German International University
Informatics and Computer Science
Prof. Dr. Slim Abdennadher

CSIS 201 - Programming II, *Spring 2025*
Jackaroo: A New Game Spin
Milestone 2
Deadline: 25.4.2025 @ 11:59 PM

This milestone is a further *exercise* on the concepts of **object oriented programming (OOP)**. By the end of this milestone, you should have a working game engine with all its logic, that can be played on the console if needed. The following sections describe the requirements of the milestone. Refer to the **Game Description Document** for more details about the rules.

- In this milestone, you must adhere to the class hierarchy established in Milestone 1. You are not permitted to alter the visibility or access modifiers of any variables nor methods. Additionally, you should follow the method signatures given in this milestone. However, you have the freedom to introduce additional helper methods as needed.
- All methods mentioned in this document should be `public` unless otherwise specified.
- You should always adhere to the OOP features when possible. For example, always use the `super` method or constructor in subclasses when possible.
- The model answer for M1 is available on CMS. It is recommended to use this version. A full grade in M1 doesn't guarantee a 100 percent correct code, it just indicates that you completed all the requirements successfully :)
- Some methods in this milestone depend on other methods. If these methods are not implemented or not working properly, this will affect the functionality and the grade of any method that depends on them.
- **You need to carefully read the entire document to get an overview of the game flow as well as the milestone deliverables, before starting the implementation.**
- Most of the methods are helper methods for others, so think wisely which methods could be used.
- Before any action is committed, the validity of the action should be checked. Some actions will not be possible if they violate game rules. **All exceptions that could arise from invalid actions should be thrown in this milestone.** You can refer to milestone 1 description document for the description of each exception and when it can occur.

1 Interfaces

1.1 Build the BoardManager Interface

1.1.1 Methods

This interface should define the following additional method signatures, which must be overridden by any class that implements it.

1. `void moveBy(Marble marble, int steps, boolean destroy) throws IllegalMovementException, IllegalDestroyException`
2. `void swap(Marble marble_1, Marble marble_2) throws IllegalSwapException`
3. `void destroyMarble(Marble marble) throws IllegalDestroyException`
4. `void sendToBase(Marble marble) throws CannotFieldException, IllegalDestroyException`
5. `void sendToSafe(Marble marble) throws InvalidMarbleException`
6. `ArrayList<Marble> getActionableMarbles()`

1.2 Build the GameManager Interface

1.2.1 Methods

This interface should include the following method signatures to be overridden by the classes implementing the interface:

1. `void sendHome(Marble marble)`
2. `void fieldMarble() throws CannotFieldException, IllegalDestroyException`
3. `void discardCard(Colour colour) throws CannotDiscardException`
4. `void discardCard() throws CannotDiscardException`
5. `Colour getActivePlayerColour()`
6. `Colour getNextPlayerColour()`

2 Classes

2.1 SafeZone Class

2.1.1 Methods

1. `boolean isFull()`: Checks whether all Safe Zone Cells are occupied.

2.2 Board Class

The validations found in this class are specifically related to the board and positions, some other validations will be through the cards themselves. Keep in mind the GameManager methods that you could utilize.

2.2.1 Methods

This class should include the following additional methods:-

1. `private ArrayList<Cell> getSafeZone(Colour colour)`: Returns an ArrayList of cells of a certain Safe Zone from the list of `safeZones` given the target `colour`, defaulting to null if Safe Zone color is not found.
2. `private int getPositionInPath(ArrayList<Cell> path, Marble marble)`: Return the index of a marble's position on a given path of cells, which could be the track or a Safe Zone, defaulting to -1 if the marble is not found on the given path.
3. `private int getBasePosition(Colour colour)`: Returns the index of the Base cell position on track for a given `colour`, defaulting to -1 if the Base is not found due to an invalid colour.
4. `private int getEntryPosition(Colour colour)`: Returns the index of the Entry cell position on track for a given `colour`, defaulting to -1 if the Entry is not found due to an invalid colour.
5. `private ArrayList<Cell> validateSteps(Marble marble, int steps) throws IllegalMovementException`: Validates the number of steps a marble can move and returns the full path of cells that the marble will take in order, from current to target keeping in mind the special cards:
   (a) `Four:`
   - Cannot move backwards in Safe Zone.
   (b) `Five:`
   - Cannot move opponent marble into any Safe Zone (mine or his), it will continue on the track keeping in mind the circular wrapping of the track (0-99).
   - When moving an opponent marble, I still cannot bypass or land on my own marbles, however, I am allowed to bypass or land on that opponent's other marbles.

