Software Design Specification

Hanabi Client G3

Authored by:
Joel Siemens <jms676>
Olivier Lacoste-Bouchet <oll096>
Troy Belsher <twb431>
Kole Barber <klb731>
Destin Astrope <dja392>

Architecture

Overview

The program will be made with using several different architectures, resulting in a Hybrid architecture. The first is a Simulation architecture based on Model View Controller (MVC) approach. The controller and the interaction model will have an event driven architecture. The AI will have a pipes and filters architecture.

Model View Controller Architecture

A model view controller architecture was chosen for how well it can be adapted to the Hanabi program. We are using model view controller to separate visual code and game state code from AI code and controller code working behind the scenes. The view will display the info stored in the model to the user. The model will contain the information about the game, including what cards are in each hand and what hints have been given about those cards, as well as the remaining time tokens and fuse tokens, the current firework stacks, the remaining turn time, and the discard pile. The view will use this information to build the UI, as well as respond to the player when they are hovering over cards during their turn to decide what to do. Since we are using two views (the game view and the discard view) that use the same data, we believe that model view controller was the right choice for our main architecture.

Event System: Controller and Interaction Model

While in game, the controller can have two states: Active and Inactive. These states are triggered by server events. The messages from the server will trigger operations on the controller and model, based on the message. The messages from the server can trigger a state from Active to Inactive or from Inactive to Active, where active is a state where the user can play.
The interaction model has several states that are changed by the user's input. The states are based on the controller's active and inactive states. They include several Active state substates: Pay, Discard, Inform Colour and Inform Number. These states will change how the mouse interacts with the on screen card objects.

Pipes and Filters: AI Controller

The AI will be have data flowing through pipes and filters. These will be computed in the AI controller. The filters will filter out moves based on a weighing system, which will then choose the best option based on the weighing system.

Other Architectures

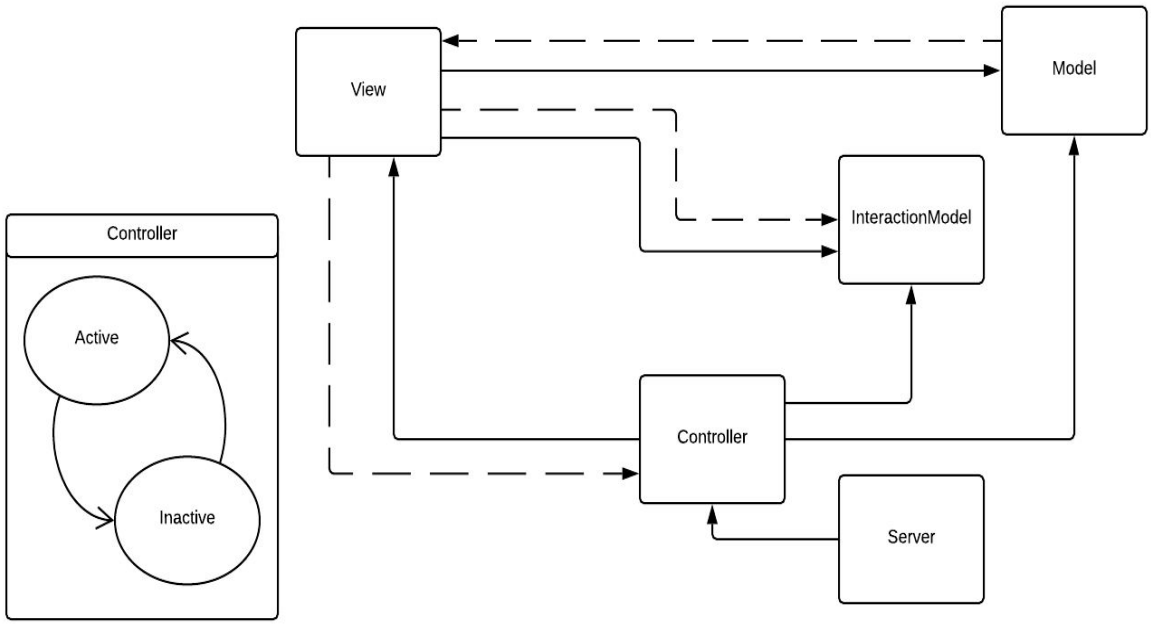
There are several other architectures that we considered, although we did not use these. A dataflow architecture doesn't make a lot of sense in this situation because each thing our program does doesn't require several methods to be chained together using each others' outputs. It might be possible to use such an architecture for our AI, with various options for moves feeding into one deciding function that tries to choose the best from among them.

