Assignment 3: Fortress Defense

Due in two parts (see course webpage 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, and do not post questions about the assignment online. Please direct all questions to the instructor or TA via course discussion group on Piazza.com. Make 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

1.1 Story

You control a peace-loving fortress which has in front of it a large field. The field is covered in a thick fog, and your enemy has positioned siege tanks in it and is attacking you! You cannot see where they are, but you can still shots at them with your fortress' gun!

1.2 Game Details

The game is played with enemy tanks located somewhere within a *10* by *10* grid of cells. Each turn, you get to fire your gun, and then the enemy fires their guns (one per tank). They start off with *N* tanks (set by command line argument); each tank occupies four connected cells forming a Tetromino (any randomly constructed one; different tanks may be different patterns). Once the enemy tanks are placed, they do not move (they are siege tanks, after all!).

Your fortress can take *1500* points of structural damage before collapsing, at which point you lose the game. The amount of damage that a siege tank does is relative to how many undamaged cells it has:

Undamaged Tank Cells	4	3	2	1	0
Damage Done	*20*	*5*	*2*	*1*	*0*

Note that even if the middle of a tank is damaged, it still counts as one tank, and continues damaging the fortress, as listed in the table. (These things are resilient!)

You cannot see the battle field due to the think fog, but your gunner can hear when your shot hits a tank. Therefore, your gunner creates a map of where you have fired, and where you have hit an enemy tank. Each turn you can see your fortress' remaining structural strength, the battle-field map, and how much damage your fortress took during the previous turn.

1.3 Game play requirements

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

- Accept 0 or 1 command-line arguments to `main()`:
  - If no arguments are provided, default to *N*=5 tanks;
    otherwise assume the first argument is an integer and use that as the number of tanks.
  - You need not do any error checking on the type/value of this parameter.
- The program randomly places the tanks on the field such that:
  - no two tanks can occupy the same cell
  - all cells of each tank are contained inside the game-board/field
  - if not all tanks can be placed on the field, the program ends.
- Each turn, the player is shown the remaining structural integrity of their fortress.
- Each turn, the player is shown a map of what is known about the game-board so far:
  - ~ indicates unknown (fog)
  - X indicates hit a tank
  - ‘ ’ (space) indicates a miss.
- The game board has the columns numbered 1, 2, …
- The game board has the rows lettered A, B, …
- The user enters their move in the form: <Letter><Number>
  - For example, `B5`, and then enter.
- For each shot the user makes, the user is told if it hits or misses.
  - If the user has already shot this cell before and it was a hit, then it is a hit again but does no additional damage to the tank.
  - If the user has already shot this cell before and it was a miss, then it is a miss again.
- Each enemy shot hits the fortress and does damage. The user is shown how much damage is suffered for each enemy shot.
  - Tanks with zero undamaged cells fire no shots.
- When a player wins or loses, the player is told they won/lost and the game exits.
  - Player wins when no tank cells are left undamaged.
  - Player loses when fortress structural strength reaches 0 (or less).
- If a player loses, they must be shown the complete game-board without the fog at the end. This display must show the location of all the tanks, and where the user missed. It may (if you like) differentiate tank cells hit from tank cells that were not hit.

1.4 Implementation/Design Constraints

- 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 wanted to have not only a text game, but also a web version and an Android version. You should be able to reuse the entire model (game logic) in a completely different UI.
  - Each class is responsible for one thing.
  - Reasonably detailed break-out of classes to handle responsibilities.
  - Each class demonstrates correct encapsulation.
  - Consider use of immutable classes where applicable.
  - Respect the command/query separation guideline when appropriate.
- The game is to use a text interface for display, and the keyboard for input.
- Implementation must follow the online style guide. 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.
- OOD Hint:
  When you have some complex state, it is often best to encapsulate it into an object. Consider having an object for storing the state of each cell of your game-board. Store a group of these to makeup the game board.
- Tetromino Hint:
  For creating the Tetromino shapes, an "easy" way to do it:
  a) pick a location to be the start.
  b) randomly add a block to that location (up, down, left, or right).
  c) randomly add more blocks, ensuring you don't try to add the same block a second time.
  - This may be easier than trying to pre-program all possible shapes and rotations into your system by hand.
  - Note you cannot just "grow" the tetrominos from one end because some shapes are then not possible (think of the shape on the right[1]).

2. Tasks

2.1 Phase 1: Design

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[2].

1. **Use case**
   - Create a use case for the game. Hint: it will be titled "Play game".
   - Provide a reasonable list of steps for playing the game from the user's perspective. For example, how to recover from incorrect user input without crashing (use a Use Case variation).
   - This must be typed on a computer and submitted as a text or PDF file named `USECASE.TXT` (or .PDF). Place this file in the `docs/` folder of your project (you will need to create this folder).

1 By Anypodetos, Public Domain, https://commons.wikimedia.org/w/index.php?curid=8269275

2 You are allowed to do the assignment individually; however, design is best done in a group, so you should strongly consider do this assignment in pairs and actively collaborating to create a design.

2. **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, or take a picture of the cards.
   - Each card must show the class name, responsibilities, and collaborators. Submit this as a .txt, .pdf, or .jpg/.png file named `CRC.TXT` (or .PDF,...)[1] in the `docs/` folder. If you take a picture, ensure that the text is clear, and that the image is less than one MB.

3. **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 some 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.
   - You need *not* update this diagram with any later changes made during implementation.
   - Submit your UML diagram as a PDF or image file named `CLASSDIAGRAM.PDF` (or .PNG, or .JPG, …) in the `docs/` folder.

4. **Explain how our OOD will work**
   - Pick two (2) 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 convincing them it's the best design. Your explanation here could be part of how you'd show that. Discuss which classes complete portions of this action/step.
   - Describe this in a .txt or .pdf file named `OODEXPLAINED.TXT` (or .PDF) in the `docs/` folder.

2.2 Phase 2: Implementation

- 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).

1 In reality, you would likely not be saving the CRC cards; however, since it's worth marks, you must submit something!

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

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

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.

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