Software Dev Project Document

By Kenneth Mallabo

Updated: 17/01/2023

This document is for the written assignment for the Software Development module. It is a project document / design document based on the previous assignment’s code.

Contents

[Requirements 1](#_Toc124880526)

[Design 2](#_Toc124880527)

[Evaluation 3](#_Toc124880528)

[Cyclomatic complexity 3](#_Toc124880529)

[Whitebox Techniques 4](#_Toc124880530)

[Basis Path Testing: 4](#_Toc124880531)

[Branch coverage 7](#_Toc124880532)

[Blackbox Techniques 8](#_Toc124880533)

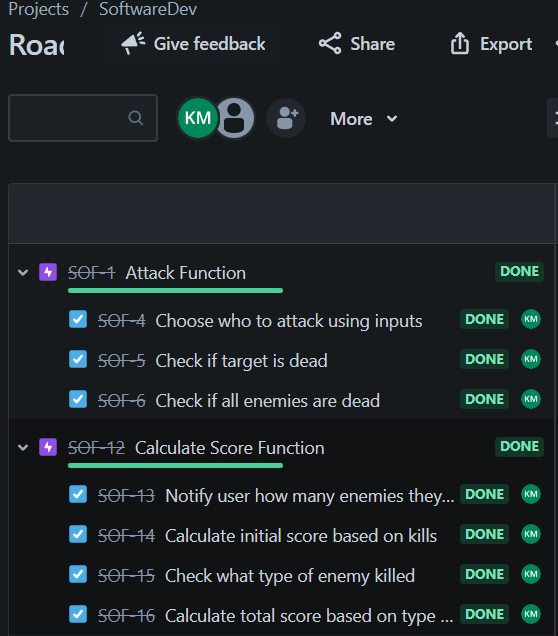
[Boundary Value Analysis 8](#_Toc124880534)

[Equivalence Partitioning 9](#_Toc124880535)

[Test code coverage statistics 9](#_Toc124880536)

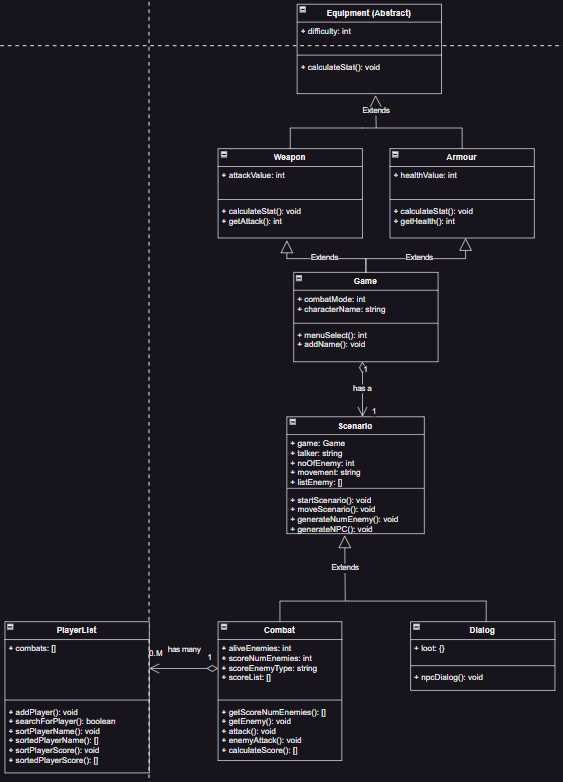
# Requirements

The application used in the requirements section is called Jira. It is utilised to create two user stories that are the most complex functions in the project. The functions are the attack and calculateScore functions that can be located in the Combat class.



# Design

There is only a slight change in the UML class diagram in comparison to the previous assignment. The Dialog class now has a new list variable called loot.



# Evaluation

## Cyclomatic complexity

The cyclomatic complexity of the textAdventureV3 program has been calculated using the following command: radon cc -s textAdventureV3.py

The top two most complex functions can be seen in the image below:

textAdventureV3.py

M 248:4 Combat.calculateScore - B (9)

M 208:4 Combat.attack - B (8)

## Whitebox Techniques

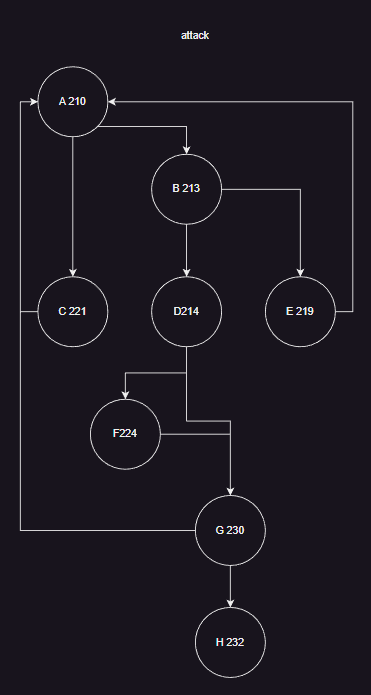
Whitebox Techniques are used for unit, integration and regression testing. This technique of trying to get the coverage of a set of test cases in order to find possible branches or paths. There are five techniques but only 2 are chosen, basis path testing and branch coverage.

