CPSC 2720 2021

[D.U.N.C]

DEVELOPERS (without) UNDERGRADS IN CS

Hutash

Cadence Meszaros
David Awosoga
Noel Roemmele
Ugonna Osuji

March 20, 2021

TABLE OF CONTENTS

INTRODUCTION

Welcome to the D.U.N.C. software developed by Developers (without) Undergrads in CS. Created by 2 Math majors, an Applied Stats major, and an MRU transfer student, this software will implement three card games - “Crazy 8’s”, “Go Fish”, and “Jungle Speed”.

Crazy 8’s

The objective of this game is to have the highest number of points after the completion of the rounds. The number of rounds is determined by the player. At the end of each round the players points will be shown, with the total points summed and displayed after the game concludes. The player with the highest score at the end of the game will be declared the winner.

Pre Game:

- Played with a standard deck of 52 cards.
- 5 cards are dealt per player.
- The remaining cards are put into a stockpile
- The top card from the stockpile will be revealed and put into a separate pile, called the play pile.

Round Structure:

- The player and the computer will alternate placing a card onto the play pile. In order to play the player must have a card in their hand that matches either the face/number or suit of the top card in the play pile.
    - The player may play an eight instead. This acts as a wild card, where the player can change the suit to whatever they desire.
- If a player is unable to play (no matching suit, number/face, or eight), then they must draw from the stockpile.
    - The player will continue to pick up from the stockpile until they are able to play.
- A player may choose to pick up a card from the stockpile as strategy, even if they do have a playable card in their hand, but they must place a card on the play pile at the end of their turn.
    - The amount of cards they may pick up is unlimited (until the deck is empty).
- A round ends when a player runs out of cards. At this time the scoring procedure is put in place.
- After a winner has been declared then the cards are reshuffled and redealt.

Scoring:

- The winner received the sum of the points from the loser’s hand, delegated as such:
    - Each eight = 50 points.
    - Each K, Q, J, or 10 = 10 points.
    - Each ace = 1 point.
    - Every other card is its face value.

Go Fish

This is a solo game, where the objective is to get the most “books”. A book is 4 of a kind, based on face/number value. The number of rounds is determined by the player. At the end of each round the winner will be declared, along with the number of books they collected. After the winner has been declared a prompt will appear asking the player if they would like to continue the game (play another round) or exit. If they choose to continue, another round will be played and their scores from the previous game will be carried over. Therefore, the books can accumulate over the rounds.

Pre Game:

- Played with a standard deck of 52 cards.
- 7 cards are dealt per player.
- The remaining cards are placed into a stockpile, face down.

Round Structure:

- A player will ask their opponent for the rank of a card that they want.
    - Requests can only be made if the player has said rank of card in their hand.
- If the player from which the card has been requested has the requested card, they must give up the card to their opponent. If the player has multiple of that card, they must give up all of them to the player who made the request.
- If the player from which the card has been requested does not have the card, they must say “Go Fish”, at which point the player that made the request will draw a card from the stockpile and place it in their hand.
    - Their turn then ends.
- If an affirmative request was made, then the player makes another card request. They may request the same rank as the previous request or a different rank.
    - They may continue to make requests until they are told to “Go Fish”, then their turn will end
- A book will be automatically counted once there are four of a kind in a player's hand.
    - A message will be sent to confirm that a book has been created, and what the rank of the book is.
- If a player runs out of cards, they must draw from the stockpile and make a request based on the card that they drew.
- The game ends when all 13 books have been collected. The winner is the player with the most books.

Jungle Speed

The objective is to get rid of all the cards in your hand. The number of rounds is determined by the player. At the end of each round the player who gets rid of their hand first is declared the winner, and the user will get a prompt asking if they would like to play another round or exit the game.

Pre Game:

- Played with a special deck of 72 or 80 cards, depending on the number of players, which consists of different colours and patterns.
- The entire deck is divided evenly between the players, face down, and the players cannot look at their cards.
- The totem is placed in between the players, in arms reach.

Round Structure:

- Players will alternate flipping the top card from their hand and placing it into their personal discard pile. This process continues until two flipped cards match in shape.
- Once two flipped cards match in shape, there will be a duel between the players.
    - The first player to press the key preselected by the player (analogous to grabbing the totem) will be declared the winner of the duel.
    - The loser of that duel will take the cards from the discard pile and place them at the bottom of their hand.
    - The player who lost the duel will start the card flipping process again
- The game ends when a player no longer has any cards in their hand.
    - This player will be declared the winner.

Special Rules:

- If a player accidentally pressed the wrong key during a duel they automatically lose the duel, taking the cards from their own discard pile and their opponents.

Other Notes:

Time permitting, the D.U.N.C. software will implement the following features:

- Different skill levels of the AI whom they are playing against.
    - For Crazy 8's and Go Fish different common strategies will be added to the levels of AI,
    - variable reflex speeds for the AI will be enabled in Jungle Speed.
- Dialogue between the player and AI may also be implemented.
- Ascii Output of cards
- The ability to play against more than two players
- For Jungle Speed, the following special cards will be reintroduced:
    - If an "All In" card is played, both players must press the totem key as quickly as possible, and the same rules apply for the loser of the duel.
    - An "All Out Card" will have all players flip a card from their hand
    - A "Colours" card will change the matching criterion from shapes to colours

PROJECT MANAGEMENT

TEAM ORGANIZATION

Team Member	**Design- Draft**	**Design- Final**	**Implementation- Basic**	**Implementation- Final**
Cadence	Reporting Lead	Design Lead	QA Lead	Phase Lead
David	Phase Lead	Reporting Lead	Design Lead	QA Lead
Noel	QA Lead	Phase Lead	Reporting Lead	Design Lead
Ugonna	Design Lead	QA Lead	Phase Lead	Reporting lead

TEAM ROLES

Phase Lead:

- Understands what needs to be accomplished in the phase.
- Attend office hours to get help for the group, regarding clarification/questions etc.
- In charge of tasks identification and delegation with approval from group members.
- Maintains communication with team members to remain on track.
- Initiate dialogue on identification and resolution of problems.
- Ensure task deadlines are met.

Design Lead:

- Compilation and collection of group design ideas.
- Ensure that team members' software design contributions meet the 2720 guidelines.
- Create and constantly refer back to UML diagrams to ensure that they represent the intended software design.

Quality Assurance Lead:

- Implement rigorous testing procedures.
- Ensures that all code meets requirements before being merged into master branch.
- In charge of test driven development for a respective phase of implementation.
- Uses boundary value analysis and equivalence partitioning for test cases.

Reporting Lead:

- Ensures that the reports are completed and properly assigned to team members.
- In charge of collecting the team members contributions for the reports.
- Scribes team meetings and disseminates actions items during meetings.
- Document team report, in accordance with team members.

RISK MANAGEMENT

Requirements/Design/Estimation

- The team planned a project that is too large.
    - To avoid this issue we will prioritize the classes deemed “essential” over the classes that we declared as “features or extras”
    - We will focus on the essentials first and then the extras, time permitting.
- The team underestimated how long parts of the project would take.
    - In the case we run into a part that is larger than we expected we will make sure we are a flexible team. Meaning that the flexible team members are able to be assigned as an aid to that particular branch to ensure its completion.
    - In addition, we can re-evaluate each part to determine if it is essential or an extra.
- Major changes to design are needed during implementation.