A call and return architecture isn't really necessary for this program because we don't need to wait for the results of an action to know what that action has done to the game state (unless it is our own turn, since we don't know the cards in our own hand), and the server does not need input from our client to progress the game. The server might operate under a style of call and return, since the game cannot progress until the person whose turn it is responds, but that isn't within the purview of our project.

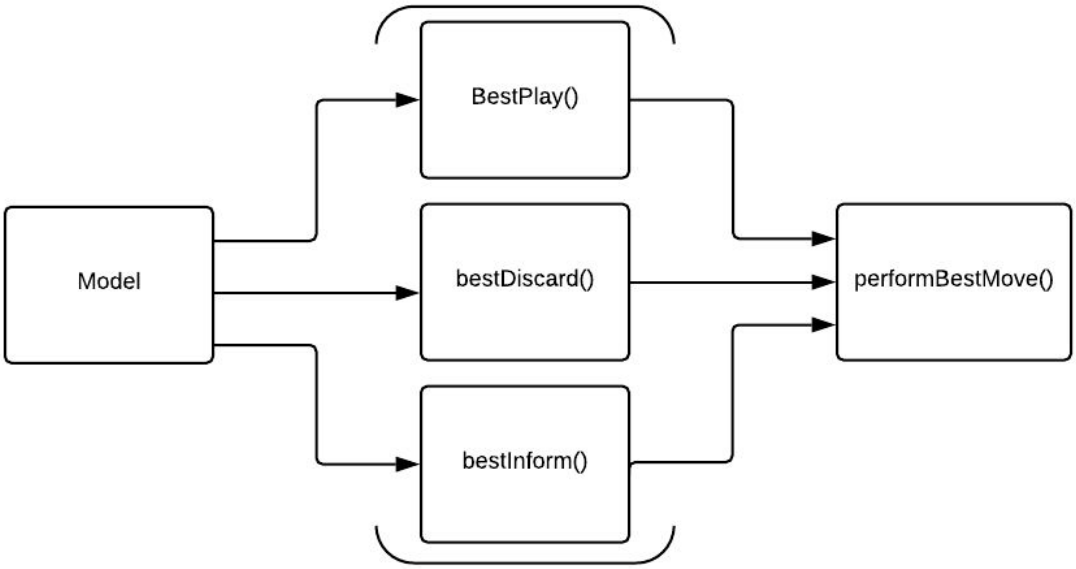
A data-centered system might make sense if our game was operating as both server and client, since we could tell each client to trigger its turn when a turn marker got to the client's player number for that particular game, and it would be easy to go looking for what information was needed. It would also, however, necessitate some level of honesty since a player's own hand would also be on the blackboard, and they would just have to be trusted not to look at it. Because the server is set up at arm's length for this project, and would not look at the blackboard, there is no compelling reason to use this architecture.

