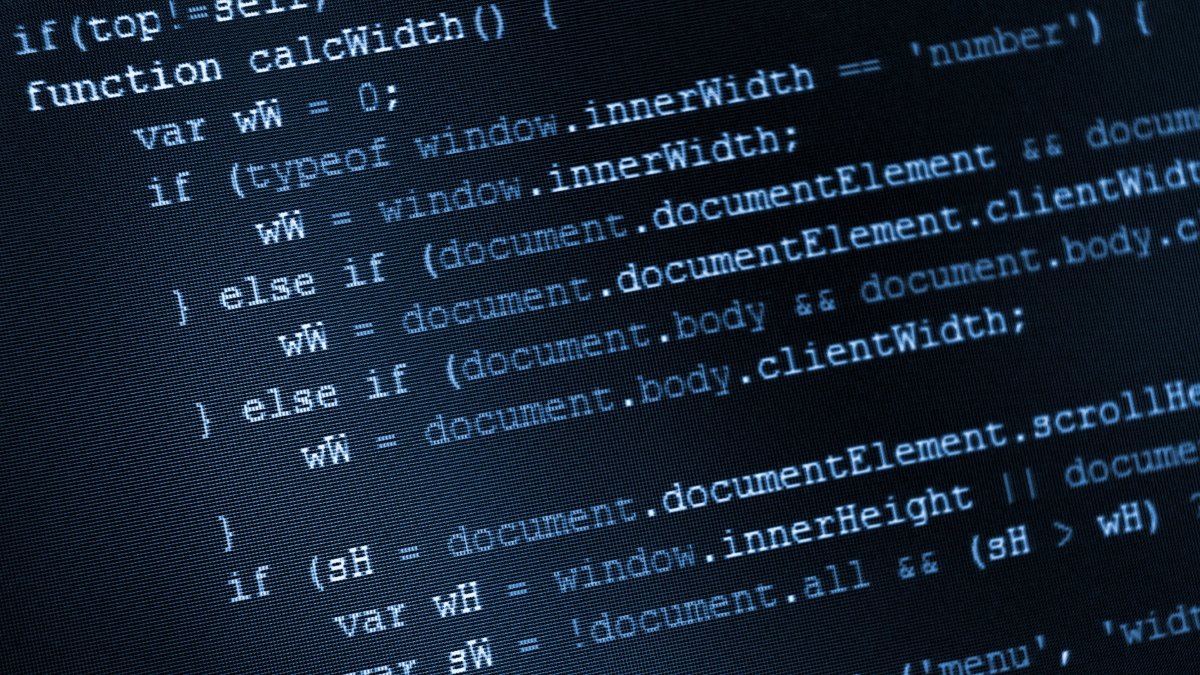
Test Flow Diagrams20 – 21

Flows, Events, Actions, States20

Test Flow Diagram example21

**Defect types**

Software can fail in a large amount of ways so defects in video games are categorized depending on how the defect is introduced, discovered or even avoided in the future. There are eight different defect types that each summarize the different software elements that go into producing game code:

**Function**

A function error is a defect that affects a games capability or the user’s experience of the game, the defect occurs when the code that provides the function is missing or incorrect. If a game has an ability or item for the player to use for example if the player had the ability to blink short distances and the player is supposed to have information that shows if the ability is available or If it’s on a cooldown, the defect occurs if the player does not get the information that is required which is usually caused by the code missing or not properly implemented. This defect type is different to an assignment defect because it is a result of usually missing code altogether whereas an assignment defect is commonly where code is there but it is somehow wrong.

**Function defect if the “else” was missing.**

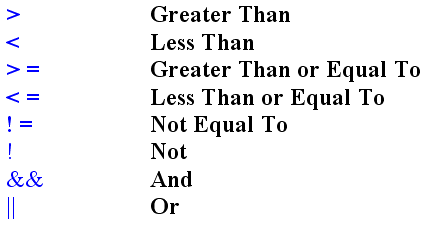
**Assignment**

An assignment defect is the result of incorrectly initializing, setting or having a missing variable in a game, this most often occurs when the game starts up because the variables are being set up during the games start up and if there is a problem with the variable the game can crash outright. In other occasions the variable can be wrong without being apparently obvious for example the player could have more lives than they are supposed to but it may not crash the game because the game dos not know the intended amount of lives. Some examples of possible assignment defects in games are:

|  |  |  |
| --- | --- | --- |
| Sports Games | RPG Games | Fighting Games |
| Initialize the score of each team | Amount of gold the player has | Health of player or AI |
| Court or pitch where the game is being played | Starting attributes of player | Map, scene of backdrop |
| Weather conditions and time of day | AI’s intended feelings towards the player e.g. friend or foe | Position of player or AI |

In most occasions assignments defects are important to avoid because they can break the game on initial startup or cause the game to be unbalanced in a way that is not intended, this is why game developers pay a lot of attention to balancing their game and its elements.

**Checking**

Checking defects occur when data is not properly validated e.g. in GML when a variable is checked in the code and one equals sign is used instead of two “=” or “==” it can mean a variable is changed instead of checked. From a player’s perspective whilst playing a game a defect like this can mean that instead of the game reducing the amount of lives that the player has when they are killed or damaged it can keep the player at fixed amount of lives because of a simple error like “lives = 2” instead of “lives == 2”. The difference between this defect and timing defects is that most checking defects occur with the misuse of operators and points like AND (&&), Less than or equal to (<=) instead of equal to (=) and Pointers (\*P).

**Timing**

Timing defects have to do with the management processes that take time to start or finish, a good example of this is where a game needs to be saved and so the player is often presented with a splash screen or a progress bar so that they know when the game is finished saving. The most common way of avoiding a timing defect is when the game starts up the player is shown a splash screen, intro or a series of publishers or developers that worked on the game, this time is used by the game to preload audio and graphics so that they are immediately available when the game needs them. These type of defects can be commonly mistaken for other defects like function defects because they can also cause similar crashes and freezes.

