CS6910
Lab 03

Goals:

- Add a menu to a graphical user interface
- Apply a design pattern to add new ways for computer 'players' to play the game
- Practice behavior-driven development to design and implement those new behaviors

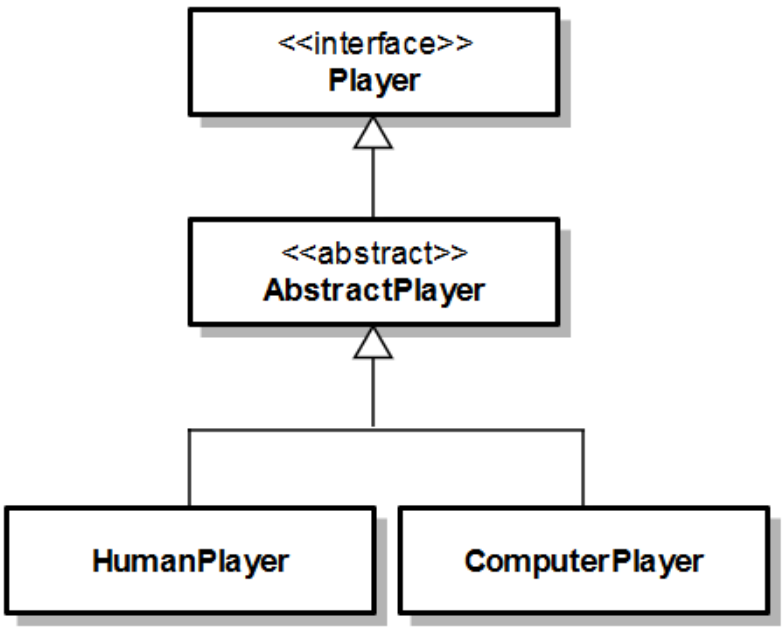
Introduction:

After seeing your feedback, it's obvious that there are many areas for improvement on this application (which is sort of the idea behind the development process). One of these improvements is to make the computer a touch 'smarter' -> rolling exactly 1 time per move becomes predictable pretty quickly. So that's what we'll focus on in this iteration. Specifically:

- Select one of many different algorithms for the `ComputerPlayer` to use
- Make it so that the computer's algorithm can be changed in the middle of a game
- Include a menu system to make it easier to make these algorithm selections

Background:

Currently the hierarchy of player classes is set up as:



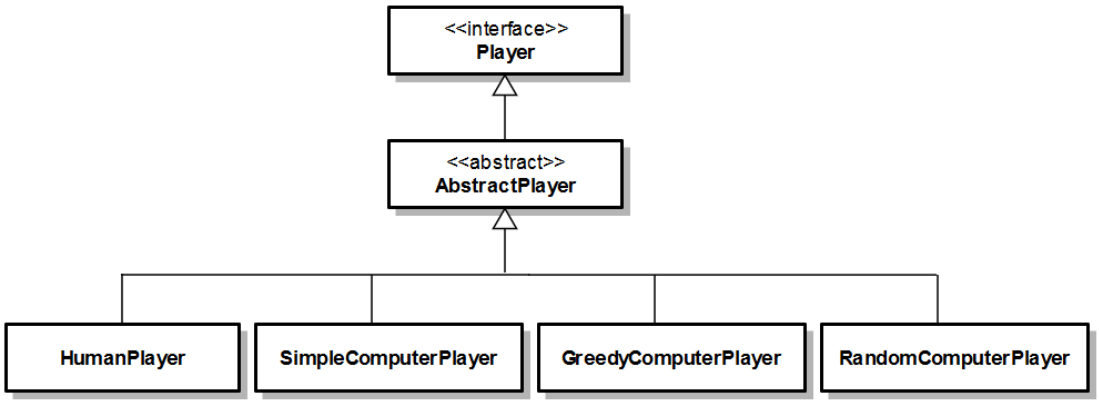
- The interface `Player` defines the method signatures for the behaviors that every type of player object must implement

- Abstract class `AbstractPlayer` implements `Player`. It implements the behaviors and data that are the same for `HumanPlayer` and `ComputerPlayer`, and defines abstract methods.

