Assignment 3: Blanket Fort Water Fight

- This assignment is **expected to be completed in pairs** (you *may* complete it individually, but that is not recommended). Doing OOD in a team is great practice.
- Do not show other students your code; do not copy code found online. Please post all assignment questions on Piazza. Make 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.

1. Game Description

1.1 The Story

It's a lovely summer day, and you are enjoying time in the park by building a huge blanket fort. Some students from the XYZ department arrive and build their own (smaller) blanket forts somewhere in the park in front of your fort. Since they love Tetris, they build their forts in the shape of polyominos. Then, they challenged you to a water-gun fight! You propose the following game:

- Since you have the biggest fort, all of the other students will be on a team together.
- When they shoot your fort with their water guns, they get some points.
- When you shoot their fort with your water ***cannon***, it destroys one block of their fort.
- Since you don't want to even look out of your fort, you don't know where their forts are, but you can hear if your water cannon has hit a fort or has hit the grass.
- To make the game "fair", you setup the scoring rules as follows:
  - Teams take turns: you shoot your water cannon once, then they each shoot their water guns.
  - If they score enough points for their team, they win.
  - But, if you first destroy every part of all of their forts, you win.

1.2 Game Details

Your opponents are the XYZ students. Opponent forts are located somewhere within a *10* by *10* grid of cells. Each turn, you fire your water cannon once, and then each opponent fires their water gun. The game starts off with *N* opponents (set by command line argument); each opponent fort occupies **five** connected cells forming a polyomino (any randomly constructed one; different forts may be different patterns).

Each opponent shot hits your fort and scores points. When the opponent team earns 2500 points, they win. The amount of points an opponent scores is relative to the number of undamaged cells in their fort:

Undamaged Fort Cells	5	4	3	2	1	0
Points Earned	*20*	*20*	*5*	*2*	*1*	*0*

Note that even if the middle of a fort is damaged, it still counts as one fort, and continues earning point, as listed in the table.

You cannot see the field or forts, but can hear if your shot hits a fort or hit the grass. This allows you to create a map of where you have fired. Each turn you can see the opponent team's points, the map of the field, and how many points each opponent shot scored during the previous turn.

1.3 Game play requirements

- There is one player and N computer-controlled opponents
- Accept 0, 1, or 2 **command-line arguments** to `main()`:
  - If no arguments are provided, default to *N*=5 opponents.
  - With 1 or more arguments, the first argument is an integer value for *N*.
    You need not do error checking on the type or value of this parameter.
  - An optional 2nd argument is the string "`--cheat`"
    - See the Cheat section later for details.
- Randomly places opponent forts on the field such that:
  - no two forts can occupy the same cell,
  - all cells of each fort are contained inside the game-board/field,
  - if not all forts can be placed on the field, the program ends with an error message.
    - The program need not backtrack on already placed forts in order to try to place all forts. It is fine for it to randomly place forts on the field and if it get to the point where another fort will not reasonably fit, the program exits with an error message. You should be able to place at least 10 forts reliably.
- Each turn, the player is shown:
  - the opponent team's score,
  - a map of what is known about the field:
    - ~ indicates unknown (it may help to think of this as fog)
    - X indicates a hit (destroyed) block of a fort
    - (space) indicates a miss (grass).
- Game board columns are numbered 1, 2, …
- Game board rows are 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 fort.
  - If the user has already shot this cell before and it was a miss, then it is a miss again.
- Opposing team shots always hit your fort and scores points. The user is shown how many points are eared for each opponent shot.
  - Opponents whose fort is destroyed (all cells damaged) do not fire shots.
- When the player wins or loses, the player is told they won/lost and the game exits.
  - Player wins when all opponent forts are completely destroyed.
  - Player loses when the opponents' team has scored 2500 points (or more) collectively.
- When the game ends, player is shown the complete game-board will all cells revealed (no "fog"):
  - This display must show the location of all the forts, and where the user missed.
  - Each fort is displayed using a unique letter (A through E if there are 5 forts).
  - Show fort cells that were hit as a lower-case letter; show undamaged cells as upper case.

1.4 Implementation/Design Constraints

- The game's OOD must be good:
  - Must have at least two packages: one for UI class(es); another for model classes (game logic).
    - **The model must not print to the screen**: it should return information to the UI.
    - **The model must not read from the keyboard**: it should be given input from the UI.
  - Each class is responsible for one thing (high cohesion).
  - Use a 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.
  - **You must use at least 3 streams in your code**.
- Implementation must follow the online style guide; see marking guide.
- OOD Hint (optional):
  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.
- Polyomino requirements:
  - You must *not* hard-code the set of polyominos; you must randomly create them.
  - Your system must be able to create all possible polyomions of the required size. For example, verify that your application can can create a `+` or `T` shape.
  - Any given game need not have all different polyomions: it is only a requirement that the system can, at least some times, generate any of the valid polyomions.

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[1].

1. **CRC Cards**
   - Create CRC cards to come up with an initial object oriented design. You can create physical cards, or do it digitally.
   - If using physical cards, do not submit the actual cards; once you have settled on a design, take a picture of the cards or type up the information on the CRC cards.
   - Each card must show the class name, responsibilities, and collaborators.
   - Create a docs/ folder in your project. Save your CRC card file here (images should be <1MB).

2. **UML Class Diagram**
   - Create an electronic UML class diagram for your OO design.
   - It is not a complete specification of the system: it should contain enough information to express the important details of your design.
   - It must include the major classes, class relationships, and some key methods or fields that explain how the classes will support their responsibilities.
   - Use a computer tool to create the diagram. You may *not* generate the diagram directly from your Java code. Suggested UML tools: Violet UML Editor (free) or Visio.

1 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.

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

3. **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 forts will be placed on the board, or what classes the UI will use to draw the game board, or how the user's move is handled.
   - Imagine that you are presenting your design to your team during a design review. Discuss which classes complete which portions of each 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. Final submission must be an IntelliJ project.
- You must start with your OOD from the Design Phase. You may modify the design as needed during implementation; you need not update your design documents to reflect your final design.
- Source Control:
  - I strongly suggest using Git to manage sharing code between teammates.
  - I suggest SFU's GitHub: https://github.sfu.ca or a private GitHub repo.

3. Cheat

If the `--cheat` command line argument is given, then:

- When the application starts, print the cheat map.
- The cheat map is like the standard game-board:
  - Rows (A-J) and columns (1-10)
  - Print '.' for locations without forts.
  - Print unique letter for each fort, starting from A.
- This format is compatible with the end-of-game board display requirements; reuse that code!
  - Optional: After the board is display, display the other team's score.
- After the cheat map is displayed the game starts normally.

4. Deliverables

The design and implementation phases are all due on one date, submitted to CourSys as a single ZIP file of your entire IntelliJ project directory. **Use the Zipper addon (see website) to generate the archive.**

Submission Details

- docs/ folder should have design files:
  - `docs/CRC.TXT` (or `.pdf`, or `.jpg`); keep images small.
  - `docs/CLASSDIAGRAM.PDF` (or `.png`, or `.jpg`)
  - `docs/OODEXPLAINED.TXT` (or `.pdf`)

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