User inputs like double clicks or repeated presses of a button can also require timing considerations but they are often dealt with using mechanisms in the game engine or they can be built in manually by the developer. In multiplayer games information needs to be sent back and forth between players and game servers and this information has to be properly managed in the right order or the games behavior can be incorrect. Sometimes when a multiplayer game is waiting for information from the server about the player it chooses to predict what is going on in the meantime, this is why some games have been said to have “rubber banding” and characters being killed round walls.

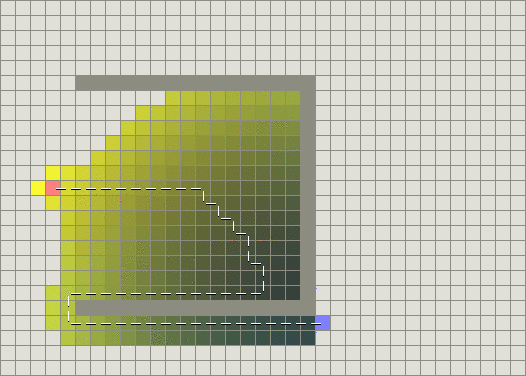
**Build/Package/Merge**

Build defects are errors that occur when the game files and data is being merged or built into an EXE, they can also occur when changing or managing game files and whilst identifying and controlling which versions get built. Often configuration management software is used to help control the use of game files, they are usually given a unique version identifier. A configuration specification is used to specify which version of which file to build, this helps the files to be identified by a single label in the config. Unlike most other defects this defect does not directly occur within the games code and won’t usually be seen by the customer because it will be fixed before any public release.

**Algorithm**

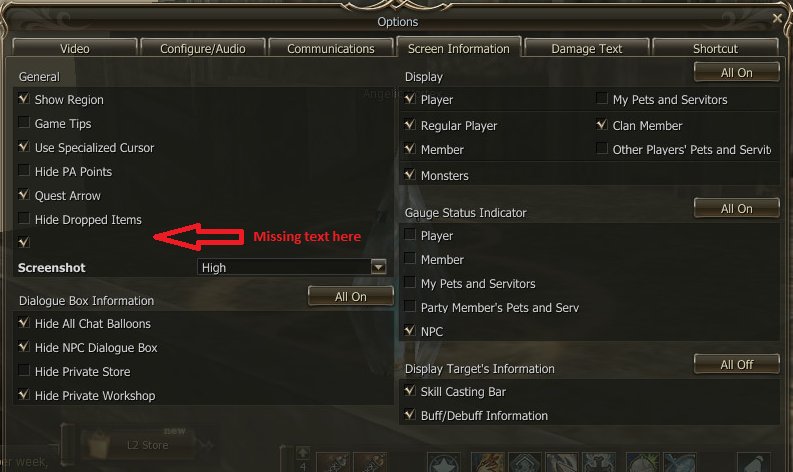
These defects occur with a games algorithm; some algorithm defects can cause game breaking bugs where the player can manipulate the game in a way that was not intended by the developer. Most games use many algorithms throughout the game which if done well will feel lifelike and smooth to the player, The AI in most video games will use algorithms to make decisions for example:

* Racing game algorithms will tell the AI when to pit-stop, change gear and use powerups.
* FPS game algorithms can control damage calculations based on factors like armour, skills and weapon types.
* Board game algorithms can control how the AI understands the rules of the game so that they do not have an advantage over the player.

The most common algorithm within games are the AI’s movement algorithm, this tells the NPC’s in the game how to avoid solid objects and walls and what path they would have to take to get to a destination the quickest, most movement algorithms also update the AI’s necessary path in real time so if they are blocked off from one route they will be able to be redirected to a secondary route.

This image shows an example of an AI’s movement algortihn working out the quickest path to the goal, the yellow squares represent where the algortihm has tried to go and the darker the square the more times the algortihm has overlapped its test, the algorithm does all these hundreds of tests in millisecons so that the AI will know where to go.

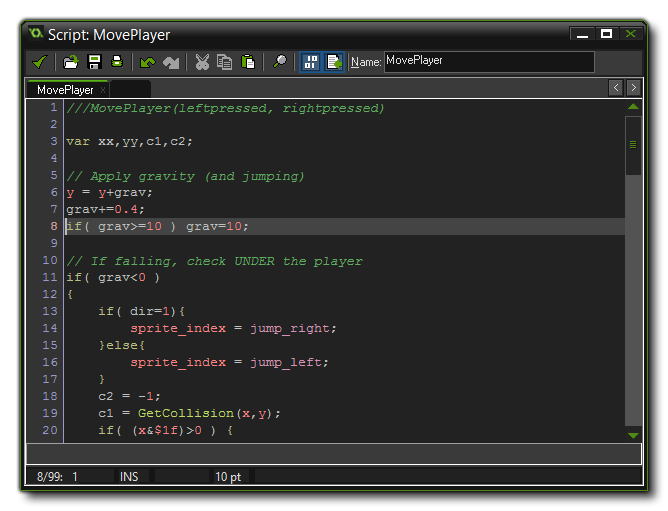
**Documentation**

Documentation defects occur with fixed data assets that are brought into the game many of these defects include: Dialog, audio, background music, cut scenes, sound effects and quest journals. These type of defects usually have nothing to do with the code within the game instead they are problems with the files and bytes being transferred to the game like audio being played, cut scenes being shown and dialog being displayed. Often the fix for these defects is to just watch or read through the cut scenes or dialog and just particularly paying careful attention to the graphics.

Language translations are a common documentation defect type in games and it is not an error with the transferring of data to the game instead the text has no problems being implemented into the game but instead it is just grammatically wrong. This is another type of defect similar to a timing defect that occurs in the code but is very easy to find out if it occurs because it will be shown during conversations with NPC’s in game or with pop-ups of information for the player, these defects are usually found in the early testing phases but can also be commonly found in open betas.

