

**Faculty of Informatics and Computer Science**

Software Engineering

**Estimation Card Game Artificial Intelligence**

By: Ahmed Hatem Tageldin

Supervised By

Dr. Ayman Ghoneim

Abstract

Estimation is a popular trick-taking card game. There are several applications on the market. However, as most of them focus mainly on the multiplayer component of the game, the Artificial Intelligence component is sometimes neglected. The objective of this project is to create an Estimation Artificial Intelligence system that is capable of simulating the expertise and unpredictability of a real human player. There are other related works similar to Estimation, such as Whist. Whist is an English trick-taking card game, however it is internationally known. Therefore, resources are more readily available for it. Instead of starting from scratch when building the Artificial Intelligence system, similar existing card games can be analysed and their Artificial Intelligence may be extracted. It can be used as a guide or even a template for further refinements. However, the fact that Estimation is region-based, may prove to be problematic in finding resources. Thus, making the job of implementation even more challenging.

# Table of Contents

[Abstract ii](#_Toc484439932)

[Table of Contents iii](#_Toc484439933)

[Table of Figures vi](#_Toc484439934)

[1 Introduction 1](#_Toc484439935)

[1.1 Overview 1](#_Toc484439936)

[1.2 Problem Statement 1](#_Toc484439937)

[1.3 Scope and Objectives 1](#_Toc484439938)

[1.4 Report Organization (Structure) 1](#_Toc484439939)

[1.5 Work Methodology 2](#_Toc484439940)

[2 Related Work 3](#_Toc484439941)

[2.1 Background 3](#_Toc484439942)

[2.1.1 Estimation Game Logistics 3](#_Toc484439943)

[2.2 Literature Survey & Analysis 8](#_Toc484439944)

[3 Proposed solution 9](#_Toc484439945)

[3.1 Overview 9](#_Toc484439946)

[3.2 Object Oriented Programming 9](#_Toc484439947)

[3.3 Technologies Used 11](#_Toc484439948)

[3.3.1 Java 11](#_Toc484439949)

[3.3.2 NetBeans 11](#_Toc484439950)

[3.3.3 GitHub 11](#_Toc484439951)

[3.4 Functional Requirements 11](#_Toc484439952)

[3.4.1 System Requirements 11](#_Toc484439953)

[3.4.2 Player Requirements 12](#_Toc484439954)

[3.5 Design 12](#_Toc484439955)

[3.5.1 Structural Model – Class Diagram 12](#_Toc484439956)

[3.5.2 Behavioural Model – Sequence Diagrams 17](#_Toc484439957)

[4 Implementation 26](#_Toc484439958)

[4.1 Program Flow 26](#_Toc484439959)

[4.1.1 Testing or Playing 26](#_Toc484439960)

[4.1.2 Start Gameplay 26](#_Toc484439961)

[4.1.3 Bidding 27](#_Toc484439962)

[4.1.4 Start Play 28](#_Toc484439963)

[4.1.5 Game State Implementation 28](#_Toc484439964)

[5 Testing and Evaluation 30](#_Toc484439965)

[5.1 Testing 30](#_Toc484439966)

[5.2 Evaluation 30](#_Toc484439967)

[6 Conclusions and Future Work 32](#_Toc484439968)

[6.1 Summary 32](#_Toc484439969)

[6.2 Future Work 32](#_Toc484439970)

[References 33](#_Toc484439971)

[Appendix 1 34](#_Toc484439972)

[Appendix 2 35](#_Toc484439973)

[Appendix 3 36](#_Toc484439974)

[Appendix 4 37](#_Toc484439975)

[Appendix 5 38](#_Toc484439976)

[Appendix 6 39](#_Toc484439977)

[Appendix 7 – Installation Guide 40](#_Toc484439978)

# Table of Figures

[Figure 1 - Scoring Table 6](#_Toc484439979)

[Figure 2 - Estimation Class Diagram 13](file:///C:\Users\Ahmad\Documents\GitHub\Estimation\%5bGP%5d%20Estimation%20Card%20Game%20Dissertation.docx#_Toc484439980)

[Figure 3 - State Design Pattern Diagram 14](file:///C:\Users\Ahmad\Documents\GitHub\Estimation\%5bGP%5d%20Estimation%20Card%20Game%20Dissertation.docx#_Toc484439981)

[Figure 4 – Round **startPlay (int cursor)** Sequence Diagram 17](#_Toc484439982)

[Figure 5 –Suit **compareWeight (ArrayList <Integer> hand)** Sequence Diagram 18](#_Toc484439983)

[Figure 6 - User **openBidding()** Sequence Diagram 20](file:///C:\Users\Ahmad\Documents\GitHub\Estimation\%5bGP%5d%20Estimation%20Card%20Game%20Dissertation.docx#_Toc484439984)

[Figure 7 - Computer **openBidding()** Sequence Diagram 22](file:///C:\Users\Ahmad\Documents\GitHub\Estimation\%5bGP%5d%20Estimation%20Card%20Game%20Dissertation.docx#_Toc484439985)

[Figure 8 - Computer **dashCall()** Sequence Diagram 24](file:///C:\Users\Ahmad\Documents\GitHub\Estimation\%5bGP%5d%20Estimation%20Card%20Game%20Dissertation.docx#_Toc484439986)

[Figure 9 - Dash Call Survey Sample Question 31](file:///C:\Users\Ahmad\Documents\GitHub\Estimation\%5bGP%5d%20Estimation%20Card%20Game%20Dissertation.docx#_Toc484439987)

[Figure 10 - Trump Suit Call Survey Sample Question 31](file:///C:\Users\Ahmad\Documents\GitHub\Estimation\%5bGP%5d%20Estimation%20Card%20Game%20Dissertation.docx#_Toc484439988)

[Figure 11 - Round deal Sequence Diagram 34](file:///C:\Users\Ahmad\Documents\GitHub\Estimation\%5bGP%5d%20Estimation%20Card%20Game%20Dissertation.docx#_Toc484439989)

[Figure 12 - Round secondRoundBids Sequence Diagrams 35](file:///C:\Users\Ahmad\Documents\GitHub\Estimation\%5bGP%5d%20Estimation%20Card%20Game%20Dissertation.docx#_Toc484439990)

[Figure 13 - Round startBidding Sequence Diagram 36](file:///C:\Users\Ahmad\Documents\GitHub\Estimation\%5bGP%5d%20Estimation%20Card%20Game%20Dissertation.docx#_Toc484439991)

[Figure 14 - User secondRoundBidding Sequence Diagram 37](file:///C:\Users\Ahmad\Documents\GitHub\Estimation\%5bGP%5d%20Estimation%20Card%20Game%20Dissertation.docx#_Toc484439992)

[Figure 15 - Session startSession Sequence Diagram 38](#_Toc484439993)

[Figure 16 - Session nextRound Sequence Diagram 39](file:///C:\Users\Ahmad\Documents\GitHub\Estimation\%5bGP%5d%20Estimation%20Card%20Game%20Dissertation.docx#_Toc484439994)

# Introduction

## Overview

Estimation is a four-player trick-taking card game. It is an Arab variation of the classic English card game Whist. The objective of the game is to estimate the amount of tricks, given a hand of thirteen cards. Each player plays one card in each hand. Whoever wins the hand collects a trick. There are thirteen tricks in each round. The person with the highest score at the end of eighteen rounds is the winner. The aim of the project is to create a playable Estimation game and simulate an artificially intelligent single player computer bot.

