Creative Project 1: Abstract Strategy Games

Specification

Background

Strategy games are games in which players make a sequence of moves according to a set of rules, hoping to achieve a particular outcome (e.g., a higher score, a specific game state) to win the game. Strategy games usually give players free choice about which moves to make (within the rules) and have little to no randomness or luck (e.g., rolling of dice, drawing of cards) involved. *Abstract strategy games* are a subset of strategy games, usually characterized by

1. Perfect information (i.e., all players know the full game state at all times)
2. Little to no theme or narrative around gameplay

Popular examples of abstract strategy games include: Chess, Checkers, Go, Tic-Tac-Toe, and many others.

In this assignment, you will implement a data structure to represent the game state of an abstract strategy game of your choice.

Learning Objectives

By completing this assignment, students will demonstrate their ability to:

- Define a data structure to represent complex data
- Write a Java class that extends a given abstract class
- Produce clear and effective documentation to improve comprehension and maintainability of a class
- Write a class that is readable and maintainable, and that conforms to the provided guidelines for style and implementation
- Use the Visual Studio debugger to go through a program line by line

Choosing a Game

You may implement any abstract strategy game you choose, subject to the following requirements:

- The game must be playable by exactly two players.
    - It is OK if the game you choose can be played by a different number of players as well, but you will implement the game for exactly two players.
- Players must take turns making moves.

- Both players must make moves following the same basic rules (i.e., gameplay must be symmetrical).
- There must be no hidden information and no randomness in gameplay.
- The game must have a clear end condition.
- When the game has ended, there must be a clearly determined winner (or the game ends in a tie).

Here are some suggestions for games to implement:

- Chomp
- Connect Four
- Toads and Frogs

See Wikipedia, Freeze-Dried Games, or Pencil and Paper Games for more inspiration. If you would like to implement a game other than the three listed above (Chomp, Connect Four, or Toads and Frogs), please post in this Ed thread to request approval. Note that you **may not implement the game tic-tac-toe** (see below). Requests for a custom game must be made by **11:59pm on Sunday, January 25** to allow enough time for review and approval before the deadline. (We will monitor the thread and approve on a rolling basis.)

NOTE: The **reflection** portion of this assignment asks you to **find someone to play your game, once finished, and observe their experience**. Note that this will take some time and coordination to arrange, so make sure not to leave this part of the reflection until the last minute!

Required Abstract Class

You will implement a class to represent your chosen game. Your class should extend that `AbstractStrategyGame` abstract class, which contains the following abstract methods:

▼ Expand

```
public abstract String instructions();
```

- Returns a string describing how to play the game. Must include how to read the game state returned by `toString()`, how to make input a move (as used by `getMove()`), how the game ends, and who wins. It should also include any additional information important to allowing someone to play the game.

```
public abstract String toString();
```

- Constructs and returns a string representation of the current game state. This representation should contain all information that should be known to players at any point in the game, including board state (if any) and scores (if any).

```
public abstract int getWinner();
```

- Returns the (1-based) index of the player who has won the game, or `-1` if the game is not over. If the game ended in a tie, returns `0`.

```
public abstract int getNextPlayer();
```

- Returns the (1-based) index of the player who will take the next turn. If the game is over, returns `-1`.

```
public abstract String getMove(Scanner input);
```

- Takes input from the parameter to specify the move the next player wishes to make. Returns a string representation of said move.
  - If the provided parameter `input` is `null`, throw an `IllegalArgumentException`

NOTE: this is an unusual way to specify this method; it would be better design to take parameters that specify the move and leave the input to a client. However, since each game might have different information needed to specify a move, we cannot specify a single set of parameters that will always work.

```
public abstract void makeMove(String input);
```

- Takes in a string representation of the move the next player wishes (the output from `getMove()`) and executes the move.
  - If the provided parameter `input` is `null`, throw an `IllegalArgumentException`
  - If any part of the move is illegal, the player's turn should **NOT** change, and the method should throw an `IllegalArgumentException`.
  - You may assume that the string is well-formatted—in other words, there is no need to throw `IllegalArgumentExceptions` for cases that aren't mentioned above.

HINT: In this method, you may find `Integer.parseInt()` helpful to convert Strings into ints. For example, `int x = Integer.parseInt("5");` will result in `x` being 5.