**Interface**

Interface defects occur when information is being exchanged or transferred within the game code, if the parameters sent don’t match with what the calling routine intended then undesired results occur. There are many ways that interface defects occur including:

1. Calling a function with the wrong value of one or more arguments.
2. Calling a function with arguments passed in the wrong order.
3. Calling a function with a missing argument.
4. Calling a function with a negated parameter value.

**If “sprite\_index” was supposed to be “sprite\_height”**

**Defect Triggers**

**Hardware configurations**

A hardware’s configuration is the allotted system resources for a specific device, most computer specialists are also able to improve the performance of hardware by allocating more system resources these features can usually be accessed via the BIOS.

Newer technology such as OS’s now have the ability to automatically allocate resources for different applications and programs that are run on the hardware, this is called Plug-and-play (PnP) and it eliminates the need for manual configuration which can sometimes cause errors if not done right and it also means that the amount of resources allocated is usually more accurate to what is needed.

**Software configurations**

Software Configuration Management (SCM) is the overall management of a software design project as it builds up into a software product or system. This includes technical aspects of the project, all level of communications, organization, and the control of modifications changes to the project plan by the programmers during the development phase.

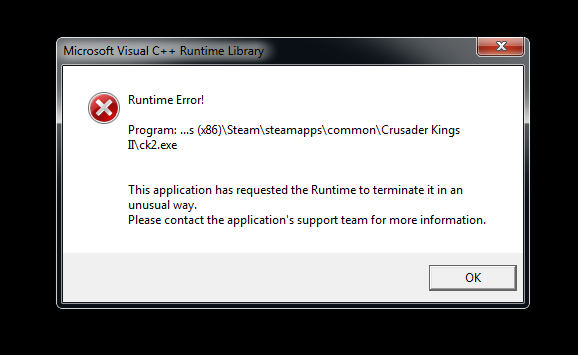
**Exception triggers**

* Audio sounds for errors and alert boxes.
* An alert box popping up at the wrong time can confuse the player and possibly be a Grade A defect.
* Can be triggered in any of the games operating regions.
* Dawn of War had an Exception defect trigger when players tried to connect to a multiplayer lobby.

Most exception triggers that the player will come across whilst playing will be handled within the game with error messages, the errors also usually have codes on them so that they can be easily distinguished and referenced later on e.g. error #404.

**Normal triggers**

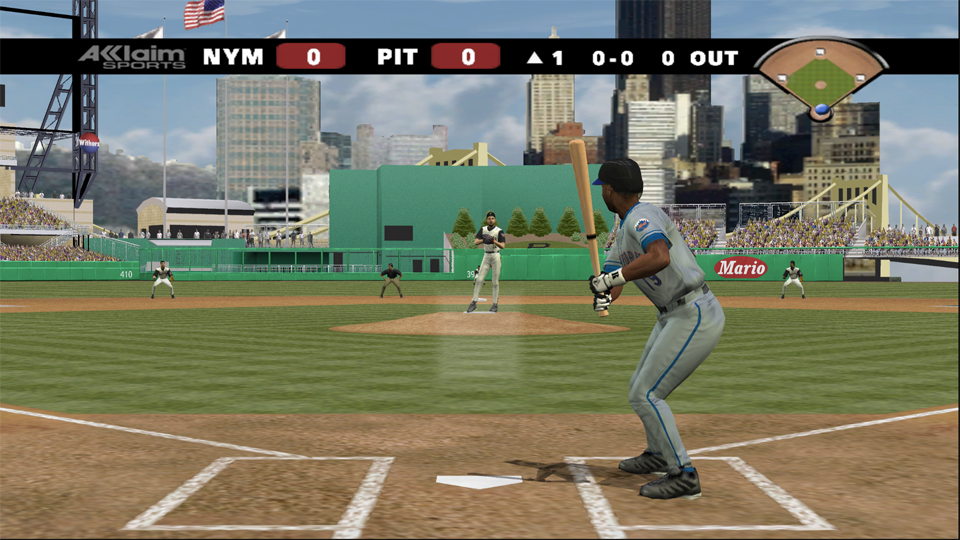
* Normal triggers occur when the game is just reading the base code which makes it function and not under any stress, configuration or exception conditions.
* Not likely to occur in real-life situations.
* If there is a defect within the normal section of the games code then it is likely to be found because it may be a Class A defect and therefore hard to miss.



Because they only occur with the base code of the game with no stressful circumstances they are extremely unlikely to make it into full release, these are commonly assignment defects that occur on game start-up.

**Restart triggers**

* Occurs when Quitting, ejecting the disk or turning of the game console or PC.
* Most games either prompt the player to be save often to ensure they don’t cause restart triggers or save often for them.
* Corrupt game data is a common defect of a restart trigger.
* For example, players have had problems with GTA 5 and corrupt data after the game had not been successfully exited or shut down.

This type of defect is usually caused by the player instead of within the internal code like most other triggers and it makes up a lot of the triggers that a player might come across because it is not as easy top test. An example of a game that had a restart trigger that gave the player an advantage is *All Star Baseball 2004* where the player could save and quit the game while they are pitching, upon starting up the game again the pitcher will have recovered all their lost energy. This kind of defect goes unnoticed if your test doesn’t go back into the game after saving, so remember the do that whenever you apply a restart trigger.

**Configuration triggers**

* Configuration triggers occur prior to the game being run.
* Date and time, System audio or Operating systems can cause Configuration triggers.
* These are Class A defects because they often don’t allow for the game to be started up or played.
* An example is Elder scrolls Oblivion which did not work on Mac OS X until it was decided to bring it to the Mac OS in 2004.

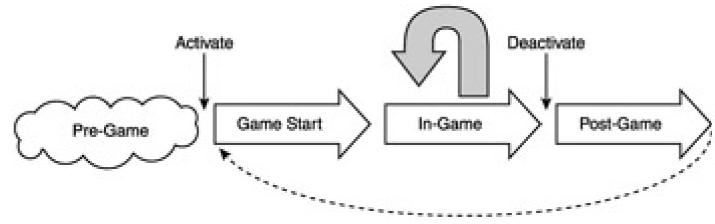
**Starting-up triggers**

* Starting up triggers Occur when the game is opened up or has started initiating.
* Initiative code like setting up variables or spawning enemies can cause start-up triggers.
* Things like new maps, powerups or enemies can also count as start-up triggers because they are also initiated on the game start-up or level start-up.
* Space engineers had a start-up defect where the game would pop up with an error message instead of starting up.