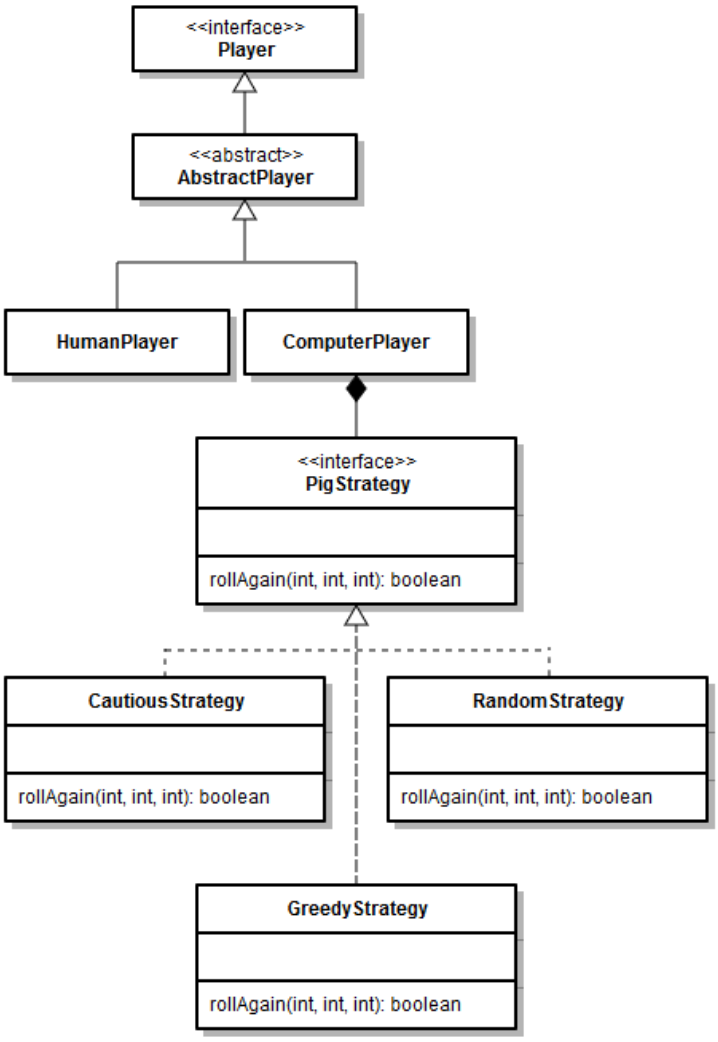
- `HumanPlayer` and `ComputerPlayer` implement the behaviors that they do differently.

There are two object-oriented approaches we could take to implement our new computer player's algorithms:

- Use inheritance to create new computer players, each with its own game-play strategy



- Use composition to let each `ComputerPlayer` object use a specified game-play strategy



- Each `ComputerPlayer` object has a data member of class `PigStrategy`
- The strategy a `ComputerPlayer` object will use is passed to it in its constructor or in a parameter to its `setStrategy` method and stored in the data member. (We'll have to code the method and data member)
- Method `takeTurn` in `ComputerPlayer` will call the `rollAgain` method of its `PigStrategy` data member

Using composition this way is called the Strategy Design Pattern. Please see this week's Readings link in Moodle for more details about Design Patterns and specifically the Strategy Design Pattern.

