Hug the Angry Jarvis Game Instructions Part 2

Introduction

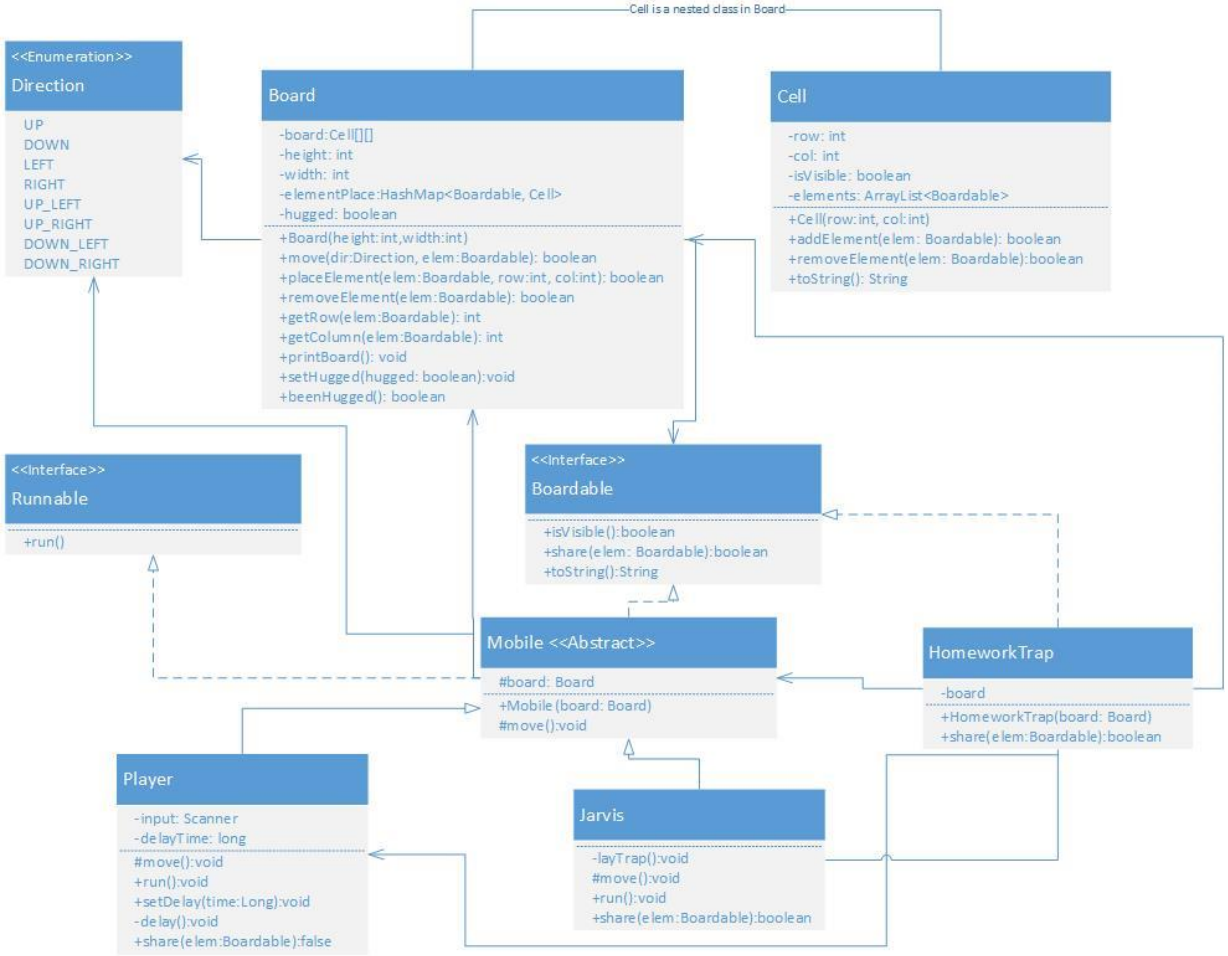
In this assignment, you have the opportunity to design and work with Java objects, interfaces, nested classes, dealing with exceptions and working with multiple Threads. Additionally, you will be using HashMap, ArrayList, Strings, Scanner, and other basic Java constructs.

This assignment is to be done individually.

Time to build "Hug the Angry Jarvis" the game. We will make several refinements as the project develops. It is very important to start this project early and make continual progress.

The Assignment

Below you will see the UML design "Hug the Angry Jarvis". Your classes MUST HAVE all of the entities (classes, methods, and fields) and they MUST HAVE the exact same names (case matters as does the order of parameters). You are welcome and possible should add additional PRIVATE methods to your classes. However, before you add additional public methods you should talk to me.



