OOP – TASK 1a– SEMESTER 1 GROUP ASSIGNMENT

# OUTLINE BRIEF

You are required to demonstrate your understanding of the key principles of the OO approach and your competence in applying the C++ programming techniques presented in Semester 1 to implement a design documented in UML.

The questions for this assignment refer to the **Case Study #1** used in some of your lectures and tutorials on UML and C++. This document contains the list of additional requirements for the system together with relevant UML documentation. You are also provided with a partial C++ implementation of the system that is available from Bb (project and source files in **OOP\_Task1a\_Skeleton.zip** together with an electronic copy of this document).

You will need to study the C++ program provided and the related UML documentation so that you can:

1. answer questions about them (Question 1),
2. provide some additional UML documentation (Questions 2 and 8),
3. apply fundamental OO principles and techniques presented this semester to refactor the C++ implementation given (Question 3),
4. extend the C++ skeleton program so that it offers additional functionality (Questions 4 to 6 for basic version and Question 9 for extended version).

This is a **group assignment** that counts for **25%** of the CW mark for the OOP module.

Any change in group composition **MUST** be approved in advance by all the group members and the teaching team. **Unless agreed with me in advance**, individual solutions will be marked but cannot be awarded more than a pass mark (i.e., 40%).

# OBJECTIVES

* Demonstrate your understanding of basic elements of UML
* Apply some essential OO facilities offered by C++ and demonstrate programming skills required to implement a small OO software system
* Relate design documentation in UML with C++ implementation of a small OO software system
* Work effectively as a member of a project team on the development and maintenance of a small OO software system
* Produce a clear and comprehensive report on work produced.

# DEADLINES

**REPORT and WALKTHROUGH BASIC VERSION (Questions 1 to 8): Week 21**  
Submit code and documentation on Bb by **Tuesday** **12 December 2017**, **11am**Attend walkthrough sessions in that week *(time to be announced)*

**WALKTHROUGH FOR EXTENDED VERSION (Question 9): Week 25**Submit code on Bb by **Monday 8 January 2018**, **10am**Attend walkthrough sessions during *(time to be announced to fit in with your group project)*

# DELIVERABLES AND WALKTHROUGH

**NOTE: Up to 10% of the mark can be deducted** if your report presentation and/or structure of electronic files provided do not follow the instructions or if you do not attend the walkthroughs.

**BASIC VERSION**

1. Submit on Bb a zip file (called **OOP\_Task1aBasic\_*yourGroupNumber*.zip**) that contains all your work for the basic version (Questions 1 to 8).

* A group **report** containing your answers to Questions 1, 2 and 8.
  1. Answers to questions about the documentation and code provided (Question 1)
  2. The UML Sequence Diagram for the **apply\_rules** operations in the **Game** class (Question 2)
  3. The UML Class Diagram for the skeleton program (Question 8)

This report is to be presented on a work document. Use a footer to indicate on each page the page number, your assignment group number (e.g. L1, U3) and names of group members. Include pictures of any diagrams drawn by hand.

**UML diagrams**: You must adopt the diagram conventions presented in class or specified in this document. However, if necessary, you can use English to complement your explanations so long as they are clear and unambiguous. You can produce the diagrams by hand or use tools such as Visio, Rational Rose or Together to help you.

* Your code for the **basic solution** of your program (Questions 3, 4, 5, 6 and 7), including **all the source files** (.h and .cpp) together with the files for your **MS Visual Studio 2017 project** (.sln, .vsxproj and vsxproj.filters). Nothing else!

**C++ code:** At the top of each file log the changes you have made to the C++ application initially given to you and clearly document new files you have added. Your C++ program MUST work in MS Visual Studio 2017 and run on the **SHU network**.

1. Contribute to a **walkthrough** for (i.e. demonstration of) the **basic version** of the C++ program. It will be arranged during week 21.

**Walkthrough**: It contributes to the final mark for this assignment and involves you demonstrating your program to your tutor, describing what it does and how it works. You will have the chance to point out interesting or innovative aspects of your work. You must be able to answer questions about your design and your code and made changes to it on request to modify some of the program behaviour.

**EXTENDED VERSION**

1. Submit on Bb a zip file (called **OOP\_Task1aExtended\_*yourGroupNumber*.zip**) that contains all your work for the extended version of your program (Question 9).
2. Contribute to a **walkthrough** for the **extended version** of the C++ program (Question 9). It will be organised during the inter-semester break.

