**Group 4**
Kristen Barrett  Josh Do
Matthew Wolff  Patrick Mancuso
Ben Graham

# Love Letter

**CSC 4444  Section 1**

## OVERVIEW

Our project is an implementation of the card game *Love Letter*, utilizing a maximum of four players (three bots and one human player). *Love Letter* is a simple, short game, where the goal is to either knock out each of the other players through various card effects or to end the game with the highest valued card.

## THE PROBLEM SPACE

### The Motivation

We wanted to discover if there was a way to optimize winning the game of Love Letter, as the game is partially based on chance as well as some aspect of probability and prediction of opponents.

### The Problem

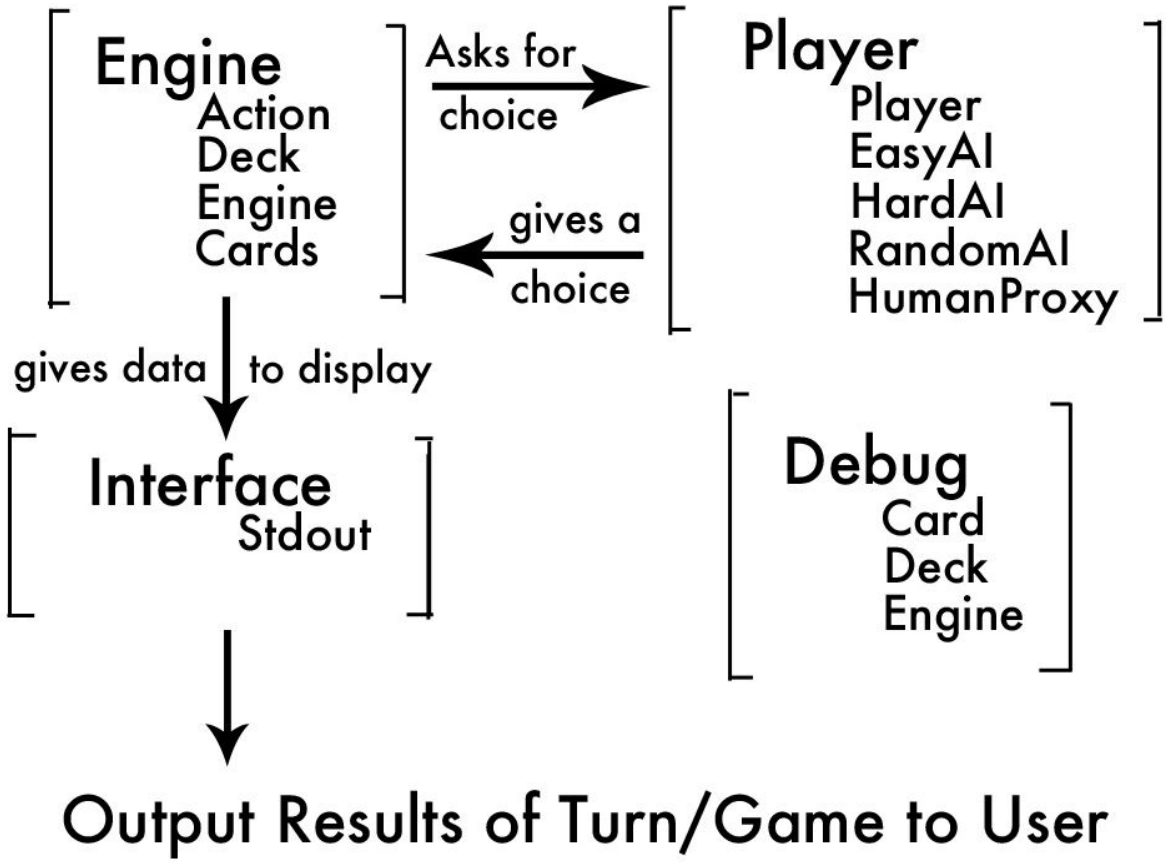
How could one win a game of *Love Letter* every single time? Despite only having eight card ranks, there are numerous combinations and strategies one could use to win; combined with the aspect of removing an unknown card from play, as well as certain player techniques, there was a quite large yet discrete solution set for ways to win the game. This became our main goal for our experimentation as well as our AI design: how could we win the game as many times as possible?

