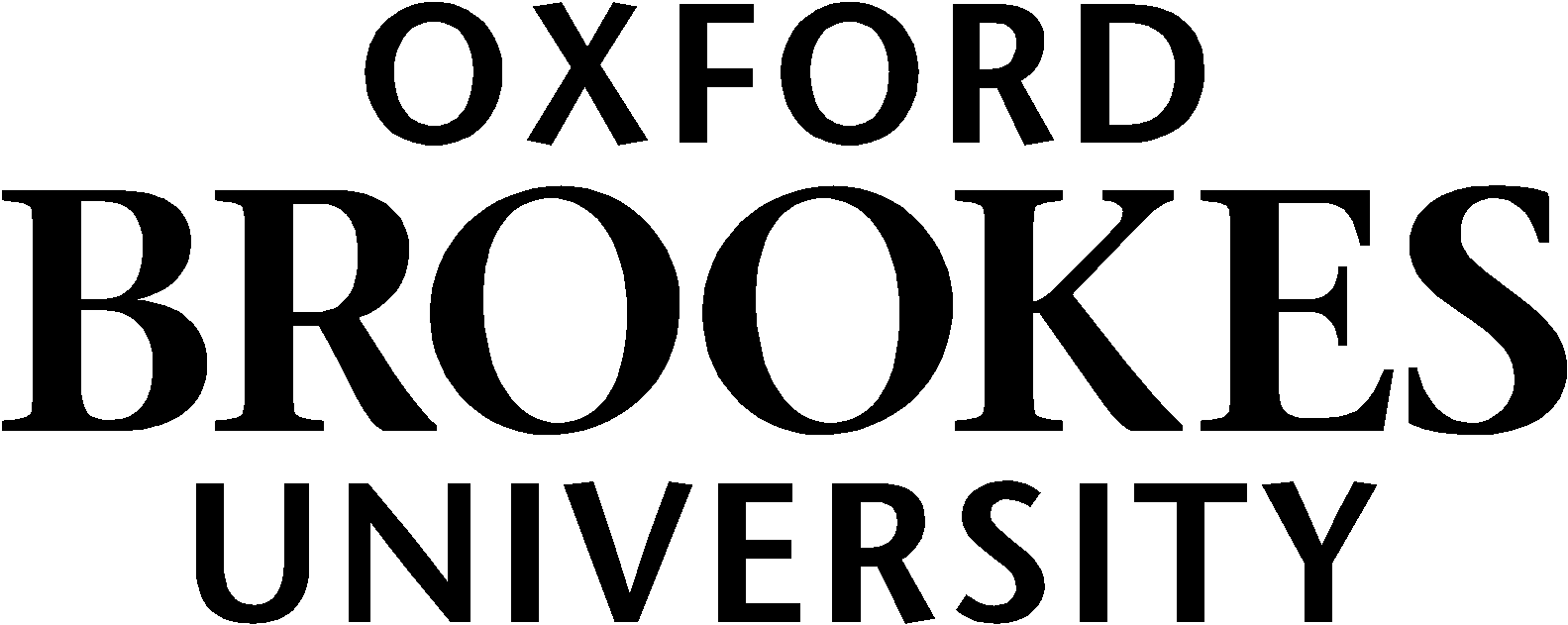
****

**Assessment cover**

|  |  |  |  |
| --- | --- | --- | --- |
| Module No: | **COMP 4006** | Module title: | **Professional Programming Practice** |

|  |  |
| --- | --- |
| Assessment title : | **Minesweeper Evaluation** |

|  |  |
| --- | --- |
| Due date and time**:** | **Friday 7th March 2025 at 1pm** |

|  |  |
| --- | --- |
| Estimated total time to be spent on assignment: | **24 hours per student** (excludes 50 hours for independent / directed study) |

**LEARNING OUTCOMES**

|  |  |
| --- | --- |
| **On successful completion of this module, students will be able to achieve the module following learning outcomes (LOs):** *LO numbers and text copied and pasted from the module descriptor*. | |
| **LO1** | Use professional tools to design, implement, test and debug a software artefact coded in a modern programming language. |
| **LO2** | Understand and apply the concepts that underpin integrated development environments such as how an IDE uses projects, packages and file structures to organise the elements of a system. |
| **LO3** | Use version control software for secure, multi-programmer development of software. |
| **LO4** | Understand basic principles of code quality (readability, correctness) and code design and apply them as part of their programming process. |
| **LO5** | Critically evaluate existing programme code in terms of code quality and design principles. |

|  |  |
| --- | --- |
| **Engineering Council AHEP4 LOs assessed (from S1 2022-23):** | |
| **B17** | Communicate effectively with technical and non-technical audiences. |

**Statement of Compliance**   
I declare that the work submitted is my own and that the work I submit is fully in accordance with the University regulations regarding assessments *(*[*www.brookes.ac.uk/uniregulations/current*](http://www.brookes.ac.uk/uniregulations/current)*)*This assignment must be entirely your own work and you are not allowed to use AI tools to complete it.