# SYSTEM REQUIREMENTS

The skeleton program for this assignment (in **OOP\_Task1a\_Skeleton.zip**) implements a small 'game' where a snake chases a mouse on a grid and tries to eat it. There are holes on the grid in which the mouse can escape.

1. The mouse is represented by the 'M' character, the snake by 'S' and each of the holes by 'H'.
2. It includes some basic functionality, such as the display of a grid with the snake, the mouse and three holes, the user can move the mouse up and down on the grid (but not outside the grid boundaries), the snake automatically chases the mouse, the mouse disappears when getting into one of the holes, the user can quit the program.
3. At the start of the game, the mouse is placed at the centre of the grid, the holes are given fixed positions and the snake is given random coordinates.
4. The game ends when the player presses the 'Q' key, when the snake eats the mouse or when the mouse escapes into a hole. An appropriate message is displayed to explain why the game ended.

The **UML Use case** in **Fig. 1** describes the top level functionality of this skeleton program.

Classes such as **Mouse**, **RandomNumberGenerator** and **Game** have been identified in the initial phases of the system design and some of them have already been (partially but often badly) implemented in the skeleton provided.

Your task is to improve the program given, provide some UML documentation and add new functionality to it.

# QUESTIONS

1. Study the UML documentation provided and the (partial) C++ code given carefully and, answer each of the following questions.

(20 marks - 2 marks each)

**NOTE:** Marks will be awarded for correctness, completeness and overall clarity of the explanation.

1. List the class relationships shown on the UML Class Diagram in **Fig. 2c** and, for each of them indicate its type and describe how it should be implemented in C++.
2. List the classes which, according to the UML Class Diagram in **Fig. 2d**, inherit the message **reset\_position** and explain why this is possible.
3. 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 these relationship.
4. Explain what the **collected** and **on\_hole** symbols used in the UML Sequence Diagram in **Fig. 3b** are, what information they represent and what is their use in message **j+2**.
5. 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.
6. 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:

bool Snake::has\_caught\_mouse() {

return is\_at\_position(p\_mouse\_->x\_, p\_mouse\_->y\_);

}

1. 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

1. Explain what would be the advantages and problems (if any) with redefining the Snake::Snake() constructor as:

Snake::Snake()

: rng\_(),

symbol\_(SNAKEHEAD),

x\_(rng\_.get\_random\_value(SIZE)),

y\_(rng\_.get\_random\_value(SIZE)),

p\_mouse\_(nullptr)

{}

1. Indicate which of the following definitions of the C++ Mouse::has\_reached\_a\_hole function is the most desirable one and explain why, indicating changes (if any) needed to make it work:

bool Mouse::has\_reached\_a\_hole(Underground ug) const {//...} //#1

bool Mouse::has\_reached\_a\_hole(Underground& ug) const {//...} //#2

bool Mouse::has\_reached\_a\_hole(const Underground& ug) const {//...} //#3

bool Mouse::has\_reached\_a\_hole(const Underground& ug) {//...} //#4

1. Since the holes never change position the three instances of the Hole class in the Underground class should be made constant. Explain what implications this would have on the definition of the Underground class’s constructor(s).
2. In the C++ skeleton program, the **Game** class has an operation called **apply\_rules**.  
   **Draw the Sequence Diagram** for (part of) this operation. It should start with the **apply\_rules** message sent to the **game** instance and represent all the interactions that take place in the system when the mouse escapes into hole #1.

(5 marks)

Marks will be awarded for:

1. overall clarity of the diagram
2. showing all the instances involved in the interaction (class identifiers and instance names)
3. showing construction and deletion of any temporary instances (if any)
4. showing all the messages involved in the interaction
5. correctly labelling the messages and showing their direction
6. showing both the type of the parameters and return value of any message
7. indicating the numbering (using the nested numbering system presented in class) and nesting of the messages activation boxes
8. placing guards where needed
9. Make use of relevant C++ OO facilities presented in Semester 1 to improve the C++ program given and in subsequent additions/amendments you make to this skeleton program. These include the following tasks:

**NOTE:** To get the marks for Question 3 you need to amend the skeleton program provided and carry on applying these guidelines/instructions to the C++ code produced for all other questions.