Another problem we discovered was defining our bots, and what knowledge would make one bot much harder to play against than the others. Like human player variations, some bots needed to remember near everything played (the Graveyard) as well as players' hands; other bots needed to figuratively give the game zero attention and simply choose cards that *seemed* best.

## TECHNICAL SPECIFICATIONS

### Program Structure & Design

Our program is split into four portions: *debug*, *engine*, *interface*, and *player*. Our *engine* controls the entire game process, from drawing, executing, and discarding cards to maintaining deck content, including individual cards and their properties. The *player* portion maintains the artificial intelligence and its level of learning, as well as human player actions. The *interface* part contains various ways for the human player to interact with the game, and the *debug* portion is our various sample code used for testing implementations. The overall structure is laid out as follows:



On launching a game, the engine goes through a cycle of actions to determine whether or not to continue the game and draw cards, or end it altogether.


As illustrated in *main.py*, an engine is generated, players are created and added to the engine, and then the game is ran with a winner selected.

On a more detailed, technical level, one could summarize a game process as such:

1. A list of players is created in order to maintain players that have lost and will eventually be removed from the round. Each player receives a card from a shuffled deck (an array of cards), and one card is randomly removed, as stated in the original *Love Letter* rules to make counting cards much more difficult.
2. On a player's turn, the player is dealt an additional card. At this point, the player can choose which of the two to play, as well as the target of the card's effects (i.e., if a Priest is drawn, whose hand to reveal).
3. Said card's perform function is called, and its effects occur in the game. The engine then discards the used card into the Graveyard, along with some additional information like who played the card.
4. Repeat 2-4 until either (a) the deck runs out of cards or (b) there is only one player remaining. The game then ends.
    a. If (a), the final players compare the value parameter of their cards; the highest value wins the round.

## Artificial Intelligence Techniques

We customized our own tree pruning algorithm from a state space of possible player card ranges. Each turn a player must choose one of the two cards in their hand to discard, so the bot need only search the heuristics of each card and compare them. The heuristic returned a value based on the outcome of a card and the target of its effect. We pruned the amount of nodes the heuristic calculations method had to visit by viewing the players' card ranges and choosing only the player that would be the best target for the card before calculating exactly what the outcome of that targeting may be. This pruning allowed for very intelligent card probabilities for actions like playing the Guard, where you must guess the target's current card. We also allowed our AI agents to maintain a memory of the state space that allowed it to see all of the actions previously performed by players and perform its heuristic calculations based on those actions. Playstyles allowed us to approach this heuristic calculation from an alternate angle, interpreting player actions to have subtly different meanings.

## Implementation & Special Cases

The implementation uses three levels of AI: random, easy, and hard. The random and easy AI's are extremely simple, so the focus of discussion will be on the hard AI.

In general, for our implementation of the hard AI, we used a range calculation to determine what possible cards each other player has, which gets recalculated after every player's turn. With each unique card, we had to determine certain edge cases so that the AI had the best possible chance of winning within the game environment. The AI had to be aware of card effects (ie: Dropping princess means you lose, having a Countess in combination with a Prince or King means you have to drop the Countess). The AI also had to decisively choose between using one card or another based upon the current game environment. Knowing when to play a high card over a low card, a risky prospect is an important characteristic of the AI. There were a couple cases where we had to specifically code for an edge case, the Baron card is a rather interesting card because when played, you and another player compare hands, whichever of you have the higher card stays in the game and the other player is out. In this case we had to make an edge case where the AI decides that if it has this card and another high card, it'll choose the baron. The Baron card is very interesting because it is the one card in the game that gives you a chance to knock out an opponent, depending on if you believe in your odds of having a higher card. While none of the cards were as complicated as the Baron, each card has its own set of rules for when a card should be played, beyond the standard logic which

## AI Types

### RANDOM AI

Always chooses a random card as well as chooses a random target for the card if it has a targeted ability. In our tests, due to the factor of blind luck, the random AI can win in surprising ways. However with extreme highs also come extreme lows. The random AI can also discard high cards with no regard for their value, including the highest card in the game the Princess, which if discarded, automatically makes you lose the game.

