Kings in the Corner Design Document

Introduction

This is the design document for the Kings in the Corner project. The project was created for CS342 at UIC. The project's creator is Lawrence Chu, who can be contacted at chu22@uic.edu.

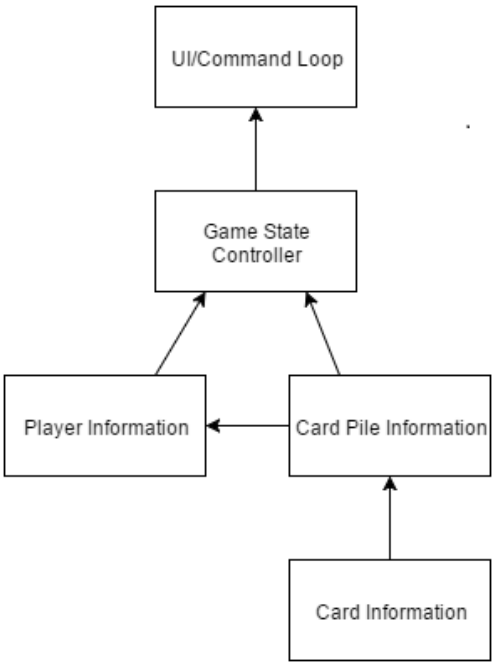
Section 1 - Purpose

The purpose of this project is to create a computer game version of the card game, Kings in the Corner, that supports a single player against a computer AI. The game follows standard rules set out at http://www.pagat.com/domino/kingscorners.html, with a target score of 25 points.

The user will interact with a text-based UI, which will present the game state, print AI moves, and allow user input for all game actions on the players' side, as well as quit, help, and about commands. On game over, the user can choose to play again or quit.

Section 2 – High-level Entities

The high level entities in this program include the UI/command loop, the game state controller, the players, the card pile classes, and the cards. Figure 1 shows how these entities interact with each other.



The UI/Command Loop handles all user input, including translating and sending information to the game state controller and error checking of user input. The game state controller handles everything else, which encompasses all game-related information, such as the players, cards, round, and points. It handles all error-checking with regard to any game action.

The major constructs of the game state revolve around the players, who have their own set of hands and points, the card piles, such as the draw deck, hand, and lay down piles, and the cards.

Figure 1

Section 3 – Design

UI/Command Loop

Usage

The UI/command loop will handle all user input and visual output with a text-based UI. Its role is to translate user input and execute the requested command, and output information about the game state to the user. When a game-related command is input, the entity will interact with the game state controller. Its scope is limited completely to user interaction, and it must interact with the game state controller to retrieve information about or make changes to the game state. This separation between the user commands and the game keeps each entity safe from unintended effects due to changes in the other entity.

Model

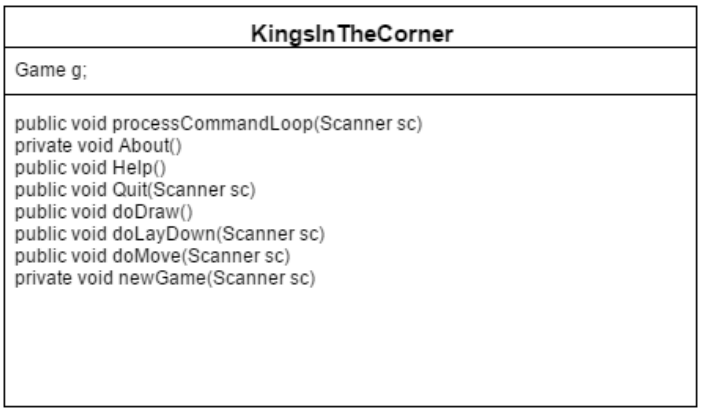


figure 2.

Figure 2 shows the contents of the UI/command loop class, KingsInTheCorner. It has only one data member, the game state controller it creates. The main function simply calls the processCommandLoop function, which will process user input and call the necessary function to execute the command. The 3 game-related commands will send the user input information to the game state controller to change the game state, and the quit, help, and about commands will quit the game, give information about user commands, and tell the user about the program.

Interaction

Figure 3 shows the interactions between the command loop and the game state controller. The interactions occur either from direct user commands, like Draw() and LayDown(), or through indirect automatic checking that occurs after one of the game-related user tasks have been completed.