Boxes & Lines MVC

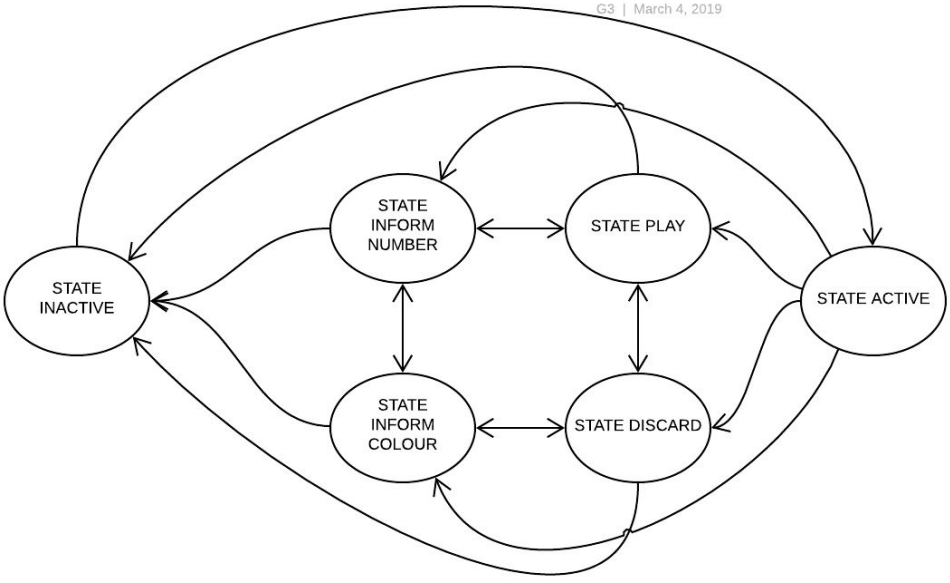
G3 | March 4, 2019



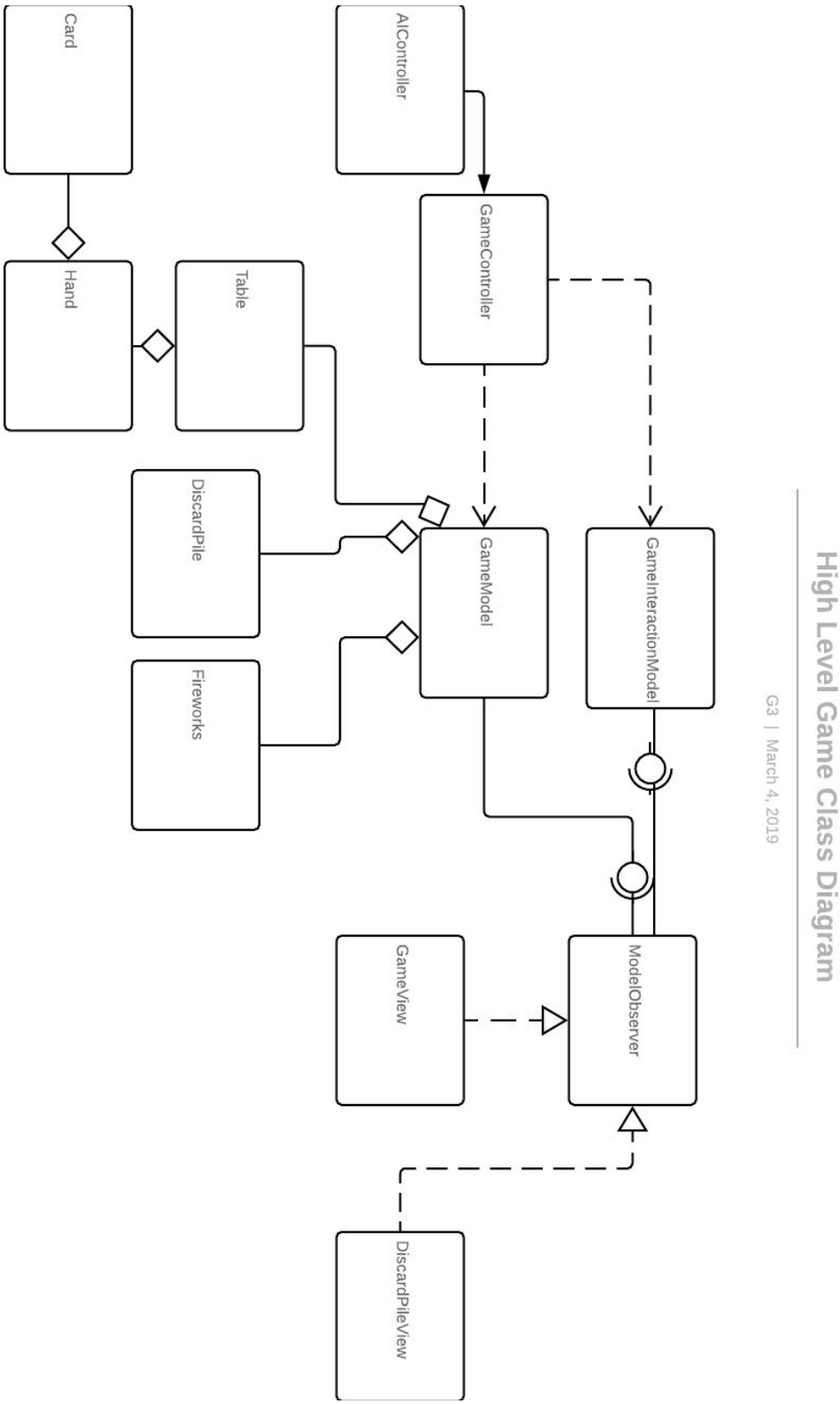
Pipes & Filters: AI Controller

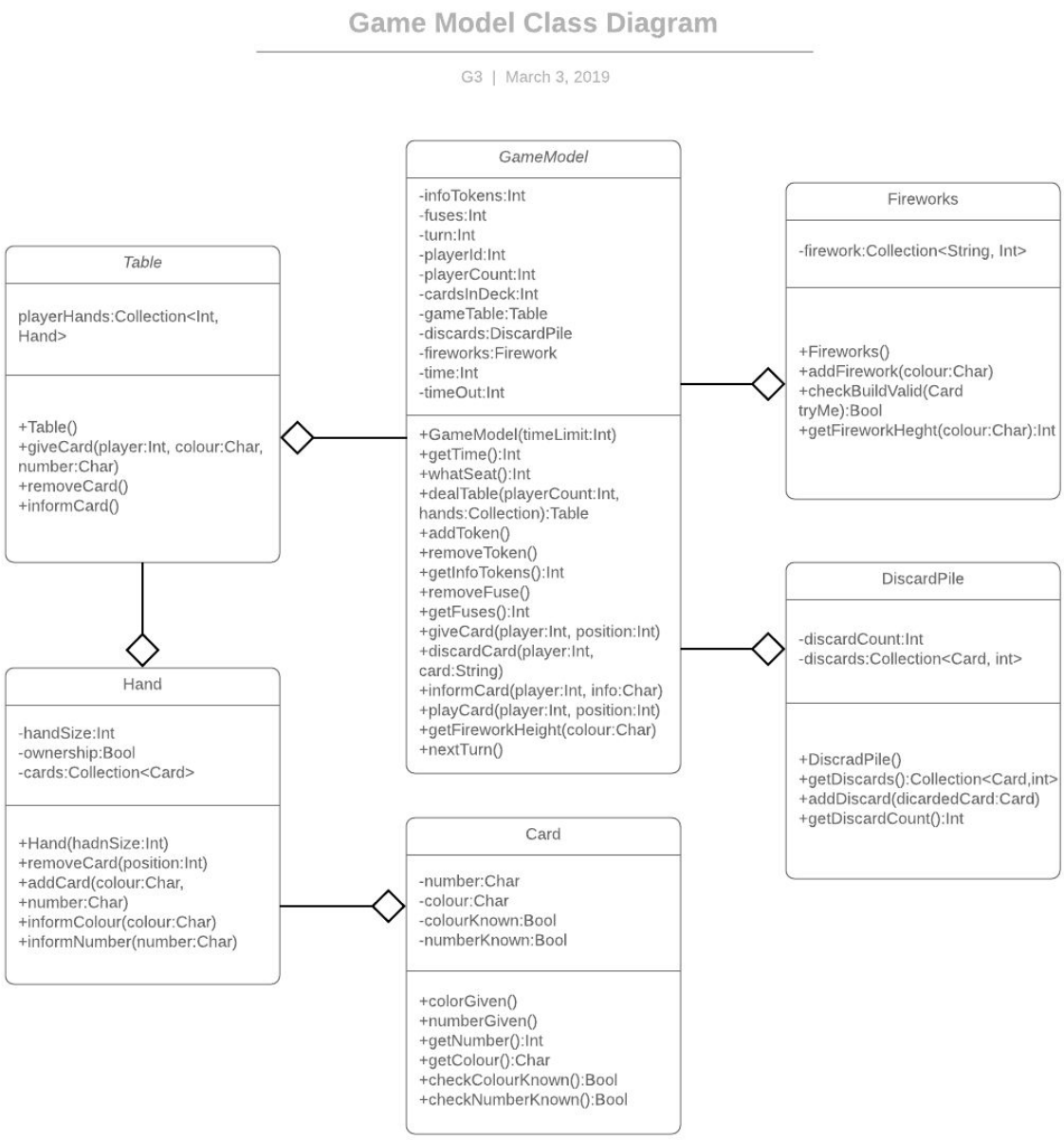


State Transition



Class Diagram





Classes and Functions

public class Card

private char number;

// number rank of the card

private char colour;

// colour suit of the card

```
private Boolean colourKnown;
        // marker for whether the colour of the card is known
private Boolean numberKnown;
        // marker for whether the rank of the card is known

public Card(char colour, char number);
        // object constructor, takes color and number value and sets both
      colourKnown and numberKnown to false.

public void numberGiven();
       // sets numberKnown to true.

public void colourGiven();
       // sets colourKnown to true.

public int getNumber();
       // returns the ranks of the card.

public char getColour();
       // returns the suit of the card.

public Boolean checkColourKnown();
       // returns whether the colour of the card has been given as information.

public Boolean checkNumberKnown();
       // returns whether the colour of the card has been given as information.
```