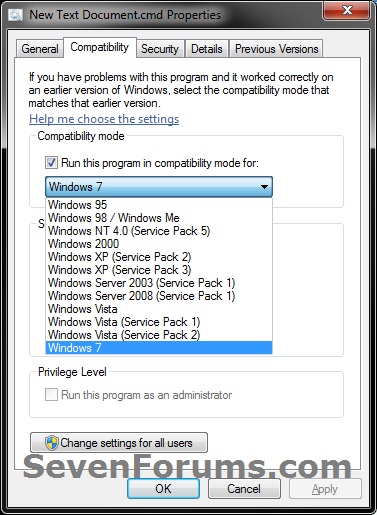
**Stress triggers**

* Occur when the game is under extreme stress related to either hardware or software.
* Memory, screen resolution, disk space, file size and network speed are all examples game conditions that can be put under stress whilst testing or playing.
* The game is played and tested under stress to understand its behaviour instead of just pushing it to a certain limit.

**Operating regions**



**Pre-game operating regions**

Pre-game operating regions represents the period before the use of the game, for mobile games this would be before the game is opened and for consoles this would be before the game disk is inserted. In both cases the user has the ability to change options and settings for the device that may impact operations within the game. An example of the settings that the user can change during the pre-game operating region that may affect the game are things like the Operating system version or type and the mode that the game is launched in e.g. Compatibility, Windowed etc. The type of defects and defect triggers that usually occur within this operating region include Build/Package/Merge and configuration triggers.

**Game start operating regions**

This region accounts for the time between the player starting the game and the game being ready to play. Within this operating region some activities can be interrupted such as cinematic sequences that show and introduction to the game. Other activities like the screen displaying a loading progress bar or animation can’t be interrupted. During this operating region the game will be performing activities which are essential to the proper operation of the game but are not visible by the player. After this process the game will be in a “ready” state where the game will wait for the player to press a button to start. Commonly defects can occur within this operating region if the player tries so skip the sequence that the game uses to load up the necessary files. The types of defects and triggers that occur during this operating region include assignment and timing defects as well as starting-up triggers.

**In-game operating regions**

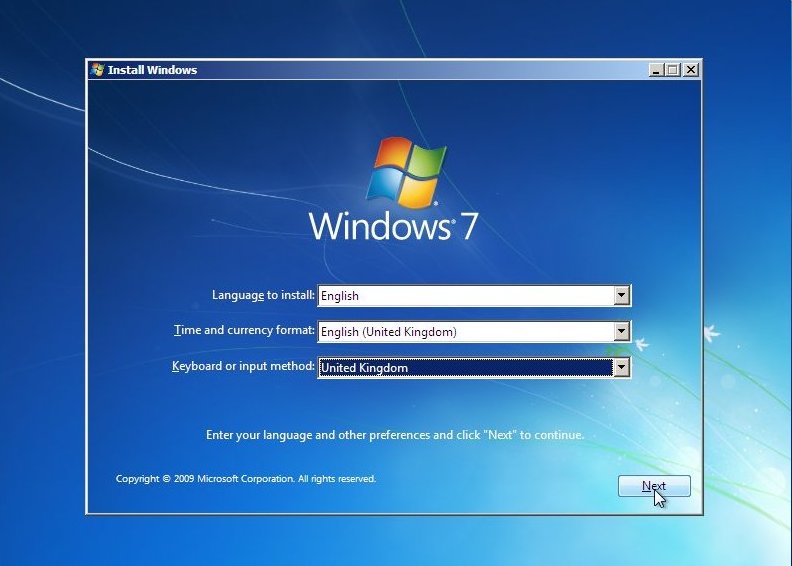
This operating region applies to all things that can be done when playing the game, some functions can only be performed once during the game and some can be repeated throughout the game. Some of these functions within this region require the player to meet a condition before they can occur, Games that include Non-playable characters (NPC’s) also manage these functions during the in-game operating region. This operating region is the most common for defects and triggers to occur because it is where the majority of the game files will be used, this means that the kind of defects and triggers that occur include Function, checking, timing, algorithm, documentation and interface defects as well as exception, normal and stress triggers.

**Post-game operating regions**

The player can end the game in multiple ways including; quitting the game without saving requires less processing whereas if the game is saved before quitting it can be harder on the software and or hardware. With portable devices where the player turns off the console using the off switch the device can perform save and shutdown operations prior to shutting off completely. Story based games often have a final cut scene or credits rolling when the player has reached the end of the game, some games reward the player with new experiences within the game once they have completed the story so that they can continue enjoying the game in other ways, this may activate code that is not used at all until the game is completed for the first time. An example of this operating region in a game is in Dawn of War 2 where the player wants to exit the game so they are presented with a dialog box that asks them if they are sure, this dialog box is primarily used to stop the player being able to exit the game with one click because they may not want to but a second purpose for this is allowing the game to prepare to shutdown beforehand so that things like game saves don’t get corrupted. The type of defects and triggers that can occur within this operating region include checking and assignment defects as well as restart triggers.

**Test phases**

**Preparation**

Before the testing team can proceed with work on the new game build, the preparation needs to be fully complete. The test equipment should have the appropriate hardware configuration planned and carried out so that the testing conditions are optimal to reduce unwanted errors that may not be caused by the game. This means that the hardware used must be completely reformatted which means that the data from previous tests like old saves or game data will be wiped from the machine or hard drive to ensure that the next tests are not affected by older files that are left behind.

For PC games it is usually more complicated to prepare for testing because there a lot of different aspects of a PC that can be modified so the versions and drivers need to be completely reinstalled e.g.

