DevelopmentInfo

The developmentInfo class holds Strings representing the name of a development upgrade with an accompanying description String. The only constructor includes all arguments, used to instantiate in the developmentInfoManager class. Setters are not included as the Strings will not be modified after initial instantiation using the constructor. Getters are included for all Strings, allowing the values to be called outside of the class if required in the future. The class also includes a displayCurrentDevInfo method to be called after a player has completed a development of an element. This displays the current development name and its accompanying description.

DevelopmentInfoManager

The developmentInfoManager class statically holds all the Strings pertaining to each required developmentInfo object and instantiates each developmentInfo object and holds them as static objects. It also holds a List<DevelopmentInfo> allDevelopmentInfo which will hold all the developmentInfo objects to be linked to their respective elements in the game. Methods for addDevelopmentInfo and removeDevelopmentInfo, each taking a developmentInfo argument, add or remove their developmentInfo object to the list respectively. The single constructor takes no arguments but uses the addDevelopmentInfo method to add all the static developmentInfo objects to the allDevelopmentInfo list. This list can then be returned using the getAllDevelopmentInfo method.

RandomEvents

The RandomEvents class is used to generate a good or bad event, adding, or removing a generated quantity of resources from a randomly selected player. The class holds static Strings denoting all the 'good events' and 'bad events', and holds static integers defining the possible resource values that may be added/deducted. The class also holds lists goodEvents, badEvents, and fees, used to store the static values. The constructor then adds all the static values to their respective lists. The generateRandomEvent method takes a List<Player> argument from which it selects a random player. It also selects whether the event to occur will be good or bad, and then selects a random event from the appropriate list. This information is then printed to screen, and the appropriate randomly generated fee (from fees list) is credited/debited to the selected player's resources. The RandomEvents class can be easily extended to add further events or fees with no alteration to the generateRandomEvent method.

StateOfPlay

The StateOfPlay class holds a private List<Player> and Board, with associated getters and setters. This StateOfPlay class is then used in the DataSaver class to save the game state.

DataSaver

The DataSaver class holds a StateOfPlay object along with a String to denote the file name the StateOfPlay will be saved to. The setStateOfPlay and setFileName methods are used to set these objects. The saveData method declares a new FileOutputStream with the set file name as an argument, and new ObjectOutputStream to write the StateOfPlay to the set file, throwing an IOException in the event of an issue with the set objects.

Serialization

The Serialization class is instantiated in the main game to save and restore the game data. It holds two static StateOfPlay objects, lastSOP and restoredSOP, and a static DataSaver object. The SaveData method takes a List<Player> players and Board board arguments and instantiates the lastSOP and dataSaver objects, with an initial null restoredSOP. The setPlayers and setBoard methods of StateOfPlay are called to save the players and board to the lastSOP, with this SOP and associated file name then being set in the dataSaver. The method then calls the dataSaver saveData method, catching a possible IOException in the event of an error saving the file. The getData method takes a String argument of the file name where the data has been saved, returning a StateOfPlay object on successful loading of the object (null otherwise). The restorePlayers and restoreBoard methods each call the getData method to set the static restoredSOP, returning the List<Player> and Board objects respectively on successful restore, with appropriate exception handling to display error messages.

Game

The Game class has a single constructor which instantiates the private PlayerManager, DevelopmentInfoManager, Message, Serialization, Board, RandomEvents, Dice, and four ElementSystem objects, as well as a Set<ElementSystem> allSystems and a Map<Element, DevelopmentInfo> myDevMap. The squares and elements comprising the board layout are then instantiated and added to the board. A List<DevelopmentInfo> allDevelopmentInfo uses the getAllDevelopmentInfo method of the DevelopmentInfoManager class to populate the list, with these DevelopmentInfo objects then linked to their respective elements in the myDevMap Hashmap. Each of the four ElementSystem objects are populated with their comprising elements, and each of these ElementSystems added to the allSystems HashSet. A List<Dice> dice is then declared with two new Dice objects added, and this dice List then forms the argument of the declared diceRoller object. A private boolean isProgress is used to track the end game conditions, with a true value returned in the event of a concluded game.

The displayStateOfPlay method calls the Board displayElementDetails method to display the final board state.

The displayIntroMessage method takes a List<Players> argument, looping through to print each player name with validation to correct for punctuation, then displays the intro String of the Message class.

The postMoveOptions method displays the options available to the current player, with a switch statement to deal with each selection. It returns a boolean endgame (initially set to false) in the event of end game conditions being met.

1. Calls a displayAll method for each ElementSystem
2. Display your Resources & Properties Owned: calls the player displayAll method
3. Develop an element: calls the developmentMenu method, where the endGame boolean can be set to true in the event of full development completion of the game.
4. Trade an Element: calls the tradeElementMenu method.
5. End your Turn: exits the switch statement, returning a false endgame boolean.
6. Save Game: calls the SaveData method of serialization
7. Quit the Game: calls the quitGame method, which passes a true boolean to endGame and exits the switch statement.
8. Displays a visual board layout String boardLayout called from the Message class

The quitGame method prints the userQuitFail String and returns a true boolean to the postMoveOptions method to end the game in the event of a player choosing to quit, as per the game requirements.