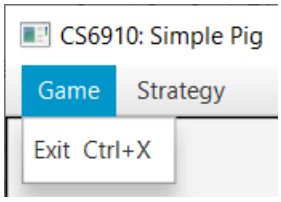
In general, it's good OO design to choose composition over inheritance, so we'll make use of Strategy because:

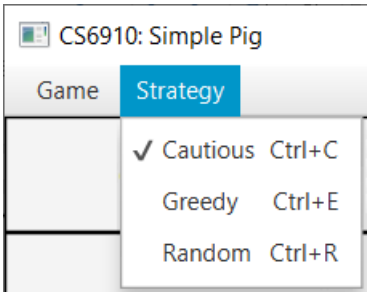
- This design makes it clear that `ComputerPlayer` objects differ in the way they determine whether or not to roll again, but are otherwise the same
- If we define a `setStrategy` method in class `ComputerPlayer`, the strategy can change during runtime. (By contrast, if we have used inheritance, the `ComputerPlayer` could only use the one strategy with which it is defined)

What to do:

We'll start with the GUI and then apply the Strategy pattern. Details will be given to get you started, but by the end we expect that you'll be able to complete the functionality following along the same design methodologies taught earlier.

The Menu Bar





1. Let's start with some GUI coding – add a menu bar to the application. For now, we'll just need two options: Game and Strategy:
   a. The File item has keyboard mnemonic G
      - Item Exit has mnemonic X and accelerator Control-X
   b. The Strategy item has mnemonic VK_S
      - Cautious has mnemonic VK_C accelerator Control-C
      - Greedy has mnemonic VK_E accelerator Control-E
      - Random has mnemonic VK_R accelerator Control-R

   HINTS:
   - Create private helper methods to build the menu bar and each menu.
   - Start by developing and user-testing the interface first (make sure it looks correct), then move on to the functionality one step at a time.

2. Add the functionality to the Exit menu item by defining an anonymous `EventHandler`. To exit the application, we'll have the `handle` method make a call to `System.exit(0)`.

   HINTS:
   - You can learn more about anonymous listeners at:
     - http://www.leepoint.net/notes-java/GUI/events/anonymous_listener.html
   - Anonymous inner classes are useful, but not always appropriate. Please see the following to learn more about them:
     - http://www.devx.com/tips/Tip/12814

3. When your Exit menu item works correctly, go ahead and commit the changes to the repository.

The `PigStrategy` interface

4. Inside package `edu.westga.cs6910.pig.model.strategies` add a new interface called `PigStrategy`

   The purpose of this new interface is to define the common interface for all the game-play algorithms for Pig. This interface will define a single method called `rollAgain` which will:
   - Accept three parameters:
     - The number of rolls already taken this turn
     - The number of points rolled so far this turn
     - The difference between the total points so far and the goal score
   - Returns true if the player should roll again, false otherwise

   HINT: Don't forget the interface and method comments!

The `CautiousStrategy` class

5. Inside package `edu.westga.cs6910.pig.model.strategies` add a new class `CautiousStrategy` that implements the `PigStrategy` interface.

   The purpose of this class is to implement the game-play strategy that always returns `false` (this is exactly what the last iteration of our application did) when asked if it should roll again. I appreciate that we're ignoring all of the parameters that are passed, but that's OK, we've just included these to make our interface more powerful (other strategies may make use of this information). There's no need for a constructor here, instead it can just use the default constructor provided.

   Once your class is complete, it's (always) a good idea to test it. Go ahead and add a new test class with an appropriate number of tests to confirm that this is working properly.

Modify the existing `ComputerPlayer` class

6. Now that we have an actual strategy (even though it's the one we're already using, we have a Java class to define it), we'll go about modifying the existing `ComputerPlayer` to work with it. This is a great way to implement new structure in your program because once you're done here, the application should run just as it did before.
   a. Add an instance variable of type `PigStrategy`
   b. Remove the existing 0-parameter constructor
   c. Add a new 1-parameter consturctor such that it:
      i. Accepts a parameter of type `PigStrategy`

ii. Does error checking to be sure the value of this new parameter is not null (otherwise it should throw an `IllegalArgumentException`)
iii. Assigns the value of the parameter to the instance variable

NOTE: Be sure your old JUnit tests still pass (if not, investigate why and fix it). Be sure to test any functionality

d. Add a new method called `setStrategy` that will:
   - Accepts a `PigStrategy` parameter, which is the strategy to be used
   - Precondition: the parameter is not `null`
   - Postcondition: the specified strategy will determine how the player will play

e. Finally modify `ComputerPlayer`'s `takeTurn` method so that it checks with the strategy to see if it should roll again. Note that part of the changes you'll need to make include replacing the existing counted loop (the for loop) with an appropriate conditional loop.