First, you will notice that *Board* has some new methods and a new field. Most of them are intuitive. The *removeElement* should remove the passed-in element from the board (meaning from the *Cell* and the *elementPlace* HashMap.) It should return true if the element was removed false otherwise. The *getRow*

and *getColumn* methods return the row coordinate and the column coordinate of the given element. They should throw an *IllegalArgumentException* if the element was not on the board. The *beenHugged* and *setHugged* are accessor methods for the *hugged* instance variable. Jarvis should be the only entity to call *setHugged*. You might also need to adjust *placeElement* and move now that *Cells* might contain multiple Boardable objects.

The *Boardable* interface also has a new method, *share*. This method will need to be implemented by all concrete implementations and dictates how objects of type *Boardable* interact if they are sharing a *Cell*. If a *Cell* already contains an element *A* when its *addElement* method is invoked with the argument *B*, then it should call *A.share(B)*. If the *share* call returns *true*, it should add B to its elements list otherwise it should return false and not add B.

A quick note about Cell's *toString*. If the Cells is not visible, it should return "#". If it is visible but empty, it should return " ". If it is visible and contains elements, it should return the toString of the last element that was added to it. A Cell becomes visible if any element is added to it that is visible. Once a cell is visible it should ever become not visible.

Mobile is a new abstract class that implements both *Boardable* and *Runnable*. Right now all of its methods should be abstract. Later we might add a method body.

The Player class is the main character of the game. Its *move* method is responsible for interfacing with the user and calling *delay*. It should call *delay* before taking any user input. The users can use the same keys for directions as the stylus from the last lab.

- "q" = UP_LEFT
- "w" = UP
- "e" = UP_RIGHT
- "a" = LEFT
- "d" = RIGHT
- "z" = DOWN_LEFT
- "x" = DOWN
- "c" = DOWN_RIGHT

However, if the user enters "s" it should indicate that they want to stay put. The *move* method is also responsible for printing board after each user input. The *run* methods should continually call *move* until Jarvis has been hugged. The *delay* method should ensure the player is not able to do anything for *delayTime* milliseconds. The very last thing *delay* should do is set *delayTime* to zero. Also, while the player is delayed any moves they entered should be ignored. The *setDelay* should set *delayTime*. This method is called by a trap. If the Player is the first element in a Cell, it cannot share its spot so its *share* methods should always return false. The Players *toString* method should return a "*".

The *Jarvis* class represents the real hero of the game, Professor Jarvis. As long as he hasn't been hugged, very 500 milliseconds Jarvis should pick some random direction and move that way. If he is unable to move in the chosen direction, he should choose another way (until all directions have been tried then he should wait for 500 milliseconds and try again.) Every 6th move Jarvis can lay a trap in one of the Cells adjacent to his current location. He should randomly pick one to place it in (and keep trying until he has placed it or until he has tried all adjacent cells). Jarvis can share his cell with anything of type *Mobile*. If

he shares with a *Player*, then he gets a hug and should set the *Board's* hugged variable (he should also print something letting the player know that they have soothed the savage Jarvis). Jarvis should not be visible, and his toString should return a "?".

HomeworkTrap is the only type of trap Jarvis can set (for now). It can share its spot with any *Mobile* (it can't share it with other traps.) If it shares its location with a *Player*, the trap should set the delay time for the player to 5000 milliseconds and then remove itself from the board (the trap has been sprung.) It should also print something explaining what has happened. If a trap shares the spot with *Jarvis*, nothing should happen. The traps are not visible and their *toString* should return " ".

In addition to what is shown in the UML, you will need to create a *Game* class. This class should contain the main method that sets up the *Board*, the *Jarvis* and the *Player* objects (these last two need to run in their own threads.) At no point should your game ever crash. Also, note that since *Jarvis* and *Player* run is separate threads some methods may need to be synchronized, I leave it up to you to figure out which ones.

Grading

I'll be watching for style, design, as well as the correct output. Your output should be neat and easily understood. At no point and time should your code crash. Make sure your code is well commented and carefully follows the class Style Guide (including a Javadoc-style comment (/** comment */) before each method). Use constants where appropriate, and focus on elegant solutions. If you find that you are writing very large methods, that likely means they should be broken into multiple smaller ones. If you have lots of repeated code, that indicates that you should write a method to do that work. When you are designing your classes, try to keep them very tightly focused.

Submitting

Submit a *zip* all of your .java files to Canvas.