Back-End Logic Classes Specifications

Last Modified: 09/12/2011

Overview

You must implement nine classes to model the following:

- player
- board location
- property
- lot, i.e. a property in a color-group, e.g. Boardwalk, Atlantic Ave., etc.
- railroad
- utility
- corner square (instances will be Go, Jail, Free Parking, and Go to Jail)
- tax square (instances will be Income Tax and Luxury Tax)
- card square (instances will be Chance and Community Chest locations)

These classes are to **model only the logic** of these entities and **not contain user interaction**. (This separation between logic and user interaction is a major part of this kind of programming. This means that if we see `System.out` *anywhere* in the code for any of the nine classes above, you will automatically fail the entire project.)

Note that these classes are interdependent; you should not expect to be completely done with the player class, for example, until you've worked on some of the others. You are expected to take advantage of both *is-a* and *has-a* relationships between classes in the back-end.

Here are further details on the requirements:

The Player Class

1. Players should know how much money they have, where they are, what token they represents them, and which properties they own. You choose the representation of the data as long as your operations and documentation support it.
2. It is useful and allowable, but not required, to maintain data members for the number of railroads and utilities. It is required to provide accessors for these values.
3. At least a default constructor is required, but initializer constructors may be added as long as they make sense.
4. Reasonable modifiers and accessors that work will all data members are required. (These don't have to be setters and getters and if there's no reason a client would directly set or get a property, it shouldn't have a setter or getter. Modifiers and accessors of sophistication are encouraged.)
5. An accessor which provides a comma-separated listing of the names of all properties owned is required. This will allow you to show a player's holdings concisely later.
6. A `toString()` is required.

Board Location Classes

1. The remaining eight classes in the bulleted list above model the board.

2. You must use inheritance and polymorphism in implementing these classes to keep your code simple. It is strongly recommended that you draw an inheritance hierarchy on paper and have it checked by the instructor, TA, or TI before writing code, as correct use of inheritance is the core of this project and you will not pass the project if this is not done correctly.
3. All locations on the board must know their address, the number of spaces counting from "GO" to that location. Other required data -- which may depend on the kind of location -- are names of locations, owners, purchase costs, color, how much each house costs, the rent structure, and house/hotel status. You choose the representation of the data as long as your operations and documentation support it.
4. Each class requires at least a default constructor. Add one or more initializer constructor(s) as necessary. (Think about what class members would have initial values upon construction.)
5. Implement the following method based upon the type of board location (more below):
   ```
   String[] getPossibleActions(Player player)
   // PRE:  player is initialized
   // POST: FCTVAL == array of options player has upon landing on
   //       this space, to be used in a menu in a user interface
   ```
6. For classes of locations which may be owned, implement a method that returns the rent due based upon the current state of the object as given in the rules above.
7. Implement modifiers and accessors needed to support rules of the game.
8. Each class is required to have a reasonable `toString()`, although it could be inherited.

More on `getPossibleActions()`

The function specified in #5 above is perhaps the most important part of the project. What happens when a player lands on a space depends on what kind of space it is and the state of the game (things like who owns what). Thus, we need a function that will return what things the user can possibly do and we implement this in the board location hierarchy, as the decision-making is based upon the state of the board location objects.

The array of options is returned to the main user interface. In our world, we'll use a console-based menu to prompt the user for his/her choice (and I provide you with a function to call from the front end on the Food for Thought and Programming Aids page). But, if we were implementing a different kind of user interface, these strings might be displayed in a popup dialog with a drop-down menu or on a part of the screen with radio buttons the user could use to make selections. The point is that the logic that drives the possible choices (the processing) is a very different kind of programming from the user interface and must be kept distinct.

Sometimes (many times) there is only one action that is possible, but other times, the player landing on the space will have choices. Because of the possibility there are choices, it's necessary to return an array of size 1 in the case that the player's response is dictated. Your implementations of this function should contained decision-making logic that considers what is appropriate based on the state of the game. (For instance, don't give the player the option to buy a property when that's not possible and later say, "Sorry, you can't buy this because..." Rather, simply only include an option to buy a property when it's a valid option.)

You may find it helpful to send in another parameter to this function; this is acceptable if the logic and documentation are clean.

Task #1: Analysis for the Back-End Logic Classes / Deliverable #1

Timeline: *Tuesday, 9/13, through Friday, 9/16*

Your first deliverable will not involve code, but instead be a document wherein you plan what you'll be doing in this project.

Save a copy of the Analysis Deliverable Worksheet (.doc) or (.pdf) or (.odt). Either complete this document electronically and print it out to turn it in or print it out first and fill it out using pencil.

This document will be due **to Doug's office by 2 p.m. on Friday, 9/16**.

Task #2: Implement the Player Class

Timeline: *Friday, 9/16, through Saturday, 9/17. (You may begin work on this once you've completed Task #1 and do not need to make use of anything from the polymorphism lecture/lab for this task.)*

Your second task is to implement the class to model a player.

No deliverable is required for this class by itself.

Task #3: Implement the Board Location Classes / Deliverable #2

Timeline: *Sunday, 9/16, through Wednesday, 9/21.*

Now, begin implementing the eight classes to model locations on the board according to the specifications above.

Create a separate test driver (or test drivers) wherein you:

- Create an array that holds the first 15 spaces on the standard Monopoly board and initialize this array appropriately.
- Test the `toString()` methods of all classes using all of these spaces.
- Test all methods of all classes.

Ultimately, the testing will not be part of your final product, but is an important step in being successful and will save you quite a bit of time in the long run.

Prepare a report according to the following specifications:

- Include the code for player and the eight board location classes, as well as the test driver.
- Put each file in its own section in the report with its own header. Start each section on a new page with a hard page break. Avoid letting pages break in the middle of methods.
- Order the sections of the report as follows: player, board location, property, lot, railroad, utility, corner square, tax square, card square.

There are two deadline and feedback options for this deliverable:

- If you've kept your portfolio up-to-date and are regularly adhering to the Programming Assignment Guidelines, I'll offer you individualized feedback at this point to help you as you proceed. Turn in your report to **to Doug's office by 1 p.m. Wednesday, 9/21** if you want this.
- Otherwise, turn in your report to Doug's office by the end of your class on **Thursday, 9/22,** to have

your report assessed for group feedback and credit.

I'll review these deliverables as noted above and prepare to do an optional project session shortly thereafter (probably Friday afternoon). I'll open this session with group feedback on trends I see and wish to address from your submissions and then have some work time where you can ask me individual questions.