## Problem Statement

There are several Estimation card game applications on the market. Most applications focus their efforts on the online multiplayer side of the game, thus neglecting the single player side. However, most of them do not contain an Artificial Intelligence component that an experienced player would find sufficiently challenging.

## [Scope](http://www.cs.stir.ac.uk/~kjt/research/conformed.html) and Objectives

The objective of the project is to create an Artificial Intelligence component for the card game Estimation. By combining expertise in playing the game and passion in Java programming, the expected outcome is an Artificial Intelligence component that can simulate the thought process and the unpredictability of an experienced human player. The main expected deliverable is a Java application. However, seeing as Android development is based on Java, an Android application may be delivered as an extension deliverable if the project can be sufficiently completed in due time. The future aim of this project is an Android application.

## Report Organization (Structure)

Section 2 covers the background of the subject, related works, their analysis and their potential usefulness to the project. Section 3 covers the proposed solution of the project, as to how the problem will be approached and the design of the implementation. Section 4 discusses the implementation and the program flow of the project. Section 5 covers the testing and evaluation of aspects of the project. Section 6 concludes the report with a summary of all the previous sections.

## Work Methodology

Plan A is to find projects fitting the same criteria and learn from their structure, their source codes and concepts. They can be used as guides. If such projects can be found, it will eliminate the need to start from scratch. For instance, other Estimation applications.

Plan B is to find projects fitting similar criteria and learn from their structure, their source codes and concepts. Although not ideal, it will provide a foundation on which to work from. For instance, variations of trick-taking games such as Whist.

Plan C is to find projects that share the same common basic attributes as Estimation. For instance, any card game application such as Poker, Hearts, Spades, etc.

Plan D is to start from scratch, which is the least ideal scenario as it involves much trial and error.

# Related Work

## Background

As per plan A, mentioned in section 1.5, the most ideal scenario is to find projects fitting the same criteria. Specifically, other Estimation projects. The main source of such related works is GitHub, which is a web-based repository service where developers can upload their projects and source codes online. The biggest problem with finding resources specifically for Estimation is that it is a region-specific card game, specifically in the Middle East. Therefore, resources are scarce, however, not non-existent.

The alternative to plan A, namely plan B, is to find projects fitting similar criteria. This may include other trick-taking games, such as Whist, Spades, Contract Bridge, Oh Hell, etc. The advantage to plan B is that these card games are international, therefore resources are much more likely to be readily available. The Artificial Intelligence component is likely to be more advanced as the game has a wider market, therefore more competition. The most useful part of finding resources is understanding various concepts of Artificial Intelligence programming, especially relating to card game science. Probabilities and statistics are likely to play an important role in the formulation of the Artificial Intelligence component.

### Estimation Game Logistics

#### Rules

The objective of the game is to estimate the amount of tricks, given a hand of thirteen cards. Each player plays one card in each hand. Whoever wins the hand collects a trick. There are thirteen tricks in each round. The card rankings are as follows, from highest to lowest: Ace, King, Queen, Jack, 10, 9, 8, 7, 6, 5, 4, 3 and 2.

#### Card Suits

There are four suits in a deck of cards. In Estimation, they are given a certain weight ranking. The ranking of the suits is as follows:

1. Suns
2. Spades
3. Hearts
4. Diamonds
5. Clubs

#### Game Structure

A game, or **Session**, of Estimation consists of eighteen rounds. The first thirteen rounds are played normally, with normal bidding rules. The final five rounds of the game are played in a fast bidding mode. The trump suit of each round is predetermined. The order of the trump suits in each of the final five rounds is as follows: Suns, Spades, Hearts, Diamonds and Clubs. Because the trump suits of the final five rounds are pre-determined, dash call may not be declared. If at any time a round is restarted, the score of the following round is incremented by two.

#### Round Structure

Each player receives a hand of thirteen cards. The objective of the game is to correctly estimate the amount of tricks that can be taken with the given hand. A round consists of thirteen hands; four cards for each hand, one from each player, totalling 52 cards. The trump suit is the suit that is elevated above its normal rank, which is chosen at the beginning of each round. At the beginning of each round, players take turns bidding for the trump suit.

#### Bidding

At the beginning of each round, players will be given a chance to declare a dash call. A dash call means that the player must not collect a single trick for the entire round. A dash call may only be declared before the trump suit bidding begins. The player on the right of the dealer begins bidding for the trump suit. A player may choose to make a bid or pass to the next player. The player with the highest bid may choose the trump suit. The bidder may also choose the trump suit as Suns, which means that there is no trump suit; the highest card of the played suit always wins. The trump suit ranking is as follows: Suns, Spades, Hearts, Diamonds and Clubs. For instance, a call of 5 Suns is greater than a call of 5 Spades, however a call of 6 Spades is greater than a call of 5 Suns. After the trump suit is chosen, the rest of the players declare their estimate bid, starting from the right of the trump suit bidder. The total sum of the players’ bids must not be equal to thirteen. The last player to make a bid estimate, or the player to the left of the trump suit bidder, has a bidding limit imposed on them. The round must either be under thirteen or over thirteen. If the last bidder makes a bid that is under or over the limit by at least two, they are flagged as a risk. If the round is under thirteen, that means that at least one player will take an extra trick. If the round is over thirteen, that means that at least one player won’t collect all of their estimated tricks. This system guarantees that at least one player will lose in each round.

#### Gameplay Flow

Once everyone has finished bidding, the trump suit bidder opens play. If the trump suit is Suns, the highest card of the played suit wins the trick. Otherwise, the highest card of the played suit wins the trick, unless someone interjects with a trump suit card. The card thrown by the player must be of the same suit. If a player does not have any cards of that suit, any card may be thrown, including trump suit cards. If more than one trump suit card is played, the highest trump suit card wins. In the fast bidding rounds, the player to the right of the dealer starts bidding in the pre-determined trump suit. The order of the trump suits in each of the final five rounds is as follows: Suns, Spades, Hearts, Diamonds and Clubs. A player wins the round if they collect the exact amount of tricks they bid at the beginning of the round.

#### Exceptions

A round may be restarted in two cases; the first of which is during call bidding, if every player chooses to pass instead of bidding. The second case is when every player loses the round. In other words, if no player collects the exact number of tricks they bid at the beginning of the round, the round is restarted. Whenever a round is restarted, the score multiplier for the following round is incremented by two.