The `AbstractStrategyGame` abstract class also contains the following implemented methods. Your class should work with the provided abstract class and should not modify it.

▼ Expand

```
public boolean isGameOver() {
    return getWinner() != -1;
}
```

- Returns true if the game has ended, and false otherwise

Your class should also include at least one constructor, which may take any parameters you deem necessary. You may also implement any additional private helper methods you like as well.

Implementation Requirements

Your game should be able to be run using the provided client program in `Client.java`. You should modify line 6 of this file to construct an instance of your class, and you may create any additional variables or data to pass to your constructor as parameters, but you should not have to otherwise modify the file. Implement your class so that this client works **as written**.

We have provided you with **two** sample implementations of tic-tac-toe (`TicTacToe1D.java` and `TicTacToe2D.java`). Both of these samples implement the correct functionality for the game, but differ in their design and underlying structure. We encourage you to look over both of these files to see some examples of how one might implement an abstract strategy game, and consider the tradeoffs that result from the two different approaches. **You may not implement tic-tac-toe as your game.**

As you implement your own abstract strategy game, you will need to consider similar tradeoffs of different approaches and make design decisions as you write your code!

Grading Guidelines

As described in the Creative Project Grading Rubric, your implementation must meet basic requirements to earn an S, and must have an extension implemented to earn an E.

> ⚠ Take extra care to ensure that the correct files are added and work with the **provided** `Client.java` and `AbstractStrategyGames.java` files

For the three suggested games, the basic and extended requirements are as follows:

Chomp

▼ Expand

Link to Wikipedia page with game rules: https://en.wikipedia.org/wiki/Chomp

Basic requirements:

- The game board is a two-dimensional grid with some squares “chomped” and some not.
- Choosing a square on a player’s turn causes all squares to the right and below it to become “chomped”.
    - i.e., If the grid’s coordinates increase to the right and down, with the top-leftmost square acting as the origin, this action consumes all squares with coordinate values greater than or equal to the chosen square's coordinates.
- Moves should alternate between players (Ex: Player 1 then Player 2 then Player 1...).
- The player who consumes the top-leftmost square (the origin) **loses** the game. This action triggers the end condition.
- `makeMove()` should take in **"<row> <column>"** where the origin (top left corner) is "0 0"

- Your program file should be named `Chomp.java`, and therefore, your class should be named `Chomp`

Extended requirements:

- The game board has three layers that must be cleared for a game to be completed.
  - The game board includes how many layers are remaining under each square, with a clear indicator of when there are no layers left on a given square.
  - Choosing a square only chomps squares on the topmost available layer within the subregion they chomp from. Chomping a square reveals the square directly below it, if there is one.
    - **NOTE:** This means you should not be prompting the user for a layer to chomp from, since it should always be taken from the topmost available layer
    - For example, chomping a square on layer 1 should only remove the other squares with strictly greater coordinate values on layer 1, and should reveal corresponding squares on layer 2. Layer 2's squares should all remain untouched by this action.
    - **NOTE #2:** This means that, based on what coordinates a player chomps from, you could potentially chomp different layers from the same game state. For example, if we start with the following Chomp board:

      ```
      3 | 3 | 1
      1 | 1 | 1
      1 | 0 | 0
      ```

      - and the player chomped the coordinate `(0,0)`, it would look like this:

        ```
        2 | 2 | 1
        1 | 1 | 1
        1 | 0 | 0
        ```

      - If the player instead chomped the coordinate `(0,1)`, it would look like this:

        ```
        3 | 3 | 1
        0 | 0 | 0
        0 | 0 | 0
        ```

    - The player who consumes the top-leftmost square from the **final** layer **loses** the game. This action triggers the end condition.

