Assignment 3: Maze Game

Due in two parts (see last section for detail; see website for dates).

- Submit deliverables to CourSys.
- Late penalty:
  - Phase 1 (design): 10% per calendar day (each 0 to 24 hour period past due), max 2 days late.
  - Phase 2 (implementation): No possibility of late submission (solution posted for midterm).
- This assignment is **expected to be completed in pairs** (you *may* complete it individually, but that is not recommended). Doing OOD in a team setting is great practice.
- Do not show other pairs/students your code, do not copy code found online. Please ask all questions via course discussion group on `piazza.com`. I suggest making your post public if a general question, private if it includes your code.
  - You may use *ideas* you find online and from others, but your solution must be your own.
- See the marking guide for details on how each part will be marked.

1. Game Description

Create a maze exploration game where the player controls a mouse in search of some pieces of cheese to win the game. However, there are some blind cats in the maze who will eat the mouse if possible. Whenever the player reaches the square with cheese in it, the mouse (player) automatically collects the cheese and another cheese appears somewhere in the maze. If a cat and mouse (player) ever occupy the same square, the player loses the game. But, since the cats are blind, they cannot track the mouse: cats move pseudo-randomly.

The player sees the maze from a top-down view. At the beginning of the game, the maze is hidden from the player, but as the player moves through the maze, squares adjacent to the player are revealed in all 8 directions.

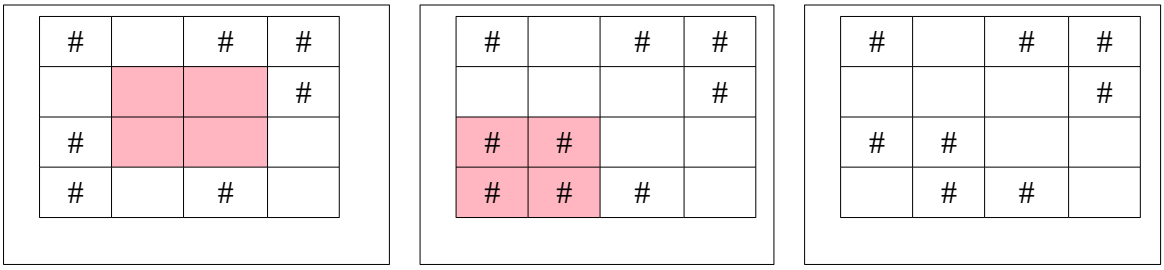
To the right is a sample display of the maze showing the walls (#), the mouse (@), the cats (!) and the cheese ($), plus unexplored spaces (.). See the course website for sample captures of the game output.

```
Maze:
####################
#    #.............#
# #  .............#
# # # .............#
#@#  #..........!..#
#  #  !............#
##   #.............#
# ##  .............#
#................$.#
#..................#
#..................#
#..................#
#...............!..#
#..................#
####################
Cheese collected: 1 of 5
Enter your move [WASD?]:
```

1.1 Game play requirements

When playing the game, the following requirements must be met:

- Maze must..
  - be 20 cells wide, and 15 cells tall.
  - have a wall all the way around the edge. This leaves 18 x 13 cells inside the outer wall.
  - be randomly generated using a maze generation algorithm, such as the ones listed on Wikipedia. Your code must have a comment, or use class/method naming which makes it clear which algorithm is being used. You must *not* copy code for generating a maze; you are to implement the algorithm yourself.
  - adapt the maze generated by your chosen algorithm with the following added requirements:
    - It must *not* have a wall in each of the four corners where the player and cats will be placed. (This is inside the outer wall which surrounds the maze)
    - Remove some of the inner walls of the maze to add cycles (loops) to the maze. (Otherwise, it is very hard for the player to navigate the maze while still avoiding cats). This means that there will not be just one path through the maze, but multiple possible paths.
    - All open (non-wall) cells in the maze must be connected, otherwise if the cheese is placed in a non-connected cell the game is unwinnable.
    - 2x2 square constraints:
      - There must not be a 2x2 square of open cells. Such an area would appear as a room in the maze, and makes the game less challenging.
      - There must not be a 2x2 square of wall cells.
      - For example, the two grids on the left are not allowed (the problematic regions are highlighted), but the one on the right is allowed. These show just a portion of the maze.