    - In this case we will schedule extra meetings throughout the week, or make existing meetings longer to compensate for the major changes.
    - In addition, the team will figure out why we had to make that change. This is for the prevention of making the same mistake.
        - Learning from our mistakes.
- Card games are too simple.
    - If we get to a point where we have finished early, but the card games are too simple we can add extra features to add complexity to the games.
        - This will require extra meetings, or longer meeting times.

People

- Addition or loss of team members.
    - In the case of losing a team member, we would retain our roles and reassign the jobs of the lost member.
    - In the case of gaining a team member, we would prep them with an orientation of our project, along with our progress to date.
        - We would then reassign jobs to even the workload among team members.
- Unproductive team member(s).
    - If we run into the issue of an unproductive team member we would:
        - Assess the reason why the team member is not contributing as much as they are expected to and figure out how to fix the issue.
            - If the team member does not like the jobs they were assigned, or struggling with them, then we can reassign them to a better fitting team member.
- Team member(s) lacking expected technical background.
    - If there are team members with less experience we can assign them to easier jobs that don't require as much technical knowledge.
        - These jobs could be:
            - Testing
            - Brainstormer/idea person
        - This allows the team member to participate in a different way, and still help the team towards the final goal.
    - Another option would be to mentor them.
        - By showing the team member an example of how to do a particular task, they may be able to follow the example and finish the task they were assigned.
- Major life events.
    - We would expect the team member to give good notice for life events that are predictable. This way we can prepare for the change.
    - In the case of an unexpected life event, we expect the team member to be transparent and communicate to the team what they need.

        - We could lighten the load or reassign some jobs to ensure that the team member can work through their personal situations.
- Conflict between team members.
    - This conflict could be a conflict of ideas, or someone taking complete control of the project.
    - To prevent conflicts from happening the team needs to have good communication and discuss issues as they arise.
        - This means retaining an atmosphere of respect and openness.
    - If a conflict cannot be avoided, we will settle disputes as quickly, and efficiently as possible.
    - A solution for conflicts would be to make decisions based on (1) consensus, (2) majority, or lastly (3) leadership.
        - In the third case the phase leader will get the final say.

Learning & Tools

- Inexperience with new tools.
    - If the tool is recommended by Dr. Anivk or presented to us during the lab we will refer to official docs to try to understand how to use the tools.
        - After we read over the docs, if there are still questions the Phase Lead will attend office hours and ask questions on behalf of the team.
    - If a minority of the team are inexperienced with the tools they will be mentored by experienced team members.
    - In addition, we can create test programs to help us understand how the tools/programs work.
- Learning curve for tools steeper than expected
    - If the learning curve is steeper than expected the team will take time outside of class to practice how to use the tools, in order to keep on track with the project.
    - There are online resources that the team can also use to help understand the tools.
    - In addition, the team can work together to help each other understand the tools.
        - This will ensure that the entire team is on the same page.
- Tools don't work together in an integrated way.
    - If the tools we are using are those suggested by Dr.Anvik or Nicole, the Phase Lead will attend office hours and ask questions on how to integrate the tools better.
    - If the suggested tools are not compatible with the teams experience, we can extend to alternative tools that may integrate better.
- Lecture topics don't align with the project.
    - If there are requirements for a phase that havent been covered in lecture yet, the team can read ahead and learn the content on their own time.
    - The Phase Lead can attend office hours and ask questions regarding topics that are unknown to the team.
    - In addition, the team can use external resources to understand the topics.

DEVELOPMENT PROCESS

CODE REVIEW PROCESS

- For push requests:
    - Anyone can push but the changes need to be reviewed by the Quality Assurance lead.
- For merge requests:
    - They will be handled by the Design Lead.
- For triaging bugs:
    - They will be handled by the Reporting Lead.
- Phase Lead will do intermittent checks of code to ensure progress and that the code is meeting requirements.
- If there are major changes that need to be done, these changes will need to be brought up during a meeting and will be decided on by the team.

COMMUNICATION TOOLS

- Use iMessage group chat to discuss minor aspects of the project, and to arrange zoom meetings.
- Every week we have 4 scheduled zoom meetings. Each meeting will be 2-3 hours long.
- We can also use the Issue tracker on gitlab to communicate.

CHANGE MANAGEMENT

- If we run into a bug, the person who was in charge of that branch will be responsible to fix it. If they cannot fix it then other team members can assist.
- After the bug has been “fixed” the code will be reviewed by the QA lead.
- Issues will be closed by the member who opened them.
- In the case of bugs resulting from incompatible code, this will be addressed by the respective team members who created it.
    - They will then work together to make code compatible.
- Status of a bug will be changed in the issue tracker by the reporting lead.
    - This will require an acceptable tag, indicating that the bug has been fixed, along with a small description/explanation.

SOFTWARE DESIGNS

DESIGN – CLASS DIAGRAMS

Refer to Appendices.

DESIGN – SEQUENCE DIAGRAMS

Refer to Appedencies.

CLASS DESCRIPTIONS

Game:

Public:

- **Default Constructor(vector <Player*> p, int decks)** will create a card game with a vector of player pointers and decks, and will initialize our player vector, number of players, and number of decks.
- **Virtual ~Destructor**
- **Virtual void resetGame():** resets the game set up when a player wants to play another game directly after they finished one.
- **Vector <Player*> getPlayers() const:** returns the player vector.
- **virtual void printRules()** will be in charge of displaying the rules of the selected game.
- **Int getNumPlayers() const:** returns the number of players.
- **Int getNumDecks() const:** returns the number of decks.
- **Void transferCards(Cardset* location, Cardset* destination, const Card* c):** will take in the initial location of the card and the final destination. This function will be in charge of picking up cards, giving cards away, putting down cards, and will be called by dealCards.
- **void setName(string s):** is the function that will allow the player to input a name of their choice. This can be letters or numbers.
- **string getName(player* player):** returns the name of the player.
- **void printScore()** will output the scores of each player.
- **void printWinner():** will output the game rankings, using the player names.
- **void dealCards(int handSize):** using the players, will deal the cards out accordingly, taking in an int parameter of how many cards need to be dealt to each player.
- **virtual void preGame()=0:** initializes game set up
- **virtual void round()=0:** controls how a round starts, functions, and ends.
- **Virtual void play():** runs the game.
- **Virtual void playerTurn(Player* p, istream& userInput):** runs a human's turn.

- **Virtual void AITurn(Player* p):** runs an AI's turn.
- **Void quitGame():** handles when a game is quit/done.
- **Int getTopNumber():** returns the rank of the top card in the stockPile.
- **Bool gameOver** = false;
- **InputHandling IH;**
- **Enum GameType** {GOFISH, CRAZYEIGHTS, JUNGLESPEED}

Protected Members

- **int numPlayers**
- **int numDecks**
- **Int HAND_SIZE**
- **vector<Player *> Players**
- **GameType type;**
- **Deck* mainDeck;**
- **Discard stockPile**

CardSet:

Public Members:

- **Enum CardSetType** {DECK, HAND, DISCARD}
- **CardSet():** Default constructor uses setCardSetSize
- **virtual ~CardSet();**
- **virtual void display() const = 0:** Fill the array with the appropriate type of cards. Implemented in StandardCardSet and JungleSpeedCardSet.
- **int getSize() const;** Getter function that returns size of deck. Return an int the size of the deck
- **void shuffleCardSet();** Randomizes the cards before they are dealt.
- **CardSetType getType() const;** getter for card type.
- **Card* getCard(const Card* c);** getter for a specific card.
- **void addCard(Card* c);** add card to a cardset
- **void removeCard(const Card* c);** remove a card from a cardset.
- **Card* getTop();** getter for the top of the main deck, or stock pile.

