School of Science

Programming Fundamentals Assignment 2B (Group of 2)

2017 Semester 1

Monster-Chase-Player-Game (25 marks)

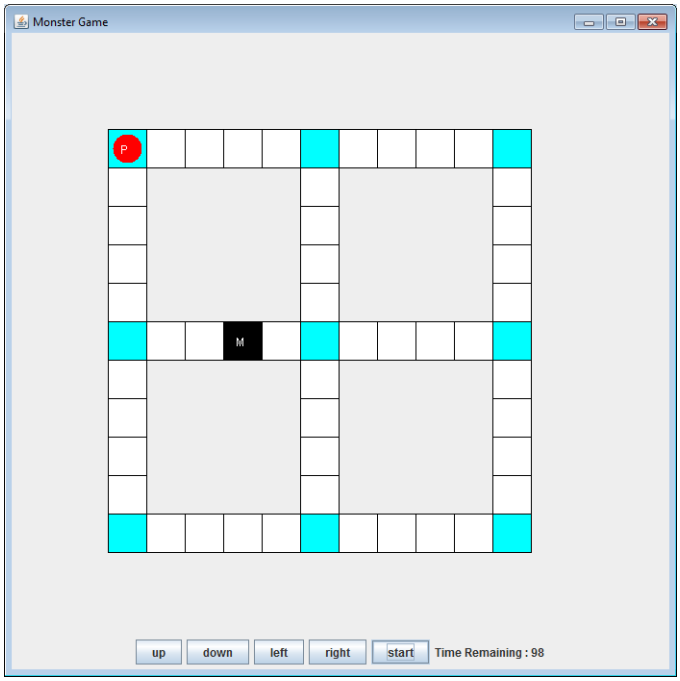
1. Objective

The main objective of this game based assignment is to familiarize students with object oriented design and programming. Game based assignments help to illustrate some of the object-oriented concepts such as inheritance and polymorphism in a fun and visual way. The game rules reflect the complexities present in real-life projects. Students are encouraged to create extensible, flexible designs by making use of encapsulation, inheritance, polymorphism, composition, abstract classes and interfaces.

The team members (exactly two) for this group assignment must be from the same tutelab. The expected breakdown of work between team members is outlined below. In your final submission include a report that highlights the extent of contribution made by each member, lessons learnt and any reflection on teamwork. This assignment will require the use of GUI class (but some start-up code is given), and you are given some freedom to change the suggested capabilities to make the game more playable and enjoyable.

2. Overview

In the stand-alone Monster-Chase-Player-Game, the player's aim is to stay alive for the required period by outrunning, escaping, jumping-over or setting traps. The monster(s) too may have many tricks such as alternating between visible and invisible periods and leaping (jumping along a row or column). The users should be allowed to combine any player capability with any monster capability at run-time to have different gaming experiences. This group assignment requires one member to develop classes and interfaces needed for player functionality while other member work on monster related functionalities. Both members are expected to work together on common functional and non-functional aspects. Marks are allocated for following good OO principles, meeting functional requirements and playability.



3. Design of Assignment Tasks and Expected Learning Outcomes

This section briefly highlights how this assignment meets course specific learning/teaching objectives:

I. User Input (Event Handling)

Handle mouse and/or keyboard events:

- Use button or key events to control player movements
- Other appropriate buttons to start, freeze, replay, escape etc.

II. Varying Player/Monster Capabilities (Algorithms, Reuse)

Allow different capabilities for Monster and Player objects:

- Analyse possible options of opponents and devise algorithms to counter them
- Use existing behaviour and state information from base classes

III. Combine different Monster/ Player Capabilities (Dynamic Binding)

Use of appropriate OO techniques that promote dynamic binding:

- Allow users to choose required behaviours/capabilities at runtime
- Use polymorphism, inheritance, composition to facilitate dynamic binding

IV. Improved Game Experience (Usability, Playability)

- Allow different skill levels by varying game duration and update frequency (delay player monster updates using Thread.sleep())
- Allow games to be paused, saved and retrieved.