    - *Hint: Generate the maze which has no cycles using an algorithm of your choice. Then, randomly remove some of the inner walls of the maze ensuring that you don't make an open 2x2 square. You must play-test your own maze generation algorithm and tweak this to make it work.*
    - *Hint: For some of these constraints, like no 2x2 square of walls, it may be easier to randomly generate a new maze if your initial generation fails to meet the requirements.*

- Game Elements:
  - The player (mouse) starts in the top left corner.
  - Cats start in each of the three remaining corners.
  - A piece of cheese is randomly placed on the board. The cheese cannot be put into a cell occupied by a wall or by the player (but by a cat is OK).
  - Cats wander the maze pseduo-randomly. They may occupy the same square as other cats, or the cheese; they do not eat the cheese. If they occupy the same cell as a player then the player

loses the game. (If the player moves into the cat's square, or the cat moves into the player's square, the player loses).
    - Cats must wander the maze and try not to immediately backtrack their steps. They need not map out the maze and never backtrack; they should just not double back on themselves immediately unless there are no other moves. Cats cannot walk through walls.
        - The cats should not be “smart” enough to track the player: they are blind (and have no sense of smell or hearing...).
- Text User Interface:
    - The outside wall of the maze must be visible to the player at the start of the game.
    - The player can always see where the cats and cheese are, even if they are in a not yet revealed cell.
    - See the sample output for the symbols you must use for displaying the maze and game elements to the user.
- Game Play:
    - The user is shown the current state of the game, including their position, cat positions, and cheese position in maze. They also see all parts of the maze they have revealed.
    - The user enters a move using one of the W (up), A (left), S (down), D (right) keys.
        - Invalid keys generate an error message and force the user to re-enter their move.
        - User input must be case insensitive ('W' or 'w' is OK).
        - You may assume that the user enters only a single character when entering a move, or you may read in a whole line/string and just process the first character. Anything is OK.
        - Use `System.in` to read in user key strokes. The user will have to press enter after each key.
    - The user may type '?' to reveal the help that is shown at the start of the program.
    - Hidden keys (cheat codes?)
        - 'm' key displays the entire map. This will be used during marking to evaluate the maze generation algorithm (we have a program to validate your maze meets the above constraints). You may either have this reveal the entire maze for the rest of the game, or just display the full maze once.
        - ‘c’ key sets the number of cheese required to win the game to 1. This will be used for marking to evaluate wining the game. You may assume that this cheat code will only be used before the first cheese is captured.
    - The user wins when they have collected 5 cheese (unless the user has used the cheat code; in which case it’s 1 cheese!).
    - The user loses when they are in the same square as a cat.
        - The user loses if they move into a cell currently occupied by the cat. (User may have been thinking that the cat would move out and allow the mouse to come in; this is not true: the cat eats the mouse!)
        - The user loses if the cat steps into the cell which the mouse (user) occupies.
        - The user loses if they move into a cell that contains a cat, even if the cell also contains the final piece of cheese they needed. (i.e., if they lose when they die).
    - When the game finishes (win or lose), the user must be shown the entire game board (reveal all cells).

1.2 Implementation/Design Constraints

- The game is to use a text interface for display, and the keyboard for input.
- The game's OOD must be good:
  - You must have two packages: One package for the UI related class(es); another package for the model related classes (actual game logic).
    - Imagine that you want to have not only a text game, but also a web version. You should be able to reuse the entire model (game logic) in a completely different UI.
    - This means that the *model* should not print anything to `System.out`; instead, have it expose methods to get the required information, and code in the UI package displays it.
    - Your model must work in problem domain concepts, such as cat/cheese/mouse/... instead of working in terms of the UI characters #, @, $, etc.
  - Reasonably detailed break-out of classes to handle responsibilities.
  - Each class is responsible for one thing.
  - Each class demonstrates correct encapsulation.
  - Consider use of immutable classes where applicable.
  - Respect the command/query separation guideline when appropriate.
- Implementation must follow the style guide (on course website). Specifically important are:
  - Good class, method, field, and variable names.
  - Correct use of named constants.
  - Good class-level comments (comment on the purpose of each class).
  - Clear logic.
  - Correct indentation and spacing.
  - No deeply nested logic: move deeply nested functionality into sub classes or other methods.