Protected:

- **CardSetType type;**
- **std::vector <Card*> cards;**
- **Unsigned seed**

Deck:

Public Members:

- **Deck():** Default Constructor

- **~Deck():** Destructor
- **Void display():** does nothing, we don't want to know the cards that are in a deck. Needed another function so that Deck is not an abstract class.

Card: Abstract Class

Public Member Functions:

- **Enum CardType** {STANDARD, JUNGLESPEED}
- **Card():** default constructor.
- **Virtual ~Card():** destructor.
- **CardType getType() const:** returns the type of a card.
- **Void displayCard():** displays the suit and rank of a card.
- **virtual string stringOfWholeCard() = 0;** returns the card information as a string.
- **Virtual bool operator==(const Card& c) = 0;** operator == overloaded.

Protected:

- **CardType type;**

StandardCard: public Card

Public Member Functions:
enum Suit {DIAMONDS, HEARTS, CLUBS, SPADES};

- **StandardCard(Suit nameOfSuit, int cardNumber):**
  - Default constructor.
  - Where the number is from 1-13.
- **Virtual ~StandardCard():** destructor.
- **void setValue(int v):** sets the value of the card.
- **void setSuit(Suit s)** sets a suit of a card.
- **int getValue():** will return the value of a card.
- **Suit getSuit ():** will return the suit of a card, by calling suitToString.
- **Bool sameRank(const StandardCard& c):** compares two cards to see if they have the same rank.
- **Bool sameSuit(const StandardCard& c):** compares two cards to see if they have the same suit.
- **String stringOfWholeCard();** returns the string version of the card information.
- **string suitToString(Suit):** returns the string version of a suit.
- **Bool operator== (const Card& c):** operator == overloaded.
- **String rankToName(const int cardRank):** returns the string version of a cardRank.
- **String cardValue():** returns the string version of the card value.
- **Const string suitValue():** returns the string version of a suit.

Private Members:

- **Suit suitName;**
- **int number;**

JungleSpeedCard: public Card

enum Colour {RED, BLUE, YELLOW, GREEN};
Public:

- **JungleSpeedCard(string cardName, Colour cardColor):** constructor
- **Virtual ~JungleSpeedCard():** destructor
- **void setColour(Colour c):** sets the colour of a card.
- **void setName(string s):** sets the name of a card.
- **Colour getColour():** returns the Colour of a card.
- **String getName():** returns the name of a card.
- **String ColourToString(Colour c);** converts the enum colour to a string.
- **String stringOfWholeCard():** converts the card information into one string.
- **Bool operator==(const Card& c):** overloaded == operator to compare two cards.
- **Bool sameColour(const JungleSpeedcard& c):** compares two cards to see if they are the same colour.
- **Bool sameName(const JungleSpeedCard& c):** compares two cards to see if they have the same name.

Private:

- **String name;**
- **Colour colour;**

Player:

Public:
ENUM PlayerType {HUMAN, AI}

- **Player(int playerID):** creates a player and assigns an ID.
- **virtual ~Player():** Destructor for Player
- **PlayerType getType() const:** get the type of player, either human or AI.
- **bool getMyTurn():** returns the value from myTurn.
- **void setMyTurn(bool turn)**: sets the value of myTurn.
- **void setTotalPoints():** assign points to a player.
- **Void setRoundPoints(const int i):** set points to a player for a round.
- **int getTotalPoints():** returns the total points earned by a player.
- **Int getRoundPoints():** returns the points of a player for a single round.
- **Int getID():** returns a players ID
- **Virtual string getName() const = 0:** returns the name of a player.

- **Bool isInHand(const int cardRank):** checks if a specific cardRank is in a players hand.
- **Hand hand :** needs to be public so that transferCards can use it as a location or destination

Private:

- **bool myTurn:** will return true or false depending on if it is that specific players turn or not.
- **PlayerType type;**
- **int totalPoints**;
- **Int pointsFromOneRound;**
- **int id;**

AI: public Player

Public member functions:

Enum Level {EASY =1, NORMAL, HARD}

- **AI(Level AILevel, int playerID):** default constructor
- **~AI():** destructor
- **void setDifficultyLevel(AI::Level l):** will set the playing level of the AI.
- **Level getDifficultyLevel():** will return the playing level of the AI.
- **String getName() const:** returns name of a player.

Protected members:

- **Level aiLevel;**

JungleSpeed: public Game

Public:

- **Bool nameCriteria:** variable to determine when to match based on card name rather than colour.
- **JungleSpeed(const int decks, vector <Player*> p);** default constructor
- **~JungleSpeed():** destructor
- **Void resetGame();** resets the decks and re-deals if a player chooses to player another round.
- **void printRules():** prints out the rules of Jungle Speed. A player can ask for the rules at anytime through the game
- **void preGame():** defines preGame set up using the players.
- **void round():** defines a round using the players.

- **double reactionTime(istream& userInput):** implement the timer function to determine the players reaction time. Returns total time.
- **Player* declareLoser(vector<double> reactionTimes):**
    - Using the players, if the player's time is greater than the AI, then the player will be declared the loser of that round.
    - Else, the AI is declared the loser.
- **Player* declareWinner(vector<double> reactionTimes):** returns the winner of a duel, by comparing the reaction times. The winner has the smallest reaction time.
- **Void Duel(vector<Player*> playersDuel, istream& userInput):**
    - Calls reactionTime
    - Calls declareWinner
    - Duel will handle the card transfer.
    - After the card transfer is complete, it will check if any hands are empty
- **Void playerTurn(Player* p, istream& userInput):** runs a human's turn.
- **Int getFlippedCardsSize():** returns the size of the flippedCards vector.
- **Void AITurn(Player* p):** runs an AI's turn.
- **Int isMatch(Card* c):** checks if there is a match between flipped cards.
- **Int getStockPileSize():** returns the size of the stockPile.
- **vector<Player*> inDuel(Card* c, int pos):** returns a vector of players who are in the duel.
- **vector<Discard*> flippedCards:** vector of the flipped cards.

CrazyEights: public Game

Public:

- **StandardCard::Suit gameSuit;** keeps track of the current suit that is getting played. Used to switch the suit when an 8 is played.
- **CrazyEights(const int decks, vector <Player*> p):** default constructor
- **Virtual ~CrazyEights():** destructor
- **void printRules():** print out the rules of Crazy 8's. A player can ask for the rules at anytime through the game
- **void preGame():** defines pregame set up using the players.
- **void scoringSystem(Player* winner):** using the players called at the end of each round.
- **Void playerTurn(Player* p, istream& userInput):** runs a human's turn.
- **Void AITurn(Player* p):** runs an AI's turn.
- **Bool hasValidCard(Player* p):** checks if a player has a playable card in their hand.