   The method should do the following:
   (a) Identify the marble's current position, whether on the track or within the player's Safe Zone.
   (b) If the marble is neither on the track nor in the Safe Zone, throw an `IllegalMovementException` indicating that the marble cannot be moved.
   (c) If the marble is on track:
      i. If the number of steps is more than the distance the marble could move to reach the end, throw an `IllegalMovementException` indicating that the rank of the card played is too high.
      ii. If the number of steps is more than distance left to my Safe Zone Entry then it means the marble will move on both track and Safe Zone, the full path should include both **in order**.
      iii. Otherwise, the track part a marble will move from current position to target is added to the full path **in order** keeping in mind the circular wrapping of the track incase of forward or backward movement.
   (d) If the marble is within the player's Safe Zone:
      i. If a marble is moving backwards in Safe Zone, throw an `IllegalMovementException` indicating the reason.
      ii. If the number of steps is more than the distance the marble could move to reach the end, throw an `IllegalMovementException` indicating that the rank of the card played is too high.
      iii. Otherwise, the Safe Zone part a marble will move from current position to target is added to the full path **in order**.
   (e) Return the full path, where the current position is at the first index and the target position is at the last index.

6. `private void validatePath(Marble marble, ArrayList<Cell> path, boolean destroy) throws IllegalMovementException`: Validates the movement of a marble along the given path. The given full path starts from the marble's current position and ends at the target position. The given boolean value destroy is true only incase of the `King` card. The method should:
   (a) Check the cells in the given full path to validate all the movement rules mentioned in the **Invalid moves section in the game description.**
   (b) Keep in mind which rule applies to path only excluding target position and which applies to both path and target.
   (c) Keep in mind the special rule-breaking card (`King`)
   (d) Keep in mind that marbles in their Safe Zone are safe from any interference so even a `King` cannot bypass or land on a Safe Zone marble.
   (e) Throw an `IllegalMovementException` with the exact reason according to each case mentioned here and in the **Invalid moves section in the game description.**
7. `private void move(Marble marble, ArrayList<Cell> fullPath, boolean destroy) throws IllegalDestroyException`: Does the actual movement logic of changing position and destroying marbles. The method should:
   (a) Remove the marble from its current cell.
   (b) Handle marble destroying keeping in mind the special cards (`King`).
   (c) Place the marble in the calculated target cell.
   (d) If the target cell is a trap, destroy the marble, deactivate the trap, and assign a new trap cell using `assignTrapCell`.
8. `private void validateSwap(Marble marble_1, Marble marble_2) throws IllegalSwapException`: Ensures that a swap between two marbles is legal. The method should throw an `IllegalSwapException` in the following cases:
   (a) The two marbles aren't on the track.
   (b) The opponent's marble is safe in its own Base Cell.
9. `private void validateDestroy(int positionInPath) throws IllegalDestroyException`: This method should throw an `IllegalDestroyException` in the following cases:
   (a) Destroying a marble that isn't on the track.
   (b) Destroying a marble that is safe in their Base Cell.
10. `private void validateFielding(Cell occupiedBaseCell) throws CannotFieldException`: This method throws a `CannotFieldException` if a marble of the same colour as the player is already in the Base Cell.
11. `private void validateSaving(int positionInSafeZone, int positionOnTrack) throws InvalidMarbleException`: This method throws an `InvalidMarbleException` if the selected marble was already in the Safe Zone or if it wasn't on the track.