### Basis Path Testing:

This is a combination of branch testing and path testing in order to make sure that all possible paths have been tested. It also provides the bare minimum number of tests cases that need to be considered and written.

**Attack function:**

1. Control flow graph:

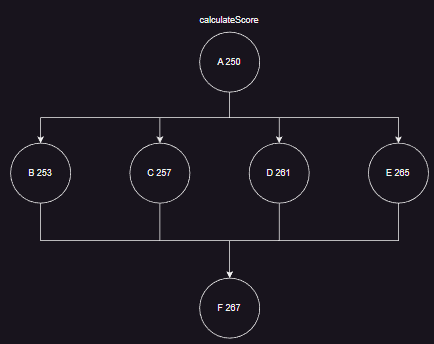


1. Cyclomatic number for attack is 5.
2. Basis set of paths for attack: AC, ABE, ABDFG, ABDG, ABDFGH.
3. Design test cases:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test ID | Test Description | Input | Initial Conditions | Expected Outcome | Actual Outcome |
| 1 | Test path AC | target = 2 | listenemies does not have 2 | "Incorrect target has been chosen" | As expected |
| 2 | Test path ABE | target = 1 | Health of target is 0 | “Chaos Marauder is already dead!” | As expected |
| 3 | Test path ABDFG | target = 1 | Target has died | "Chaos Marauder 's health has dropped to 60”  “Chaos Marauder has died” | As expected |
| 4 | Test path ABDG | target = 1 | Target has been hit | "Chaos Marauder 's health has dropped to 60” | As expected |
| 5 | Test path ABDFGH | target = 1 | All enemies in listenemies are dead | “Chaos Marauder has died”  "All enemies has been slayed!" | As expected |

**calculateScore function:**

1. Control flow graph



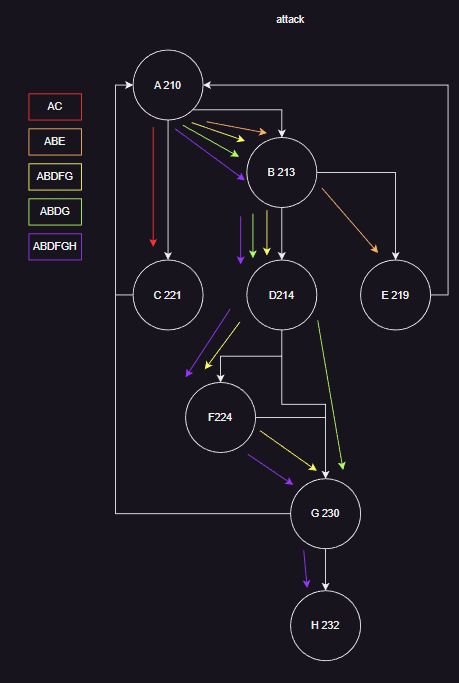
1. Cyclomatic number for calculateScore is 4.
2. Basis set of paths for calculateScore: ABF, ACF, ADF, AEF.
3. Design test cases:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test ID | Test Description | Input | Initial Conditions | Expected Outcome | Actual Outcome |
| 1 | Test path ABF | scoreNumEnemies = 1 | Enemy type is a Chaos Marauder | Score = 80 | As expected |
| 2 | Test path ACF | scoreNumEnemies = 1 | Enemy type is a Slaver | Score = 20 | As expected |
| 3 | Test path ADF | scoreNumEnemies = 1 | Enemy type is a Pirate | Score = 40 | As expected |
| 4 | Test path AEF | scoreNumEnemies = 1 | Enemy type is a Ninja | Score = 60 | As expected |

### Branch coverage

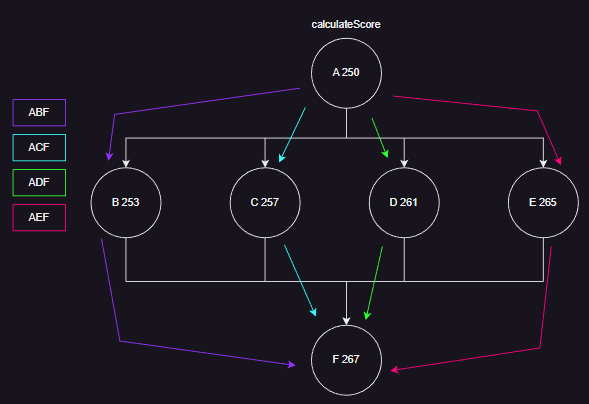
This is achieved by the number of decision branches that are covered by a set of test cases. Getting 100% branch coverage is the goal and it is determined by ensuring that every decision branch is used at least once in a test.