GoFish: public Game

Public:

- **GoFish(const int decks, vector<Player*> p):** default constructor
- **Virtual ~GoFish():** destructor
- **Void resetGame();** resets the decks and re-deals if a player chooses to play another round.
- **void printRules():** will print the rules of Go Fish to the screen
    - A player can ask for the rules at anytime through the game
- **void makeBook(const int cardRank):** "creates" books for each player, using the players
- **void goFish():** using the players, when a request is denied, goFish will call transfer cards and take a card from the stockpile and place it in the player's hand.
- **Void playerTurn(Player* p, istream& userInput):** runs a human's turn.
- **Void AITurn(Player* p):** runs an AI's turn.
- **Void requestHandler(Player* requester, Player* requestee, vector<Card*> v):** checks to see if a players turn should end of keep going.
- **Int hasBook(Player* p):** checks if a player has a book in there hand.
- **Vector<vector<int>> previousAsk:** contains the previous asks of all players. Used for AI strategy.

Private:

- **Int books;**

CardGenerator:

Public:

- **CardGenerator();** default constructor
- **virtual ~CardGenerator();** destructor
- **StandardCard* makeStandardCard(const int typeOfSuit, const int valueOfCard);** creates a standard card
- **JungleSpeedCard* makeJungleSpeedCard(const std::string s, const int i);** creates a jungleSpeed card.

CrazyEightsAI: Public AI:

Public:

- **CrazyEightsAI(Level playerLevel, int playerID);** default constructor. Sets the level of the AI and sets the ID
- **virtual ~CrazyEightsAI();** destructor
- **Card* strategy(Card* c):** returns the card the AI will play based on its strategy.

Private:

- **vector<Card*> legalCards:** contains all the playable cards in a player's hand.

DeckGenerator:
Public:

- **DeckGenerator();** default constructor
- **virtual ~DeckGenerator();** destructor
- **Deck* makeDeck(const Game::GameType t);** makes a deck, the type of deck will depend on the GameType
- **Deck* makeDeck(const Game::GameType t, const bool twoPlayers);** used for making a 72 card deck for JungleSpeed. Used when there are only 2 players.
- **Deck* makeStandardDeck(Deck* d);** makes a standard deck of cards.
- **Deck* make72JungleSpeed(Deck* d);** makes a JungleSpeed deck without the SPECIAL cards.
- **Deck* makeExtraJungleSpeed(Deck* d);** makes a JungleSpeed deck with the SPECIAL cards.

Discard: Public CardSet:
Public:

- **Discard():** default constructor
- **~Discard():** destructor
- **Void Display():** displays the top card in the discard pile.

GameActions:
Public:

- **GameActions();** default constructor
- **virtual ~GameActions();** destructor
- **std::vector<Card*> makeRequest(Player* receiver, const int cardRank);** allows a player to make a request for a card in GoFish
- **StandardCard::Suit mostPrevalentSuit(vector<Card*> hand):** CrazyEightsAI uses this to figure out which suit to change it to when an 8 is played.

GoFishAI: Public AI:
Public:

- **GoFishAI(Level playerLevel, int playerID);** default constructor. Sets the level of the AI and the ID.
- **virtual ~GoFishAI();** destructor
- **Card* strategy();** returns the card they want to ask for based on the AI's strategy.
- **Player* ChoosePlayer(Game* g)**; is used in the AI strategy to pick a player to make a request to.

Private:

- **Unsigned seed:**
- **Int storePlayer:** stores the player that the AI would like to make a request to.

Hand: Public CardSet:
Public:

- **Hand();** default constructor
- **~Hand();** destructor
- **void display() const;** displays a players hand.
- **std::vector<Card*> getHand();** returns the hand of a player.

Human: Public Player:
Public:

- **Human(std::string playerName, int playerID);** default constructor. Sets a players name and ID
- **virtual ~Human() {}** destructor
- **void setName(std::string s);** allows a player to input a name of their choice
- **std::string getName() const;** getter function for the players names

Private:

- **std::string name;**

JungleSpeedAI: Public AI:
Public:

- **JungleSpeedAI(Level playerLevel, int playerID);** default constructor, sets the AI's difficulty level and ID.
- **virtual ~JungleSpeedAI();** destructor
- **double getResponseTime();** Returns the response time of the AI.
- **void setResponseTime(Level l);** Sets the response time of the AI depending on its difficulty level.

Private:

- **double responseTime;**

UI:
Public:

- **Static const int MAX_OPPONENTS = 3;**
- **InputHandling IH:**
- **UI()**; default constructor
- **~UI();** destructor
- **Game::GameType chooseGame();** Determines which game the player wants to play. Returns the selected Game.

- **Game* createGame(istream& userInput):** creates and runs the desired game by calling the constructor.
- **const int getNumOpponents(istream& userInput);** gets the number of AI's
- **const int getNumDecks(istream& userInput);** gets the number of decks (hardcoded to 1 deck)
- **Player* createHuman(std::string s)**; Creates a new Player, returned to the game constructor
- **Player* generateAI(Game::GameType t, AI::Level l, int id);** Generates AI depending on game choice. Implemented in GoFish, JungleSpeed, and CrazyEights
- **std::vector<Player*> createPlayers(Game::GameType gameChoice, istream& userInput);** creates the players vector

InputHandling:
Public:

- **InputHandling():** default constructor.
- **~InputHandling():** destructor.
- **Bool isLegalInt(const string temp):** checks if the inputted int is a digit.
- **Void clearStream(istream& userInput):** clears the input stream.