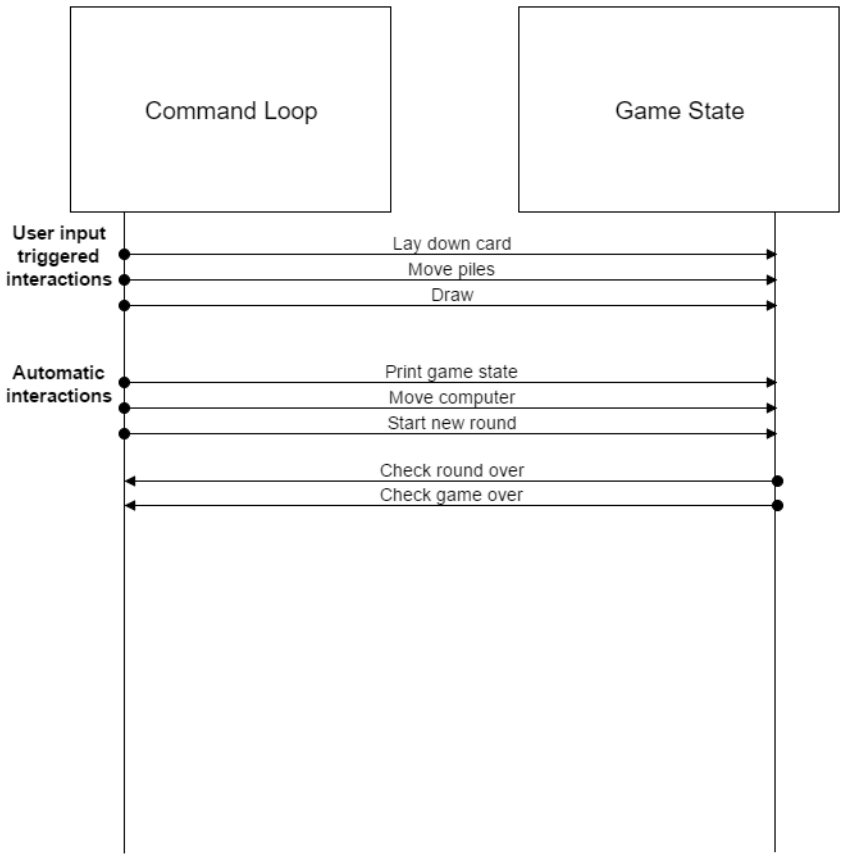


figure 3.

Game State Controller

Usage

The game state controller handles all information related to the game, and acts a go-between for the user/command loop and the different components of the game state, such as the players and cards. The controller provides the UI/game loop with high level access to game information and game mechanics that can change the game state. It then processes the request and manages the lower level components to execute the necessary game mechanics and save the changes to the game state as required. The game state controller is meant to provide a higher level abstraction that facilitates understanding in the game mechanics and makes working with the higher level game mechanics/user commands less error-prone by reducing the need to worry about smaller details.

Model

Figure 4 shows the main content of the Game class, which contains all the necessary data needed in the game and defines functions for each high level game mechanic in the game, which the command loop will call.

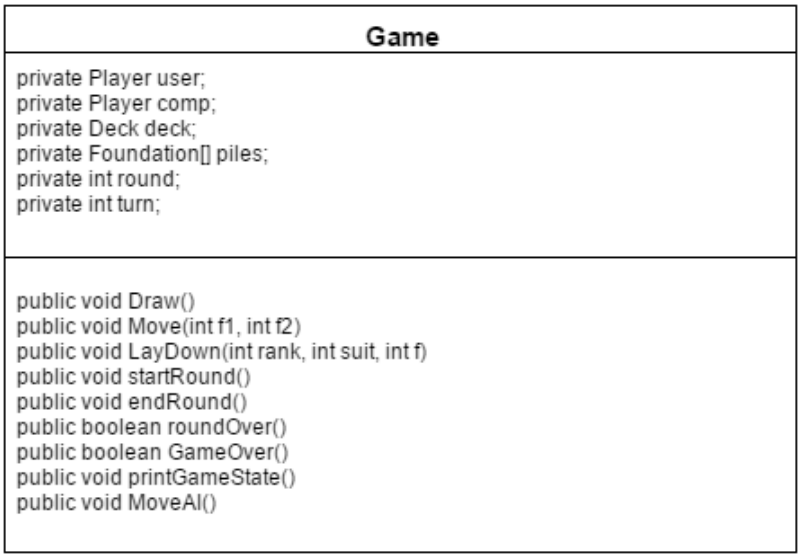


Figure 4.

