COMP 2150 - Winter 2021 - Assignment 3

Due April 1 at 11:59 PM

Mar 13: added note on number of cards.

Description:

In this assignment you will be implementing *"whodunit?"*, a game that involves deduction to determine the “who, where and how” of a murder. Each player makes guesses about the crime, and then other players refute that guess, if possible, based on information they have on cards that have been dealt to them. The game "*whodunit*?" is mildly similar to a well-known boardgame called "Clue". For a three-minute introduction to the board game Clue, please see this video: https://www.youtube.com/watch?v=sg_57S4l5Ng

You will implement the game in Java using interfaces and proper object-oriented programing. It will be a text-based game that allows one human player to play against a group of computer players.

Game Simplifications:

"*whodunit*?" is simpler than Clue. Most notably, there will not be a game board or dice in "*whodunit*?", so there is no movement in the game. Additionally, the players do not take on character identities in the game – players and suspects are separate in "*whodunit*?". The game will thus consist of players taking turns making guesses about the identity of the murderer, the weapon and the location of the murder. The first person to guess correctly wins the game. Also in "*whodunit*?", a turn consists of either a suggestion and an accusation (not both).

Major Assignment Parts

There are three major parts to the game.

1. **Players**: you must create two classes for game players. You are responsible for implementing one class for human players and one class for a computer player. These two player classes should use **IPlayer interface** to implement a specific set of methods. The Player interface represents the methods necessary for players of the game.

   Note: You **must** use the **IPlayer** interface and your players must satisfy the interface. Additionally, you should not use additional methods to access the player classes outside of the interface methods – your project must work identically if other (working) player classes are substituted for your classes. Failure to do so may result in lost marks. You cannot modify the interface.

2. **The Model**: this class manages all of the players of the game, all the cards in the game, and the turns of the board.

3. The main method: This will create the players, the cards and the model and then call the model to start the game. The game should allow one human player to play against a group of computer players. This can all be static code. The main method should ask how many computer opponents are desired before starting the work of the model.

   Note: the number of cards is not fixed – the model should be able to work for any non-zero number of cards of each category. Your final submission should set a moderate number of cards for each of the three categories (typically, 5-8 cards per category). These Card objects (see below) should be created automatically without asking the user in the main method.

Basic Classes

You should write two basic classes that help the game:

1. **Card**: a card is one of the cards of the game. It has two fields: a type (Weapon, Suspect or Location) and a value.
2. **Guess**: a guess is made by a player when they want to learn more information about the game. A guess consists of three cards, and a boolean variable for whether the guess is a suggestion or an accusation. (An accusation is the final guess of the game for a player – if they are correct, they win the game. If they are incorrect, they are eliminated from the game.)

These classes should have constructors that accept the required information. They should also have getters for the information, but no setters.

Model

The model is the portion of the program that stores the players and the cards, and runs the game. The model will give all players a number between 0 and n-1 where n is the total number of players. Players are arranged in order 0 through n-1 around the table. That is, after player i's turn, it will be player (i+1)%n's turn.

The model will create, shuffle and distribute all of the cards to the players, as well as holding the three “answer” cards (the who-where-how cards).

The model should receive info from the main method: it should get all the players and all the cards from the main method. No other setup should be done by the main method – all of the game initialization should happen in the Model.

The model must, during set up, “seat” the players at the table. That is, the model should assign indices to each of the players (see #1 below in the description of Player methods). Indices can be arbitrarily assigned. The model is also responsible for choosing the answer for the game (the victim, the location and the weapon) and then distribute all remaining cards to the players. Similar to the original board game, cards are distributed to players based on their index (i.e., player 0, then player 1, then player 2, …) and it is ok if some players receive one more card

than other players. All cards given to players are combined and distributed, not distributed separately by type (Weapon, Suspect, Location). This means that some players may not receive any cards of a type, which is ok.

Further, the start of the game should be initiated by the main method by calling a method from the Model class. See the pseudocode in the **Game Pseudocode** section below to see how the game is run. Once a game is played, the main program should terminate.