APPENDICES

APPENDIX A: FIGURES AND TABLES

Figure 1: All classes

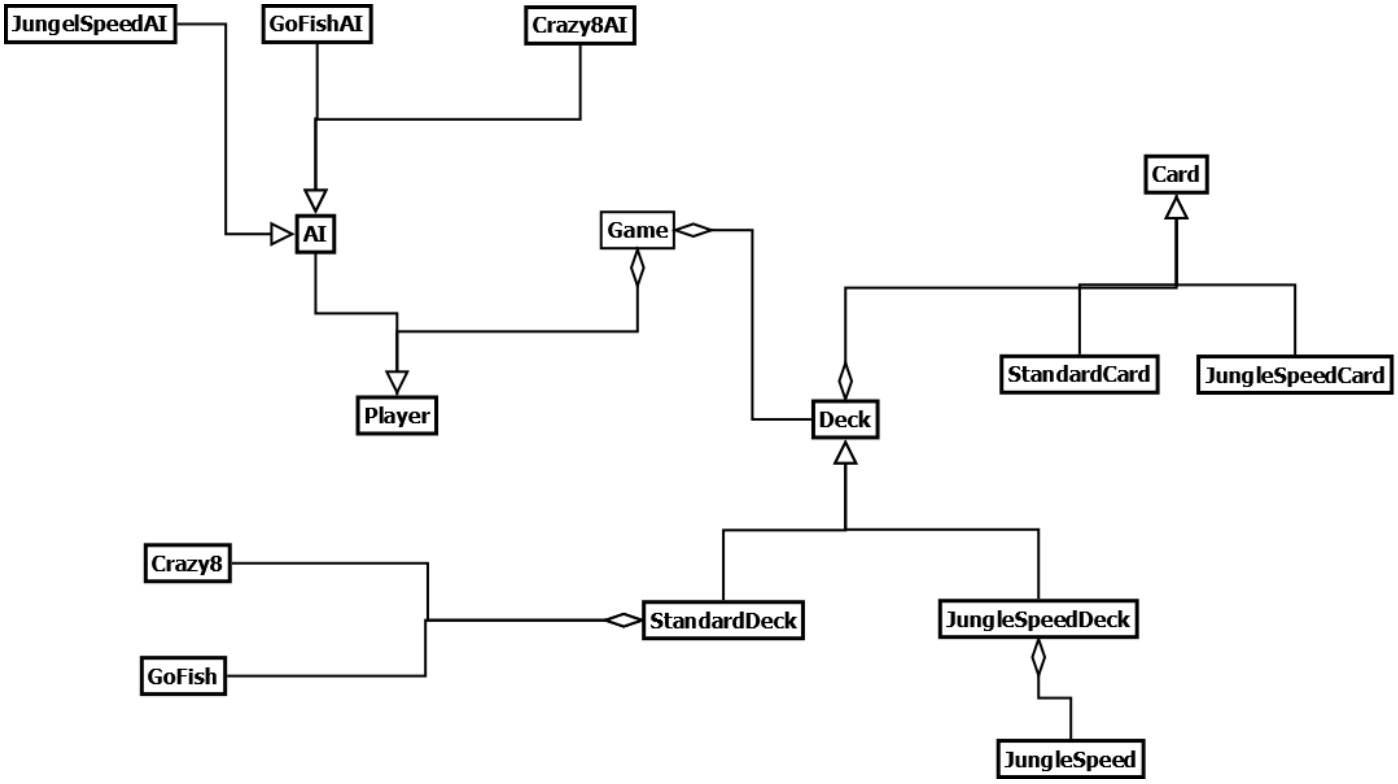

Figure 2: Players and AI's

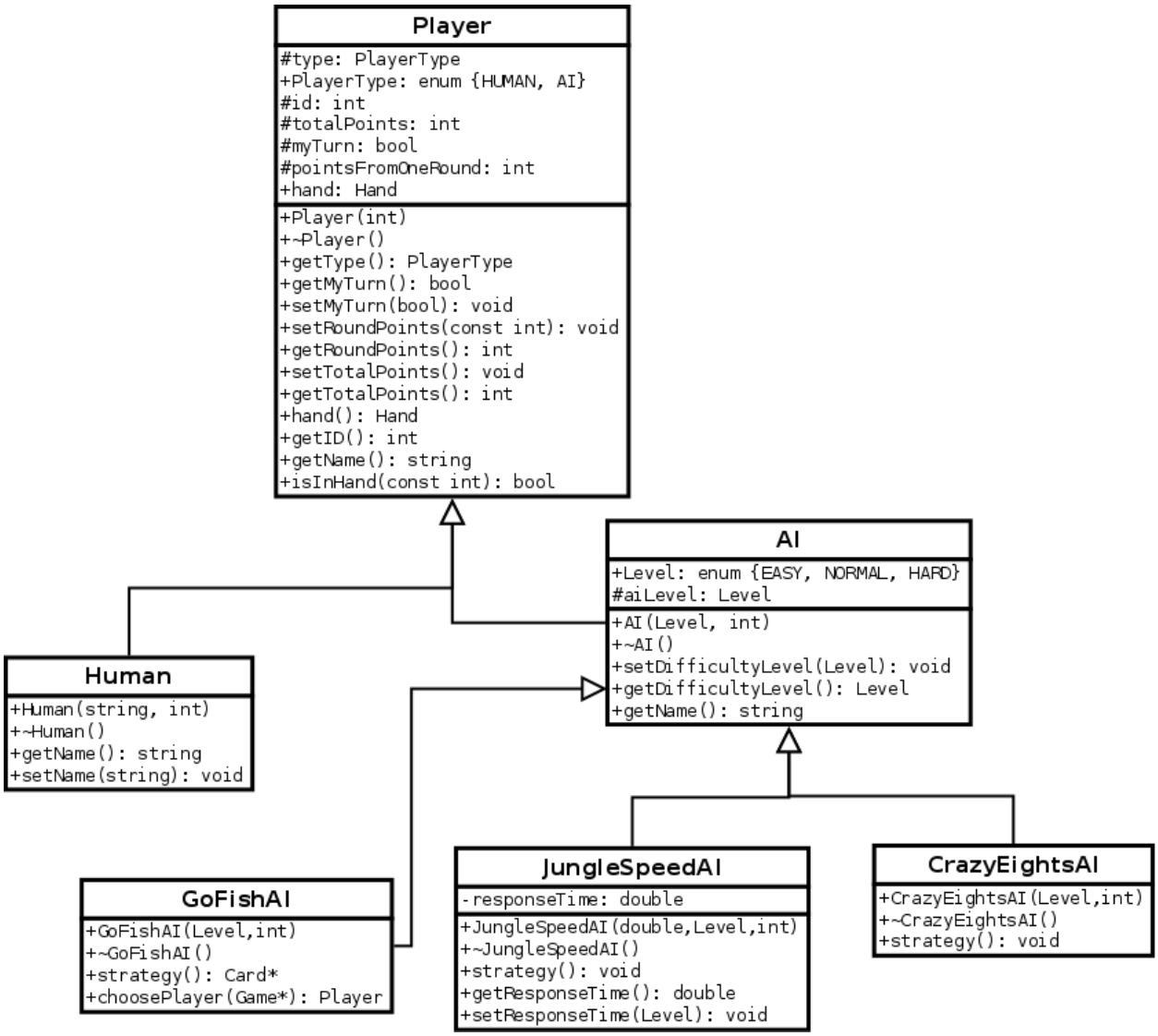

Figure 3: Game, it's related functions, and it's children

