SOFTWARE DESIGN DOCUMENT

Dream Team

DEREK RANDALL
MAURO CHAVEZ
STEFAN DOUCETTE
JEFFREY PAULS

Table of Contents

REVISION HISTORY

Revision Number	Revision Date	Nature of Revision	Approved By
1	07/25/2020	Table of Contents Amended to include Revision History	Derek
2	7/29/2020	Updated Initialize Game diagram with loop clauses and reorganized class objects	Mauro

1. Architecture

Game Manager: When we moved from the requirements phase to the design phase, we took on some feedback around our use cases that we used to redesign our structure. In our updated UML class diagram, we now have an overarching subsystem called Game Manager which allows a user to access Gameplay (by starting the game), Question Bank (by viewing questions), or quit the program. This change makes more sense from an end-user perspective, and it allows us to consolidate several use cases. The Game Manager subsystem consists of just one class, GameManager, and the instances of this class will serve as boundary objects for the other two systems. It is how the user will interact with and access the gameplay or question bank.

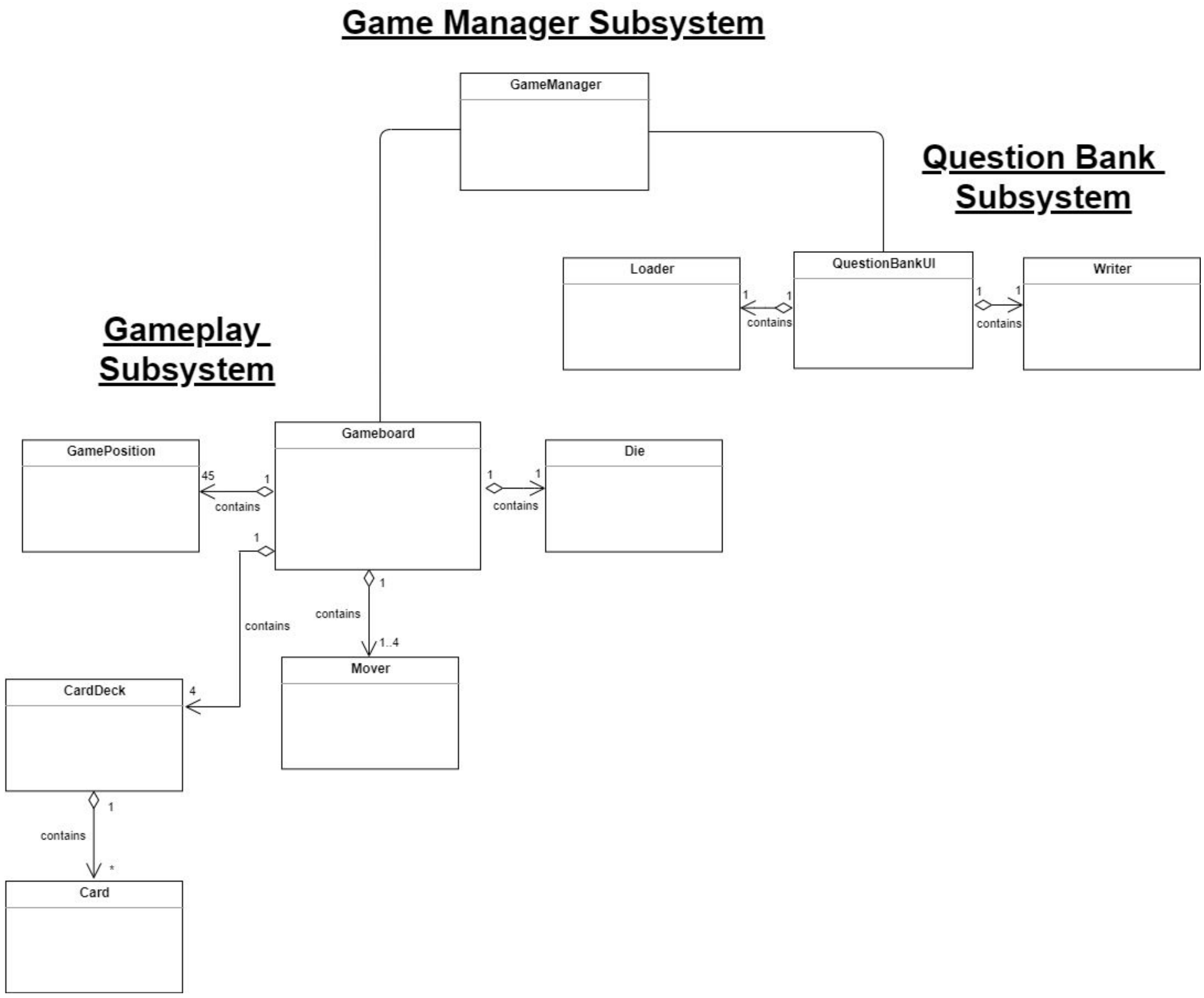
Gameplay: The gameplay subsystem is by far the largest portion of the software, and it will also be where a user spends the majority of their time while using the program. It consists of 7 classes which are all centered around the main driver, Game Board. The Game Board is a singleton class, as there should never be more than one instantiation of it at any given time, and is the starting point of most of the various interactions in this subsystem, and as shown in our UML Class Diagram, we have defined it as an aggregator of the Game Position, Card Deck, Mover, and Die Classes.