* A fresh installation of the latest version of the Operating system, this includes any patches or security updates.
* The latest drivers for all components of the computer. Since this is necessary for all drivers not just the usual video card and sound card drivers it also includes drivers for the chipset, motherboard and Ethernet card etc.
* Any “helper-apps” or middleware that are necessary for the game to run optimally must be installed with their latest versions, these can range from DirectX to Battle Eye service.

The reason why this is such an important phase of testing is because is it the foundation of all further testing and if a problem occurs during this phase than it can carry through each and every test phase and be hard to fix or even cause all testing data to be invalid which can severely ruin a games schedule. An example of this is an old game company had an internal QA team that regularly used the testing PC’s for things like Email, IRC, Web browsers and Unreal Tournament after running tests on up and coming PC titles which ended up leading to wasted time chasing bugs that were not caused by the game and instead caused by problems brought up by Email and file sharing programs that ran in the background and taxed the system resources and the network bandwidth.

**Alpha testing**

Alpha testing is the beginning stage of testing and it describes the bare bone essentials of the game packaged together for testing, this also means that during alpha testing the game is still far from completion because of placeholder assets, missing game segments or below-target framerate. During the course of alpha testing the game is fine-tuned, features are play tested and revised and systems developed by separate programmers are linked together.

The result of this very early version of the game build means that it is riddled with bugs and defects which can be overwhelming for a QA tester but well-planned and documented test suites should bring structure to the overall test phase. Over the course of alphas testing all modules of the game should be tested at least once and performance baselines should be established like frame rate and load times, during alpha testing having baselines for performance helps to determine how much needs to be done to get the games performance up to what was planned for a full release, during the early stages of developing a 3D action game an average of 20 – 30 frames per second (FPS) can be acceptable but the common release target is a solid 60 frames per second with no prolonged dips when there are a greater-than-usual amount of characters and special effects on screen. Entry criteria for Alpha testing include:

**30 FPS**

* All major game features exist in the build and can be tested.
* A tester can navigate the game along some path to finish.
* The code passes at least 50% of the platform Technical Requirements Checklist (TRC).
* The game is compatible with most specified hardware and software.

**beta testing**

Beta testing being the phase after alpha means that the developers should have a good idea of the game that they are creating, this means that there should be no new features implemented and what is already in the game build should be perfected. The term “Beta Testing” is commonly referred to any outside testing done by players, this is somewhat true because there are different types of beta testing including Internal bets testing and External beta testing, these two different testing types are really separate phases because internal beta testing where the games features are being thoroughly tested after being implemented in alpha testing these features will include Audio, Interface and platform compatibility. An example of Closed and Open beta testing done for a game is Overwatch which had a closed beta followed by an open beta which helped to test the games features bust most importantly the servers as well as the game balance so that upon full release of the game the servers would be able to handle the amount of players.

External beta testing is a separate later phase because it involves players outside the development team having access to the game to test features, there are also two different types of external beta testing:

**Closed beta**

A closed beta test means that only a certain amount of players can have access to the game and it usually requires some sort of sign-up, these closed beta tests are used for finding and reporting a whole plethora of bugs and defects like Checking, Build/Package/Merge, Algorithm, Documentation, Interface.

**Open beta**

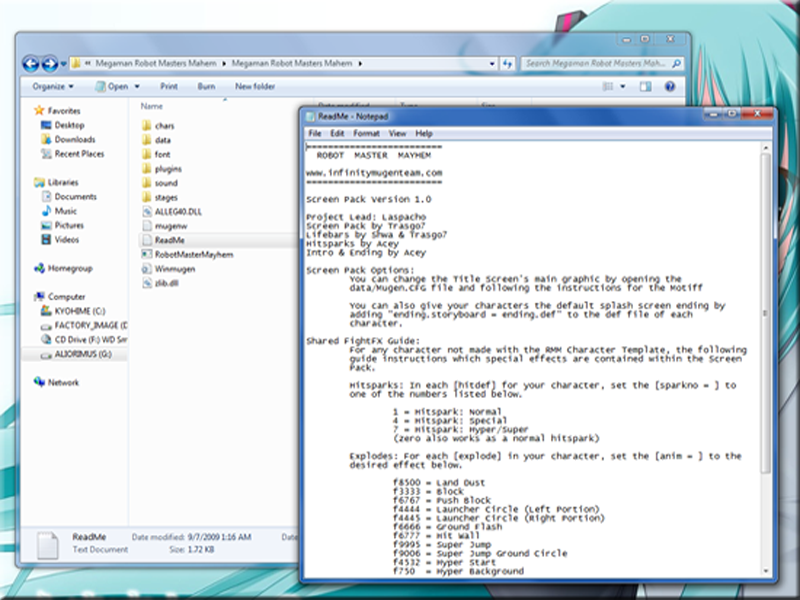
The difference between closed and open beta is within closed beta main game features are tested using a large player base but with open beta testing the main testing purpose is with the networking for the game e.g. the net-code and the amount of players that the game servers can hold with no connection problems. The type of defects that are tested within an open beta usually include defects that are directly linked to online games e.g. Timing and Checking.

Entry criteria for Beta testing include:

* All features and options are implemented.
* The code passes at least 100% of platform Technical Requirements Checklist (TRC).
* All controllers work.
* The game logic and AI is final.

**Gold testing**

Gold testing is the phase where the game build is starting to resemble a fully finished retail ready game, within this testing phase there will be multiple game builds called Gold Master Candidates (GMCs) which are versions of the final game build which will be sent off to be manufactured. Commonly the testing for this phase will be working through all the test suites again or as much as can be done in the time left before release.

Entry criteria for Gold testing include:

* 90% of all known bugs should be fixed including all of the grade A defects like Crashes, hangs and major function failures.
* Any known open issues have a workaround that has been documented in a readme.txt file for PC users.
* Release-level performance has been achieved (60 fps frame rate).