**EXPLANATION OF AND FIX FOR THE LONG METHOD CODE SMELL**

|  |
| --- |
| An example of a long method can easily be spotted in the playGame() function it contains the inputs and outputs, the game rules and the display all in one method. As seen in the image below:    Screenshot 1: The playGame() function and its role.  We could make those points into methods but also making sure we don’t add any unnecessary “transition ”methods by making plenty of small methods. We’ll refactor the function by using the extract method, taking clumps of code and turning them into methods.    Screenshot 2: Screenshot illustrating why I chose to proceed with the extract method.    Screenshot 3: Example of the extract method; it is also made clear what the purpose of the new method through its name.  We could add a boardConfig() function for the board’s dimensions and the amount of mines on the board, a display() function to get the full board displayed and a text message if the player won or lost at the end of the game, a playerhitMine() to return a Boolean expression of if the player has hit a mine or not during its turn and finally playerMove() to get the player’s move coordinates. Then call all these functions in the playGame() making the function easier to understand having all of its core mechanics displayed individually in different functions with easy to understand name and readable code. If necessary the incline method (see screenshot 2) can be used to “fuse” smaller methods if to many extract methods were made but in this case it is not necessary. |

**EXPLANATION OF AND FIX FOR THE REPETITIVE CODE SMELL**

|  |
| --- |
| Now for an example of duplicated code we’ve got the addMine() and makeMove() functions which process neighbouring squares in the same way but separately. The code makes it so that the code could go out-of-bounds or in an already selected spot, leading to duplications.    Screenshot 4: Duplicated code explained and methods to get rid of it.  In this case, simply using the extract method again would be the best way of fixing the duplicated code. We could create a getNeighbours() function that takes care of checking the squares neighbouring the spot where doing so also avoids the if-statement redundancy in both functions. I have used the same refactoring method twice but it is the simplest and most promoted refactoring method in the book.    Screenshot 5: Motivation for using the extract method. Through this I saw that the book favours shorter methods which I also find simpler to write and easier to understand. |

**OTHER CODE FLAWS AND FIXES**

|  |
| --- |
| Some functions have got some deeply nested code, with chains of if-statements and for loops, again from the addMine() and makeMove() functions. This makes code so much harder to understand other people trying to read the code.    Screenshot 6: Author explaining why nested statements make code harder to understand, with a similar example and provides a simple solution on how to make it easier to comprehend, for anyone else trying to read the code.  There is also some very poor error handling, in this code notably in the playGame() and makeMove() functions. There isn’t any way for the playGame() function to handle out of bounds coordinates which would lead an IndexError, same with the makeMove() function; furthermore if a spot that is already selected is selected again, it will also cause the program to decrement self.selectableSpots leading to incorrect logic as it is missing input validation. The index error can be fixed by adding an extract method mentioned previously. To avoid the playGame() assuming that all moves are valid we can use a try-except method, why an Exception instead of return code, it’s to make it easier to process for the programmer as it doesn’t have to be handled right away.    Screenshot 7: Author explaining why using Exceptions is better than using return codes    Screenshot 8: As it is said on the screenshot, it facilitates error processing  This is not mentioned in the book but I’d like to add that, according to PEP8, the class names should be PascalCase and not camelCase; BoardSpot and BoardClass instead of boardSpot and boardClass respectively. |

