CPSC 223J — Java for C++ Pgmrs — Project #2

Project #2 – The Nim Game

Introduction

Now that we've released our first game (Number-Guess) on our new platform (the Games Master), it's time to do several things.

Firstly, if needed, we want to copy the project #1 code base to a project #2 directory, and then maybe clean it up a bit so that designing, testing, and debugging our next add-on game is easier (and hence faster to do). Don't do too much.

Secondly, we will be **adding** a new game, Nim. Nim is a game of pebble piles. On each turn, a player picks a pile and takes one or more pebbles away (including up to the entire pile). The game ends when a player takes away the last pebble, to win the game. (NB, you can find out more about Nim on the web: e.g., at Wikipedia.) Piles are a bit harder to display, so we will use rows of "pebbles".

Note, the Number-Guess game should still be working in this project.

Games Master Coupling to Nim

Modify the Games Master (GM) listen() method to offer an option to play Nim. Modify the GM's respond() method to create a Nim game object and call Nim's repl() method when the Nim game is selected. When the Nim game ends, it will return from its repl() method to its GM respond() caller.

Nim Class Hierarchy

Create a Nim class that inherits the Repl (or ARepl class, depending on how you named it) ABC. Now, the Nim class has a repl() method (including its body). Implement the 6 repl-called sub-methods in the Nim class (their headers are declared in IRepl) so that the Nim game will work.

Nim Player Interaction

Our initial Nim game will have the player play the "Nim AI Brain". (It will be a level-0 Brain, basically random.) When playing Nim, we want to present the "board" position for the current state of the game before each move (for the user and for our AI Brain).

The board has numbered rows (from 1 upward) for each pile, and each row contains a 'o' character for each pebble in that row's "pile". Here is an example board position with three of the original four rows still filled, and with a gap after every three pebbles (to make it easier for the user to count them):

```
1: ooo o
2:
3: ooo ooo oo
4: oo
```

After presenting the board position for the player, we want the player to be prompted for a "move" choice. The choice consists of two numbers: a row/pile number and the count of pebbles to take. If the player chooses 0 for the row, we consider that the player has chosen quit the game early. Otherwise, if the player picks a non-existing row number, or picks an empty row, or picks a non-positive count of pebbles, then the game (the listen() method) tells the player that this is an illegal move and to please make a different move. If the player picks a row with pebbles and any positive count, then we take that many pebbles (or all of them if the count is bigger) from that row/pile.

The players (user and AI Brain) alternate making moves.

Nim AI Brain

This first version of the Nim game has a level-0 AI Brain. This means that the while the Brain always makes a legal move, beyond that its "thinking" is random. To implement this, your Brain can do one of two things to pick a legal row.

The first way has only simple steps: loop picking a random row until we pick a row with at least one pebble in it. For example, if there are four rows, then the Brain loops: picking a number from 1 to 4 until a non-empty row is picked. Once the row is successfully picked, then pick a random number of pebbles from the row. For example, if the row has 3 pebbles in it, pick a random number from 1 to 3.

The second way has a complicated row-picking step. First, get a count of the number of non-empty rows. Now, without looping, pick a random number of that size. For example, if of the four rows, only three are non-empty, then pick a random number from 1 to 3. If you pick, say row 2, then walk the set of rows, checking for non-empties, to find the second **non-empty row number** – and that is your Brain's row pick. Once the row is picked, pick the number of pebbles as above.

Because it is simpler, we recommend the first way. (NB, Get to Working Code Faster.)

Listening

Nim's listen() method will

- Display the current board position (resulting from Setup, or the Brain's move)
- Display a prompt to input a row and count (from the user)
- Input the user's row and count
- Display the user's move (e.g., "User removes 2 pebbles from row 3")

The listen() method will then construct a Nim_Move object (containing the row and pebble count) and put it in a Nim object slot for use by Nim's respond() method.