### EASY AI

Simple reflex agent, always playing card with lowest value--tries to maximize its chance of winning by holding the highest rank card in its hand. Has no memory of past rounds or other cards played. The easy AI will randomly pick players for targeted cards and does not look into the graveyard to see what cards have already been played. Due to the conservative nature of the easy AI bot, it tended to lose games mostly because it never took huge risks, didn't properly use the Priest, and does not try to guess what players have in their hands.

### HARD AI

Full state space, keeps track of past rounds (using the environment), and any other publically available information to predict other players hands and optimize its choices, also has two playstyles: aggressive and defensive. Aggressive takes risks and tries to eliminate other players when given a reasonable chance at succeeding. Defensive plays it safe and only plays high risk, high reward cards (baron and king) when the only other option is to lose. Other than cards like the Baron and King where you compare/switch hands the aggressive and defensive bots are the same. The Hard AI keeps track of the possible cards each player could have and makes choices based on its knowledge of other players. For example, if the hard AI knows player 2 has a countess and the hard AI has a guard it will play the guard on player 2 and guess countess in order to eliminate that other player. This list of possible cards each player can have is pruned by the cards in the graveyard, cards in the bots hand and the actions of other players. (if one player plays a baron and loses the bot knows the winner of that must have a card with value greater than the card discarded)

## PROJECT COMPONENTS

### Software Tools, Packages, & Libraries

The project is implemented entirely in Python and has no external dependencies besides the Python standard library. Testing was done against Python version 3.5.2, though the project should be compatible with Python versions 2.7.x.

The developers used PyDev (an Eclipse plugin) for development and PyCharm (a variant of IntelliJ's IDEA platform). Git and GitHub were used to collaborate and organize workload.

https://github.com/matthewjwolff/LoveLetter

## Functionality Demo

```
You have been dealt a Guard
What will you play?
1. Handmaid
2. Guard
>2
On whom  will you play that?
0. Jerry Lewis
1. RandomAI0
>1
What card do you guess?
0. Guard
1. Baron
2. Priest
3. Handmaid
4. Prince
5. King
6. Countess
7. Princess
>7
Player RandomAI0 has played Countess on Jerry Lewis
You have been dealt a King
What will you play?
1. Handmaid
2. King
>2
On whom  will you play that?
0. Jerry Lewis
1. RandomAI0
>1
Player RandomAI0 has played Handmaid on RandomAI0
You have been dealt a Princess
What will you play?
1. Guard
2. Princess
>1
On whom  will you play that?
0. Jerry Lewis
1. RandomAI0
>1
What card do you guess?
0. Guard
1. Baron
2. Priest
3. Handmaid
4. Prince
5. King
6. Countess
7. Princess
>2
Player RandomAI0 has played Baron on Jerry Lewis
The winner of the game is Jerry Lewis
```

# APPENDIX

## PLAYER

```
class Player(object):
    '''
    This is the base class for all Player classes.

    Types of players may include local players using a GUI, computer player
    using an AI, networked "player" proxy that receives moves from a nonlocal
    player, and networked "player" client that sends his moves to the proxy
    that is acting on his behalf.
    '''

    def getAction(self, dealtcard, deckSize, gravestate, players):
        '''
        Callback from engine to get a player's choice
        '''
        raise NotImplementedError

    def assignHand(self, card):
        '''
        The engine has given the player his starting hand
        '''
        self.hand = card

    def notifyOfAction(self, action):
        '''
        On another player's move, this method is called for all other players
        to serve as notification that a move occurred.
        '''
        raise NotImplementedError

    def priestKnowledge(self, player, card):
        '''
        When the Priest is played, add knowledge of a target player's
        hand to self. Specifically for use with HardAI and bots with state history.
        '''
        raise NotImplementedError
```