V. Tracking Registration Details and High Scores in a Database

- Store and retrieve registered user records and high score details

VI. Exception Handling (Error Handling and Input Validation)

Improved error/exception handling:

- Propagate exceptions when it cannot be handled by a method/class
- Printing appropriate error messages (such as when data files do not exist)
- Verifying user input is in the expected range

VII. Group Project (Team working skills, Testing, Debugging)

Team design, task delegation, unit testing and documentation:

- Come up with designs that allow low coupling and high cohesion
- Develop detailed classes/methods and refactor as necessary
- Develop test harnesses to facilitate unit testing (not necessary to use JUnit)
- Develop good coding style/documentation (comments)
- Develop unit and mock tests to facilitate smooth integration.
- Logging and/or printing out status information

4. Dynamic Game Play Options

You are presented with the skeleton code for a simple monster game. The design for the skeleton code is presented in the class diagram attached. You are expected to use inheritance, composition abstract classes and interfaces to incorporate additional capabilities. At the beginning of the game, the desired additional capabilities for player and monster must be specified. If only one or two capabilities are selected, the player can choose the additional player and monster capabilities (must be the same number as the player).

5. Detailed Assignment Requirements

Playing the Game

You are presented with the skeleton code for a simple monster game. The design for the skeleton code is presented in the class diagram attached. You are expected to use inheritance, composition and abstract classes to allow additional capabilities to be provided. At the beginning of the game, the desired additional capabilities for player and monster must be specified. If only one or two capabilities are selected, the player can choose the additional player and monster capabilities (must be the same number as the player).

Player Extensions Required

Player needing energy to move

- Player needs 2 calories of energy to make each move.
- Moving over golden nougats found in each cell provide 6 calories. Nougats exist in all the cells in the beginning.
- Player starts with 40 calories.
- Player cannot move when it runs out of energy (just wait to be eaten-up or time-up)
- Player wins if player survives the game duration
- Allow the initial calories, calories needed for moving and calories for nougat to be varied.

1. Capability to skip over multiple cells

- Allow player to skip over multiple cells by pressing the (same) button key repeatedly within the stipulated time slot. But maximum number of cells is limited to three. Each additional step in a move takes up double the energy (2 for 1 step move, 2+ 4 for 2-steps move, 2+4+8 for 3-steps move)

2. Capability to put traps (all rules for 1 apply)

- Allow player to put a trap in its current cell. If a monster moves into that cell it will be prevented from moving or eating anything for 5 time units (from the time monster moved into the active trap). A player can move over a trapped monster. Putting a trap costs 50 calories and will last for 10 time units. A player cannot place a new trap while the previous trap is still active.

Monster Extensions Required

Smart Monster with better chase strategy

- Considering the energy level and remaining time
- Consider player capability; for example if the play has the ability to lay traps, assume the monster can query the board about the distance to the nearest active trap.

Additional Monster Capabilities

1. Invisible capability

- Allow monster to toggle between visible and invisible for periods between 5-10 units of time.

2. Leaping capability

- This monster can leap and eat any player in its sight (horizontal and vertical)

Group Features Required

These additional features are related to usability, file and exception handling.

- Allow users to specify player and monster types needed for the game
- Allow user to vary game parameters (such as the timeslot and the time duration)
- Allow game to be paused, restarted and played again.
- Allow user registration (name, address, username and password
- Allow user to login specifying username and password
- Track scores (wins, losses)
- Save and retrieve user details and score history
- Allow use of keys (instead of buttons) when appropriate
- Allow game state to be saved and retrieved

6. Assignment Demos, Submission and Marks Allocation

The overall design is to be done jointly by both partners. It is expected one member will implement the Player related functionality while the other does the Monster related ones. Utility classes (such as Cell and Grid) must be shared by all team members. Each member should be able to demonstrate their contribution to Player and Monster classes and utility classes for progress marks.

Bonus Marks (up to 3)

- Allowing options for more complex grids
- Saving and restarting the game.

No bonus marks will be allowed if basic features are not completed.

Marks Allocation
Contribution Declaration (See appendix 2)
Each member will be required to sign a document stating their overall contribution and the classes, design or functionalities they contributed.