Question Bank: The question bank subsystem is where a user will be able to view and/or change the questions in the question bank. The system has a User Interface that allows a user to access the Loader and Writer classes which will manage interactions with a back end directory of text files. We have defined the Question Bank UI as an aggregator of both the other questions.

For more detailed information on classes and their attributes and methods, please see the Class Specifications (Section 3)

2. Static Design (Class Diagram)

Note: Details of each class are described in detail in the next section.



3. Class Specifications

** The following class specifications use Python syntax to describe class constructor method headers: __init__(parameters)*

Die

Attributes:

- num_sides: int

Methods:

- __init__(num_sides: int)
  - Class constructor taking the number of sides on the die as input
- roll()
  - Simulate the rolling of a num_sides sided die by randomly generating a number between 1 and num_sides inclusive

Card

Attributes:

- type: str
- question: str
- answer: str

Methods:

- __init__(type: str, question: str, answer: str)
  - Class constructor that takes strings specifying the type, question, and answer of the card. Implementation Note: The type string should be an enumerated type so that we can ensure all classes use the same type labelings.

CardDeck

Attributes:

- type: str
- cards: List[Card]

Methods:

- __init__(type: str)
    - Given a type string, initialize a deck of cards with an empty list of cards

- load_questions_from_file(path_to_question_bank: str)
    - Load the most recent question bank from the preset local path where the Writer class saves files to. Questions only of a specific type will be loaded where the type is specified by the CardDeck's type attribute. Info will be loaded from the question bank file system and each question and answer pair will be used along with the CardDeck's type to inititalize a card object and append it to CardDeck's cards attribute. It will be this method's job to scramble the order of the Cards in CardDeck's cards list.

- deal_card(type)
    - Select the first item in the CardDeck's card list, delete it from the list, and return it

GamePositions

Attributes:

- side_length: int
- center_index: int
- total_perimeter: int
- matrix : List[List[str]]

Methods:

- __init__(side_length: int)
    - side_length will by default be set to 11. Constructor will determine the center_index of the matrix to be identified as the center of the board and the position that all spokes converge on. It will also use the side_length to determine the total_perimeter of the game board. Using this information it will construct a matrix (in Python a list of lists) and label the types of each legal position.

- print()
    - Used to print the game board to the command line for validation purposes

- find_next_position(start_pos_x: int, start_pos_y: int, spaces_to_move: int, direction: str)
    - This method will take a start x and start y position, supplied from data gathered from a mover object by the GameBoard, and figure out where the end x and y positions are on the matrix after a movement is carried out in

the specified direction. In the case that the movement path crosses a spoke, use ask_user_for_path() to figure out what direction to move in.

- ask_user_for_path()
    - Give user the opportunity to decide what direction to move in if their movement path crosses a position where a spoke meets the board perimeter

- get_position_type()
    - Look up the corresponding type to the position the mover has moved to. This can be one of 4 colors (representing question categories), a roll again space, or the final space when all wedges have been collected.