Modify the `Main` class

7. You probably noticed that as soon as you changed the `ComputerPlayer`'s constructor, this class stopped compiling. We'll fix that next by including code to create a `CautiousStrategy` object and then pass this object to the `ComputerPlayer` constructor. The application should compile now.

   Note that this means that when the application starts, it will always use the `CautiousStrategy` until the user selects a new one.

Modify the class `PigPane`

8. Because we have the ability to set a cautious strategy, let's go ahead and include code inside the application's menu system to set it (for later when we have other strategies to choose from).

   Add an anonymous `EventHandler` to the `Cautious` menu item. In its `handle` method, pass a new `CautiousStrategy` object to the game's computer player's `setStrategy` method.
9. Now is a good time to run the application and make sure it works as expected (like it did before). Once you're convinced things are working properly make another commit to your repository with a descriptive commit comment.

The `RandomStrategy` class

10. Add a new class `RandomStrategy` that implements the `PigStrategy` interface.

The purpose of this class is to implement the game-play strategy such that it is randomly decides (with equal likelihood) whether to roll again or not. Note be sure to use `Math.random` here for the decision.

Modify the class `PigPane`

11. It's time to code the GUI's menu system so that a `RandomStrategy` can be used.

    Add an anonymous `EventHandler` to the `Greedy` menu item to set the computer player's strategy to a new `GreedyStrategy` object.

12. Go ahead and run your application to be sure it's working as expected. One thing you'll notice right away is that you'll only ever be able to see the last roll of the die and the turn total. This is OK for now.

    Once you're happy with the application's ability to set strategies appropriate, go ahead and make a commit to your repository.

The `GreedyStrategy` class

13. Last, but not least is the GreedyStrategy class. Follow the pattern that we've used in the previous two strategies and implement this class such that the computer player will always attempt to roll at most three times. Notice how this time we do actually need to make use of some of the data passed to `rollAgain`.

14. When you have finished implementing the class, modify the GUI's menu so the user can choose this strategy. Run the application to be sure it performs as expected. When you're convinced it's working properly, go ahead and commit to the repository.

Update the User Guide

15. Because our application now has some new functionality, go ahead and add a new page to your Web User Guide to describe the changes. (Hint: describe all items in the new menu system).

Assumptions / ReadMe File

At any point in time you feel like there are assumptions that you need to make when doing this (or any) assignment, please create a text file named `ReadMe` and store this file in the top-level of your project (so when I unZip your project, I will see this file at the same level as the '`bin`' and '`src`' folders). Inside this file just list out what assumptions you felt that you needed to make when doing the exercise. Also explain any design decisions that you are concerned about and feel the need to explain. Note that I will *not* be grading this document – so if you don't include one, that's fine. I'm just giving you this opportunity to help explain what you are thinking ahead of time.

And of course, if your assumptions aren't completely fair ("I assumed that this is a stupid exercise and no one would ever want such a crappy piece of software" or "I saw that there were just so many flaws in the Starter File, that I threw it away and created my own application from scratch"), then I'll be sure to contact you about that and give you the opportunity to fix your files before moving ahead. Also, it's probably worth reading through the information about Grading below.

Submitting

When you are finished, make one final commit to your repository, be sure that all of the class files are saved, close Eclipse, and Zip the folder holding your Eclipse project using 7-Zip. Give your file the name 6910*YourLastName*Week03.zip. Upload the Zip file to the appropriate link in Moodle.

Grading:

Again, this is a pass/fail course – which is nice because it will allow us to talk about smaller issues that may come up along the way without worrying about maintaining a perfect 100% score. Once things are 'graded', please visit the Moodle Gradebook to see the comments I've left. Those submissions that receive 'Unsatisfactory' will need to get the issues fixed and approved by me (by sending your work via email) before proceeding to the next unit.

Remember, all work must be marked as Satisfactory to pass this course.