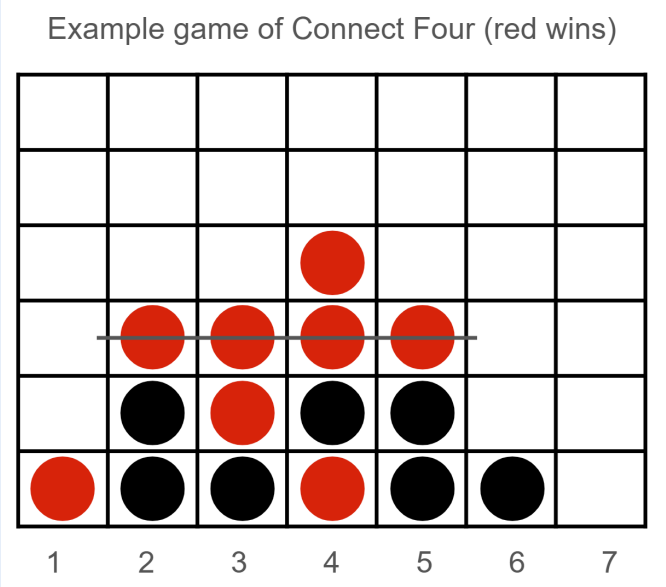
Connect Four

▼ Expand

Link to Wikipedia page with game rules: https://en.wikipedia.org/wiki/Connect_Four

Basic requirements:

- The game board is a 7-wide by 6-tall grid.
- Players' tokens are placed in the bottom-most available space in the column selected for a move.
    - Your program should only be prompting the user for column input. The program should **not** take any row input.
    - The column should be 1-based indexed. (i.e., the leftmost column is 1 and the rightmost column is 7)
- Moves should alternate between players (Ex: Player 1 then Player 2 then Player 1...).
- Four of the same player's tokens in a row, either horizontally or vertically, trigger the end condition.



- `makeMove()` should take in **"<column>"**, for example, a move of "1" attempts to drop a new piece in column 1.
- Your program file should be named `ConnectFour.java`, and therefore, your class should be named `ConnectFour`

Extended requirements (choose at least one):

- At the start of each turn, the active player may choose to either remove one of their tokens from the board or place a token (but not both). The player is able to remove one of their own tokens from the bottom row of any column, shifting all other tokens in that column (if any) down by one row.
    - `makeMove()` should take in **"<move> <column>"**, where <move> is either "A" or "R", correlating to adding or removing, respectively. As such, a move of "R 3" will attempt to remove the bottom token from column 3.
- Four of the same player's tokens in a row diagonally trigger the end condition (in addition to the horizontal and vertical end conditions).

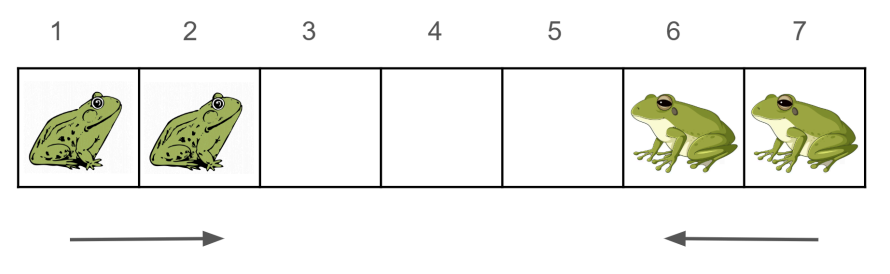
Toads and Frogs

▼ Expand

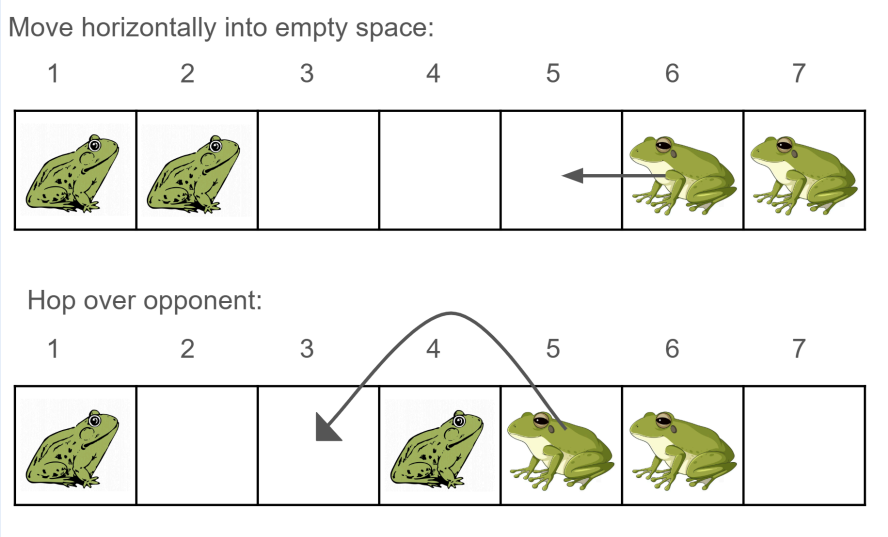
Link to Wikipedia page with game rules: https://en.wikipedia.org/wiki/Toads_and_Frogs