2. Tasks

2.1 Design Phase

Complete the following steps to create an object oriented design for this application. You should be doing this with a partner and engaging in a collaborative design process[1]. For submitting these design files, first create an IntelliJ project, and then right-click on it in the Project view and select New --> Directory, and name it `docs`.

1. **CRC Cards**
   - Create CRC cards to come up with an initial object oriented design.
   - Do not submit the actual cards, but once you have settled on a design, you must type up the information stored on the CRC cards, take a picture of the cards, or submit a digital version of the work.
   - Each card must show the class name, responsibilities, and collaborators. Submit this as a .txt, .pdf, or .jpg file named `CRC.TXT` (or .PDF,...)[2] in the `docs/` folder. If you take a picture, ensure that the text is clear, and that the image is less than a meg.

2. **UML Class Diagram**
   - Create an electronic UML class diagram for your OO design.
   - The diagram should not be a complete specification of the system, but rather contain enough information to express the important details of your design.
   - Your diagram must include the major classes, all class relationships, and a few key methods or fields that explain how the classes will support their responsibilities.
   - You must use a computer tool to create the diagram. You may *not* generate the diagram directly from your Java code. Suggested UML tool: Violet UML Editor (free) or Visio.
   - Submit your UML diagram as a PDF or image file named `CLASSDIAGRAM.PDF` (or .PNG, or .JPG, …) in the docs/ folder.

3. **Explain how our OOD will work**
   - Pick two interesting (non-trivial) actions/steps that the game must support and explain how your OOD supports this.
   - For example, explain how the board will be drawn, or how the user's move is handled.
   - Imagine that you are presenting your design to your team and pursading them it's the best design. Your explanation here could be part of how you'd show that.
   - Describe this in a .txt or .pdf file named `OODEXPLAINED.TXT` (or .PDF) in the docs/ folder.

1 You and your partner should work together on coming up with a design.
2 In reality, you would likely not be saving the CRC cards; however, since it's worth marks, you must submit something!

2.2 Implementation Phase

- Implement the game in Java.
- You must start with your OOD from the Design Phase, but you may modify the design as needed. You need not update your OOD documents to reflect your final design.
- Source Control:
  - If you'd like to develop using SVN, by creating a project in CourSys, it will automatically create a repository for your group.
  - If you'd like to develop using GIT, you can use SFU's GitLab: https://csil-git1.cs.surrey.sfu.ca/
    (Do not use GitHub unless you keep your projects closed source; do not share your code publicly during the course).
- *Hint: When running your program, try running it from IntelliJ's terminal vs the "run" menu: the screen may re-draw faster making the text-based game more playable.*

3. Deliverables

The design and implementation sections have separate due dates (to encourage design *before* implementation!). Each section must be submitted as ZIP file of your project. Submit to CourSys.

3.1 Phase 1: Design

Submit a ZIP file of your IntelliJ project. You should have some code working by here, but you will only be marked on:

1. `docs/CRC.TXT` (or .PDF, or JPG); Keep images small!
2. `docs/CLASSDIAGRAM.PDF` (or PNG, or .JPG)
3. `docs/OODEXPLAINED.TXT` (or .PDF)
4. Any code you've written so far is fine; it won't be marked.

To submit, create a group in CourSys and invite your partner. Only one person per group need submit.

3.2 Phase 2: Implementation

- Your project must build a JAR file.
- No late submissions possible for the implementation phase.
- Submit a ZIP file of your entire project; see course webpage for details.
  - Include the `docs/` folder as well as all your code.

May be done individually. Still need to create a group, but just for you.

Please remember that all submissions will automatically be compared for unexplainable similarities.