Responding

Nim's respond() method is more complex because of the Brain. It will do this:

- If the user's move is a "quit", mark the game done (a slot) and return
- Update the current board position using the user's move
- Display the new board position (resulting from the user's move)
- Check for a win (for the user) and if so mark the game done and return
- Display a note to indicate waiting for the Brain's move
- Get the Brain's row and count move (see how below)
- Display the Brain's move (e.g., "Brain removes 4 pebbles from row 2")
- Check for a win (for the Brain) and if so mark the game done and return
- Update the current board position using the Brain's move

Note, before implementing the Brain, you could let the user supply the Brain's answer.

Nim Coupling to the AI Brain

During Nim setup, you should create a Nim_Brain object and put it in a Nim slot. The Nim_Brain class's get_move() method is passed the Nim object, itself, and returns a Nim_Move object. The get_move() method will pick a random legal row and a random count of that row's remaining pebbles. Get_move() will then create a new Nim_Move object with this row and count and return it.

In order for the Brain to do this, the passed Nim object should be interrogated about its current board position: how many rows it has and how many pebbles each currently has (hence some Nim getter methods are needed for this). However, the AI Brain should not be able to change the contents of the Nim object.

CPSC 223J — Java for C++ Pgmrs — Project #2

Nim Board and Setup

The Nim board should be an Arraylist (in a Nim slot) of ints. Each row corresponds to an element of the array. Each element contains the number of pebbles that row currently has.

During setup, a random number, N, of rows should be chosen in the range 3 to 6. (These rows are numbered for the user from 1 to N.) Then for each row separately, a random number of pebbles should be chosen in the range 3 to 8. (Note, it might help code clarity to ignore the array's [0] slot.)

For example, if the random choice for rows is 4, and then for those 4 rows the random pebble counts are 4, 3, 8 and 5, then the initial board position would be displayed as follows:

```
1: ooo o
2: ooo
3: ooo ooo oo
4: ooo oo
```

Development Note: We still get "**it's coded but doesn't compile**" submissions. To minimize debug time, we suggest using Add-a-Trick (1-3 executable statements for the added "trick", and print to see what's changed), then recompile/run/see. And write stub methods that output their name if you need to call a new sub-fcn when you're adding a "trick" to the caller. These can be filled in with later "tricks".

Academic Rules

Correctly and properly attribute all third party material and references, if you used any. Do not make calls to functions (e.g., "exec()") which trigger the O.S. to execute an external program.

Submission

Your submission must, at a minimum, include a plain ASCII text file called **`README.txt`** (e.g., title, contact info, files list, installation/run info, bugs remaining, features added) all necessary source files to allow the submission to be built and run independently by the instructor. [For this project, no unusual files are expected.] Note, the instructor doesn't necessarily use your IDE.

All source code files must include a comment header identifying the author, author's contact info (please, no phone numbers), and a brief description of the file.

Do not include any IDE-specific files, object files, binary executables, or other superfluous files.

Place your submission files in a **folder named** `X-pY_lastname-firstinitial`. Where X is the class course number (e.g., 123 for CS-123) and Y is the project number (eg, 9 for Project #9) For example in CS-123 for Project #9, if your name were Tom Cruise, then you would use

```
123-p9_Cruise-T
```

Then zip up this folder. Name the .zip file the **same as the folder name**.

Turn in by 11pm on the due date (in the Titanium post) by **sending me email** (see the Syllabus for the correct email address) with the zip file attached. The email subject title should also include **the folder name**. [NB, If your emailer will not email a .zip file, then change the file extension from .zip to .zap, attach that, and tell me so in the email.] Please include your name and campus ID at the end of the email (because some email addresses don't make this clear). If there is a problem with your project, don't put it in the email body – put it in the README.txt file.

Grading

- 50% for compiling and executing (running) correctly with no errors or warnings
- 35% for clean and well-documented code (but only if it compiles and runs)
- 10% for a clean and reasonable **`README`** file (but only if it compiles and runs)
- 5% for successfully following Submission rules