**End of certification testing**

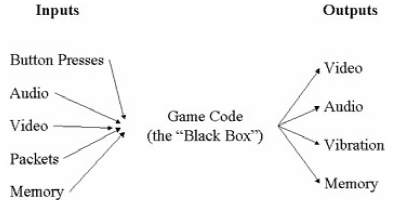
Once gold testing is complete a clean Game Gold master candidate (GMC) is sent to the platform manufacturer for final certification, the platform manufacturer e.g. (Microsoft, Nintendo and Sony) commonly conducts tests on games prior to releasing them on their platforms this is to ensure that the game is up to their standards and most importantly follows the age rating guidelines that were given by the developers for example if the game was specified to be a PG game then the rating that the platform manufacturer gets from the game rating companies like PEGI or ESRB. At the end of certification testing the Platformers manufactures QA team will write up a bug report of the games defects, this bug list will be used by the representatives of the publisher to dictate what bugs have time allotted to then for fixing. The bugs that will be fixed will be put into a “Must-Fix” list that will be given to the development team to avoid spending time on minor bugs that may cause the code to be at more risk to defects.

**Regression**

Regression testing is the process of having software updates and patches implemented into the final released game build, Publishers don’t like patches because they potentially add to the overall cost of the project but they don’t earn any more money for the companies. Developers don’t like them because they can be perceived by consumers as a failure of the game because it might need fixes after release. The upside of having Regression testing that allows for patches to the game build is that the development team can polish of the games features, as well as polishing off the game it also means that the player base of a game may be happy with how the developers are continuing development of the game.

**Testing Processes**

**Black box testing**

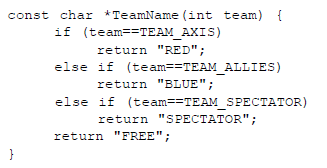
Black box testing refers to testing that is done from outside the application e.g. no knowledge or access to the games code. Game testers don’t commonly find defect within the games code instead they use the same input devices that the normal player would have, this means that Black-box testing is the most cost-effective way to test extremely complex networks of systems and modules that even the simplest of video games represent. The image below illustrates some of the various inputs that can provided to a video game and the outputs that you receive back. The most basic of inputs are positional and control data like button presses and cursor movements. These input devices include; Joysticks, Keyboard, Mice, Dance pads, Bass fishing controllers, Maraca controllers and drum controllers. Inputs can also include audio types like microphones in headsets or attached to the game controller. Video inputs can also be incorporated from usb cameras like or motion sensing hardware like Kinect. Secondary input from other players can come from a separate controller, a local network or the internet, finally memory cards and hard drives can provide stored data such as saved games or option settings.

**White box testing**

As opposed to Black-box testing the games tester will have access to the games code which allows them to predict the manner in which the code will influence the game and other parts of the code, this can be very difficult because it is nearly impossible to account for the player feedback loop. There are situations where white-box testing would be a more practical and necessary that black box testing, these situations include:

* Tests performed by developers prior to submitting new code for integration with the rest of the game.
* Testing code modules that will become part of a reusable library across multiple games and/or platforms.
* Testing code methods or functions that are essential parts of a game engine or middleware product.

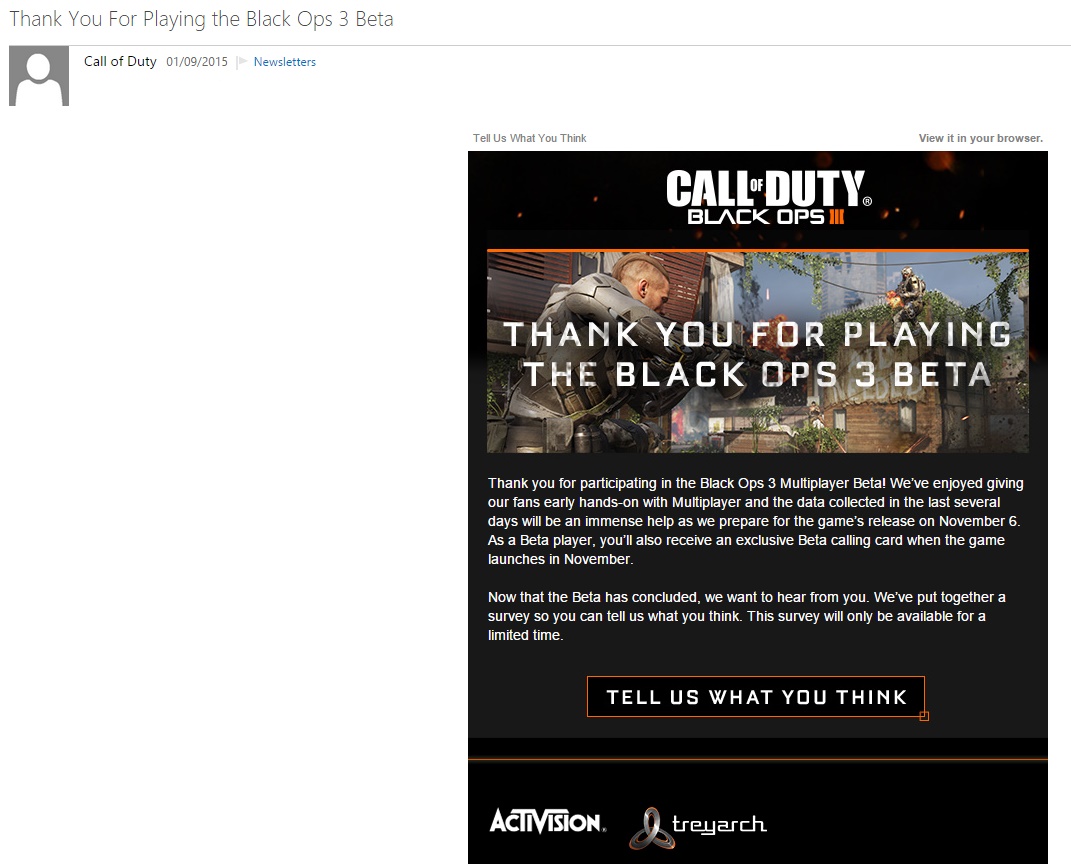
Below is an example of White box testing done on a selection of code from the game *Castle Wolfenstein: Enemy Territory:*

Four white box tests are required to properly test these lines of codes, the first test would be to call the “TeamName” function with the “TEAM\_AXIS” parameter to check if the “RED” string is returned. The second test should be passing the “TEAM\_ALLIES” parameter to check if the “BLUE” string is returned. The third test passes “TEAM\_SPECTATOR” to check if “SPECTATOR” is returned. The final test passes “TEAM\_NONE” to check if “FREE” is returned. These four tests will test each line of code at least once and also test the behavior of both the “true” and “false” branches of each IF statement.

