**1 A. List the class relationships shown on the UML Class Diagram in Fig, 2c and, for each of them, indicate its type and how it should be implemented in C++**

There are two types of relationships present. These are aggregation and composition.

The player and the score class have a composition relationship. The Player has a score. The whole class is the Player and the part class is the Score.

In order to implement this relationship, the Player class has an instance of the Score class as its data members. Then the Score (part) object is initialised in the constructor of the Player (part) class and destroyed when the Player class goes out of scope.

The game and the score class have an association relationship.

To implement this relationship, the Game class contains a pointer to the instance of the Player class. This instance is private in the Game class. In the constructor of the Game class, the Player's pointer is set to null. Also in the Game class, there should be a function which sets the Player instance. The parameter for this function is a player pointer which is used to initialise the values of the player instance in the Game class.

**1 B. List the classes which, according to the UML Class Diagram in Fig. 2d, inherit the message reset\_position and explain why this is possible.**

The classes that inherit the reset position are Mouse, Nut and Snake. This is possible the base class is the MoveableGridItem and the Mouse, Nut and Snake are the derived class which mean these 3 classes inherit all the public functions of the MoveableGridItem base class and since reset position is a public function, the 3 derived classes can inherit this.

**1 C. Using only the information provided in the UML Sequence Diagram in Fig. 3a, list the class relationships that must exist for the interactions represented on the diagram to be possible and, for each of them, explain how you know this. You do not need to indicate the nature of this relationship.**

The game class needs to talk to the mouse class because the Game class is telling the mouse class to run the function called can\_collect\_nut in which the parameter is a nut object.

This would mean that the Mouse class must talk to the Nut class due to the parameter which is a nut object.

The Game class also talks to the Nut class because according to the sequence diagram, the Game class is telling the Nut class to run the function disappear() if the got\_nut value is true

**D. Explain what the collected and on\_hole symbols used in the UML Sequence Diagram in Fig. 3b are, what information they represent and what their use in message j + 2 is**

The collected and on\_hole are both placeholders represented in the sequence diagram that store the returned boolean values from the 2 functions.

the on\_hole placeholder is set to true if the mouse is at the same co-ordinates as the hole. The Game talks to the Mouse class and tells the Mouse class to run the function has\_reached\_a\_hole(ug:Underground) accepting an underground object. if they both have the same co-ordinates, on\_hole placeholder is set to true.

This is the same for the Nut so if the mouse is at the same co-ordinate as the nut, the collected placeholder is set to true.

In message J+2, if both of these placeholders are set to true (i.e the nut has been collected by the mouse and the mouse has reached a hole), the Game class tells the Mouse class to run the function, escape\_into\_hole() so the mouse can exit the maze.

**1 E. List the #include directives that should be placed in the two files used for the Snake module (i.e., Snake.h and Snake.cpp) and explain why each of these directives is needed there.**

In the snake cpp file, there should be an “#include Snake.H”. The reason for this is that so that this allows the cpp implementation file to access the private and public declaration of the functions and the data members so that it can implement the functions declared and make use of the data members in the header file.

The #include “constants.h” file would be used to implement the symbol of the snake and #include “RandomNumberGenerator.h” file would be used to set the random position of the snake.

In the snake header file, there needs to be #include “RandomNumberGenerator.h” in order to create the random number generator variable as private. ASK ABOUT THE MICE HEADER DECLARATION IN THE SNAKE.H FILE. WHY DO WE NEED IT AND WHY IS IT WHEN WE REMOVE THAT FROM THE SNAKE HEADER FILE, THE PRIVATE MOUSE VARIABLE DOESN’T GIVE AN ERROR?

**1.F The C++ Snake::has\_caught\_mouse function is currently defined as follows in the skeleton application given. Describe all the improvement(s) that should be made to this definition and what other parts of the code will also need to be changed for the function to work as required, obey the OOP guidance and be safe: NO IDEA**

**1.G In the skeleton application given, list all the functions that would be called (directly or indirectly, explicitly or silently) if the C++ statements on line 1, line 2 and line 3 were to be added in some client code as follows. Explain why and describe what would happen.**

{

Snake snake; //line 1

Snake another(snake); //line 2

//...

} //line 3

**Snake snake; //line 1**

This will run the following functions:

* The snake constructor to create the snake object
  + random number generator constructor to create the rng object that is private
    - In the constructor, the next function is the seed() function which will initialise the pseudo random number generator
  + next, it will run the position\_at\_random() function in the snake class
    - in this function, it will run the get\_random\_value() function twice from the random number generator class to get random values for x and y.

**Snake another(snake); //line 2**

This will run the following functions

* the snake copy constructor implicitly since the snake class file hasn’t defined it.
  + random number generator constructor to create the rng object that is private
    - In the constructor, the next function is the seed() function which will initialise the pseudo random number generator
  + next, it will run the position\_at\_random() function in the snake class
    - in this function, it will run the get\_random\_value() function twice from the random number generator class to get random values for x and y.

**} //line 3**

* Will run the deconstructor for the 2 random number generator because lifelines are synced (the random number generator object is part of the Snake class so this will be destroyed with it )
* Snake deconstructor both for the copy snake and the snake on line 2 and 1 respectively

These will be run because the snake object has gone out of scope so will be destroyed.

PLEASE GET THIS CHECKED WITH JAMIE AND SEE IF ANYTHING IS MISSING

1.H

Advantages:

No need to have position at random function since the x and y co-ordinates of the snake are being initialised in the constructor having access to those private variables

Problems:

However, the random number generator constructor won’t have been run straight away. This will only happen when the symbol, x, y, and the mouse pointer have been initialised. This means that x and y value for the snake will not be random. Only time it will be random is when the set up function for the game is run due to the position at random function in that game function.

In other words, this will run the same function twice.

ANYTHING ELSE?

1.I

#3 is the best option because having a reference to the underground object means there is no unnecessary copy being made for that object. Also, using const keyword means that there is confidence that the underground object in that function won’t change the state of the object. For example, properties values cannot be changed.

In the mouse header, the prototype for that function needs to be changed to pass a constant underground reference object and make sure the function is constant.

Another change would be that is that the inner function is at position would be constant.

1.J