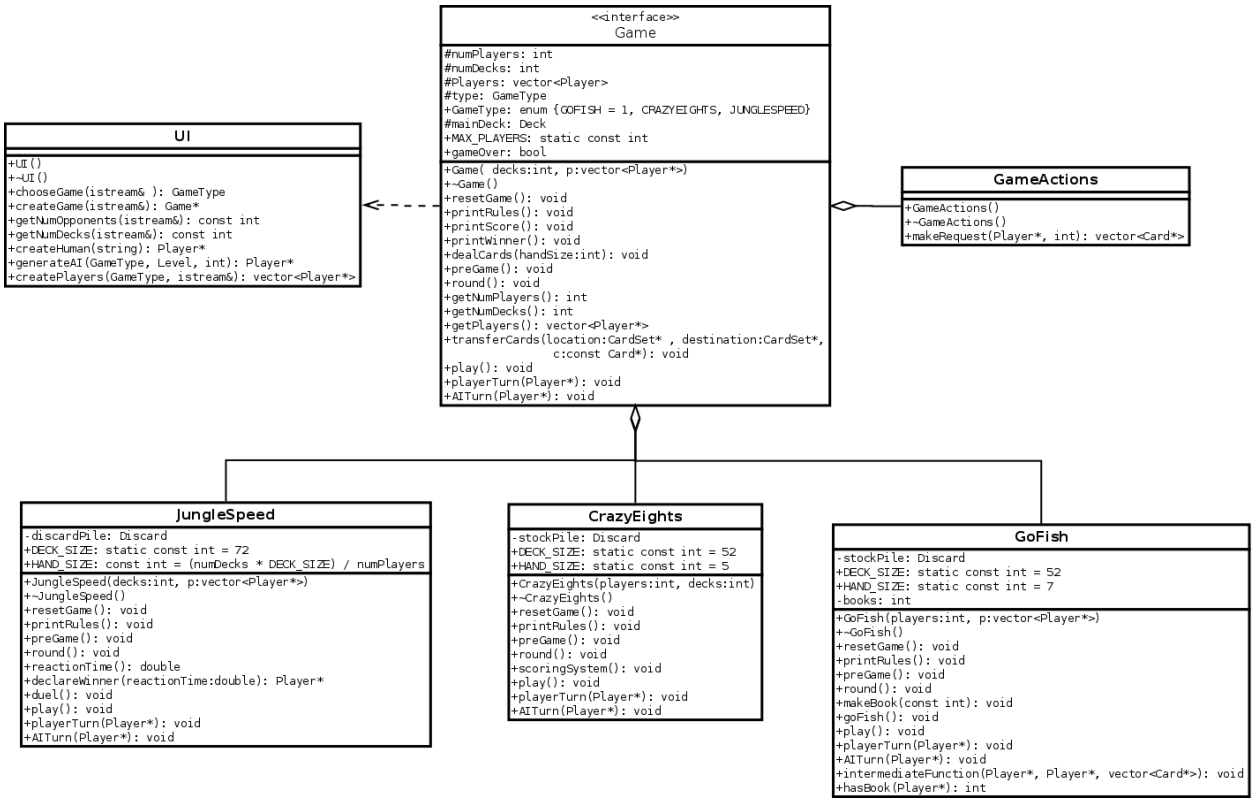

Figure 4: DeckGenerator, CardSet and it's children

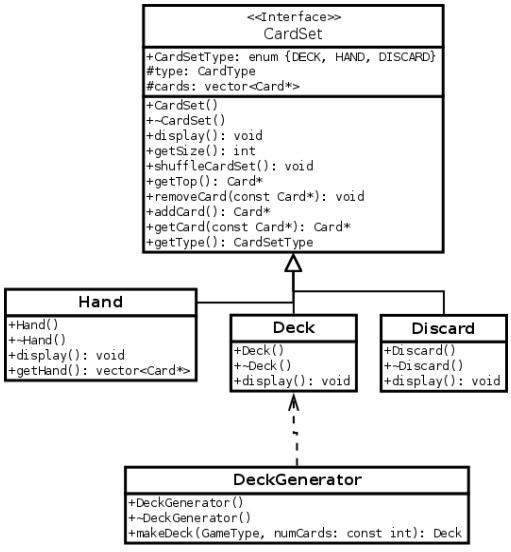

Figure 5: CardGenerator, Card and it's children

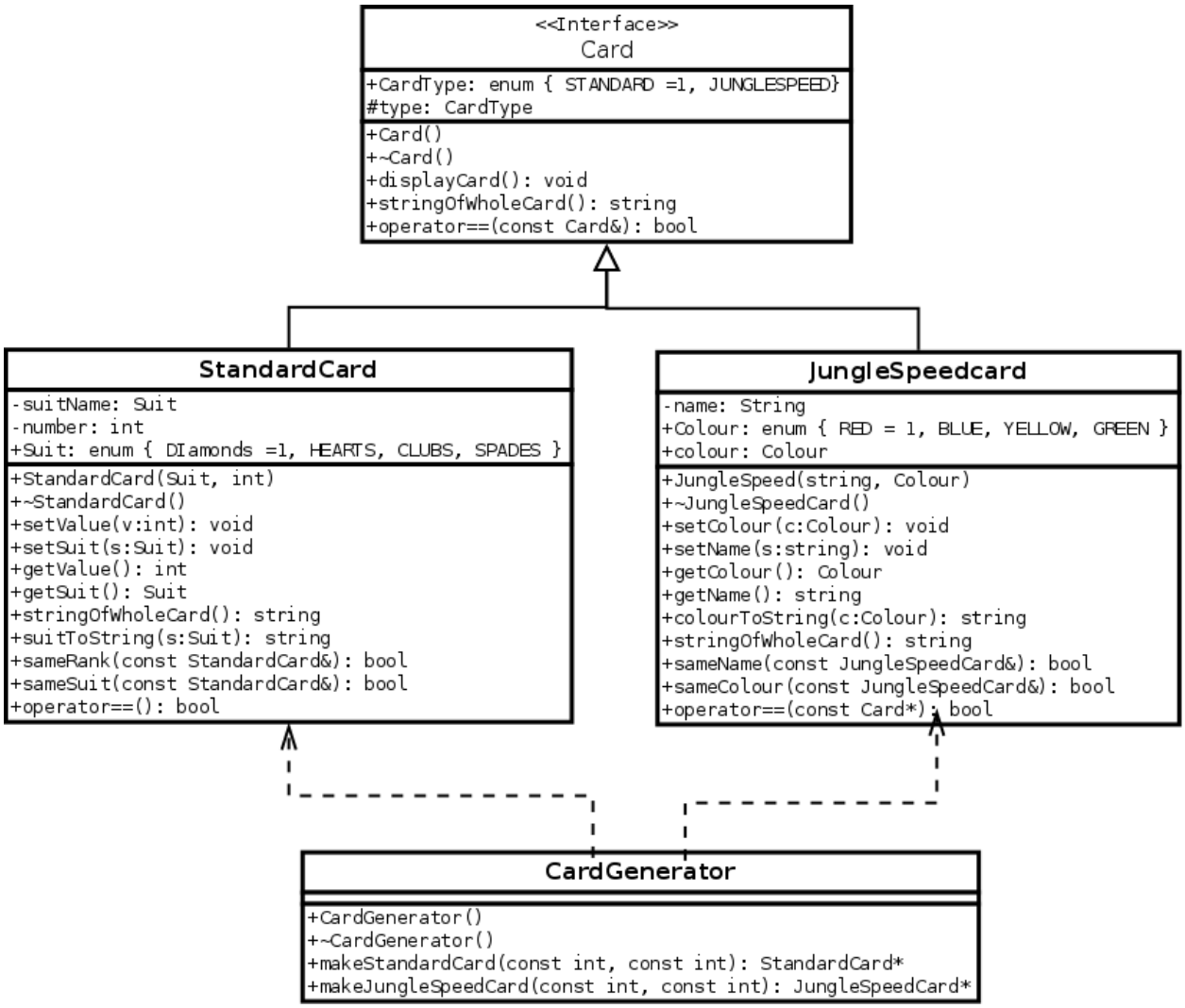