#### Scoring Model

The scoring model was adapted from an Estimation Calculator application. [1]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Case** | | | | **Scoring Equation** |
| Win | Caller / With | Call > 7 | Only winner |  |
| Not only winner |  |
| Call <= 7 | Only winner |  |
| Not only winner |  |
| Not Caller | Dash Call | Only winner |  |
| Not only winner |  |
| Normal Bid | Only winner |  |
| Not only winner |  |
| Lose | Caller / With | Call > 7 | Only loser |  |
| Not only loser |  |
| Call <= 7 | Only loser |  |
| Not only loser |  |
| Not Caller | Dash Call | Only loser |  |
| Not only loser |  |
| Normal Bid | Only loser |  |
| Not only loser |  |

Figure 1 - Scoring Table

#### Notes

* **With** refers to the case when a player bids a number of tricks equal to the caller. For instance, if the trump suit caller bids 5 Suns and a player bids 5 tricks, they are considered to be **With.** The scoring calculation for the caller and the **With** players is the same.
* The *call* variable refers to the amount of tricks the player bid.
* The *risk* variable refers to the amount of tricks the last bidder called. If a player bids 2 tricks under or over the limit, they are flagged as a risk. If a player bids 4 tricks under or over the limit, they are flagged as a double risk. If a player bids 6 tricks under or over the limit, they are flagged as a triple risk. Each risk is multiplied by 10 and added to the score. For instance, if a player bids 3 tricks under the limit, they are flagged as a double risk; when the score is calculated, 20 points are added to the score, before the multiplier.
* The *difference* variable refers to the absolute value of the predicted number of tricks subtracted from the actual number of tricks collected.
* If only one player wins, 10 points are added to the score calculation, before the multiplier. The same calculation applies if only one player loses.
* The *Multiplier* variable may be affected by several factors. If any of the following phenomena takes place, the*Multiplier* variable is incremented by two:

1. The previous round had no winners
2. The previous round had every player pass in the bidding phase
3. Three players called **With** the trump suit caller
4. Two players declare Dash Call in one round

## Literature Survey & Analysis

Existing Estimation projects are scarce, ones with open-source code are non-existent. There exist three Estimation card games on the Google Play Store, the most relevant of which is Pocket Estimation. [2] Pocket Estimation boasts 500,000 downloads; it is the most popular Estimation card game on the market. However, despite that fact, its single player mode does not pose a challenge to an expert player.

Projects similar to Estimation are much more likely to yield useful results. One such project is Whist. Whist is a classic English trick-taking card game originating from the 18th century. There exist several Android applications for the game Whist. The most popular of those applications is aptly named “Whist”. The creator of this game has uploaded the source code and the documentation of the game to GitHub. The original source code language is Java. All things considered, this is the main source that will be used for the program structure and architecture. The creator of this project is Nikos Tsaousis. [3] Within the project uploaded to GitHub, he has included the code architecture (Model View Presenter), the class diagram, the techniques used in the creation of the Artificial Intelligence computer opponents and the software design patterns. In comparison to the other Estimation projects, Nikos’s application is much more helpful as it is documented and explained. The only downside of this project is that it is very complex.

# Proposed solution

## Overview

As an expert Estimation player, the experienced gained over the years will be very useful in creating the Artificial Intelligence component. Naturally, as the program is developed, there will be several versions of the Artificial Intelligence with increasing complexity and difficulty as more time is spent on the code. Each version may be used as a benchmark to test against the final product. The game may have several difficulties, with the final version being the hardest; each version modelling a slightly increased difficulty with different tactics, strategies and approaches. The final version should be able to simulate the difficulty and unpredictability of playing against a real player.

Another potential solution is one that actively learns from the player, gauging their level of expertise, evolving and adjusting its difficulty accordingly. However, this type of system is likely to be more complex because of the learning agent. It may use the advantage of having three computer opponents versus one human opponent. Thus, allowing it a much broader perspective of the game instead of the perspective of one opponent.

The proposed solution is a form of expert system. An expert system is a form of artificial intelligence that simulates the knowledge and decision-making skills of a human expert. An expert system consists of three elements: a knowledge base, an inference engine and a shell interface. The knowledge base is comprised of facts and rules. The inference engine is the reasoning and deduction based on the facts and rules of the knowledge base. Given the rules of the game and a collection of facts provided by an expert, the system should be able to analyse the environment then deduce a decision that maximises chances of success.

## Object Oriented Programming

Object oriented programming (shorthand OOP) will be used to create this project. OOP is used to define a system that models how humans interact with the world around them, by defining everything in terms of classes and objects, which are instances of said classes. A class may have attributes and methods, which serve to manipulate the class’s attributes. This is essentially the blueprint of an object. There are four major principles of OOP: Abstraction, Encapsulation, Inheritance and Polymorphism.

Abstraction is defining a high-level view of the operations of an object and hiding details of the exact implementation. The purpose of abstraction is to provide a common generalisation to be used by different classes. A good rule of thumb to remember is “program to an interface, not an implementation”. Abstraction has several advantages, such as reusability, readability and elimination of code redundancies. An abstract class is easier to understand than a concrete class as it hides extraneous details. The higher the abstraction, the lower the amount of implementation details. Encapsulation is similar to abstraction, it is the process of combining attributes and the operations that manipulate them in one place. The purpose of encapsulation is to hide the details of the underlying implementation of operations and control the way in which data members are accessed or changed. Encapsulation can be achieved by using access modifiers to control access to data attributes and their operations. One way to do this is to set the access modifiers of attributes and their operations to private. This is called data hiding, which is denying access to the internal representation of an attribute to any entity outside the class. Another way to achieve encapsulation is to make attributes private and their operations public, therefore providing a public interface with which to interact with attributes. This protects attributes from being changed in problematic ways.

Inheritance is a mechanism that allows an object to receive, use and in other words, inherit, the attributes, operations, behaviours and properties of another class. The inherited class is called the base class, or superclass. The inheriting class is called the derived class, or subclass. Inheritance has several advantages, such as reusability and the elimination of code redundancies. A subclass may access its superclass’s attributes and methods, but a superclass may not access its subclasses’ attributes or methods. Some programming languages allow multiple inheritance, where a subclass can have multiple superclasses. The final major principle of OOP is polymorphism. Polymorphism is a mechanism that allows any inheriting class to change the implementation of any inherited operation. There are two types of polymorphism: static (compile-time) polymorphism and dynamic (run-time) polymorphism. An example of static polymorphism is method overloading. Method overloading is a mechanism that changes the signature of a method; this allows the user to have a method of the same name but with different parameters, such as number of parameters and type of parameters. An example of dynamic polymorphism is method overriding. Method overriding is a mechanism that allows any subclass to change the implementation of any inherited method. Dynamic polymorphism and inheritance are closely related.

## Technologies Used

### Java

Java is an object oriented programming language, developed by Sun Microsystems. Java is one of the most widely used programming languages in the world, as it supports a wide variety of libraries and APIs. The main reason for using Java is that it is the same language that is used by Android Studio. This allows for the project to be adapted into an Android application, which is the future aim of this project.

### NetBeans

NetBeans is an integrated development environment (IDE) that was developed by the same company that developed Java; Sun Microsystems. It is one of the most popular IDEs in the world. There are several advantages to using NetBeans. It has a wide array of plug-ins, built-in and custom user-made plug-ins. NetBeans also has code analysis, debugging tools and code generation features. It also has a built-in GUI designing tool.

### GitHub

GitHub is a web-based repository hosting site. It provides version control and source code management. GitHub is most useful when used with a group of developers working on the same project, however in this project, it will be used to create many versions as it is being developed.