Interaction

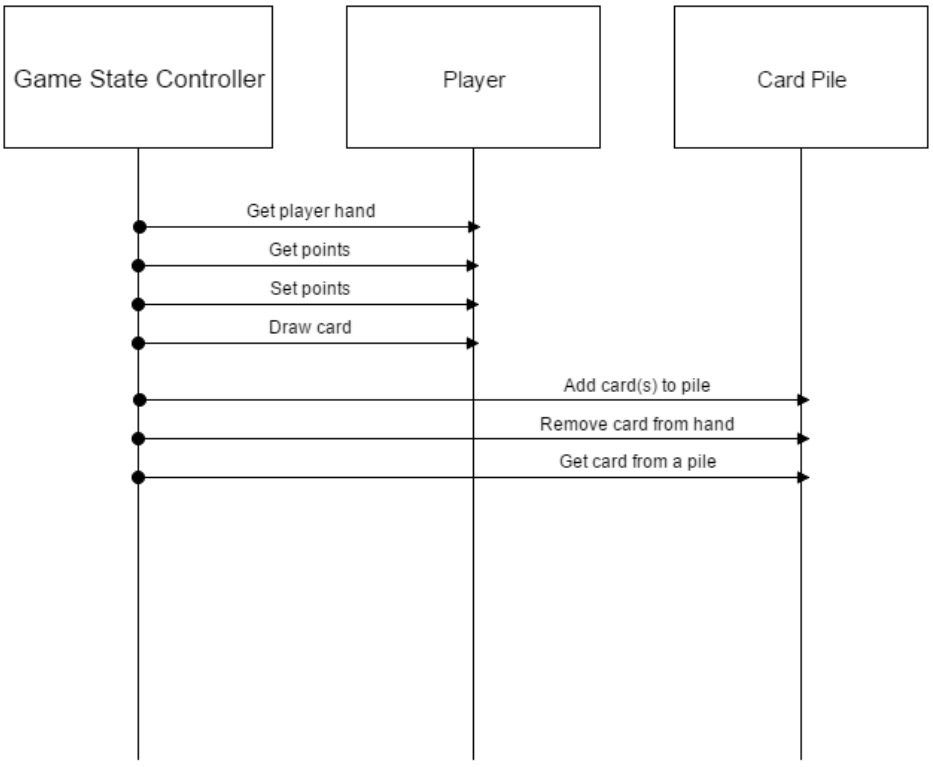


Figure 5.

Figure 5 shows all the interactions the game state controller has with the different game objects. It does not directly call any of the Card class methods, instead relying on the player and card pile classes to manage those details.

Player Information

Usage

The player object is responsible for holding all information tied to a player, namely the hand and the penalty points accumulated. The player class provides access to this data and stores any changes to it. The hand is manipulated through its own class, but the player class has the functions necessary to change point totals. The player class provides data encapsulation for the player, making sure only the game state controller can access the hands and points of each player.

Model

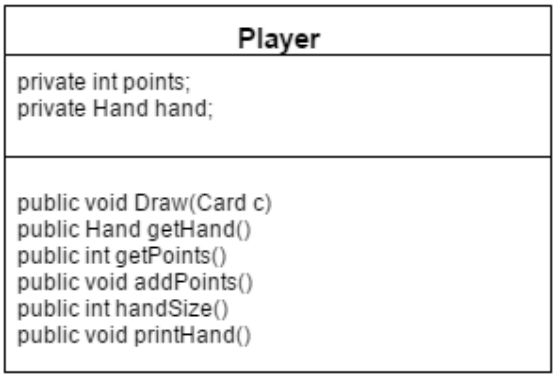
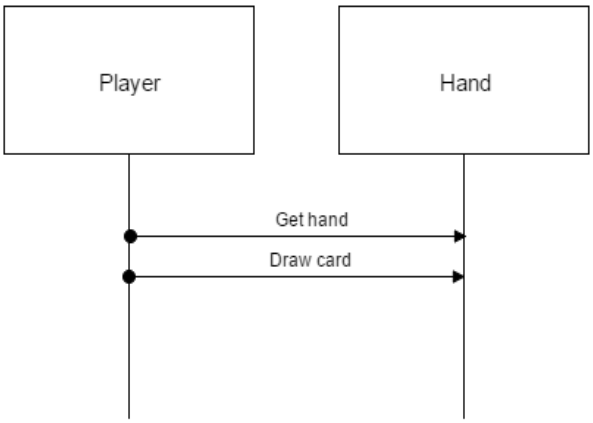


Figure 6.