Figure 6: How to Start a Standard Deck Game

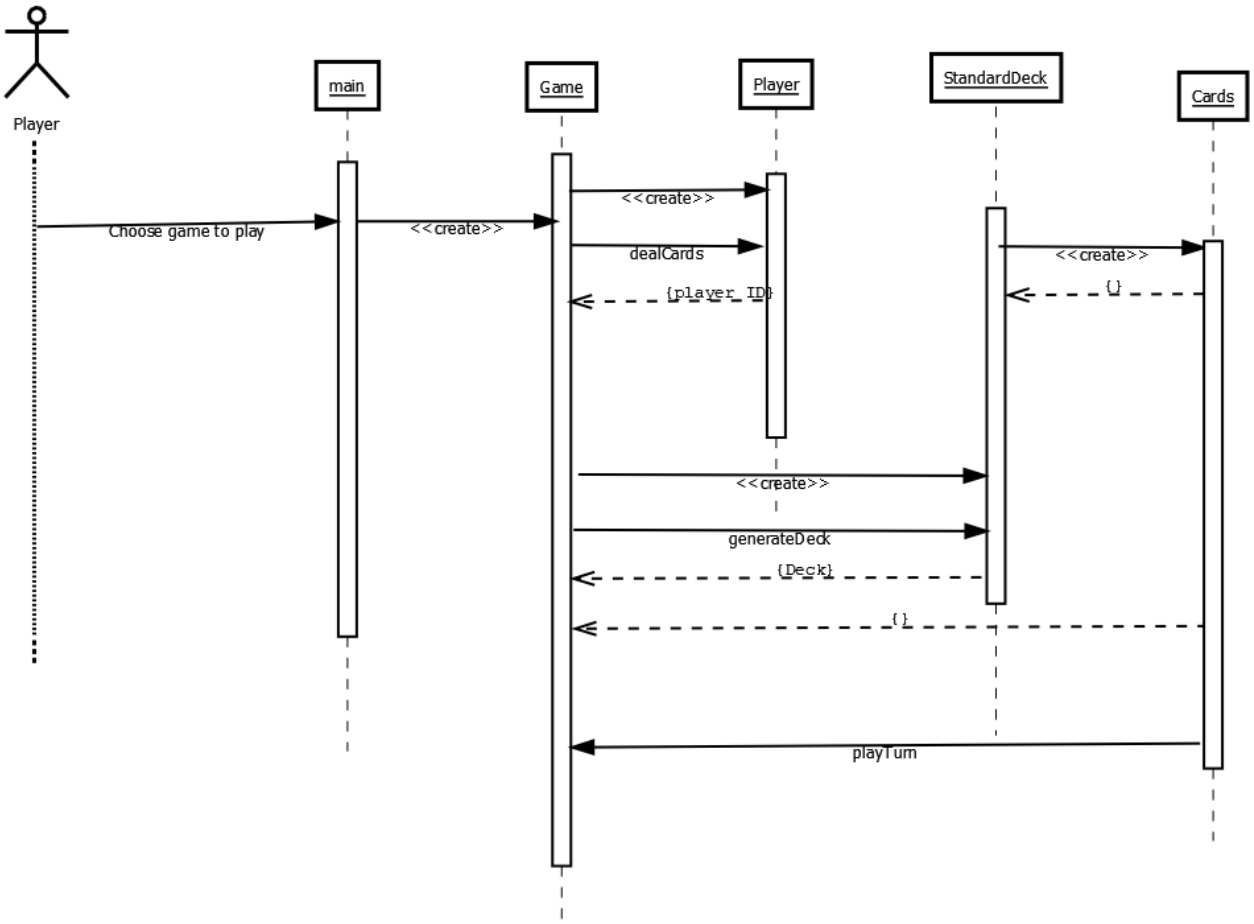
Player
main
Game
Player
StandardDeck
Cards
Choose game to play
<<create>>
<<create>>
dealCards
{player_ID}
<<create>>
{}
<<create>>
generateDeck
{Deck}
{}
playTurn