**Smoke testing**

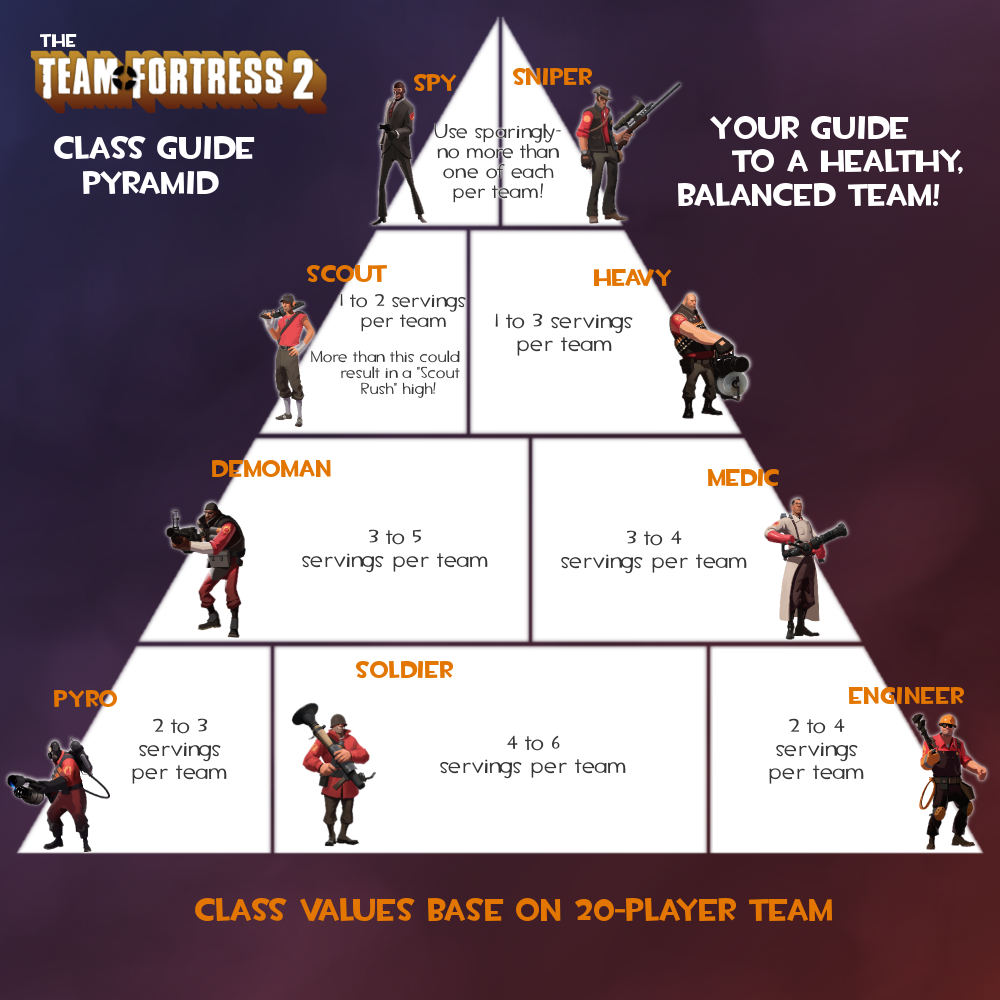
Smoke testing is usually the next step after accepting a new build and it can commonly be automated because it refers to the series of test cases that are run before more rigorous tests are done on the game build. The name smoke testing was given because this stage of testing is used to determine if “smokes” (malfunctions) when the game is run. At the minimum this test phases should consist of Launching and loading the game, this means that a tester should launch the game and spend a minute playing around with each module like the options and main menu, if the game launches with no obvious errors or performance issues than it has passed smoke testing and should be put forward for further testing. An example of smoke testing is with a game like Bioshock Infinite the game should be launched and the menus should be navigated to test for issues like FPS, Crashes or visual bugs.

**Cleanroom testing**

Clean room testing is a technique extracted from a software development practice known as cleanroom software engineering, in game development cleanroom testing refers to testing the game in a way that replicates how a player would interact with the game. To help realistically mimic how the player interacts with the game for testing purposes a lot of research is necessary because it gives the testers and plethora of good examples of how players play the game, these types of research include bar charts of how many times certain menus were opened, Key-macros that were defined and used by players and matches or races that may take longer at the end of the game due the increasing difficulty.

A good example of cleanroom testing is when players that take part in a games beta might be asked to answer surveys about the experience they had with the game and the types of missions that they prefer.

**Play testing**

Play testing is very different to the other types of testing that are involved in game Quality assurance because it involves questions and tests that do not specifically have one plain answer like “Does this button do its function?” instead it is more opinionated and open ended in the way it should be answered. The questions asked within play testing should be things like:

* Is the game too easy?
* Is the game easy to learn?
* Are the controls intuitive?

And most importantly:

* Is this game fun?