Mover

Attributes:

- mover_name: str
- curr_x_pos: int
- curr_y_pos: int
- wedges: List[str]
- on_spokes: bool
- mover_color = str

Methods:

- __init__(name: str, mover_color: str, start_pos_x: int, start_pos_y: int)
    - Using the input parameters creates a mover, initializing the wedges attributes as an empty list and the on_spokes bool as false

- update_pos(new_x: int, new_y: int)
    - Update the position of the mover by taking in a new_x and new_y positions to overwrite the current values stored in the curr_x_pos and curr_y_pos attributes

- get_pos()
    - Retrieve the current position of the user’s mover

- add_wedge()
    - Add a wedge pertaining to the current space’s color when a user answers a question of that type correctly

- render()
    - Given the rendering of matrix of GamePositions, render the mover object at its current x and y positions

GameBoard

Attributes:

- instance : (static class variable pointing to instance of GameBoard)
- players: List[Mover]
- green_deck: CardDeck
- red_deck: CardDeck
- white_deck: CardDeck
- blue_deck: CardDeck
- die: Die
- game_positions: GamePositions

Methods:

- __init__(num_players: int, player_names: List[str])
  - Initialize a red, blue, green and white deck of cards along with a Die and GamePositions object. Initialize a list of movers for each player using the num_players and player_name attributes. Assign colors at random and have all players start in the center of the board. This method is not to be used publicly

- instance()
  - Returns a pointer to the singleton object stored in GameBoard's instance pointer. If the pointer is null, it calls the class constructor to initialize the new object and store it under the instance pointer.

- main_gameplay_loop()
  - Loop through movers allowing each to take a turn. Only continue on to the next mover once the current mover cannot take anymore turns.

- take_turn()
  - Facilitate a user rolling the die and picking a direction to move in. This will also move the user's mover, ask the user a question, get the user's response, reveal the answer, and record if the user got the question correct.

- present_die()
  - Ask the user to roll the die via UI element. Upon user trigger, a random number will be displayed showing results of roll before mover position is updated.

- ask_user_direction()
  - Ask the user which direction they want to move in.

- display_question()

    - Show user the question and get their answer

- display_answer()
    - Show the user's answer, show the correct answer to the question, and ask the user to confirm if it is correct

- report_end_of_turn()
    - Report via a UI component that the current player’s turn is over and wait for user input as acknowledgement before returning

- report_end_of_game()
    - Report via a UI component that the game is over

- draw_board()
    - Initialize the actual game board with both perimeter and interior spaces. Space types include 4 colors and ‘roll again’ spaces. If the board is already drawn, update the board with the new properties and draw again.

QuestionLoader

Attributes:

- question_file_path: str
- question_collection: list[Dicts[str: str]]

Methods:

- __init__(path_to_question_files)
    - Given the path to the location of the flat files storing questions for the game, make a call to the private _load_questions(). method to load all questions in the form of dictionaries with the following keys (one per question): “type”, “question”, “answer”.

- _load_questions(path_to_question_file)
    - Using the path_to_question_file, find the file and load it’s contents (questions) in the form of dictionaries with the following keys (one per question): “type”, “question”, “answer”. Append each of these dictionaries to a list and return the list.

QuestionWriter

Attributes:

- path_to_files: str

- question_collection: list[Dicts[str: str]]

Methods:

- __init__(path_to_questions: str, question_collection: list[Dicts[str: str]])
    - Class constructor that takes a path to the location of the question files for writing. Also takes in a question_collection as created by a QuestionLoader object. This means that one should only ever use a QuestionWriter after a QuestionLoader has done its work.

- insert_question(q_type: str, q: str, a: str)
    - Add a question to the question_collection by creating a new dictionary of the form {“type”: q_type, “question”: q, “answer”: a} and append it to the question_collection list

- write_questions_to_files(self)
    - Take the current contents of the question_collection and write them to the files specified by the path_to_questions attribute

QuestionBankInterface

Attributes:

- path_to_questions: str
- loader: QuestionLoader
- question_writer: QuestionWriter

Methods:

- __init__()
    - Using the internal path_to_questions class attribute, initialize a QuestionLoader and then pass the question_collection attribute of the QuestionLoader into the constructor for the QuestionWriter

- add_question(q_type: str, q: str, a:str)
    - Wrapper around calling QuestionWriter’s insert_question method

- save_questions()
    - Wrapper around calling QuestionWriter’s save_questions method

- view_questions()
    - Display contents of question_collection

GameStarter

Attributes:

- None

Methods:

- __init__()
  - Initialize GameStarter object

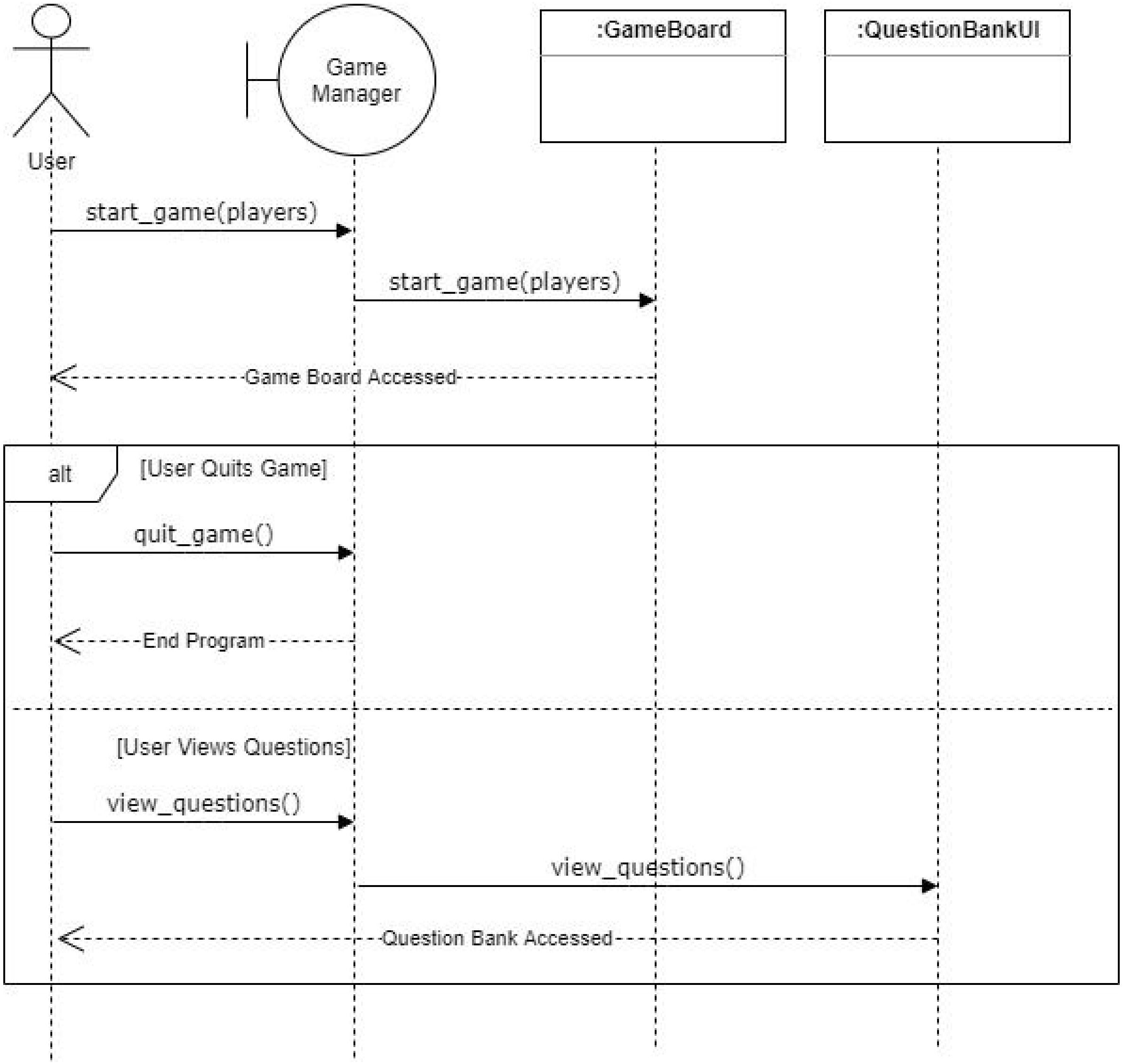
- draw_ui()
  - Draw menu system that allows users to start a new game or view questions