Basic requirements:

- The game board is a 1x7 grid with 2 toads at the leftmost 2 spots and 2 frogs at the rightmost 2 spots
    - The board should be 1-based indexed. (i.e., the leftmost position is 1 and the rightmost position is 7)



- Toads may only move right, and Frogs may only move left
- A move consists of either moving to the next square (if empty) or "hopping" over an opponent's piece into an empty square.
    - For example, frogs may only move to an empty space on the left or hop over a toad on the left into an empty space



- The player who can't make any more valid moves on their turn **loses** the game. This action triggers the end condition.
- Moves should alternate between players (Ex: Player 1 then Player 2 then Player 1...).
    - Toads should always go first
- `makeMove()` should take in **"<index>"**, for example, a move of "1" moves the piece from index 1 (if it exists). Based on the positioning of the other pieces, it should either:
    - If no piece in front of it: move one square towards the direction it's facing,
    - If an opponent's piece is in front of it, and there's an empty space after that, jump over

    an opponent's piece, or
    - throw an `IllegalArgumentException` in all other cases.
- Your program file should be named `ToadsAndFrogs.java`, and therefore, your class should be named `ToadsAndFrogs`.

Extended requirements (choose at least one):

- Each piece can only move three times before it becomes inactive.
- After a hop, the piece is now stunned and can't move for one turn. Moving by one space still behaves as normal (no "stun").

Other Games

If you would like to implement a different game, you will need to specify what the basic and extended requirements will be as part of your proposal. Your proposed requirements should be similar in scope and complexity to the requirements for the three suggested games. Post in this Ed thread to propose a different game.

Testing Requirements

On this assignment, you are required to write 2 of your own test cases that test different aspects of your game. Those test cases should be contained within their own method in your Testing class. We've provided you with examples within `ExampleTesting.java` to give you an idea of how testing could be done. **Our requirement for these tests is that one tests for a win condition (a situation where a player wins) and another tests for illegal moves (a situation where a player tries to make an illegal move).**

WARNING: We have provided a test case that simply tests to see if you've uploaded `Testing.java` and it fails no tests. It does not check whether or not you've met the testing requirements for this assignment.

WARNING: We did not provide behavior tests for this assignment. Therefore, you should still run your program and play your game *thoroughly* to ensure that it meets the expected requirements. If you are implementing a custom game, you do not need to pass the file check test (since it only checks that you have a file named `Chomp`, `ConnectFour`, `ToadsAndFrogs`.

NOTE: We will never grade your `Testing.java` file on Code Quality or for comments. This also means you are allowed (and encouraged) to use lambdas in order to test exceptions using `assertThrows`. See comments in `ExampleTesting.java` for more details on using lambdas in this context!

Assignment Requirements

For this assignment, you should follow the Code Quality guide when writing your code to ensure it is readable and maintainable. In particular, you should focus on the following requirements:

- You should make all of your fields private, and you should reduce the number of fields to only

those that are necessary for solving the problem.

- You should avoid hard-coded numbers in your implementation. Instead of hard-coding specific numbers, which we call using magic numbers, it's usually better to use a variable or some property of an object.
- Each of your fields should be initialized inside your constructor(s).
- You should comment your code following the Commenting Guide. You should write comments with basic info (a header comment at the top of your file), a class comment for every class, and a comment for every method other than main.
    - Make sure to avoid including *implementation details* in your comments. In particular, for your object class, a *client* should be able to understand how to use your object effectively by only reading your class and method comments, but your comments should maintain *abstraction* by avoiding implementation details.
- Any additional helper methods created, but not specified in the spec, should be declared ***private***.

⭐ [GRADED] Abstract Strategy Games

This slide **IS** GRADED.