    In addition the class overrides the `BoardManager` interface's methods with the following functionalities:

12. `void moveBy(Marble marble, int steps, boolean destroy) throws IllegalMovementException, IllegalDestroyException`: Responsible for moving a given marble a number of steps on track, Safe Zone or both, handling collisions and special cards in the process. If destroy is set to true, it indicates a `King` card. The method should utilize the helper methods already done.

13. `void swap(Marble marble_1, Marble marble_2) throws IllegalSwapException`: Swaps 2 marbles on track after validating the swap using the helper method.
14. `void destroyMarble(Marble marble) throws IllegalDestroyException`: Destroys a marble by removing it from the track and sends it back to the Home Zone of the owner player (his collection of marbles). If the marble doesn't belong to the same player, it should validate the destroy using the helper method first.
15. `void sendToBase(Marble marble) throws CannotFieldException, IllegalDestroyException`: Places the marble in its player's Base Cell. If Base Cell is occupied, it validates fielding using the helper method and then destroys any opponent marble that is positioned in that Base Cell.
16. `void sendToSafe(Marble marble) throws InvalidMarbleException`: Sends a marble from their current location to a random unoccupied cell in their own Safe Zone after validating the action using the helper method.
17. `ArrayList<Marble> getActionableMarbles()`: Returns all the marbles that are on the track as well as marbles in the Safe Zone of the current player.

2.3 Card Class

2.3.1 Methods

These methods vary across subclasses, so any subclass with unique functionality should override these methods in accordance with the behaviors specific to each card type as described in the **Game Description**.

1. `boolean validateMarbleSize(ArrayList<Marble> marbles)`: Validates whether the given list of marbles contains the correct number of marbles that each card is supposed to act on (0, 1 or 2 based on the type of card).
2. `boolean validateMarbleColours(ArrayList<Marble> marbles)`: Validates whether the given list of marbles contains the correct color of marbles that each card is supposed to act on (same color as the player or not based on the type of card). In case of a `Jack` Card having two marbles, the order of marbles does not matter.
3. `abstract void act(ArrayList<Marble> marbles) throws ActionException, InvalidMarbleException`: Performs the specified action of the `card` on the selected marbles. Consider using specific methods from the board and game manager interfaces to assist with this functionality.

2.4 Deck Class

2.4.1 Methods

1. `static void refillPool(ArrayList<Card> cards)`: Adds a collection of cards to the existing card pool. This static method receives an ArrayList of Card objects and appends them all to the cardsPool. Used to refill the cardspool with the cards in the firepit once empty.
2. `static int getPoolSize()`: Returns the count of the cards that are remaining in the `cardsPool`.

2.5 Player Class

2.5.1 Methods

1. `void regainMarble(Marble marble)`: Adds a specified marble to the player's collection of marbles which acts as the player's Home Zone.

2. `Marble getOneMarble()`: Returns the first marble without removing it, if any, from the player's marble collection that acts as the player's Home Zone. If no marbles are available, it should return null.

3. `void selectCard(Card card) throws InvalidCardException`: Checks if the given card is available in the player's hand and sets it to the `selectedCard`. Throws an `InvalidCardException` if the card does not belong to the current player's hand.

4. `void selectMarble(Marble marble) throws InvalidMarbleException`: Selects a marble to be used in the game by adding it to the `selectedMarbles`. Throws an `InvalidMarbleException` if trying to select more than two marbles.

5. `void deselectAll()`: Clears all selections, including the selected card and any selected marbles, resetting the player's choices.

6. `void play() throws GameException`: Executes the selected card's action using the chosen marbles. Initially, it checks if a card has been selected; if not, it throws an `InvalidCardException`. It then validates the number and color of the selected marbles are appropriate for the selected card; if not, it throws an `InvalidMarbleException` stating the reason. Upon passing all checks, the method allows the selected card to act with the selected marbles.