**HINT:** Tackle these tasks in the order given as some of them depend on each other. For example task f can only be successfully achieved after the changes for tasks c, d and e have been made.

1. Store each class in the system in a separate module (named after the class).  
   Add all the required include directives where appropriate - but only those that are necessary.

(3 marks)

1. Make sure each data item used in the program is declared in an appropriate scope.
2. Remove all global variables.
3. Use local variables when possible.
4. Ensure that any class data class member used is necessary and part of the class structure (i.e., wherever appropriate, use local variables to member functions instead of class data members).
5. Place constant declarations in the modules that need them.

(3 marks)

1. Declare class members as private wherever possible:
2. Make all data members private to provide encapsulation and, if necessary, provide public accessor or mutator functions to provide access to that data.
3. Declare function members as private when only used as support functions in their class.

(2 marks)

1. Create and use well-defined constructors.
2. Use with explicit initialisation list - whenever appropriate.
3. Make conversion constructors explicit.
4. Remove constructors that are not needed (i.e., not used or automatically provided when the compiler built-in version is sufficient).

(4 marks)

1. Use static class members where appropriate:
2. Make the **RandomNumberGenerator** data member in the **Snake** class a **static** instance.
3. Use **static** member functions where possible and appropriate.

(2 marks)

1. Declare class member functions as const (wherever possible and appropriate) and apply appropriate parameter passing and return value to communicate data across functions
2. Prefer const reference or const pointers parameters for class instances used as input parameters (instead of 'passed-by-value')
3. Return const value when appropriate.

(3 marks)

1. Declare class attributes and instances as const (wherever possible and appropriate).

(2 marks)

1. Add the operator<< and operator>> functions to serialise instances of the Game class. These could be used (e.g., from the main function) to respectively read simple data about the game (i.e., at least snake’s and mouse’s positions) from a text file called **Game.txt** and store it back into that text file as shown below:

//for output: save game into file

ofstream fout;

fout.open("Game.txt", ios::out);

if (fout.fail())

cout << "\nAn error has occurred when opening the file.";

else

fout << game; //insertion operator<< for Game instances

fout.close();

//…

//for input: read game from file

ifstream fin;

fin.open("Game.txt", ios::in);

if (fin.fail())

cout << "\nAn error has occurred when opening the file.";

else

fin >> game; //extraction operator>> for Game instances

fin.close();

(2 marks)

1. Identify and remove any unnecessary dependencies between the Hole class and other classes in the system. Only the **Underground** class should know about and manage the holes (i.e., no other class should be aware of the Hole class).

(4 marks)

1. Explain why it would not be a good idea to remove the direct relationship between the Snake and the Mouse classes.

(1 mark)

1. **Modify the C++ program given** so that, the snake has a tail made of 3 cells represented by the symbol 'T'. To start with the tail is ‘coiled’ under the head but as the snake moves its tail appears, following the head.   
   To do this, you need to **modify the Snake class**, i.e., use a vector to store the tail and add (at least) the get\_tail\_symbol and move\_tail functions to respectively provide a copy of the tail symbol and make the tail follows the snake head at all time.   
   Your implementation should match the **UML Class model of the Snake class in Fig 2b** and the (partial) **UML** **Class Diagram** in **Fig. 2a**.

(4 marks)

Marks will be awarded for:

* ensuring that your implementation matches the UML Class Model given (e.g., attributes identifiers and types and operations' identifiers, return type (if any) and type of parameters (if any), level of access).

1. **Modify the C++ program given** so that there is a nut on the grid in position (8, 9) represented by the 'N' character. The mouse collects the nut by walking over it. When the nut has been collected it disappears. The mouse needs to collect the nut before it can escape in one of the holes.   
   To do this you need to **add a new class** called **Nut**.

Your implementation should match the **UML Class model of the Nut class in Fig 2b** and the (partial) **UML** **Sequence Diagrams** in **Fig. 3a and 3b.**

(6 marks)

Marks will be awarded for:

* ensuring that your implementation matches the UML Class Model given (e.g., attributes identifiers and types and operations' identifiers, return type (if any) and type of parameters (if any), level of access).
* ensuring that your implementation matches the (partial) UML Sequence Diagrams given (e.g., develop new public member functions in the Mouse class such as can\_collect\_nut to match corresponding messages in **Fig 3a**.)