## ENGINE

```
class GameEngine(object):
    '''
    The engine registers players, instantiates the game, runs the gameplay
    loop, and executes actions that transition the game from state to state.
    '''

    def __init__(self):
        '''
        Constructor
        '''
        self.origplayers = []
        self.deck = Deck()
        self.running = False
        self.grave = []
        self.discarded = None

    def addPlayer(self, player):
        self.origplayers.append(player)

    def runGame(self):
        # make a NEW list
        self.players = list(self.origplayers)
        assert len(self.players) >= 2
        for player in self.players:
            player.assignHand(self.deck.getCard())
        # discard one
        self.discarded = self.deck.getCard()
        self.running = True
        while self.running == True :
            for player in self.players :
                player.handmaidenFlag = False
                card = self.deck.getCard()
                # I changed my mind, no deep copying

                # Note, API change. Player is notified of the card dealt,
                # remaining cards in the deck, the graveyard, and the
                # list of players
                action = player.getAction(card, len(self.deck.shuffled),
                                          self.grave, self.players)
                # update the player's hand
                if action.playedCard == player.hand:
                    # If the player chose to play the card he had kept,
                    # then replace the card in his hand with the card from
                    # the deck
                    player.hand = card
                action.playedCard.perform(action, self.players, self.grave, self.deck)
                # Tell other players that a play occurred
                for oplayer in self.players:
                    if oplayer != player:
```

```
                    oplayer.notifyOfAction(action)
            self.grave += [action]
            # End the game if nobody remains or the deck is empty
            if len(self.players) == 1 or self.deck.size()==0:
                # Yes I could make this into a proper while loop
                self.running = False
        winner = self.players[0]
        # TODO: handle ties?
        for player in self.players:
            if player.hand.value > winner.hand.value:
                winner = player
        return winner
```

## INTERFACE

```
class StdoutInterface(object):
    '''
   An implementation of CLI user input using print statements
    '''

    def priestCallback(self, player, card):
        print("Player "+str(player)+" has a "+card.__class__.__name__)

    def notifyCallback(self, action):
        print("Player "+str(action.doer)+" has played
"+action.playedCard.__class__.__name__+" on "+str(action.target))

    def actionCallback(self, dealtcard, deckSize, gravestate, players):
        print("You have been dealt a "+dealtcard.__class__.__name__)
        chosen = False
        cardChoice = 0
        chosenCard = None
        playerChoice = 0
        guessChoice = 0
        while(not chosen):
            print("What will you play?")
            print("1. "+self.proxy.hand.__class__.__name__)
            print("2. "+dealtcard.__class__.__name__)
            cardChoice = int(input(">"))
            if cardChoice > 2 or cardChoice < 1:
                print("Bad choice")
            else:
                chosen = True
                chosenCard = self.proxy.hand if cardChoice == 1 else dealtcard

        chosen = False
        while(not chosen):
            print("On whom  will you play that? ")
            for i in range(len(players)):
                print(str(i)+". "+str(players[i]))
            playerChoice = int(input(">"))
```

```
            if playerChoice < 0 or playerChoice > len(players):
                print("Bad choice")
            else:
                chosen = True
        if type(chosenCard) == Guard:
            # they chose a guard, better see what they want to guess
            chosen = False
            while(not chosen):
                print("What card do you guess?")
                for i in range(len(cardTypes)):
                    print(str(i)+". "+cardTypes[i].__name__)
                guessChoice = int(input(">"))
                if guessChoice < 0 or guessChoice > len(cardTypes):
                    print("Bad choice")
                else:
                    chosen = True
        return Action(self.proxy, chosenCard, players[playerChoice], cardTypes[guessChoice])

    def __init__(self, name):
        self.proxy = HumanProxy(self.actionCallback, self.notifyCallback,
self.priestCallback, name)
```