2.6 CPU Class

2.6.1 Methods

For simplicity, the method in this class is readily available to use on cms. Here's a detailed explanation of what the method does:

1. `void play() throws GameException`: Method that automates the game process by attempting to play cards using available marbles. It initiates by retrieving actionable marbles and the player's current hand of cards. The cards are shuffled to randomize the selection. For each card:
   (a) The card is selected.
   (b) The potential counts of marbles (0, 1, or 2) that can be acted upon is computed based on the card's marble size validation.
   (c) The counts are shuffled to randomize the selection process.
   (d) For each count:
      i. If zero marbles are needed and valid, it executes the card's action with no marbles and terminates if successful.
      ii. For one marble, it shuffles the actionable marbles and attempts to execute the card's action with each until one succeeds.
      iii. For two marbles, it considers pairs from the shuffled list, attempting the card's action for each valid pair until one pair successfully triggers the card's action.
      iv. The act method may throw exceptions which are non-critical and should not halt the process. To manage this, each invocation of the act method is wrapped in a try-catch block. Exceptions are caught and ignored, allowing the game to continue with further attempts.
      v. If non of the cards at hand are playable and nothing was executed, the method sets the selected card with the first card at hand without acting upon it again.

2.7 Game Class

2.7.1 Methods

This class should include the following additional methods:-

1. `void selectCard(Card card) throws InvalidCardException`: This method allows the current player to select the given `card`.
2. `void selectMarble(Marble marble) throws InvalidMarbleException`: This method allows the current player to select the given `marble`.
3. `void deselectAll()`: This method allows the current player to deselect all previously selected card and marbles.
4. `void editSplitDistance(int splitDistance) throws SplitOutOfRangeException`: Sets the `splitDistance` to the given value, throwing a `SplitOutOfRangeException` if the provided value is outside the appropriate 1–6 range. Particularly used for the Seven card's movement when a player chooses to split their move between two marbles. The `splitDistance` represents how far the first marble moves, with the remainder applied to the second marble (7 - splitDistance).
5. `boolean canPlayTurn()`: Checks whether the player's turn should be skipped by comparing their hand card count against the `turn`.
6. `void playPlayerTurn() throws GameException`: This method allows the current player to play their turn.
7. `void endPlayerTurn()`: This method ends the current players turn after it was skipped or after they've played their selection by:
   (a) Removing the current player's selected card from their hand and adding it to the `firePit`.
   (b) Deselecting everything the current player has selected.
   (c) Moving on the next player and setting them as the current player.
   (d) Starting a new turn once all players have played a card and the play order is back to the the first player.
   (e) Starting a new round once 4 turns has passed by resetting the turn counter.
   (f) Refilling all players' hands from the deck when starting a new round.
   (g) Refilling the Deck's card pool with the cards in the firepit and clearing it if the cards pool has fewer than 4 cards to draw.
8. `Colour checkWin()`: Checks whether one of the players have won or not by checking if any player have their Safe Zone completely occupied by their marbles. The method should return the color of the winning player if a player has won, and `null` if no player has won yet.

   In addition to overriding the `GameManager` interface with the following functionality:

9. `void sendHome(Marble marble)`: Regains a marble back to the owner's Home Zone (collection of marbles).
10. `void fieldMarble() throws CannotFieldException, IllegalDestroyException`: Gets one marble from the current player. If the marble is null it should throw a `CannotFieldException`. Otherwise, it attempts to field the marble into the board by sending it to the Base Cell of the player, if the marble is fielded, it is then removed from that player's Home Zone (collection of marbles).
11. `void discardCard(Colour colour) throws CannotDiscardException`: Discards a random card from the player with the given colour. If the player has no cards in hand to be discarded, a `CannotDiscardException` is thrown.

12. `void discardCard() throws CannotDiscardException`: Discard a random card from the hand of a random player other than the current.
13. `Colour getActivePlayerColour()`: Returns the colour of the current player.
14. `Colour getNextPlayerColour()`: Returns the colour of the next player.