1. **Modify the C++ program given** so that the program records some information about the current player and his/her score.  
   The system asks for the player’s name at the beginning of the game and sets the initial score to 0.   
   The game can be played several times during the same session. The player' score increases by one each time the mouse escapes and decreases by one when the snake catches the mouse before it dies or the mouse can escape. The player's name and current score are displayed just below the grid at all times. At the end of each individual game the player is asked whether he/she wants to continue. If the user enters the letter 'Y' or 'y', the game starts again with the same player. If he/she enters any other letters the session ends.  
   To do this, you need to **add two new classes to the program** to represent the player of the game and his/her score.   
   Your implementation should match the specification given above and the (partial) **UML Class Diagram** for the **Player** and **Score** class in **Fig. 2c**.

(5 marks)

Marks will be awarded for:

* ensuring that your implementation matches the description of the functionality given above.
* ensuring that your implementation matches the UML Class Model given (e.g., attributes identifiers and types and methods’ identifiers, return type (if any) and type of parameters (if any), level of access and relationships between the two classes).
* respecting natural role and responsibility for each class(e.g., handling all interactions with the user of the system (input and output) through the Interface class).

1. Modify the program so that a **hierarchy of GridItem classes** is developed and used to capture the common features to all items represented in the grid.   
   Your implementation should match the description of the various objects shown on the grid and the **UML Class Model** for the **GridItem** hierarchy of classes in **Fig. 2d**.

(10 marks)

Marks will be awarded for:

* ensuring that your implementation matches the UML Class Diagram given (e.g., attributes identifiers and types and methods' identifiers, return type (if any) and type of parameters (if any), level of access and relationships between the classes).
* implementing the various grid items with static inheritance– using public derivation and redefining inherited functions where appropriate.

1. **Produce the UML Class Diagram(s) for the C++ program you have produced for questions 3 to 7**. This needs to include:

* The complete class models of the **Mouse**, the **Snake** and **Game** classes (with the details of their attributes and operations)
* A class diagram with all the classes in the system (represented just by their icon – a box with their names) and their relationships (including weak dependencies).

An incomplete diagram is givenin **Fig. 2a**.

(5 marks)

**NOTE:** You can create independent class models for each class in the system (with details of their attributes and operations) and represent the relationships between the classes in a separate diagram showing only the class icons (i.e., boxes labelled with the class names) and the links between them.

Marks will be awarded for:

* showing ALL the class models with their identifiers, all their attributes and methods
* indicating both the type of the parameters and return value of the methods as well as the type of the attributes
* indicating the level of access of any class member (**+** for public and **-** for private members)
* showing ALL the relationships between classes using appropriate links (e.g., diamonds arrows for aggregation, clear head arrow for specialisation, dashed line arrows for simple dependency) and labelling these links when appropriate.

NOTE: If a software tool is used to create the diagrams, and the tool follows a different convention for showing access levels, a key should be provided with the diagram.

# EXTENSIONS

1. Apply all of these techniques and principles when implementing some of the following additional functionality:

**NOTE**: You do not have to do all of these to get the **maximum 20 marks** associated with this question.   
This question will be assessed during a walkthrough. You will need to hand in an electronic copy of your code to your tutor during the walkthrough. Up to a maximum of **10 marks** may be deducted if you do not contribute to the walkthrough or produce the required electronic files.

(4 marks each, up to a maximum of 20 marks)

