COMP 210 - Lab 6 and Homework 5

Spring 2016

This week we finish our work with the Puzzle State problem by finishing the hierarchy from lab5 and developing a simple CLI interface. A solution to last week's lab and homework can be found on the server if you need or wish to work from it.

Lab 6

Due by the end of day Friday

For this lab you'll start working up to the top of the PuzzleState problem with the goal of completing PuzzleState class and there by finishing the hierarchy and a functioning *canCross()* method. The overall design has been updated and diagrammed in figure (1). You'll find an implementation of everything on that diagram excluding the PuzzleState in the course directory.

Your goal for lab is to complete PuzzleState such that you can properly test *canCross* on a variety of puzzles. Towards this end you must implement some new elements to our design:

- A STATIC FACTORY METHOD called *createInitialState*[1] for constructing the initial state given the flashlight's battery power and the speed of all the people stored in an array.
- A mutator method for Puzzle State that that sets the implementation to be used by the factory. The default can be either array or list. An enumeration type[2] *GroupImp* is used to enumerate the implementation options and a private static field[3] is required to store the current implementation preference.
- A Group selector for PuzzleStates.

[1] Factories build things. This method build instances of the class in which they're defined

[2] `https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html`

[3] `https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html`

The logic behind Factory methods is to abstract away construction details. In this case, the only important details about the initial state are the flashlight's battery power and the speed of each person. Identifiers for each person can be counted off as part of the construction process and all objects start on side A. A constructor or factory method can maage this for you. Finally, the question of which group implementation to use and how to best to construct that Group is completely managed by the PuzzleState through the factory method and the new field. Users can set the implementation with the mutator and then the factory method will then construct that initial object in whatever way it sees fit. From the user perspective, no Group need be explicitly instantiated. The factory method carries

out the grunt work and if the Group is actually needed for some reason, then the user can select whatever group gets constructed by the factory method. To fully control the construction of PuzzleStates we can make the actual constructors private and restrict their use to the Factory method. This means we cannot directly test the factory but must instead test that the PuzzleState constructed by the factory has the expected behavior. In this case, that means checking things like the Group and the result of canCross. We'll discuss the details of the factory method in class Friday so focus on clear documentation and testing of the PuzzleState behavior.

Homework 5

Due before the next lab

For homework you'll implement a basic CLI-based UI for our program. We'll consider two different ways of invoking the program:

1. At the CLI, users pass the name of a CSV file where the first line is the flashlights' battery power and the second line is the speed of each person separated by commas
2. Users pass the battery power as the first argument followed by one or more speeds for people.

In both cases the result of canCross is reported to the standard output in a descriptive manner[4].

Notice the first option takes a single argument and the second takes at least two. This should allow us to easily differentiate which case we're dealing with without using special flags like we typically see at the CLI. Completing this program gives us further practice with Arrays[5] while also giving us a chance to work with CLI arguments[6] and File I/O[7]. While we can certainly design a class or two to assist with the UI logic, for now we'll just cram it all in the *main* or PuzzleState.

[4] something like, "this puzzle is solvable" would suffice

[5] `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html`

[6] `https://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html`

[7] `https://docs.oracle.com/javase/tutorial/essential/io/scanning.html`

Updated UML Diagram

The UML diagram has been updated based on this week's tasks and the solution to last week's lab found on the server. Notice there is one new element to the diagram, underlined methods. Something that is underlined is STATIC. In this case both the factory and the main methods are static methods.

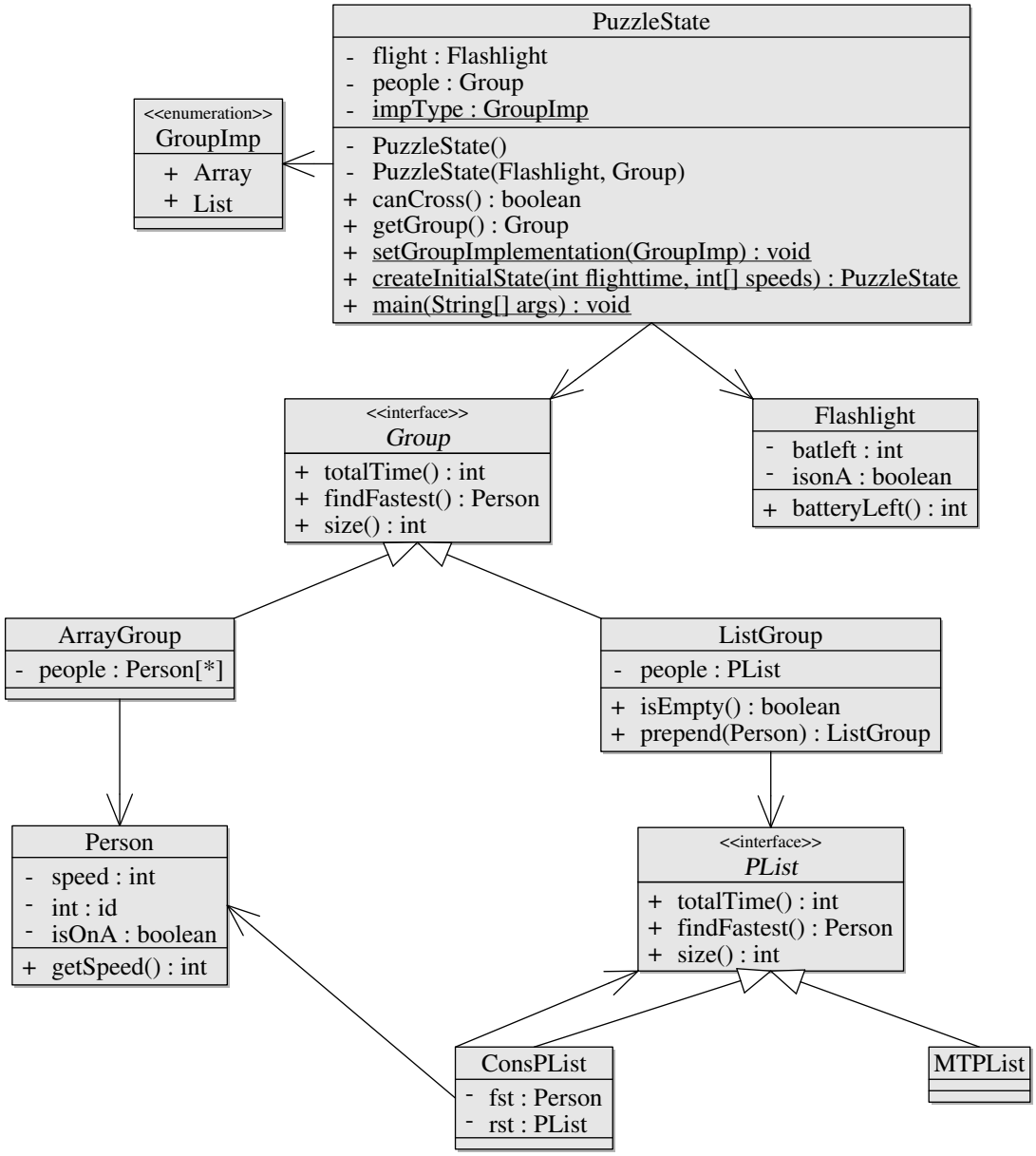

Figure 1: A Framework for Exploring the Puzzle Problem