**Action Function:**



There is 100% branch coverage.

**calculateScore function:**



There is 100% branch coverage.

## Blackbox Techniques

Blackbox techniques are utilised to show functions are operational by making sure proper inputs also produces correct outputs. It tries to find errors putting entering inputs that are expected by someone of has no knowledge of the inner workings of the program’s code. The two blackbox techniques are Boundary Value Analysis and Equivalence Partitioning.

### Boundary Value Analysis

This technique attempts to create test cases in order test boundary values and outside the boundary values. Boundary values are the limits of the equivalence classes.

**Action function:**

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Description | Expected Result | Actual Result |
| 1 | target = 0 | "Incorrect target has been chosen" | As expected |
| 2 | target = 1  but target’s health is 0 | "Chaos marauder is already dead!" | As expected |
| 3 | target = | "Incorrect target has been chosen" | As expected |
| 4 | target = test | "Incorrect target has been chosen" | As expected |
| 5 | target = 1  target exists | "Chaos marauder’s health has dropped to 20!" | As expected |

**calculateScore function:**

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Description | Expected Result | Actual Result |
| 1 | score = 1  scoreEnemyType = Chaos Marauder | Score = 80 | As expected |
| 2 | score = 1  scoreEnemyType = Slaver | Score = 20 | As expected |
| 3 | score = 1  scoreEnemyType = Pirate | Score = 40 | As expected |
| 4 | score = 1  scoreEnemyType = Ninja | Score = 60 | As expected |

### Equivalence Partitioning

Equivalence partitioning consists of two parts, creating a new test case until all valid equivalence classes are covered and the other is creating a new test case until all invalid equivalence classes are covered.

**Action function:**

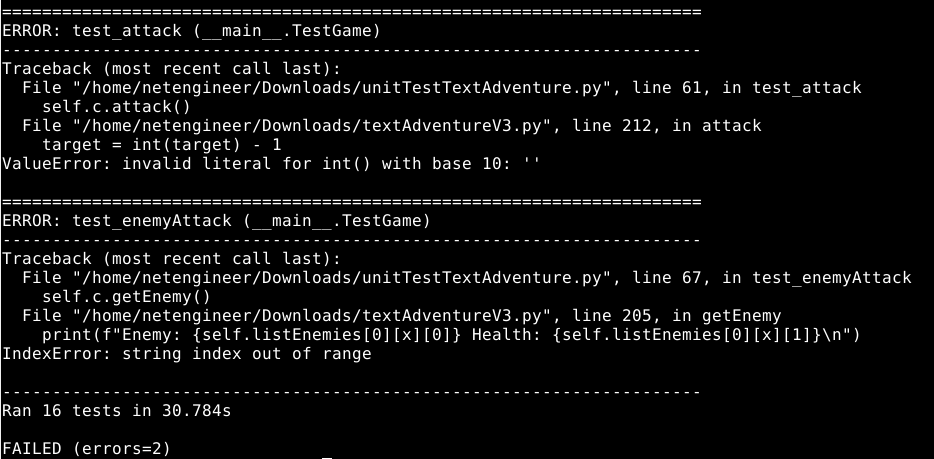
|  |  |  |
| --- | --- | --- |
| Specification condition | Valid equivalence class | Invalid equivalence class |
| target int | 1-4 target | 0 target, > 4 target |
| listEnemies list objects | 1-4 listEnemies | 0 listEnemies, > 4 listEnemies |

**calculateScore function:**

|  |  |  |
| --- | --- | --- |
| Specification condition | Valid equivalence class | Invalid equivalence class |
| scoreNumEnemies int | 1-4 scoreNumEnemies | 0 scoreNumEnemies, > 4 scoreNumEnemies |
| score int | 20-320 score | 20 < score, > 320 score |

## Test code coverage statistics

The test code coverage statistics achieved by the unit test code can be performed running the following command in a terminal window: coverage run -m unitTestTextAdventure.py.



A bonus is that creates the Coverage report which analyses the test code and generates statistics about the kind of testing of the application code is used by the test code. This can be achieved by the command: coverage html.