## Functional Requirements

### System Requirements

* Programming the game rules
* Programming the game structure
* Integrating exception handling
* Create bidding system to manage normal and fast bidding

### Player Requirements

* Manage score calculation model and display player scores
* Facilitate computer and user player management
* Create artificially intelligent computer bot capable of analysing the game and making decisions to maximise chances of success, which includes:
  + Dash Call system
  + Bidding system

## Design

### Structural Model – Class Diagram

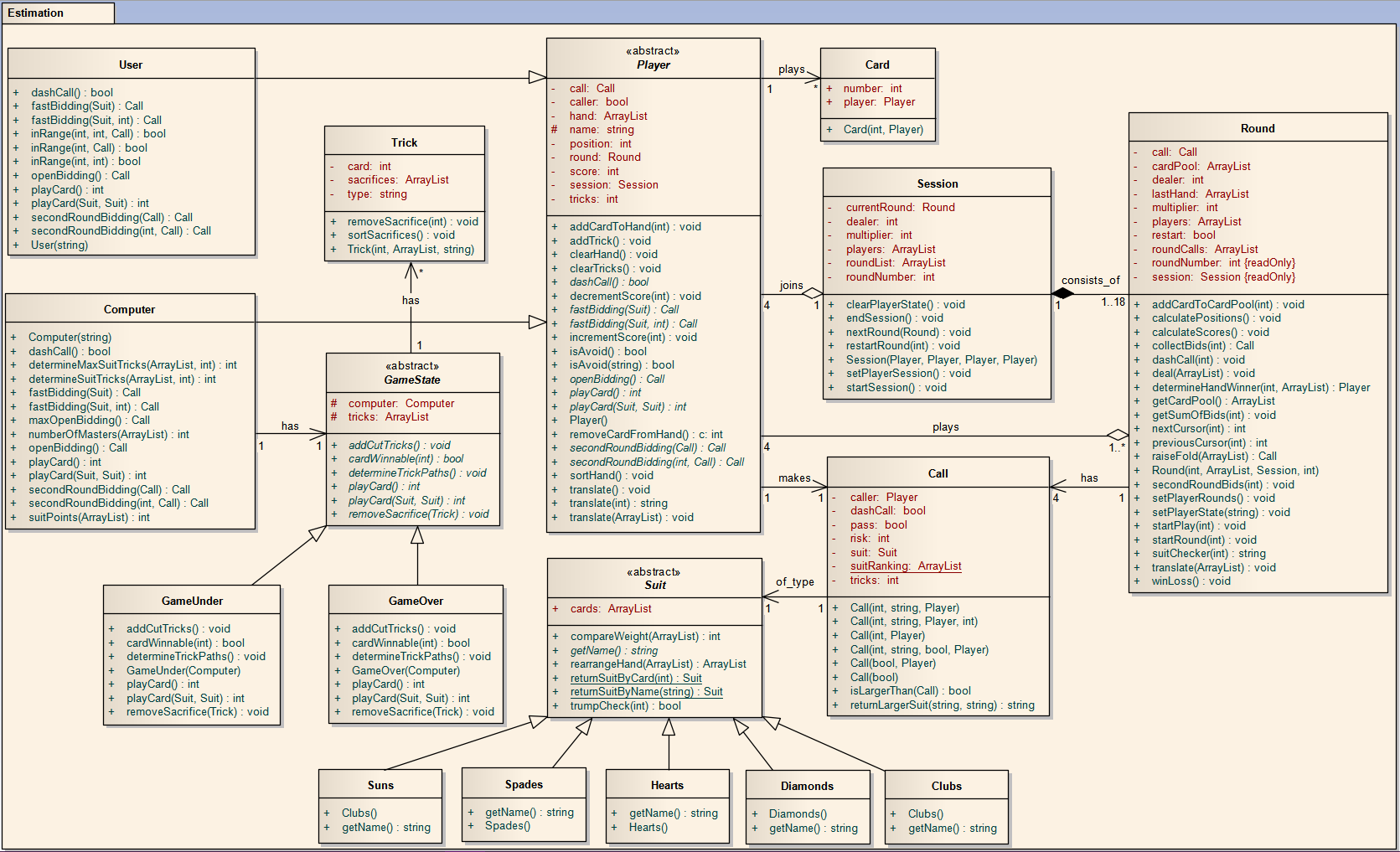


Figure 2 - Estimation Class Diagram

#### State Design Pattern

The State design pattern is a behavioural design pattern. It is used to alter an object’s behaviour when its state changes. In this project, it is used to change the way a Computer object plays a card. In an Estimation round, there are two possible states. The game is either over the limit (13 tricks) or under the limit. Depending on the state of the game, the Computer object approaches the game in a different way. When the game is over the limit, the Computer object is more aggressive with its approach, taking tricks as soon as possible. If the game is under the limit, the Computer object is more conservative with its approach, delaying trick collection as much as possible.



Figure 3 - State Design Pattern Diagram

The state in this case is the GameState class. The concrete classes are GameUnder and GameOver. The context is the Computer class. It contains an object of the GameState class. The **playCard** method in the Computer class calls the **playCard** method in the GameState object. In each Round, after the bidding session is finished, the states of all the Computer players are set.

#### Abstraction and Polymorphism

Abstract classes are to be used in the project. An abstract class must have at least one method declared as abstract, meaning they have no implementation. This forces any inheriting subclass to implement any abstract methods. Another feature of abstract classes is that they may not be instantiated, however they may be used in polymorphism. This is particularly useful as it allows the programmer to avoid making the distinction between the subclasses; instead, the programmer may use the general abstract superclass to refer to any subclass without limiting options. In particular, the Player class and the Suit class.

##### Player Class

The decision to make the Player class abstract came as a result of the varying implementations of its child classes. The User subclass represents human players, whose implementation requires active input from the user. For instance, a human player must decide on a call to bid on, thus entering a number and a Suit. On the other hand, the Computer subclass, which represents the artificially intelligent automated Player, has a different implementation than the User subclass. For instance, a Computer Player would automatically analyse the hand dealt to it and calculate the optimal bid in such circumstances without any prior input from the human Player.

##### Suit Class

The Suit class is a different case, as it serves a different function. It is also an abstract class, because its subclasses only have one element that varies. Each Suit has a set of cards which mark its domain. For instance, a Spades class’s domain is from the card 40 to the card 52. This behaviour is implemented in each class constructor. All the methods are inherited as they perform the same operation, but the numbers on which the operations are performed are what varies from subclass to subclass. The only exception to the Suit rules is the Suns class. Its domain is the whole deck of cards, from card 1 to card 52. That is because it does not allow a trump suit.

#### Relationships

* User-Player relationship – the Player class is specialised into a User class. This class is for human players. The User class overrides a number of methods for a more specific implementation. This is an example of inheritance.
* Computer-Player relationship – the Player class is specialised into a Computer class. This class is for computer bot players. The Computer class overrides a number of methods for a more specific implementation. This is an example of inheritance.
* Player-plays-Card relationship – a Player plays one Card at a time. However, over the course of one round, each player plays thirteen Cards. Therefore justifying the multiplicity: one Player plays many Cards.
* Player-joins-Session relationship – this relationship is an example of aggregation. A Player may exist outside of a Session. The object construction occurs outside of the Session class. A Player may be in only one Session at a time. Each Session is comprised of exactly four Players. Therefore justifying the multiplicity: four Players join one Session.
* Session-consists of-Round relationship – this relationship is an example of composition. The object construction occurs inside the Session class; a Round may not exist without a Session. A typical Session of Estimation is eighteen Rounds, however a Session may be ended early. One Round can only belong to one Session. Therefore justifying the multiplicity: one Session consists of one-to-eighteen Rounds.
* Player-plays-Round relationship – this relationship is an example of aggregation. A Player may exist outside of a Round. Each Player can only play in one Round at a time. A Round of Estimation must have four Players. Therefore justifying the multiplicity: four Players play one-to-many Rounds.
* Round-has-Call relationship – one Round has four Calls, one Call for each Player. The main Call for the Round is in a variable of its own, named *call*, while the rest of the Calls are stored in an ArrayList. Therefore justifying the multiplicity: one Round has four Calls.
* Player-makes-Call relationship – every Round, each Player makes one Call. Therefore justifying the multiplicity: one Player makes one Call.
* Call-of type-Suit relationship – one Call can be of one Suit type. Therefore justifying the multiplicity: one Call of type one Suit.