---

public class DiscardPile

```
private int discardCount;
        // tracks the number of cards in the discard pile

private Collection<Card,int> discards;
        // A collection to store the discarded cards and how many of each card has been
        discarded

public DiscardPile();
        // Constructor method instantiates the collection and sets the discardcount to 0

public Collection<Card,int> getDiscards();
        // returns the collection of discarded cards
```

```
  public void addDiscard(Card discardedCard);
        // adds a newly discarded card to the collection

  public int getDiscardCount();
        // returns the amount of cards in the discard pile
```

---

public class DiscardPileView

```
  public DiscardPileView();
        // constructor for view to display discard pile

  public void setModel();
        // sets a model for the view to observe

  public void drawCard();
        // code to draw a discarded card

  public void drawDiscardPile();
        // method to draw all discarded cards
```

---

public class Fireworks

```
  public Collection Fireworks;
        //a container for individual fireworks
  public Fireworks();
        //initializes all fireworks with height set to 0 for each

  public void addFirework(char colour);
        //increments a fierwork by 1 on its height

  public Boolean checkBuildValid(Card tryme);
        //checks to see if a play is valid and a firework can be successfully increased

  public Collection getFireworks();
        //returns the collection of fireworks
```

---

public class GameController

```
  private GameModel model;
        // link to game model
```

```
    private GameView view;
            // link to game view
    private GameInteractionModel iModel;
            // link to interaction model
    private int state;
           // current game state, whether or not it is the user's turn
    private final int STATE_INACTIVE;
           // not the user's turn
    private final int STATE_ACTIVE;
           // the user's turn

    public GameController();
       // constructor.  will set up the model, view, and interaction model.

    public void handleMessage(String jsonMessage);
       // handles a message from the server, adjusting the model as needed.

    public Collection parseJSON(String jsonMessage);
       // parses a message from the server into a collection of usable information.

    public void sendJSON(Collection move);
       // converts a collection of information into a json message and sends that message to the
server

    public void setModel(GameModel model);
       // sets up the game model

    public void setiModel(GameInteractionModel iModel);
       // sets up the interaction model

    public void setView(GameView newView);
       // sets up the game view

    public void endGame();
       // ends the game, and creates a new window with endgame information
```

---

public class AIController extends GameController

```
    private Boolean canInform;
           // true if there are >0 info tokens
    private Boolean canDiscard;
           // true if there are <8 info tokens
```

```
    public AIController();
        //constructor, will initialize canInform, canDiscard

    public void checkAvailableMoves();
        //updates the booleans

    public Collection<String> bestPlay();
        //finds the best play card move, returns info about the play

    public Collection<String> bestInform();
        //finds the best Inform move

    public Collection<String> bestDiscard();
        //finds the best Inform move

    public performBestMove();
         //finds the best move between info, discard, and play card.
```

---

public class GameInteractionModel

```
    private Collection observers;
        // collection of observers to be informed of game state changes
    private Card selected;
        // marker for which card is hovered over by the user
    private int state;
        // marker for state variable for the view
// state information that will affect what UI components are active at any given time.

    private final int STATE_INACTIVE;
        // not the user's turn.  only the discard pile can be used.
    private final int STATE_ACTIVE;
        // user's turn base state.
    private final int STATE_PLAY;
        // user's turn, play action selected.
    private final int STATE_DISCARD;
        // user's turn, discard action selected.
    private final int STATE_INFORM_COLOUR;
        // user's turn, inform colour action selected.
    private final int STATE_INFORM_NUMBER;
        // user's turn, inform number action selected.
```

```
public GameInteractionModel();

public void setSelected(Card selectedCard);
    // sets a card as being hovered over for the view's use.

public void addObserver(Observer anObserver);
    // add an observer to the list of obvservers

public void removeObserver(Observer anObserver);
    // remove an obvserver from the list of observers

public void notifyObserver(Observer anObserver);
    // notify all observers in the observer list of a change.
```