1. At the beginning of each game the holes are placed on the grid in fixed positions. The mouse, nut and snake are placed at random on the grid, in such a way that they do not overlap between themselves and that they do not cover any holes. The holes remain at their fixed positions throughout, the snake chases the mouse and, now, the mouse needs to collect the nut before it can safely escape in one of the holes. The dimensions of the grid should be set as constants in your application and the program should work as expected even if these dimensions are amended (to sensible values).
2. The player can put the game in and out of ‘cheat’ mode at any time by pressing the 'C' key. To start with the cheat mode is off. When the ‘C’ key is pressed, if the game was in normal play, it goes on cheat mode, i.e., the snake stops moving but the rest of the game continue as normal. When the same key is pressed again, the snake can move again and the game resumes as normal again. A “CHEAT MODE ACTIVATED” message is displayed while the game is on ‘Cheat’ mode. When the cheat mode has been used in one game, the score for that game will not be recorded.
3. When the mouse enters a hole, it automatically disappears (into the underground) and randomly reappears out of another hole on the grid (i.e., the ‘exit’ hole is selected randomly, but cannot be the same as the ‘entry’ hole). The snake notices and tries to resume the chase. If this means backtracking on its own tail, it may be temporarily stuck until the mouse is in a position that allow it to move again towards it.
4. The mouse can no longer walk over the nut, but it can push it from behind. To escape the mouse needs to push the nut into one of the holes (instead of collecting it) then follow it into the hole. When pushed onto a hole, the nut disappears. To ensure that the mouse is always able to push it, the nut must remain at least one space away from the wall at all time. That is, it must not initially be set right next to the wall, even when it is randomly placed (as in Question 9a) and the mouse should not be able to push the nut too near the wall.
5. The player can save and reload the game at any time by pressing the 'S' or 'L' keys respectively.   
   When the ‘S’ key is pressed, game data (moveable items' coordinates and player's score) is saved in a file named after the player (with the **.txt** extension). If a file for that player already exists, its contents is overwritten. If this **.txt** file didn’t not exist, it is automatically created.   
   When the ‘L’ key is pressed, previously saved data for this player is reloaded. If there isn’t an appropriate **.txt** file available, an appropriate message is displayed ("NO SAVED GAME AVAILABLE") and the game remains unchanged.
6. The player can ‘undo’ the effects of the last arrow key pressed (i.e., the mouse and snake's moves and any related side effects) at any time by pressing the 'U' key. This command can only ‘un-do’ one move, the last one (if any). If nothing can be undone (at the start of the game or because an 'undo' operation has already taken place) an appropriate message ("NO UNDO POSSIBLE") should be displayed.
7. Any other interesting feature you may think about.

**NOTE**: Please check with your tutor that it is challenging enough to be worth 4 marks.

**NOTE**: Please check with your tutor that it is challenging enough to be worth 4 marks)

**NOTE**: Please check with your tutor that it is challenging enough to be worth 4 marks)

Marks will be awarded for:

* ensuring that your C++ implementation follows the guidelines provided for the basic versions and use the techniques presented in the module,
* producing a clear OO design (e.g., reuse of existing methods, appropriate level of access to member functions, avoiding unnecessary dependencies between classes, appropriate allocation of responsibility to classes and appropriate implementation of relationships between classes, natural class public interface),
* following the design guidelines given in lectures (e.g. implement specialisation (is\_a) relationships with public inheritance, association (knows\_a) relationship with pointers to instances and composition (has\_a) relationship with instance data member),
* respecting natural role and responsibility for each class (e.g., handling all interactions with the user of the system (input and output) through the Interface class),
* producing a robust system by checking that possible problematic situations are dealt with appropriately (e.g., trying to undo when the game has not yet started, by reusing code where possible and by avoiding hard-coded values such as grid size or hole coordinates) and
* producing readable code (e.g., consistency of style, comments, meaningful identifiers, program layout).

# APPENDIX 1: USE CASE DEscription

**"Top level" Use Case**

1. The system shows the grid with the grid items displayed on it.
2. The system waits for a key
3. The user presses an arrow key.
4. The system updates the position of the mouse and moves it in direction of the key (i.e., up, down, left or right) when possible.
5. The system updates the position of the snake to make it get closer to the mouse (it can move in diagonal).
6. The system redraws the grid and goes back to step 2.

**Alternate course #1:**

at 3:

* 1. The user enters the 'Q' key.
  2. The system indicates that the user ended the game.

1.3 The system ends the application.

**Alternate course #2:**

at 3

2.1 The user enters a key that is not recognised (i.e., different from an arrow key or the 'Q' key).

2.2 The system goes back to step 2.

**Alternate course #3:**

at 4

3.1 The move requested is not possible (i.e., pushing mouse out of the grid boundaries) the system goes directly to step 5.

**Alternate course #4:**

at 4

4.1. The system updates the position of the mouse and recognises that the mouse can escape underground into a hole.

4.2 The mouse disappears.

4.3 The system indicates that the mouse has escaped.

4.4 The system ends the application.

**Alternate course #5:**

at 5

5.1. The system updates the position of the snake and recognises that the snake has caught the mouse.

5.2 The system indicates that the mouse has been eaten.

5.3 The system ends the application.

Fig. 1: Use Case describing top level interaction between the user and the system.