### Behavioural Model – Sequence Diagrams

The project will be dependent on many methods and functions, therefore only the most fundamental and important methods will be listed below.

#### Round – startPlay

Figure 4 – Round **startPlay (int cursor)** Sequence Diagram

**Explanation**

This method is responsible for the distribution of play during each round. After call bidding and second round bidding has taken place, the Round class initiates play. The cursor parameter is used to determine the person who opens play. The player who won the trump suit bid opens play, then the players take turns to play a card. Each card played is added to an ArrayList, in order, to keep track of the cards played in each hand. After four cards have been played, the winner of the hand is determined and that player has their tricks incremented. Then the hand is cleared to make space for the next four cards. This process is repeated thirteen times, as each player has thirteen cards, totalling 52 cards altogether. This method has no return.

#### Suit – compareWeight

Figure 5 –Suit **compareWeight (ArrayList <Integer> hand)** Sequence Diagram

**Explanation**

This method serves to be one of the most important functionalities in the system. It is responsible for comparing a set of four cards together and returning the largest card. The Round class calls this method when determining the hand winner after each hand. The first step is to sort the given hand, this is to reduce the potential amount of processing that the main algorithm has to do. As it is a list of integers, it can be sorted using the default **Collections.sort** comparator. The main algorithm has a series of if conditions with the method **trumpCheck(int)**, which returns a true value if the given integer is in the Suit object’s card domain. (See Section 3.3.1.1.2 Suit Class). For instance, the card 51 (King of Spades) would be in the domain of the Spades Suit. If all of the cards in the hand are in the domain of the object Suit, then the highest card wins. If not all of the cards in the hand are in the domain of the object Suit, then only the cards in the domain are compared. When this method is called in the Round object, the person who plays the first card in the hand is the one who decides the Suit object whose **compareWeight** method is called. This guarantees that there would be at least one card that fits the **trumpCheck** condition. For instance, if the first card in the hand is the Ace of Spades, the Spades **compareWeight** method gets called. There is only one exception to this rule, which is if one of the following players (not the first player) does not have a card of the played Suit and interjects with a trump card (based on the Round trump suit), the Round object calls the **compareWeight** method of the trump suit. This would mean that only the trump card played would be in the domain, meaning that it would be the highest card. This method returns an integer, the highest card.

#### User – openBidding

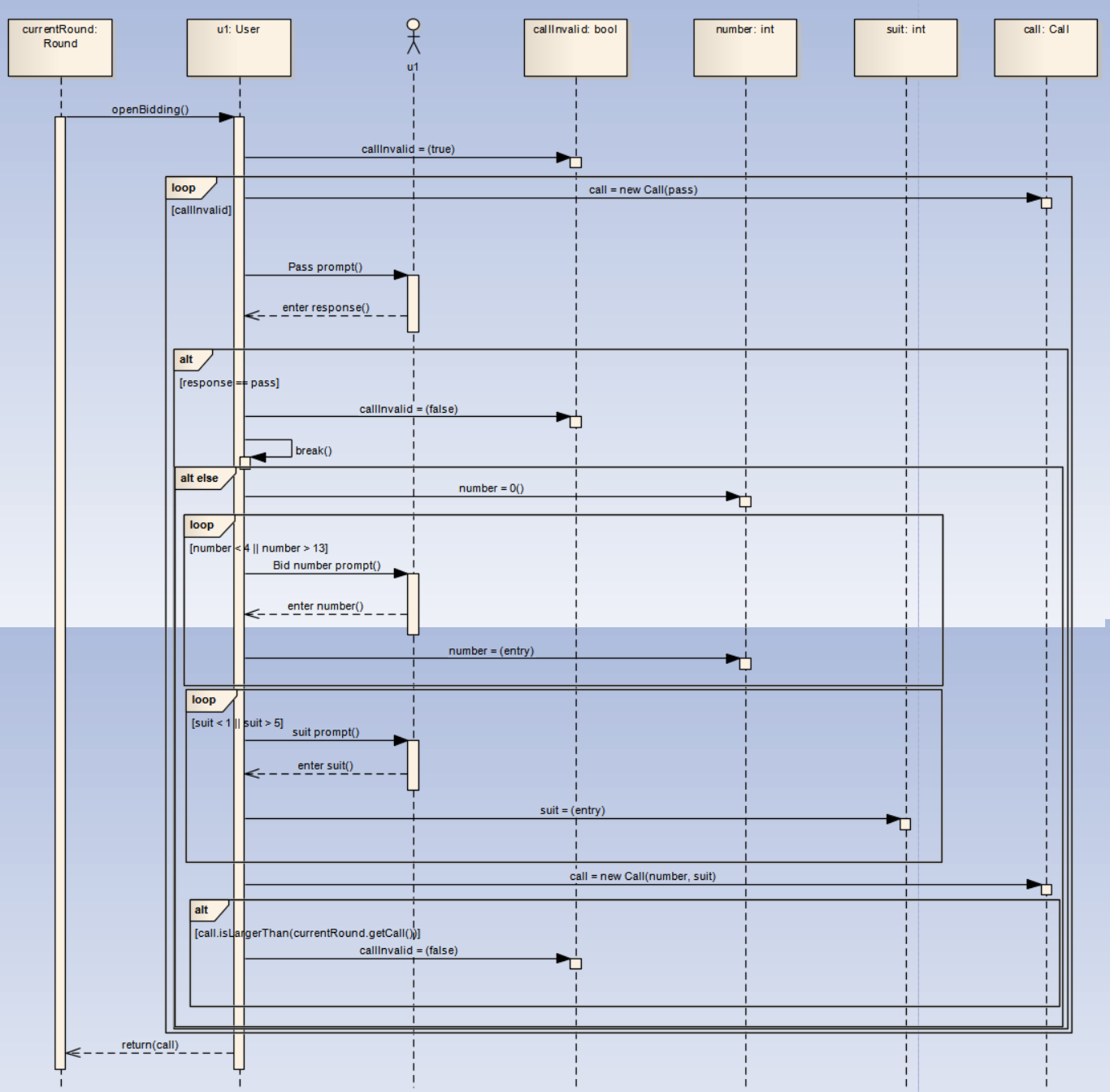


Figure 6 - User **openBidding()** Sequence Diagram

**Explanation**

This method allows human user Players to participate in the trump Suit bidding round. The Round object calls this method. The main while loop has a condition that forces the user to either pass or make a valid call. As long as the call is invalid, the loop will never break. The user is first allowed to pass on bidding, which would break the loop and result in a “Passed” call. Otherwise, the user is then directed to choose a number to bid between four and thirteen, as the trump Suit call cannot be less than four or greater than thirteen. After the number has been chosen, the user is then prompted to choose a Suit to bid on. The call is then created and compared with the biggest trump Suit call made so far in the Round. If it is bigger, then the loop is broken. This method returns the call made by the user.