- start_game()
  - Initialize GameBoard object

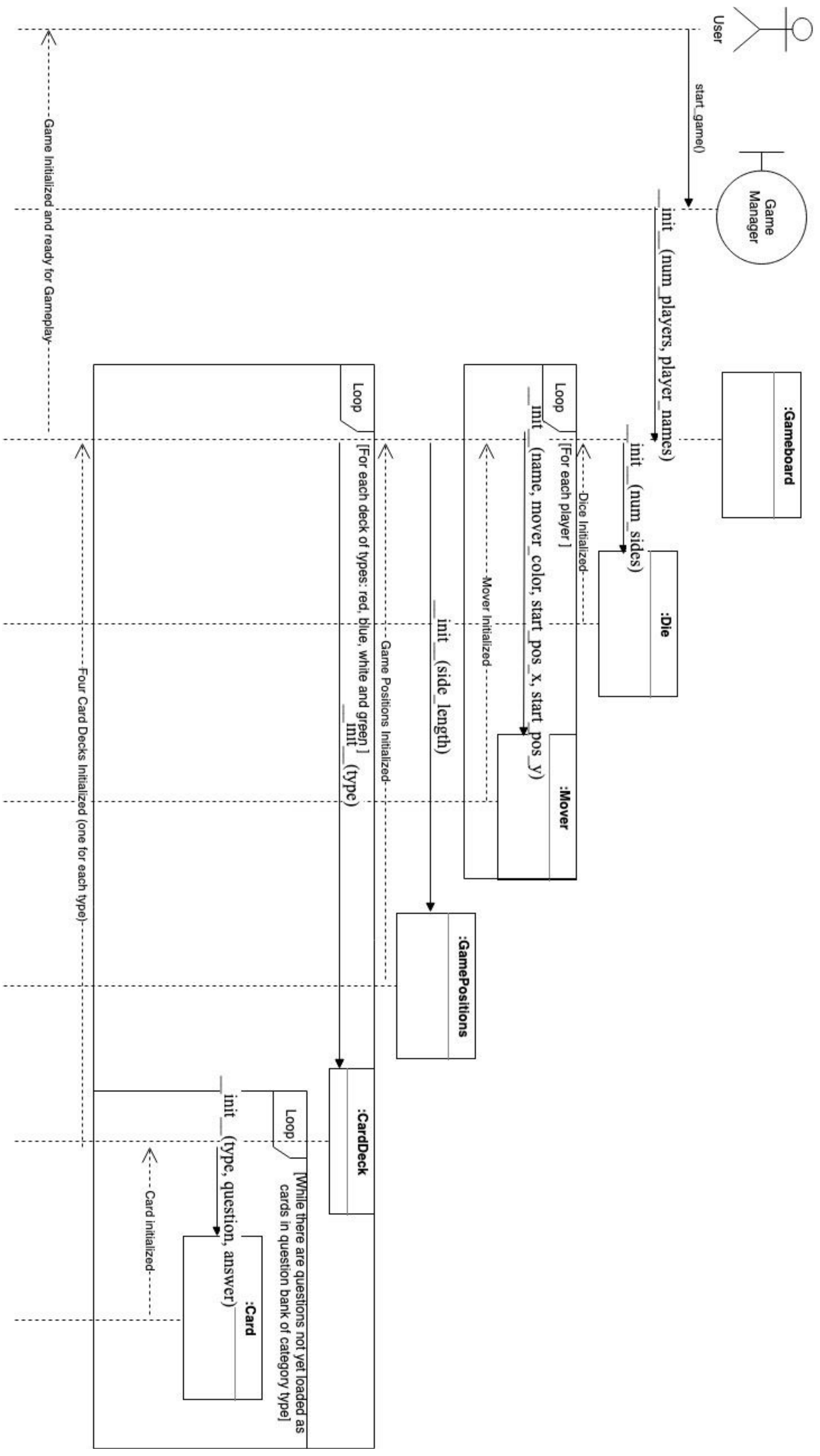
- view_questions()
  - Initialize QuestionBankInterface object