Figure 6 shows the data members and main functions for the player class. Unlike the other classes, Player gives direct access to one of its object data members, the hand, with the getHand() method. This breaks off a small layer of data abstraction, but reduces the amount of methods necessary in the Player class, all of which would simply be redirected calls to the hand class anyway.

Interaction



The player is primarily a storage object for the hand and points for each player, and does not interact much with other objects, as can be seen in Figure 7. It will facilitate drawing cards into the hand, but otherwise simply gives access to the hand to the game state controller.

Figure 7.

Card Pile Information

Usage

The card pile class is a parent class to the 3 types of card collections in the game: the draw pile/deck, the 8 foundation piles, and the player hands. The card pile class, and each of its children, is responsible for holding and accessing the information pertaining to them as well as functionality to manipulate the piles. The hand and foundation classes also are responsible for determining valid card additions/removals based on their class characteristics. Outside of printing a card's information, only the card pile classes will interact directly with cards, which provides an additional layer of data abstraction.

Model

As seen in Figure 8, the Hand, Deck, and Foundation classes all inherit from the card pile class and share methods to add cards, print cards, and check the pile size. However, they each also have specialized methods as well, such as specific methods to remove cards and access cards from each type of pile. The nKings data member in the Hand class helps with scoring and determining lay down validity.

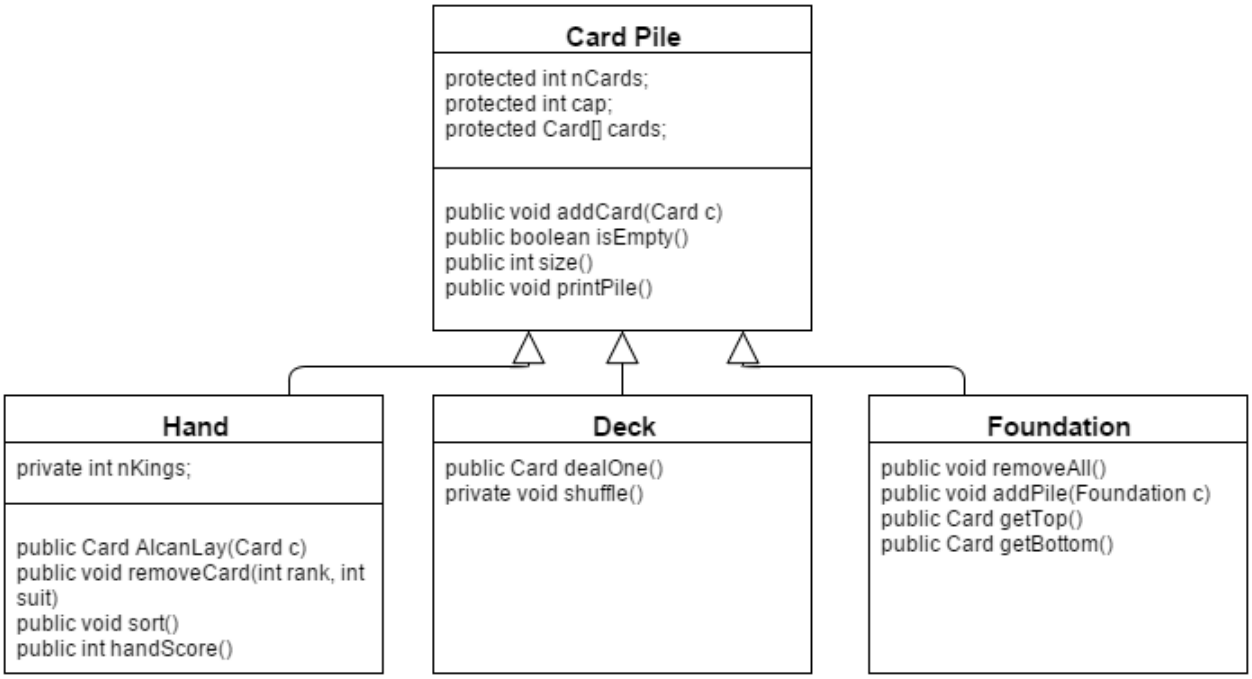


Figure 8.