**USE OF DEBUGGER TO UNDERSTAND PROGRAM**

|  |
| --- |
| First of all, I’ve saved the original minesweeper program with inputs and then a modified version with the inputs set as constants by making a commit git repository of both versions of the code.  So now onto the debugging of the functions \_\_str\_\_ and addMine to understand what they do. First the \_\_str\_\_ function:  I’ll place a breakpoint at the playGame() call line as it calls all the functions necessary for the game to function, thus we’ll step into the function step over every other function until reaching line 108, in which I’ll step into; we can see the local variables that were previously changed to constants.    Screenshot 9: Line to step into to get in the \_\_str\_\_ function.  Now we step into the \_\_str\_\_ function, we can see that there is a returnString to store the board and a divider to divide the board’s rows    Screenshot 10: First lines of the \_\_str\_\_ function.  After doing an iteration of the first for-loop, we can see the returnString change by adding a column divider followed by the column number (i), this loop will continue until the loop has got all of the columns also extending the divider until it’s as long as the first row of column’s length    Screenshot 1: Entering the first loop.    Screenshot 1: Entering the second loop.  Now onto the rows, the first divider will be added to the returnString followed by the first row value.    Screenshot 1: Entering the third loop.  Now entering the next loop, it iterates for each (x;y) coordinate a divider which is then appended to the returnString for if there is a mine and the coordinates target a selected spot, if the coordinates only target a selected spot and if none of those parameters apply.    Screenshot 1: Third loop, first iteration.  After iterating every single coordinate, the two last rows of code append a last divider and the horizontal divider to the returnString and returns the board through the returnString.    Screenshot 1: Function full completion and final board output.  Next the addMine() that I’ve previously stated needed a little change to its code, but without tweaking it, let’s try to understand how it works through the debugger:  I’ll proceed in the same way to step into the addMine() function by stepping into the playGame() call, then stepping into the board variable on line 106.    Screenshot 1: Line 106, used to step into the addMine() function.  We are now in the \_\_init\_\_ function which randomly assigns coordinates, which can be seen in the local variables tab, to a mine whilst also making sure there isn’t already a mine in that spot; I then step into the self.addMine().    Screenshot 1: Line in the \_\_init\_\_ function used to step into the addMine().  In the addMine() function we can see, the value of the mine and its “state” (True or False) directly updated to set its value to -1 and state to True; the value to respect the game rules as that is the value of a mine in a Minesweeper game and the state so that the program can recognize the mine.    Screenshot 1: First loop after updating the values of the randomly selected mine.  Then follow a succession of loops and if-statements which can be hard to understand but if we go through them line by line make sense. The first loop is to iterate through the row above, same row and row below, then checking if those values are in range. Then goes through the first sets of if-statements, which check if the values above and beneath the mine; if they aren’t mines their value is updated by adding 1. All whilst making sure said spots are part of the board.    Screenshot 1: If-statements checking the state of the surrounding spots.  And finally, the values of the left and right spot are also checked and updated accordingly.    Screenshot : If-statements checking the state of the left and right spot.  This all repeats itself for as many mines are in the board.    Screenshot : Shows how many iterations are done, being as many as there are mines in the board. |

**USE OF DEBUGGER TO FIND BUGS**

|  |
| --- |
| I’ll now go over the bugs in the \_\_init\_\_ and makeMove() functions; first of all the \_\_init\_\_ function:  The \_\_init\_\_ function should initialize a new object when a new instance of BoardSpot is called, here only the selected value is changed whereas the value and the fact that the instance is a mine is set through a class-level variable instead of an instance-level variable. This means that any change to the “mine” and “value” variables could change the values of those variables in every instance; to fix that they should both be instance-level variables in the \_\_init\_\_ function. The “selected” variable is in the \_\_init\_\_ function but is also a class-level variable which shouldn’t be the case either. This bug, also causes an inconsistency at the \_\_str\_\_ function which calls an instance-level “value” variable when it is in fact a class-level (which it shouldn’t be).    Screenshot : Screenshot simply showing the \_\_init\_\_ function without any instance-level variables but the “selected” variable. (I couldn’t get a better screenshot to demonstrate the bug explained above)  And finally the makeMove() function once more, the issue is already known but we’ll use the debugger to figure out where exactly the problem stems from:  Here I’ve decided to put the same coordinates twice in the test spots to demonstrate the fact that the same values can enter the function more than once. After a first iteration, it seems that the (0,2) spot is a mine.    Screenshot : First test coordinates (0;2) spot going through the makeMove() function.  Despite the fact that the program has already been over the (0;2) spot it checks it again thus decrementing the amount of selectable spots and returning the same output of the spot being a mine. This recurrent selection of the same spot could create unnecessary recursive calls and could even lead to an infinite loop. To avoid that a simple if-statement to check if the selected spot has already been selected is sufficient.    Screenshot 24: Screenshot demonstrating the reiteration of the function for the same value iterated previously. |