The returnDevelopableElements method takes an argument of the current player, declaring and returning a List<Element> developableElements, populated by looping through the allSystems Set, with an if statement to check the player ownsFullSystem method, adding the elements of said system to the developableElements list if so. The elements are then sorted by board position using the player sortElements method.

The noDevelopmentsToMakeChecker method takes the List<Element> developableElements populated by the returnDevelopableElements method, returning a true boolean if the list is empty (i.e., the player has no developments to make). A message appropriate to this scenario is displayed. The method then checks if any elements in the developableElements list are fully developed, adding them to a new List<Element>. These fully developed elements are then removed from the developableElements list, with a true boolean returned in the events of all the player's elements being fully developed (i.e., list size zero), with an appropriate message printed in this case.

The developElement method takes Element and Player arguments, checking if the player has insufficient resources to develop the element and that the element is fully developed (preventing development if so, with appropriate message printed). If the player has sufficient resources, the method removes the resource cost for development from the player using the player removeResources method and increases the elements development level using element increaseDevlevel method.

The developmentMenu method return a boolean gameWin in the event of all elements being fully developed. It declares a List<Square> boardLayout populated via the board getSquares method. It also populates a List<Element> developableElements via the returnDevelopableElements method. A do...while loop presents options for the player to select any of their developableElements for development, initially calling the noDevelopmentsToMakeChecker to break the loop if the method returns a true boolean. When the player chooses a valid element number for development, the developElement method is called, and the displayCurrentDevInfo method of the element is called from myDevMap. After each development, a check is carried out to see if all elements are fully developed, setting the gameWin boolean to true, calling the displayEpilogue method and breaking the loop. The noDevelopmentsToMakeChecker is also called after each development to break the loop if a true boolean is returned.

The displayEpilogue prints the epilogue Strings from the Message class with the delay method of Message called as needed between each String.

The tradeElementMenu method takes arguments of the current player and a List<Player> players of all players of the game. Within a do...while loop, a Set<Element> is populated by the player getSquaresOwned method, with a List<Element> playerElementList populated from the Set. These elements are then sorted via the player sortElements method, and info displayed by the player displayPropertyOwnedInfo method. If this list is empty an appropriate message is printed and the do...while loop broken, exiting the trade menu. A for loop lists these elements as trade options for the player. If the player selects a valid element to trade, a TreeMap<Integer, Player> is populated and used in a do...while loop to display players the current player may trade with.

When the player selects a valid player to trade with, another do...while loop asks the selected player if they would like to purchase the element, calling the tradeElement method if yes, and printing an appropriate message if no. The trade menu will loop until the player chooses to exit to the main postMoveOptions menu.

The tradeElement method takes two Player arguments, the current player, and the buyer, as well as the element to trade. The method checks if the buyer can afford the trade (preventing the trade if not, with appropriate message printed). If the trade is valid, the method calls the player removeSquare method for the current player, and the player addSquare method for the buyer, also calling addResources and removeResources for player and buyer respectively. Dialogue is printed describing the trade.

The start method controls the main flow of the game. It takes a List<String> usernames, initially checking for an empty userNames list (in the event of a game restore scenario), calling the Serialization restorePlayers and restoreBoard methods if empty. The players' current location is then set to their location on the restored board. If usernames is populated, the createPlayers method of PlayerManager is called, with the players list populated via the getPlayers method of PlayerManager. Each player's starting position is then set to the first index of the board getSquares list. The displayIntroMessage method is called, and the first player's setCurrentTurn boolean set to true, with their setHasMoved boolean set to false. A for loop cycles through each player turn for the list of players, with this loop repeated via an outer while loop which tracks the isProgress boolean, continuing until and end game condition is met. Within the for loop, the player isCurrentTurn and hasMoved booleans are checked such that in event of a restored game, the start method will continue from the most recent save point in the game. The diceRoller roll method is called to determine a squaresToMove integer, which is then used as an argument of the board move method. After move, the player bankruptCheck method is called to check for a rent charge which may have left the current player with negative resources, setting the isProgress boolean to false and breaking the loop (ending the game). The current player's setHasMoved boolean is then set to true, the player displayAll method called and the game saved via the serialization SaveData method. The postMoveOptions method is then called, returning further game end conditions to set isProgress to false if returned. Once postMoveOptions has concluded, current player's setCurrentTurn and setHasMoved methods are called to set both booleans to false, the next player's currentTurn set to true. The game then saves again, and at the end of each loop of player turns, the generateRandomEvent method of RandomEvents is called. The bankruptCheck method of player is then called for all players to check whether the random event has left a player with negative resources. If so, the randomEventBankrupt String of Message is called, and isProgress is set to false.

The while loop continues until isProgress breaks the outer loop, finally calling the displayStateOfPlay method and printing a game over message.

GameManager

The GameManager class holds private PlayerManager and Game objects. The start method declares a List<String> usernames, populated by the PlayerManager getUserdata method. The start method then instantiates the Game and calls the start method of game with the usernames argument. The single constructor takes a PlayerManager argument, using this to set the private PlayerManager pm.

ArtemisLiteSystem

This class contains the main method of the project, where a new artemisLiteSystem object is instantiated via the constructor, which declares new PlayerManager and GameManager objects, and calls the GameManager start method to initiate the game.