4. Dynamic Design (Sequence Diagrams)

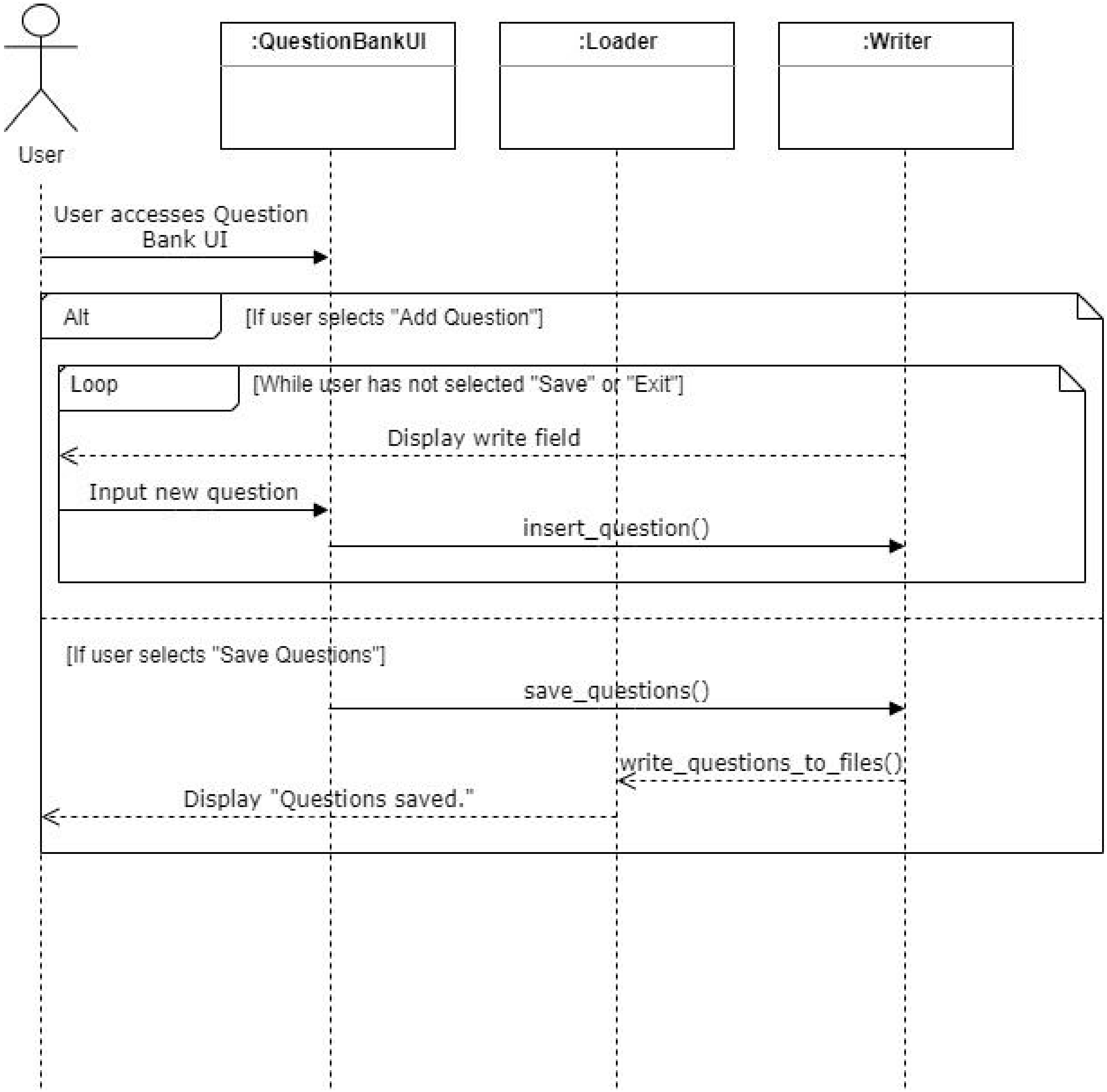
4.1. Manage Game



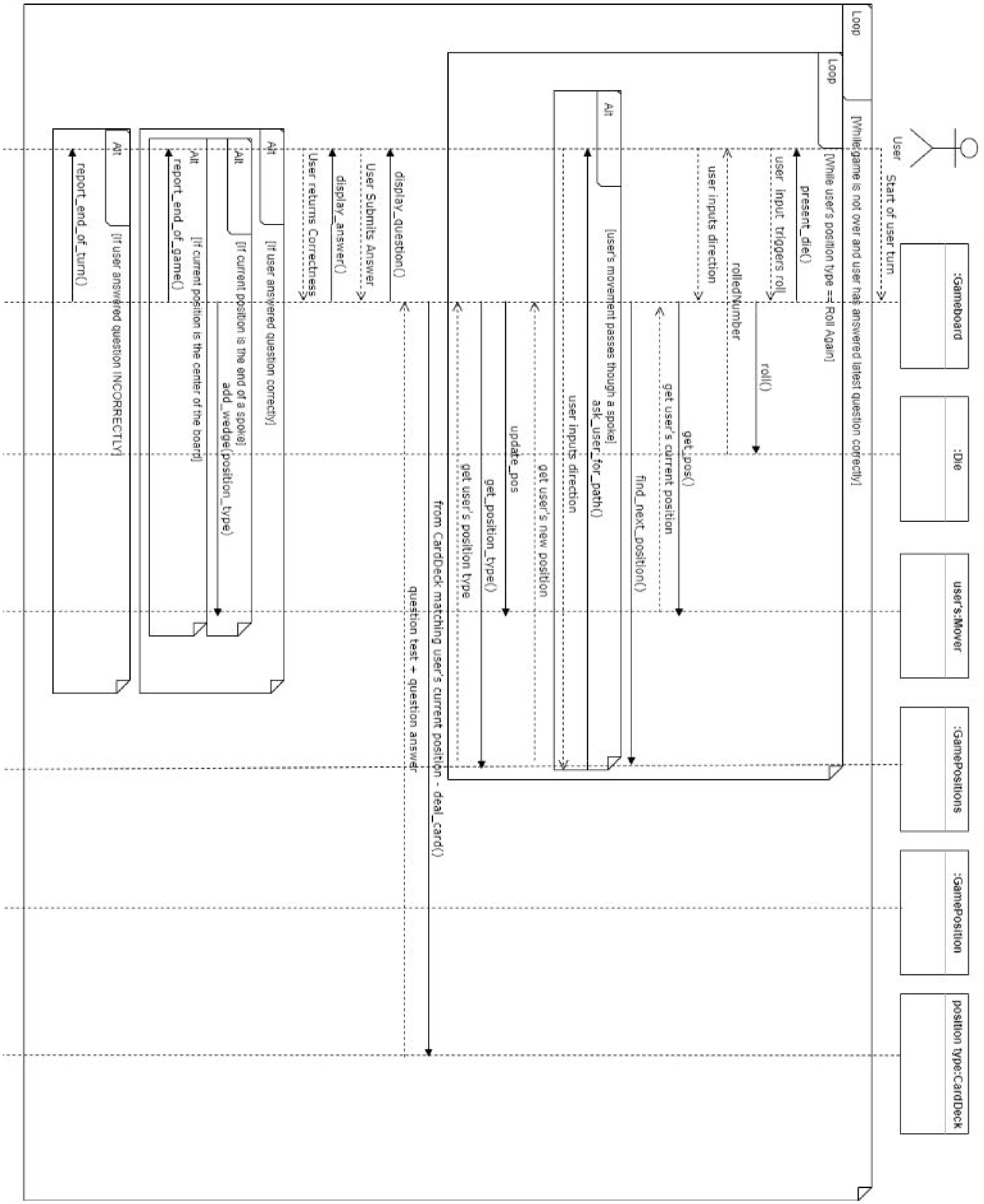
4.2. Initialize Game Board


User
start_game()
Game Manager
:Gameboard
__init__(num_players, player_names)
__init__(num_sides)
:Die
Dice Initialized
Loop
[For each player]
__init__(name, mover_color, start_pos_x, start_pos_y)
:Mover
Mover Initialized
__init__(side_length)
:GamePositions
Game Positions Initialized
Loop
[For each deck of types: red, blue, white and green]
__init__(type)
:CardDeck
Loop
[While there are questions not yet loaded as cards in question bank of category type]
__init__(type, question, answer)
:Card
Card Initialized
Four Card Decks Initialized (one for each type)
Game Initialized and ready for Gameplay

4.3. Modify Question Bank


User
:QuestionBankUI
:Loader
:Writer
User accesses Question Bank UI
Alt
[If user selects "Add Question"]
Loop
[While user has not selected "Save" or "Exit"]
Display write field
Input new question
insert_question()
[If user selects "Save Questions"]
save_questions()
write_questions_to_files()
Display "Questions saved."

4.4. Take Turn


User
:Gameboard
:Die
user's:Mover
:GamePositions
:GamePosition
position type:CardDeck
Loop
[While game is not over and user has answered latest question correctly]
Start of user turn
Loop
[While user's position type == Roll Again]
present_die()
user input triggers roll
roll()
rolledNumber
user inputs direction
get_pos()
get user's current position
find_next_position()
Alt
[user's movement passes though a spoke]
ask_user_for_path()
user inputs direction
get user's new position
update_pos
get_position_type()
get user's position type
from CardDeck matching user's current position - deal_card()
question test + question answer
display_question()
User Submits Answer
display_answer()
User returns Correctness
Alt
[If user answered question correctly]
Alt
[If current position is the end of a spoke]
add_wedge(position_type)
Alt
[If current position is the center of the board]
report_end_of_game()
Alt
[If user answered question INCORRECTLY]
report_end_of_turn()