**Unit Testing Strategy**

**For (Pit-Stop)**

**Jason Smith**

**Student No: 17702485**

**BSHC Year 2**

Unit testing can be performed using two different approaches. Black-box testing is when components are tested using an external point of view and white-box testing is where components are tested using an internal point of view. (Crispin and Gregory, 2014) The segment that I will be testing is the pit-stop process within the racing game. The pit-stop section of the racing game is an important element of the race where a players car can temporarily leave the race-track during the race to have their fuel replenished and have their cars worn tyres replaced by new ones. This procedure will enhance the performance of the players car and benefit them for the remaining duration of the race. The process itself becomes automated when the players car enters the pitlane and stops at the repair point. The only control that the player will have over the process is when they click a button to start the process and then to click a button to end it.

It is in the players best interest to have the pit-stop completed in the fastest time possible so that they may re-join the race without falling too far behind. For this reason the process must be fast, fluent and error free. The pit-stop segment of the game must take a players car as an input when it drives into the pit-lane and output the car in an optimal fully repaired state. Any other outcome which results in the player not re-joining the race must be seen as a failure as it will negatively affect the players gaming experience.

**The first test** **case** will analyse the players car driving into the pitlane and out the other side without stopping at the repair point. This test should cater for the possible scenario where a player could have entered the pitlane accidently but did not wish to make a pit-stop. An example of this would be as a result of a collision during the race which knocked the players car off course and into the pit-lane.

The result of this test will only be judged as successful if the players car can just drive past the repair point in the pit-lane without stopping and re-join the race. This is because the automated process should not begin until they stop at the actual repair point and not until then.

**The second test** **case** will analyse a scenario where a player drives his car into the pit-lane and stops at a repair point but already has a full fuel load (10 units of fuel) and tyres which are already in perfect condition (Level 10 tyres). Even though this players car does not require additional units of fuel or a change of tyres it was important that the system recognises this and allowes the player to re-join the race without delay. This is the only outcome where the result of this test can be declared a success.

Because the code is written in a way that allows the player to end the pit-stop process as soon as their fuel and tyre levels are full, then the player can end the pit-stop at the click of a button and re-join the race with minimal time wasted. So the car in this test case should be treated the same as any car that had just completed the pitstop process.

An example of the code is shown below.  
While (fuel < 10 && tyres < 10){  
 message = “Repairing, please wait.”;  
 fuel++;  
 Tyres++;  
}  
message = “Repairs complete, click to re-join race”;

**The third test case** will analyse how a players car should exit the pit-lane and re-join the race once their repairs were completed. Because the pit-lane exits back onto the race track where a race is in progress, there is a risk of collision with other cars at this point. To prevent this a Boolean value isClear() is used to determine if the pit-lane exit is clear and safe to exit. A test should involve repeatedly taking the pitstop process in a large number of games and safely exiting the pit-lane without incident.

The result of this test can only be declared a success if the player is able to safely re-join the race each time after completing their pit-stop.

Unit testing is used in situations where software developers want to test individual segments of code before integration. Some advantages of unit testing are that we can segregate each part of the program and test that every part is working correctly before integration testing. Issues can also be detected early and resolved without effecting the other pieces of code. Bugs discovered in unit testing cost much less to fix than bugs that are discovered later on. (Crispin and Gregory, 2014)

Disadvantages of unit testing are that it is not possible to simulate and test every possible scenario. Unit testing needs to be done in addition to further testing such as integration testing and it is time consuming to develop effective unit tests for many different scenarios.

Testing is an integral part of the software development process. Before the release, it is essential to make sure that all program components are functioning correctly and that the entire system operates as expected. Unit testing is an effective and valuable testing strategy but only if it is preformed in conjunction with other testing strategies. This type of testing will not catch every bug in the system but it is still an important part of the overall testing process. This is because the purpose of each sort of test is different, and every kind of test catches different types of bugs.

**Bibliography**

**Crispin, L. and Gregory, J**. (2014). *Agile testing*. 1st ed. Upper Saddle River, N.J: Addison-Wesley.