Figure 2: Sequence for one round of Crazy 8's

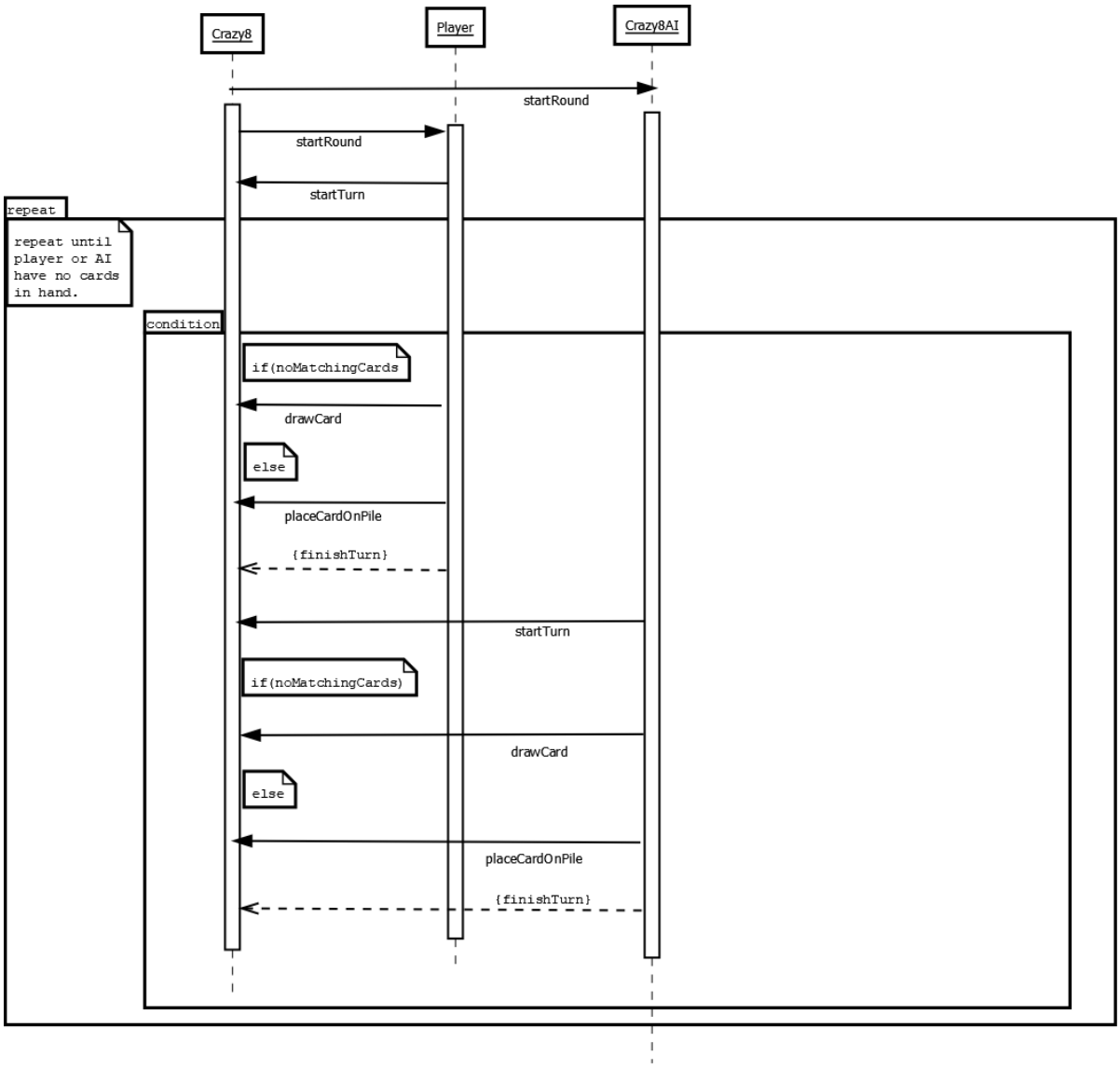

Figure 3: Sequence Diagram for one round of Go Fish

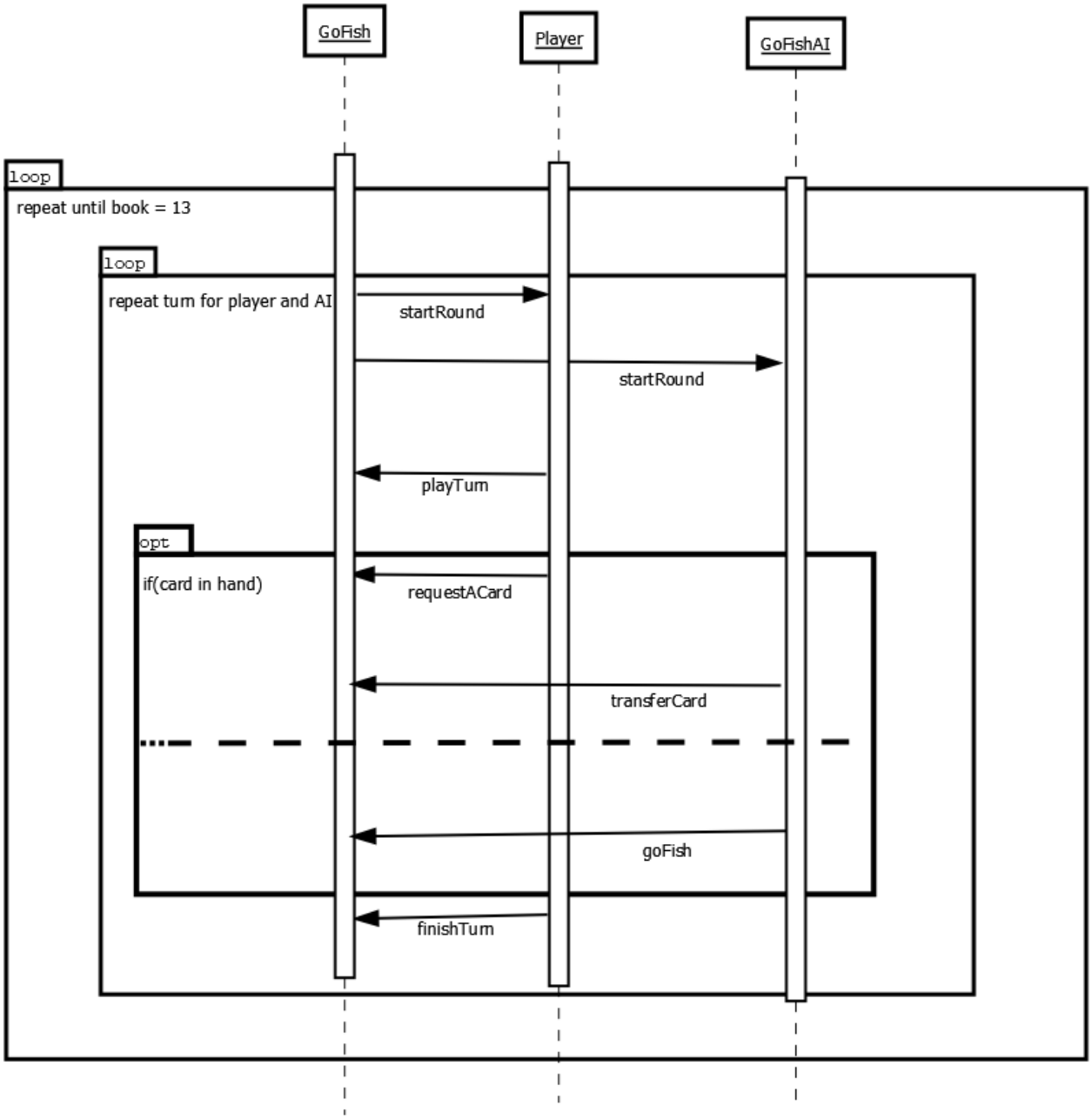

Figure 4: Sequence Diagram for one round of Jungle Speed

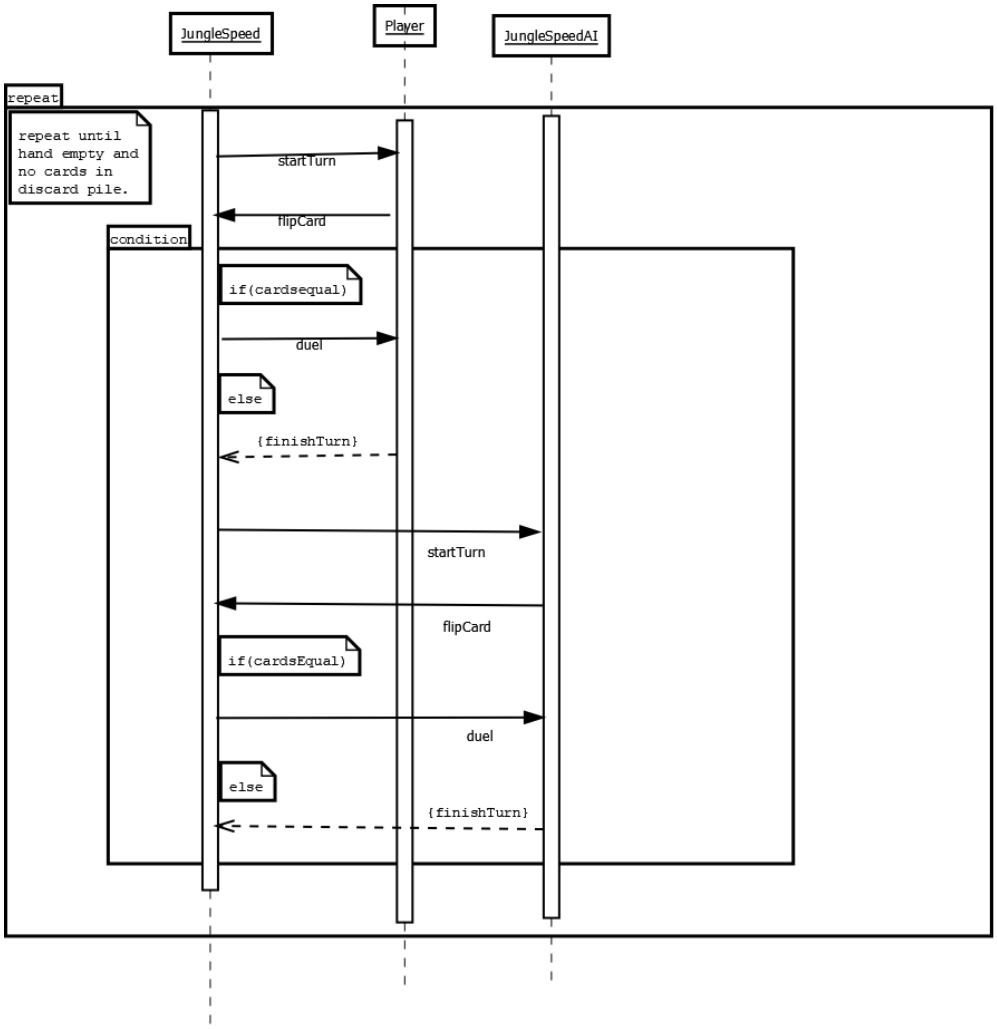

Figure 5: Sequence Diagram for after a game ends