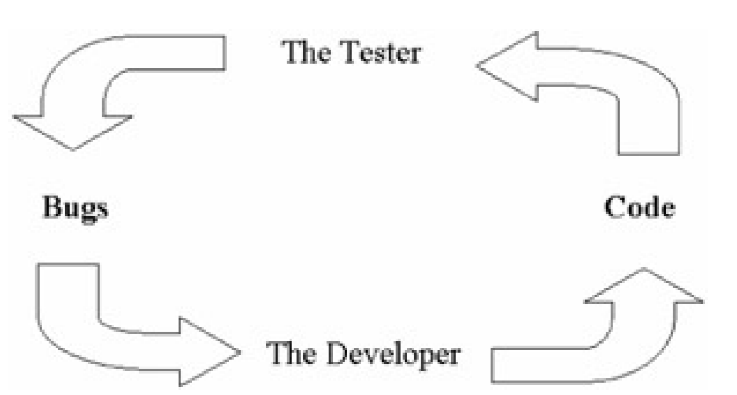
This means this type of testing covers aspects of the game like balancing e.g. is the game challenging but not frustrating? The overall balance of the game also refers to the mechanics and element within it like characters, weapons and fighting styles, examples of things to test are:

* Balance between Orcs vs Humans e.g. each race should have attributes that work well against each other like Strength over speed and agility stamina over health.
* Melee fighters vs ranged fighters e.g. for the game to be well balanced players have to make up their own mind on what class they are interested in playing, this means that each class should be equally useful and practical in all situations so that one class does not overpower the other throughout the game.

**Ad hoc testing**

Ad Hoc is a Latin phrase that can be translated as “to this particular purpose”. In game development this refers to the giving the game tester full reign over the types of tests that they do on the game build, during testing the user will subconsciously think “what will happen if I do this…?” and Ad Hoc testing allow for them to answer those questions by removing the list of bugs found with the build and having them wander through the game as if it was a maze. There are two types of Ad Hoc testing, the first being “free testing” which was previously explained as giving the tester full control over the type of testing done. The second type is directed testing, which is intended to solve a specific problem or find a specific solution. A good example of this type of testing is to sometimes have external people to test the game e.g. other employees of the company, this means that they will be a set of fresh eyes on the game that will see it and its bugs differently to anyone else.

**Testing life cycle**



1. **Plan and design test:**

Planning often occurs at the beginning of the test phases but should be revisited each time the testing is redone. The type of questions that should be asked are what has changed in the design specification since the last build? And what features have been cut? The scope of testing should ensure that no new issues were introduced in the process of fixing prior to release.

1. **Prepare for testing:**

Everything that makes up the game should be aligned together to have an almost complete version of the game for testing. Often build/package/merge defects could occur here because files and applications are brought together and problems with compatibility may be encountered.

1. **Performing the test:**

The test suites will be run against the game build and If a defect is found the tester will test things around the bug to be certain that they have as much information as possible for the bug report, the more work put in during the testing the easier the bug report will be. Testing around the defect is crucial to the process so that all aspects of the problem are noted to save time later on otherwise defects that are found may not be completely fixed and could possibly come up again in a later test and continue to be a problem.

1. **Report the results:**

Log the results of the test in the bug report and also log the completed test suite so that it won’t be gone over again unnecessarily. Being clear with all aspects of the bug will help yourself and others later ion locate and fix the bug.

1. **Repair the bug:**

The developers and programmers will be doing this step but the tester should be available at this stage to discuss the defect in detail if the bug report was not fully understood. The bug report should be the only necessary document that is needed to fully understand the cause, result and features of a bug within the game.

1. **Return to step 1:**

With new bugs and results a new build of the game can be plan and designed.

**Combinatorial testing**

Getting the necessary amount of testing just right and not too much or too little is and extremely important part of game testing and combinatorial testing is one of the largest tools used to help testers manage exactly that. The most important aspect if combinatorial testing is to cut down the amount of testing that need to done on the game, because the testing of a game is held to strict deadlines it means that not every possibility of a game can be individually tested because that will take far too much time so the purpose of combinatorial testing is to use pairwise combinatorial testing to find defects and gain confidence on the fame software while keeping the test sets small relative to the amount of functionality they cover. “Pairwise” means that each parameter that is used for testing needs to combined with each other once. The types of parameters that are commonly used in combinatorial testing include:

* Game events
* Game settings
* Gameplay options
* Hardware configurations
* Character attributes
* Customization choices

**Combinatorial testing example**

With the tables on the right I started with the top table and marked out each possibility with the maps, guns and states this left me with 16 different tests which was too much because I did not need to test each gun and each map together. I needed to test each state with each map because there may be faults with how the states interact with the map. Shortening down the table to 8 tests left me with each map being tested twice with different states and different guns. This testing table will help the game testing to go quicker and reduce the amount of unnecessary tests being done which helps with time and money.

**Test Flow Diagrams**

Test Flow Diagrams (TFDs) are visual models that represent a games behavior from the player’s perspective. These diagrams help with testing because a tester can travel through the diagram to exercise the game in both familiar and unexpected ways.

One of the main upsides to using Test Flow Diagrams is that they are graphically simple and appealing which means that it describes the complexity of how a games events can function in an easy to understand diagram. These Test Flow Diagrams are made up of elements like Flows, Events, Actions and States that outline the different type of functions that occur with games.

**Flows**

Flows are simply the connecting lines that join each game element together within the TFD, they will have an arrow to show which way the games elements lead into each other e.g. picking up a sword would lead to swinging the sword and not the other way round.

**Events**

Events are the operations that the game receives from the player like picking up an object or casting a spell, the three factors that should be considered when including a new event are:

* Possible interactions with other events.
* Unique or important behaviors associated with the event.
* Unique or important game states that are a consequence of the event.

**Actions**

Actions temporarily occurs between events and are commonly shown on a flow line to symbolize that they are temporary which means that they can only be perceived, detected or measured when they occur and will not persist over time. Examples of actions include Sounds, visual effects, game controller feedback and network information sent over a multiplayer server e.g. opening a chest in a room would be an event but particle effects that fly out of the chest would be an action.

**States**

States represent persistent game behavior and are re-entrant, As long as you don’t exit the state you will continue to observe the same behavior and each time you return to the state you should detect the exact same behavior.

**Test Flow Diagram example**

This diagram shows the flow of an experience the player will have in the game, it explains how that experience will be tested and shows how many options the player has in that room. Test flow diagrams help to plan the basics of the game code because if the programmer looks at this diagram it will tell him exactly what needs to be in the game.

****