# APPENDIX 2: Class Diagram

Snake

Game

-snake

-mouse

-rng

*pursues*

Mouse

- x: Integer

- y: Integer

- symbol: Character

**<<construct>>**

+ new\_mouse()

**<<query>>**

+ get\_x(): Integer

+ get\_y(): Integer

+ get\_symbol(): Character

+ is\_at\_position(x: Integer, y: Integer)  
 : Boolean

+ is\_alive(): Boolean

+ has\_escaped(): Boolean

+ has\_reached\_a\_hole(ug: Underground)

: Boolean

**<<update>>**

+ dies()

+ escape\_in\_hole()

+ scamper(k: Character)

- position\_in\_middle\_of\_grid()

- updatePosition(x: Integer, y: Integer)

RandomNumberGenerator

Interface

*communicates through*

Fig. 2a: (Incomplete) Class Diagram of the system.

Nut

- x: Integer

- y: Integer

- symbol: Character

- collected: Boolean

**<<construct>>**

+ new\_nut(x: Integer, y: Integer)

**<<query>>**

+ get\_x(): Integer

+ get\_y(): Integer

+ get\_symbol(): Character

+ has\_been\_collected(): Boolean

+ is\_at\_position(x: Integer, y: Integer)  
 : Boolean

**<<update>>**

+ disappears()

Snake

- x: Integer

- y: Integer

- symbol: Character

- rng: RandomNumberGenerator

- pmouse: Mouse\*

- tail: vector<???>

+ get\_x(): Integer

+ get\_y(): Integer

+ get\_symbol(): Character

+ is\_at\_position(x: Integer, y: Integer)  
 : Boolean

+ spot\_mouse(m: Mouse)

+ has\_caught\_mouse(??): Boolean

+ chase\_mouse(??)

- position\_at\_random()

- move\_tail(??)

- set\_direction(dx: Integer, dy: Integer)

//…

Fig. 2b: (Partial) Class Models of the Nut and Snake classes.

Game

*is played by*

-score

Score

- amount: Integer

**<<construct>>**

+ new\_score()

**<<query>>**

+ get\_amount(): Integer

**<<update>>**

+ update\_amount(a: Integer)

Player

- name: String

**<<construct>>**

+ new\_player()

**<<query>>**

+ get\_name(): String

+ get\_score\_amount(): Integer

**<<update>>**

+ update\_score\_amount(a: Integer)

*In* ***Player***

***name*** *should be constant*

Fig. 2c: (Partial) Class Diagram showing the Player and Score classes.

- collected: Boolean

//...

//...

//...

- alive: Boolean

- escaped: Boolean

+ new\_mouse()

+ is\_alive(): Boolean

+ has\_escaped(): Boolean

+ has\_reached\_a\_hole(??): Boolean

- position\_in\_middle\_of\_grid()

//...

- tail: vector<MoveableGridItem>

//...

+ new\_snake()

+ has\_caught\_mouse(??): Boolean

+ chase\_mouse(??)

- position\_at\_random()

- move\_tail(??)

//...

GridItem

- symbol: Character

**<<query>>**

+ get\_symbol(): Character

//...

- x: Integer

- y: Integer

**<<query>>**

+ get\_x(): Integer

+ get\_y(): Integer

+ is\_at\_position(x: Integer, y: Integer): Boolean

**<<update>>**

+ reset\_position(x: Integer, y: Integer)

+ update\_position(dx: Integer,

dy: Integer)

//...

Mouse

Snake

Nut

Hole

- x: Integer

- y: Integer

**<<query>>**

+ get\_x(): Integer

+ get\_y(): Integer

+ is\_at\_position(x: Integer, y: Integer):

Boolean

//...

FixedGridItem

*In* ***FixedGridItem***

***x*** *and* ***y*** *should be constant*

MoveableGridItem

*In* ***GridItem***

***symbol*** *should be constant*

Fig. 2d: (Incomplete) Class Model of the GridItem hierarchy of classes.