#### Computer – openBidding

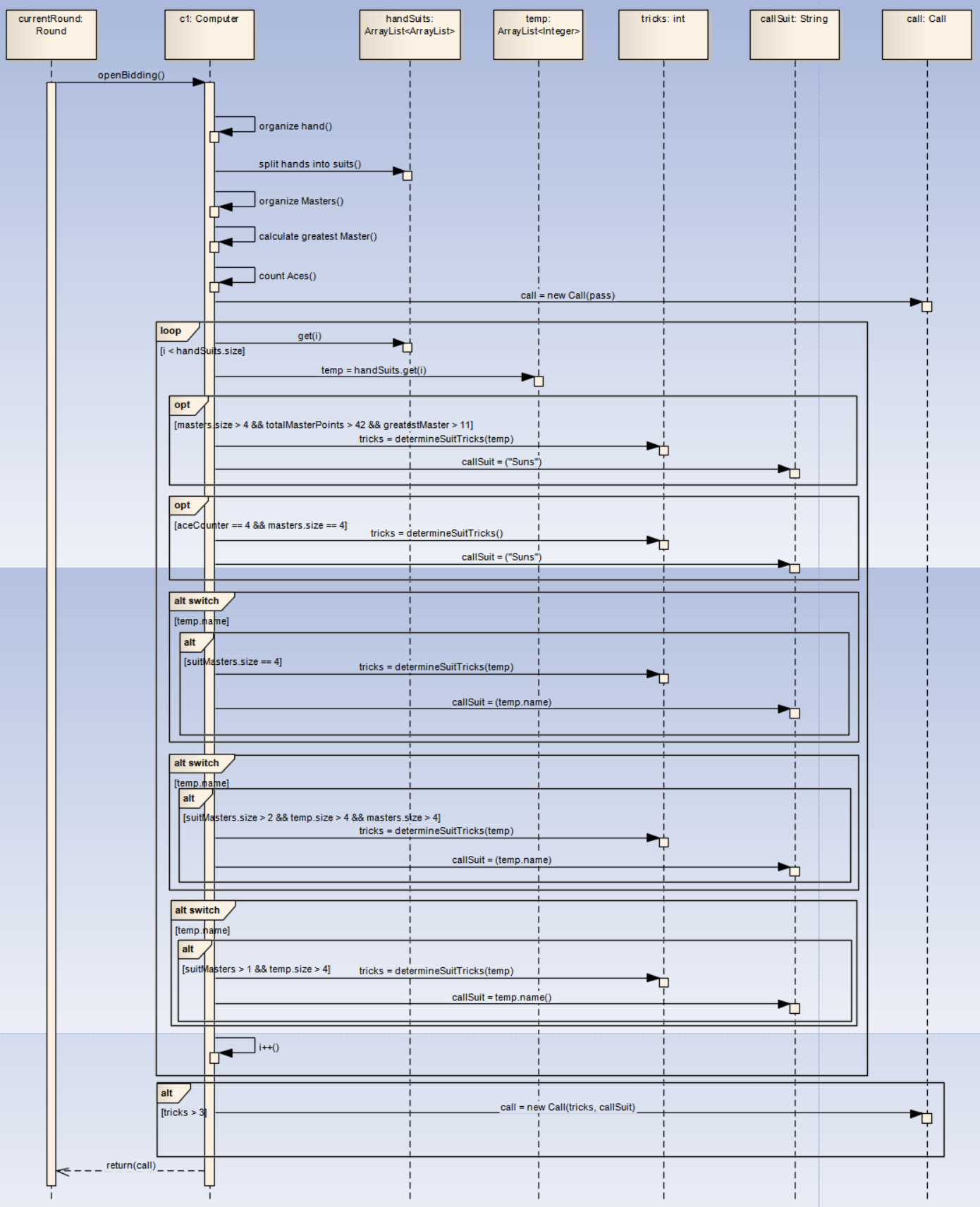


Figure 7 - Computer **openBidding()** Sequence Diagram

**Explanation**

This method allows the computer Players to participate in the trump Suit bidding round. The Round object calls this method. The first step is to organize the hand into several lists and variables. Some organizing operations include counting the Aces in the hand, counting the masters, counting total master points, putting the cards in the hand into their respective Suit lists, putting masters into their respective master Suit lists and determining greatest master in each Suit. After organizing the hand, a loop traverses the Suit list and each Suit is subjected to five cases, each with specific criteria which must be satisfied:

1. Total amount of masters in hand is greater than 4  
   Sum of master points in hand greater than 42  
   Greatest master in hand higher than a Queen  
   *Call Suit = Suns*
2. Hand contains a total of 4 Aces  
   *Call Suit = Suns*
3. Suit contains a full house (Ace, King, Queen and Jack)  
   *Call Suit = Suit (Spades/Hearts/Diamonds/Clubs)*
4. Suit has more than 2 master cards  
   Suit has more than 4 cards  
   Hand has more than 4 master cards in total   
   *Call Suit = Suit (Spades/Hearts/Diamonds/Clubs)*
5. Suit has 2 or more master cards  
   Suit has 5 or more cards   
   *Call Suit = Suit (Spades/Hearts/Diamonds/Clubs)*

In each case, the tricks are counted by calling the method **determineSuitTricks (suit)**. The result is then put into a variable *tricks* and the call Suit is put into a variable *callSuit*. After the case loop, the method then checks if the *tricks* variable is greater than 3, as the smallest possible trump Suit call is 4. If it is greater than 3, then the call is made using the *tricks* and *callSuit*. Otherwise, the call is a “Passed” call. The method then returns the call generated.