---

public class GameModel

```
private int infoTokens;
        // must be an integer between 0 and 8, these tokens are a resource used to give other players hints.
private int fuses;
        // initialized to 3, the team loses one for each mistake.  once fuses reaches 0, the game ends.
private int turn;
        // marks the player number of the player whose turn it currently is.
private int userID;
        // this client's player position at the table
private int playerCount;
        // number of players in the current game
private int cardsInDeck;
        // counts the remaining cards in the deck.  If the deck runs out, each player gets one more turn before the game ends.
private Table gameTable;
        // holds the table state, containing hands, cards, and revealed information for each player.
private DiscardPile discards;
        // keeps track of which cards have been discarded over the course of the game.
private Fireworks fireworks;
        // maintains the firework stacks.
private int time;
        // turn timer
private int timeLimit;
        // turn time limit time for this game
```

```
public GameModel(int timeout);
    // constructor.  Will initialize timeLimit, infoTokens, fuses, and cardsInDeck.

public void dealTable(int playercount, Collection startinghands);
    // initializes the table for the game at the start.  playercount will
    indicate the number of players for this game, and startinghands
    will bring in the parsed collection of other players' hands.

public void addToken();
    // increases the available information tokens by one.

public void removeToken();
    // reduces the available information tokens by one.

public int getInfoTokens();
    // returns the current remaining number of information tokens.

public void removeFuse();
    // reduces the number of fuses by one.  If fuse count reaches 0, triggers game end.

public int getFuses();
    // returns the current number of fuses remaining.

public void giveCard(String newcard);
    // places newcard into the hand of the current player.

public void discardCard(int player, int handPosition);
    // discards the card in the given posiiton in the current player's hand.

public void informCard(int player, char info);
    // marks the chosen card in the chosen player's hand with the the information

public void playCard(int player, int handPosition);
    // attempts to play the card in the chosen position to the fireworks stack.

public int getFireworkHeight(char color);
    // returns the current height of the firework stack of the given color.

public void nextTurn();
    // incrememnts the current player counter.  must only be called after all results from current
turn are completed.
```

```
    public int currentTurn();
        // returns the seat number of the player whose turn it is currently.

    public int getTime();
        // returns the remaining time for the current turn.

    public int playerSeat();
        // returns the userID seat number
```

---

public class GameView

```
    private GameModel model;
            // link to model
    private GameInteractionModel iModel;
            // link to interaction model

    public GameView();
        // constructor, will draw all components

    public void setModel(GameModel aModel);
        // sets the link to the game model

    public void setInteractionModel(GameInteractionModel aModel);
        // sets the link to the interaction model

    public void drawTable();
        // draws the game table

    public void drawHand();
        // draws the players' hands

    public void drawCard();
        // draws the players' cards

    public void drawInfoTokens();
        // draws the remaining information tokens

    public void drawFireworks();
        // draws the current status of the fireworks stacks

    public void drawFuses();
        // draws the remaining fuses
```

```
public void drawStatus();
    // draws status bar with timer and cards left in deck
```

---

public class Hand

```
private int handSize;
        // the amount of cards in the player hands
private Collection<Card> cards;
        //a collecton that will store the current cards in the hand

public Hand(int handSize);
        // constructor.  Will initialize collection cards, handSize.

public void addCard(char colour, char number);
        // adds a card with colour and number to collection cards

public void removeCard(int position);
        // remove the card at position in collection cards

public void informColour(char colour);
        // will set the colourKnown to true to each card object in collection cards that belongs to
input colour

public void informNumber(char number);
        // will set the numberKnown to true to each card object in collection cards that belongs to
input number
```

---

public class Table

```
private Collection<int,Hand> playerHands;
        // collection of player hands, indexed by player seat numbers

public Table();
    // constructs player hands by giving each player their cards.  constructs own player's hand
by giving them blank cards.

public void giveCard(int playerID, char colour, int number);
    // adds a card with the given rank and suit to the chosen player's hand.
```

```
   public void removeCard(int playerID, int cardPosition);
      // removes the card in the chosen player's hand from the chosen hand position.

   public void informCard(int playerID, String type, char info);
      // informs the chosen player about cards in their hand of the chosen type.
```

---

public class EndGameView

```
   private void display();
         //gathers game end state information and congratulatory message to display to all
players.

   private void closeGame();
         //takes player back to the main menu and closes connection to the game and server.
```

---

public class GameMenu

```
   private Button createGameButton;
   private Button joinGameButton;
   private Button howToPlayButton;
   private Button exitButton;
      // These buttons are used to call their respective methods in the class.

   private int game-id;
   private String token;
      // These fields are user entered data used for establishing a server connection.

   private Socket serverSocket;
   private Model aModel;
   private InteractionModel iModel;
   private View aView;
   private Controller aController;
      // These fields will be set by createGame() or joinGame(), these are the main instances of
our in-game program.

   public GameMenu();
      // Constructs the main game menu

   public void createGame();
      // Creates a connection with the server and creates a game as well as instantiates aModel,
iModel, aView, aController, game-id and token.
```

```
    public void joinGame();
        // Creates a connection with the server as well as instantiates aModel, iModel, aView and
aController.

    public void howToPlay();
        // Presents a window with the game rules.

    public void exit();
        // Closes the entire program.
```