Interaction

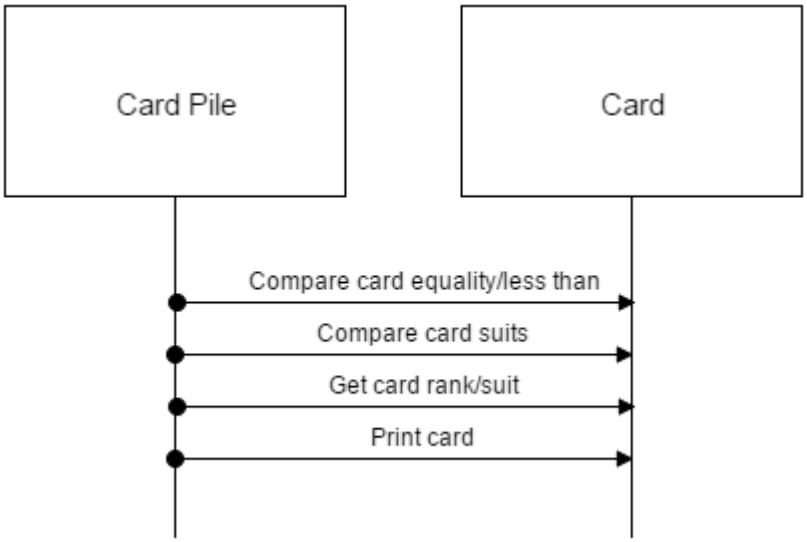


Figure 9.

The card pile classes hand all direct interaction with the cards. They will call all necessary methods to compare cards and check validity of any attempted card additions.

Card Information

Usage

The card object stores the rank and suit information of the card and is responsible for providing comparison functions between cards. Thus, all the details regarding checking whether a move is valid comes from a comparison method within the card class, which helps abstract these lower level details from the higher level game mechanics, and encapsulates card placement-related error checking within the class.

Model

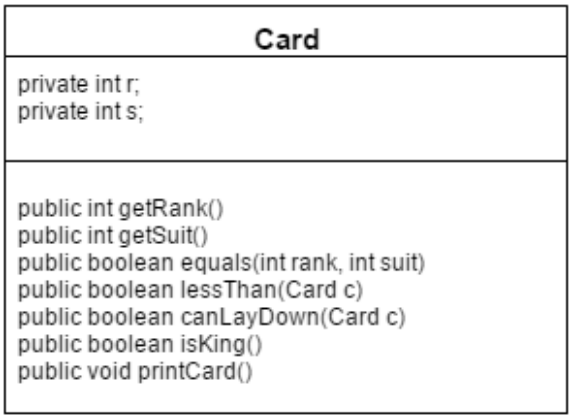


Figure 10.

The card class contains the necessary rank and suit information to distinguish the 52 cards, which are stored as ints to ease comparison. In addition to getters, the card class has multiple comparison functions to aid in both sorting and checking valid moves. It will also print cards, converting the rank and suit to characters as needed.

Section 4 – Benefits, Assumptions, Risks/Issues

Benefits:

- Data encapsulation – all data members for each object used in the game are private
- Data abstraction – there are several layers of abstraction in the game mechanics and the higher level objects rely on calls to lower level objects to manage the details
- Inheritance – the program uses inheritance with the card pile class to simplify some of the code
- Well defined roles – each class has a clearly defined role that is easy to understand from the name
- Clear separation between game mechanics and UI – this makes it easier to work on each one separately, which could be useful for a GUI for example

Risks:

- Methods specific to game – many of the methods are tailored specifically to this version of the game, and thus classes like the card class or card pile class do not necessarily have functionality built in to work for a general card game
- Lacks some sanity checking functionality – for example, there is no check that a card removed from one pile is added to another pile or vice versa. The logic of the game makes it so the add/remove code for cards will be nearby so it is relatively easy to check, but with future additions, things may get messier and a sanity checker would be useful.

Assumptions:

- We are following the ruleset at http://www.pagat.com/domino/kingscorners.html