Download starter code:

C1_AbstractStrategyGames.zip

Testing Requirements

Remember that you're required to write two tests, each within their own method in `Testing.java`. More information on this requirement can be found in the spec.

WARNING: We did not provide behavior tests for this assignment. Therefore, you should still run your program and play your game *thoroughly* to ensure that it meets the expected requirements. If you are implementing a custom game, you do not need to pass the file check test (since it only checks that you have a file named `Chomp`, `ConnectFour`, `ToadsAndFrogs`.

NOTE: We will never grade your `Testing.java` file on Code Quality or for comments. This also means you are allowed (and encouraged) to use lambdas in order to test exceptions using `assertThrows`. See comments in `ExampleTesting.java` for more details on using lambdas in this context!

Reflection

The **reflection** portion of this assignment asks you to **find someone to play your game, once finished, and observe their experience**. Note that this will take some time and coordination to arrange, so make sure not to leave this part of the reflection until the last minute!

The following questions will ask you practice **metacognition** to reflect on the topics covered on this assignment and your experience completing it. For each question, focus on your plan and/or process for working through the assignment along with the CS concepts. Think about things like how you organized your working time, what sorts of things tended to go wrong, and how you dealt with those errors or mistakes.

Please answer all questions.

Question 1

REQUIRED
You MUST answer this question to receive credit for the assignment

Which game did you implement?

- ( ) Chomp
- ( ) Connect Four
- ( ) Toads and Frogs
- ( ) Other

Question 2

Describe how you implemented the state of your chosen game and **why** you chose that implementation.

Question 3

Describe an alternate implementation you could have chosen for your game. Include at least one specific detail about this alternate implementation (e.g. what fields would you have? How would you represent a move being made?) .

Then, list at least one advantage and one disadvantage you think this alternative has compared to the implementation you chose.

Question 4

The following 3 questions will be concerned with HCI (Human-computer Interaction) a subfield of CS related to application design and guiding principles for good design.

You're going to conduct a "behavioral mapping" study with your current implementation. (This is just a fancy way of saying "observation of someone playing your game"). You'll have to follow the following steps:

1. First, get a user to play your game - peer, friend, family, roommate, TA at the IPL, etc.
2. Tell them that you'd like them to test out a game that you made, and their goal is to win.
3. Provide no further instruction than that (you want their unbiased reactions and attempt to use your system without explanation).
4. Play the game with them (~5 mins in length).

While playing, take note of the following questions:

- Were there any instances in which the user was confused as to how to play the game or how the game worked?
- Were there any instances in which the user made a mistake / was confused about how to enter input for the game?

Report your findings (indirect feedback) here along with how the user of your study relates to you (peer, friend, family, roommate, TA at the IPL, etc.)

Question 5

Now, ask the user for direct feedback regarding your implementation of the game. Some questions include:

- Did you understand how to play the game from the provided instructions alone?
- What was most confusing about how to record a move?
- In a perfect world, how do you think the interface should change to make this game easier to play?

Report your findings.

Question 6

Based on your responses to the previous two questions, if you were to make one change to your implementation centered around **usability**, what would it be? (This answer doesn't have to be feasible with your current knowledge of Java / CS - can be anything!)

Question 7

What skills did you learn and/or practice with working on this assignment?

Question 8

What did you struggle with most on this assignment?

Question 9

What questions do you still have about the concepts and skills you used in this assignment?

Question 10

About how long (in hours) did you spend on this assignment? (Feel free to estimate, but try to be close.)

Question 11

Was any part of the specification or requirements unclear? If so, which part(s), how was it unclear, and how could it have been made more clear?

Question 12

[OPTIONAL] Do you have any other feedback, questions, or comments about this assignment?

(Note that we may not be able to respond to questions here, so please post on the message board if you would like a response!)

Final Submission

Final Submission

Fill out the box below and click "Submit" in the upper-right corner of the window to submit your work.

Question

I attest that the work I am about to submit is my own and was completed according to the course Academic Honesty and Collaboration policy. If I collaborated with any other students or utilized any outside resources, they are allowed and have been properly cited. If I have any concerns about this policy, I will reach out to the course staff to discuss *before* submitting.

(Type "yes" as your response.)