# APPENDIX 3: SEQUENCE DiagramS

game   
: Game

apply\_rules()

mouse   
: Mouse

i. got\_nut:= can\_collect\_nut(n: Nut): Boolean

nut   
: Nut

i+1. [got\_nut] disappear()

*Possibly some other interactions here...*

*Possibly some other interactions here...*

Fig. 3a: (Partial) Sequence Diagram showing what happens when the mouse collects the nut.

game   
: Game

apply\_rules()

mouse   
: Mouse

j. on\_hole:= has\_reached\_a\_hole(ug: Underground): Boolean

nut   
: Nut

j+2. [collected and on\_hole] escape\_into\_hole()

j+1. collected:= has\_been\_collected()  
 : Boolean

*Possibly some other interactions here...*

*Possibly some other interactions here...*

Fig. 3b: (Partial) Sequence Diagram showing what happens when the mouse escapes on the hole after having collected the nut.

# APPENDIX 4: FEEDBACK AND MarkING ShEET

|  |  |  |  |
| --- | --- | --- | --- |
| **GROUP**  **NUMBER** |  | **NAMES:** |  |

## BASIC version

|  |  |  |  |
| --- | --- | --- | --- |
| **1. Questions on documentation provided** | | **/20** | |
| **2. UML Sequence Diagram** starting with the **apply\_rules** message when the mouse escapes into hole #1. | | **/5** | |
| **3. Use of relevant C++ OO facilities to improve the C++ program given and amendments** | | | |
| a) Separate module for each class |  | **/3** | |
| b) Use appropriate scope for all data items used |  | **/4** | |
| c) Appropriate level of access (private or public) for class data and function members |  | **/2** | |
| d) Use static class members |  | **/2** | |
| e) Correct constructors - with initialisation list |  | **/4** | |
| f) Appropriate use of const class member functions, const reference and pointer parameters and const return value |  | **/3** | |
| g) Appropriate use of const class attributes and instances |  | **/2** | |
| h) Add Game public operator << and operator >> |  | **/2** | |
| i) Removal of unnecessary class dependencies with Hole class |  | **/4** | |
| j) Justify keeping relationship between Snake and Mouse classes | *\* In preparation for question at walkthrough* | **/0\*** | |
| **4. C++ - with modified Snake class** | | **/4** | |
| **5. C++ - with Nut class** | | **/6** | |
| **6. C++ - with Player and Score classes** | | **/4** | |
| **7. C++ - with inheritance for a hierarchy of GridItem classes** | | **/10** | |
| **8. UML Class Diagram for the C++ program you have produced**. | | **/5** | |
| **Presentation of report and code** | |  | |
| **OTHER COMMENTS** (See front page of report and general feedback provided)  **SUB TOTAL MARK** | | **/80** |

## Extended version

|  |  |  |
| --- | --- | --- |
| **9.** Apply all of these techniques and principles when implementing one or more of the following additional functionality | | |
| a) The snake, mouse and nut are placed at random on the grid at the beginning of each game. They should not overlap between themselves or cover any of the holes which remain at their fixed positions. |  | **/4** |
| b) The can be put in and out of ‘cheat’ mode at any time by pressing the 'C' key. A “CHEAT MODE ACTIVATED” message is displayed while on ‘Cheat’ mode |  | **/4** |
| c) When the mouse enters a hole it automatically disappears into underground and randomly reappears out of another hole on the grid (i.e., the ‘exit’ hole is random, but not same as ‘entry’ hole). |  | **/4** |
| d) To escape the mouse needs to push the nut into one of the holes (instead of collecting it) then follow it into the hole. When pushed onto a hole the nut disappears. The nut cannot touch a wall at any time. |  | **/4** |
| e) The player can save and reload the game at any time by pressing the 'S' or 'L' keys respectively. Only one version of the game can be saved at any time. If no saved game available, a suitable message is displayed. |  | **/4** |
| f) The player can ‘undo’ the effects of the last arrow key pressed (i.e. both the mouse and snake move and any related side effects) at any time by pressing the 'U' key. Only un-do the last move (if any). If no undo possible, a suitable message is displayed. |  | **/4** |
| g) Other interesting feature you may think about… |  | **/4** |

**NOTE**: You do not have to do all of these to get the **maximum 20 marks** associated with this question.   
This question will be assessed during a walkthrough. You will need an electronic copy of your code to hand in during the walkthrough.

|  |  |
| --- | --- |
| **OTHER COMMENTS**  **SUB TOTAL MARK** | **/20** |