Players

A specific set of methods are declared in the IPlayer interface. Both Player classes (Human and Computer) should implement all the methods declared in IPlayer interface.

```
public interface IPlayer {
    public void setUp( int numPlayers, int index, ArrayList<Card> ppl,
                ArrayList<Card> places, ArrayList<Card> weapons);

    public void setCard (Card c);

    public int getIndex();

    public Card canAnswer(Guess g, IPlayer ip);

    public Guess getGuess();

    public void receiveInfo(IPlayer ip, Card c);
}
```

The six methods work as follow:

1. A method **setup** that receives five parameters: the number of players in the game, the index of the current player (what player number am I?), a list of all the suspects, a list of all the locations and a list of all the weapons.
2. A method **setCard** that indicates that the player has been dealt a particular card.
3. A method **getIndex** that returns the index of the player (e.g., I am player 2).
4. A method **canAnswer** which has two parameters: a player index i and a guess g. This method represents that player i (which is different from the current player) has made guess g. The current player is responsible for "answering" that guess, if possible. The method should either return a card (which the current player **must** have in their hand) or null, to represent that the current player cannot answer that guess.
5. A method **getGuess** (no parameters). If this method is called, it indicates that it is the current player's turn. The method should return the current player's guess for that turn. The guess may be a suggestion or an accusation.
6. A method **receiveInfo** which has two parameters: a player index i and a card c. This represents that the current player has made a guess (previously) and the player at index i has one of the cards of the guess, c. If no player can answer any information about the current player's previous guess, then c is null and i=-1. Otherwise, c is guaranteed to be

a card from the current player's previous guess and i is a valid index. This method is guaranteed to only be called after a call to getGuess by the current player.

The human player class must prompt the user for information when appropriate. For instance, the method canAnswer should show the human player the current guess and the current index i, show the player the cards they have that they can show player i and ask which card they wish to show. The program is not responsible for managing the human player's deduction (i.e., it does not need to track which cards they have been show by other players, etc.).

You must also implement a computer player. The computer players should make reasonable guesses. To ensure this, the computer players must pass the Junit tests described below. Beyond the requirements below, any behaviour of the computer player that follows the basic rules of the game is acceptable, except that computer players should never make accusations unless they are sure that the accusation is correct.

Unit Tests

To ensure that the computer player is making reasonable guesses, you must write a JUnit test file that demonstrates the following facts:

1. If a computer player has no cards, then canAnswer should return null.
2. If a computer player has exactly one card from a guess, canAnswer should return that card.
3. If a computer player has more than one card from a guess, canAnswer should return one of the cards.
4. If a computer player is given all but n cards (for some number n > 2 that you should choose) from the set of cards, a call to getGuess should return a guess that does not contain any of the cards that the player has been given. That is, an initial guess from a computer player must consist of cards it does not have.
5. If a computer player is given all but three cards from the set of cards, a call to getGuess should return the correct accusation (not a suggestion).
6. If a computer player is given all but four cards from the set of cards, a call to getGuess should not return an accusation. However, if receiveInfo is called with one of the four cards, then after that, a second call to getGuess should return the correct accusation.
7. If a human player is given some cards, and then canAnswer is called with a guess that includes one (or more) of the cards the player has, the method must return one of those cards (that is, the human player cannot give a card that they do not have in their hand – this will be achieved through input validation in your implementation).

For these tests, you should use a non-trivial set of cards (for people, places and weapons). You should also use a setup method. In JUnit, you should annotate one method with @BeforeEach with code that should be done (individually) before each method.

Each test (1-7) should be a separate method. You must use assertions to establish the statements above – no output should be produced other than output from the assertions. Name each method as testX with X from 1 to 7 for ease of verification.

You may write additional tests if you like, but are not required to.

Game Pseudocode

Here is the pseudocode for how the game progresses. Note that this pseudocode does not include details on every call to methods of the players.

```
while game is not over:
  ask active player for their guess
  if guess is an accusation:
    test if accusation is correct:
      if correct, game is over, active player is the winner.
      if not correct, active player is removed from the game.
      (game is over if there is only one player remaining. active player can be human)
  else:
    ask players if they can respond to the guess.
      (starting with next player after active player, including removed players)
    if another player can answer the guess:
      provide answer to active player
    else:
       report that no answer is possible to the active player
  if game has not ended:
    move to next active player.
inform all players about the outcome of the game
 (player and guess if someone guessed correctly, player only if all other players are
  eliminated)
```

Data Structures

In this assignment, you are not required to build you own data structures. You should use ArrayLists (with types) to store the information, as illustrated in the Interface.

You may find the Collections.shuffle() method helpful for shuffling ArrayLists of cards.

Other

A file of prompts that you can use for the human player have been included.

Submission

Submit all your source code for all classes you implement. For ease of compilation, you must provide copies of all classes and the interface in a zip file. You should also submit a readme.txt file (in the zip file) that describes exactly how to compile and run your code from the command line, and how to run the JUnit test file on aviary. The markers will be using these instructions exactly and if your code does not compile directly, you will lose marks. Each class must be in its own file.

You MUST submit all of your files in a zip file. Submit all files on UM Learn.