#### Computer – dashCall

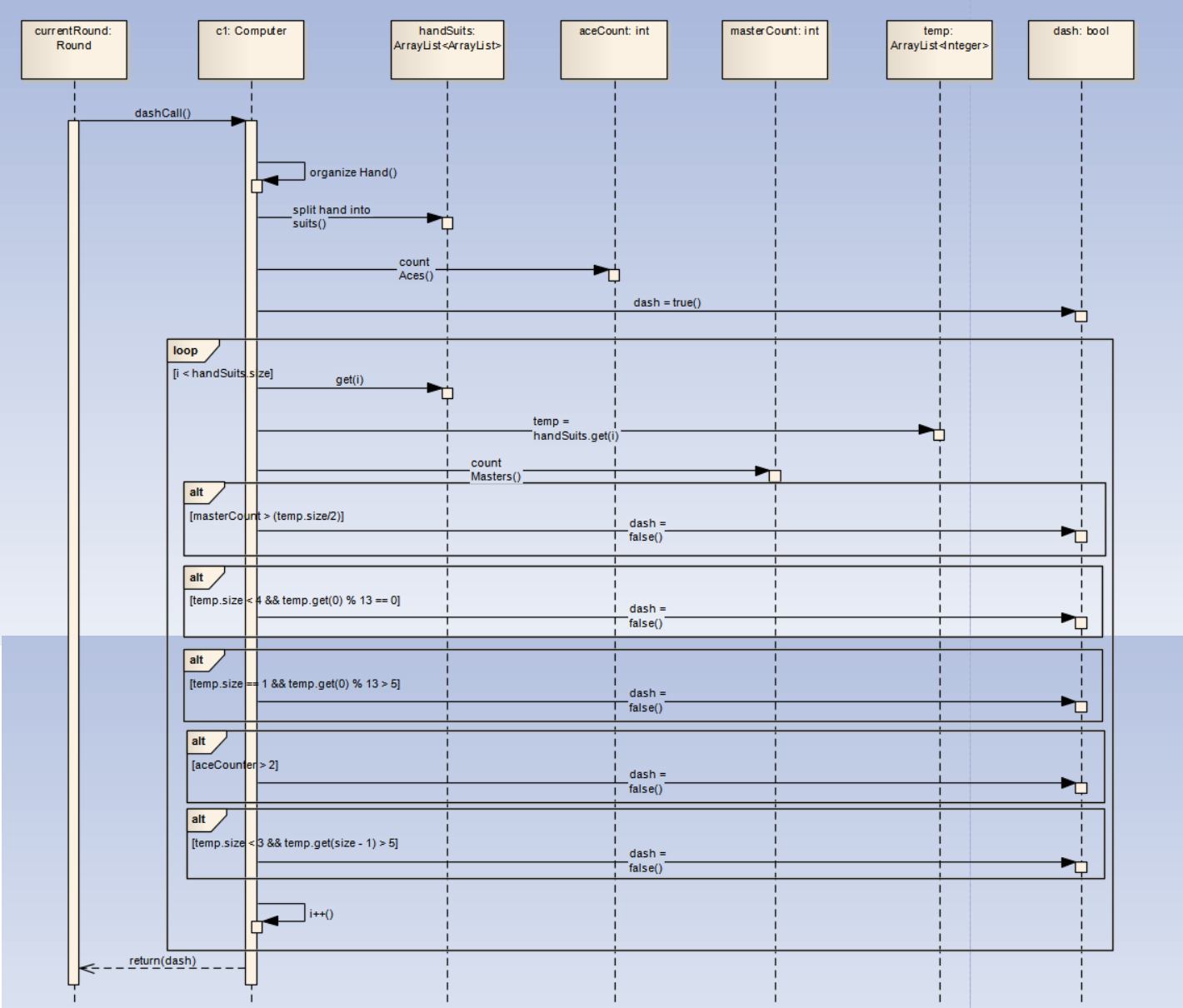


Figure 8 - Computer **dashCall()** Sequence Diagram

**Explanation**

This method allows the computer Players to declare a dash call. This method is called by the Round object before bidding begins. The first step is to organize the hand into several lists and variables. Some organizing operations include counting the Aces in the hand, counting the masters and putting the cards in the hand into their respective Suit lists. A Boolean variable named *dash* is used as a flag to determine whether or not dash call can be declared; it is initially set as true. After the hand is organized, a loop traverses the Suit list and each Suit is subjected to a series of five cases, each with specific criteria which must be satisfied:

1. The amount of master cards in the Suit is greater than or equal to half
2. Suit has less than 4 cards  
   One of the 4 cards is an Ace
3. Suit has 1 card that is greater than 5
4. Hand contains more than 2 Aces
5. Suit has less than 3 cards  
   Smallest card in the Suit is greater than 5

Inside each of these cases, the *dash* variable is set as false, to indicate that dash call is not safe to be declared. This method then returns the *dash* variable.

# Implementation

The artificial intelligence component was designed in the model of an expert system. An expert system consists of a knowledge base, an inference engine and a shell interface. An important concept that was used thoroughly in this project is modular arithmetic. Modular arithmetic was used to determine the value of a card. For instance, the cards 13, 26, 39 and 52 are all Aces because the (card % 13) value of each of them is 0. The cards 12, 25, 38 and 51 are all Kings because the (card % 13) value of each of them is 12 and so on.

## Program Flow

### Testing or Playing

The main function gives the user two options: to display the test functions or to play the game. There are three test functions, Dash Call, Trump Suit Call and Trick Determination. If the user chooses to play the game, they are given a choice of playing with three other users, which is used mainly for testing and experimenting, or playing with three other computers. After the players are decided, a session is created with the chosen selection of players and the session is started with the **startSession** function in the Session class.

### Start Gameplay

After the Session is created, the round number is set to 1 and the dealer is set to 0. The *dealer* attribute represents the player to the right of the card dealer. In a real game of Estimation, the player to the right of the card dealer is the one who opens bidding. After this, a new Round is created with the dealer as 0 and the **startRound** function of the Round class is called. The first thing that is done is to clear the player state. While this is not a problem in the first round, however in every other round, the players still retain information from the previous round, such as tricks. The **deal** function deals cards to the players; this function adds numbers from 1 to 52 in an ArrayList, then **Collections.shuffle** is used to ensure randomness and then the cards are distributed among the players. The player who is marked by the cursor (Array position) may now open bidding.

### Bidding

The **startBidding** function first checks if the round number is less than 14, otherwise it would go into fast bidding mode. (See Section **2.1.1.3** Game Structure). If the round number is less than 14, then players are given a chance to declare a dash call before bidding starts, so the **dashCall** function of the class Round is called. This function loops through the list of players, asking each one if they can declare a dash call. If two people declare dash call in one round, then no more players may declare dash call and the multiplier of the round is incremented by 2. After players are given a chance to declare dash call, the **collectBids** function is called. This function traverses the player list, starting with the cursor position, and collects a bid from each one. A bid may either be a “Pass” or a trump suit call. If all players choose to pass, then the round is restarted and the multiplier is incremented by 2. Each time a trump suit call is made, it is compared with the round call (largest call made thus far) if there is one, otherwise it is set as the round call. If the call made is bigger than the round call, then the bigger call replaces the round call as the new round call. The function keeps track of each player who made a call and gives them a chance to increase their call above the current round call or pass. After a round call has been decided, the second round of bidding may commence, so the **secondRoundBids** function of the Round class is called with the cursor parameter. The *cursor* now points to the player to the right of the round caller.

The **secondRoundBids** function first checks if the round number is less than 14, otherwise it would go into fast bidding mode. The function determines who the last bidder is, the player on the left of the round caller. A loop traverses the player list, starting at the cursor, and calls the **secondRoundBidding** function of the Player class, which determines the amount of tricks the player can get with the hand they have, given the suit. The last bidder has a limit of tricks imposed on them, as the total sum of all bids must not equal to 13. (See Section **2.1.1.5** Bidding). After second round bidding has finished and the round state has been determined, the state of each computer player is set, then the round caller opens play.

If the round number is greater than 13, then fast bidding takes place of normal bidding. The **startBidding** function has a provision that takes care of this. Instead of the function **collectBids** being called first, the **secondRoundBids** function is called instead, which has a provision to handle fast bidding rounds differently. The suit is pre-determined. Instead of calling the **collectBids** function or the **secondRoundBidding** function of the Player class, the **fastBidding** function is called. The call of each player is collected. The player with the biggest call is set as the round caller. If more than one player chose the biggest call (e.g. 4 Suns and 4 Suns), then the player who made the call first is chosen as the round caller.

### Start Play

The **startPlay** function is responsible for the distribution of play during each round. A loop of 13 iterations repeats a set of instructions. The cursor parameter is used to determine the person who opens play. The player who won the trump suit bid opens play, then the players take turns to play a card. Each card played is added to an ArrayList, in order, to keep track of the cards played in each hand. After four cards have been played, the winner of the hand is determined and that player has their tricks incremented. The player who wins the hand is the one who opens play next. The winner of each hand is determined by the **compareWeight** function of the Suit class. (See Section **3.5.2.2** Suit – compareWeight) The Suit class, whose **compareWeight** function is called, is decided by the player who opens play. If the player plays a Spades card, then the Suit is Spade and all players have to follow the Suit. If anyone is avoid of the Suit that is played, they may play any other card, including trump cards. If any player plays a trump card, then the suit of that trump card is called. This does not apply if the trump suit of the round is Suns. After the winner has been determined, then the hand is cleared to make space for the next four cards. If the state of the round is under limit, players are more conservative in their play, until someone loses (collects extra tricks). This function has a provision which changes the state of the Computer players in the round if someone loses in an under limit round.