Week	Marks Allocation
10 (Lab)	Team progress mark for having subdivided the work and made substantial progress. You are required to show the progress made by each student in developing his or her classes – **2 marks** for the whole team but all members must be present to get this mark.
12 Demo (book a 30 mins slot)	Individual Marks based on two complete classes and two main functionalities **(3 x 4) = 12 marks** (based on code & explanation) Group Marks based on completeness, correctness, error validation, robustness, usability and good teamwork/research **(6 marks)** Feature based Bonus Marks – **up to 3 marks**
Week-12 Sunday 11.59pm May 28th Final Submission through BlackBoard.	Marks based on refactored code, improved design diagrams, clear logic and documentation (including lessons learnt and personal reflection on teamwork and how the process can be improved) **(5 marks)** **Leadership bonus up to 10%** of the overall mark for anyone making 60% or more contribution in the project – this is based on the contribution percentage signed by all members (standard 50 % for 2 members)

So How do we Get started?

A good starting point is to load and run the supplied starting code in BB under 2017 Assignments (Assign2BStartupcode) to have some fun (Note the monster is chasing along the shortest path). Then study the UML diagrams and the code attached.

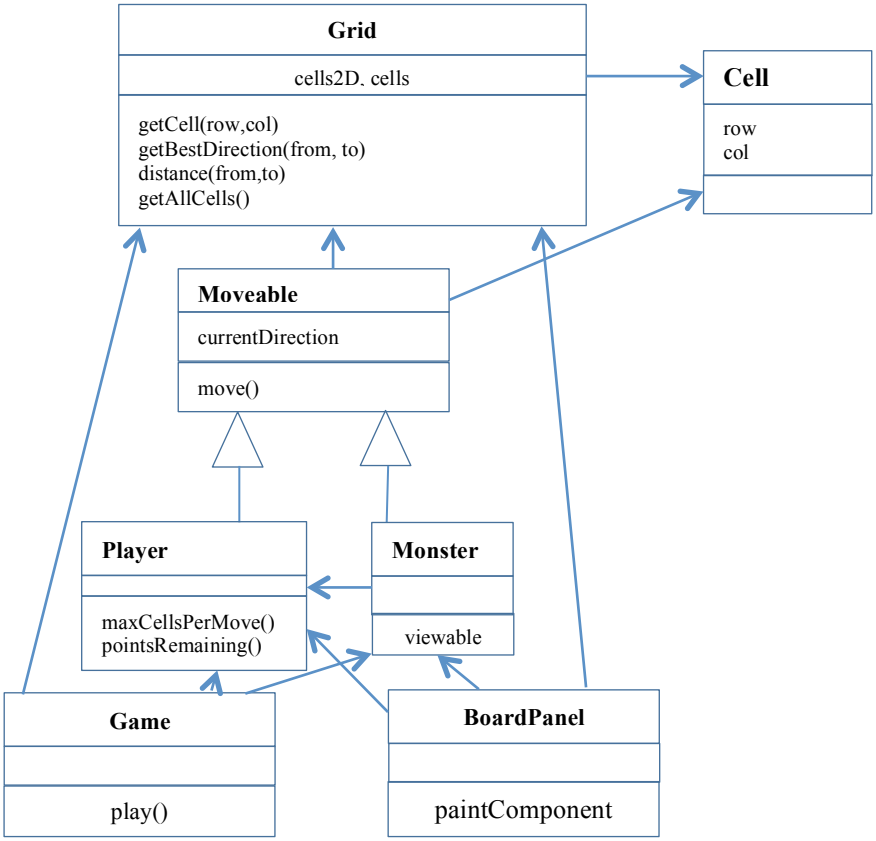
Appendix I

Contribution Form to be signed and submitted with the Assignment

Your contribution should be based on the outcomes (mainly) and the hours spent.

	Student Number	Name and Signature	Contribution Design/Leadership	Contribution Programming	Contribution Testing	Contribution Overall (%)
1						
2						
3						

Appendix II (Class Diagram) for the Skeleton Code



This design is intentionally made simple to help you get started. You are expected to add/alter classes, methods, associations and attributes as you implement other required functionality and refactor the design.