### Game State Implementation

The states decide what tricks can be won and how to win them. On creation, the constructor calls the **determineTrickPaths** function, which serves to create a list of the tricks possible and their sacrifices. A sacrifice is a card that is thrown in order to allow any cards that are higher to be played. If a card does not have any sacrifices, then it is not counted as a trick. For instance, a King must have at least one sacrifice, to allow for the Ace to fall. A Queen must have two sacrifices, in order to allow for the King and the Ace to fall. If the **determineTrickPaths** function results in a list of tricks whose size is less than the player’s call, then the **addCutTricks** function is called. The **playCard** method analyses the player’s hand and deduces the best possible next move. For instance, if the player is the caller and the player’s hand contains a full house, then the strategy is to play all the cards in the full house in order to drain the other players of their trump cards. This eliminates the possibility that someone might interject play with a trump card and cut, therefore increasing the chances of winning a less likely trick with several sacrifices. One such example is the Jack, which requires three sacrifices, one for each master higher than itself.

Once a player reaches their desired amount of tricks, the game state changes to GameUnder, even if the state of the round is over limit. This is to ensure that the player does not collect any extra tricks. The main characteristic of the GameUnder state’s approach to play is that the lowest card possible is always played. The lowest card possible is determined by the other cards that are already played and the cards already in the hand. For instance, if the previous cards played are 2, 3, 4 and 5, then 6 is the lowest possible card. If the hand of cards already played contains a card that is high, then the highest possible card, while it is also lower than the high card that was played, is played. This is where the bulk of the knowledge base is. An if condition exists for each case.

# Testing and Evaluation

## Testing

There are three functions which will be available for testing in the program: Dash Call, Trump Suit Call and Trick Determination.

* Dash Call – a function will produce a number of hands and output the decision of the algorithm as to whether or not to declare dash call.
* Trump Suit Call – a function will produce a number of hands and output the decision of the algorithm in the format of a Call (number + suit).
* Trick Determination – a function will deal cards to a set of four players and set a randomly chosen suit as the trump suit. Each player will then produce a number of tricks and their potential sacrifices.

## Evaluation

Seeing as the source of the other Estimation games are not freely available online, a comparison between the two would be impractical without the completion of this project as a mobile application. Therefore, a less comprehensive method of evaluation will be used; partial system evaluation. This project contains one feature which is not included in other games: the Dash Call. Another feature which, in my opinionated point of view as an expert Estimation player, is not properly implemented in other games is accurate trump suit calling. The two features will be evaluated.

A survey was designed and distributed to a set of Estimation players. [4] The survey received 18 responses. The purpose of the survey was to determine the level of acceptance of the results produced by the algorithms. The survey is comprised of two sections, 10 questions each. The first section is about the dash call function. Given a hand of cards, the person chooses whether or not they would declare a dash call. The hands used for the survey were generated randomly using the testing functions listed above. All the hands used for the dash call survey were declared by the algorithm as a dash call. The purpose of the survey is to test the acceptance of the dash call function.

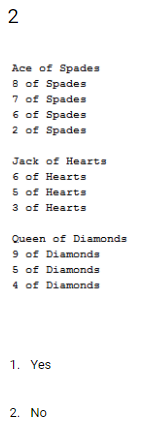
The second section of the survey addressed the trump suit call function. Given a hand of cards, the person chooses whether or not the Call provided by the algorithm is optimal. If not, the person chooses whether they think the call is too high or too low.

Figure 9 - Dash Call Survey Sample Question

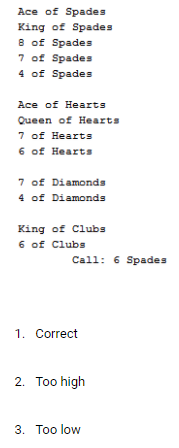


Figure 10 - Trump Suit Call Survey Sample Question

* The results for the first section showed a 59% acceptance rate, with a total of 106 votes from a potential 180 choosing yes.
* The results for the second section showed a 73% acceptance rate, with a total of 131 votes from a potential 180 choosing yes. There was a total of 49 non-yes answers. 84% (41 from 49) of those believed that the call given was too high. The remaining 16% (8 from 49) non-yes answers were that the call given was too low.

# Conclusions and Future Work

## Summary

Estimation is a popular Arabic card game. However, the fact that it is region-specific poses a problem in finding resources for it. An alternative solution to that problem is to find similar games, such as Whist, and using it as a guide to help create an Artificial Intelligence component. The main artificial intelligence concept used in this project is the concept of expert systems. Expert systems are comprised of a knowledge base, an inference engine and a shell interface. This project provides all three. However, one of the things that this project does well is using the concepts of object oriented programming, such as abstraction and polymorphism. This project takes a broad view of the problem of a challenging Estimation computer player.

## Future Work

This project aims to serve as the ground work for an Estimation application. Thus far, it only relies on the concept of expert systems for the artificial intelligence component. However, in the future it may incorporate more advanced concepts, such as heuristic analysis and multi-threading. I believe that this project has promise as it combines two things that I am passionate about.

References

1. Mina Zaklama, *Estimation Calculator* <https://play.google.com/store/apps/details?id=com.zaklama.calculateestimation>
2. Fanella Productions, *Pocket Estimation* <https://play.google.com/store/apps/details?id=com.fanellapro.pocketestimation.android>
3. Nikos Tsaousis, Whist is a trick-taking, real-time multiplayer card game  
   <https://github.com/tsanikgr/whist>, December 2015
4. <https://goo.gl/forms/f1R3GmqlQ6a0qvi33>

# Appendix 1

Figure 11 - Round deal Sequence Diagram

# Appendix 2

Figure 12 - Round secondRoundBids Sequence Diagrams

# Appendix 3

Figure 13 - Round startBidding Sequence Diagram

# Appendix 4

Figure 14 - User secondRoundBidding Sequence Diagram

# Appendix 5

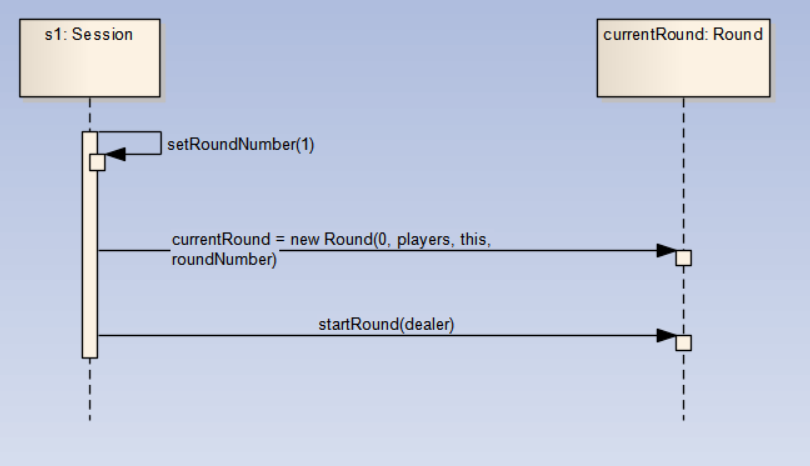


Figure 15 - Session startSession Sequence Diagram

# Appendix 6

Figure 16 - Session nextRound Sequence Diagram

# Appendix 7 – Installation Guide

The project comes with two files; a .bat file and a .jar file. It requires a Java version 1.8 or higher to be installed. To execute it, run the “**RunEstimation.bat**” file. Both of the files “**RunEstimation.bat**” and “**Estimation.jar**” must be in the same directory. This project can also be run on NetBeans and opened for the source code.