Kalaha in Java:
An experiment in Artificial Intelligence and Networking

Bachelor Project in Computer Science
IT Department of Roskilde University, Denmark
Students: Stephan O’ Bryan & Christos Karavasileiadis
Supervisor: Keld Helsgaun
Date: 16-12-2009
# Table of Contents

1. Abstract .......................................................................................................................... 4

2. Terminology ..................................................................................................................... 5

3. Introduction ..................................................................................................................... 7

4. Problem Analysis ............................................................................................................. 8
   4.1. Specification Requirement ......................................................................................... 8
      4.1.1. Functional Requirements ................................................................................... 8
      4.1.1.a) Single Player ................................................................................................. 9
      4.1.1.b) Multi Player ................................................................................................. 10
   4.2. Game Strategy ......................................................................................................... 11
      4.2.1. Game Trees ....................................................................................................... 11
      4.2.2. Minimax Algorithm ......................................................................................... 13
      4.2.3. Alpha-Beta Pruning ......................................................................................... 18
   4.3. Java Networking ....................................................................................................... 23
      4.3.1. Server-Client Model in Java ............................................................................... 24
   4.4. Design Considerations ............................................................................................. 27
      4.4.1. Flexibility .......................................................................................................... 27
      4.4.2. Artificial Intelligence and Speed .................................................................... 28
      4.4.3. Networking ....................................................................................................... 29
   4.5. The GUI .................................................................................................................... 30

5. Description of the Program ............................................................................................ 31
   5.1. Single Player General Classes ................................................................................. 31
       5.1.1. Class Move ........................................................................................................ 31
       5.1.2. Class GamePosition ......................................................................................... 31
   5.2. Single Player Concrete Classes ............................................................................... 34
       5.2.1. Class KalahaMove ......................................................................................... 35
       5.2.2. Class KalahaPosition ..................................................................................... 35
       5.2.3. Class KalahaGame ......................................................................................... 37
   5.3. Multi Player Server Documentation ........................................................................ 38
       5.3.1. Class Game_Server ......................................................................................... 39
       5.3.2. Class GameController .................................................................................... 40
       5.3.3. Class Table (Interface) .................................................................................. 40
       5.3.5. Server_Main Class ......................................................................................... 42
   5.4. Multi Player Client Documentation .......................................................................... 43
       5.4.1. Class KalahaPositionMultiplayer .................................................................... 44
       5.4.2. Class KalahaGameMultiplayer ...................................................................... 45
   5.5. Graphical User Interface Documentation ............................................................... 47
       5.5.1. Class KalahaGameGUI .................................................................................. 47

6. User Guide ....................................................................................................................... 49
   6.1. Kalaha Rules ............................................................................................................. 49
   6.2. User Guide- The Client .......................................................................................... 50
   6.3. User Guide- The Server .......................................................................................... 51

7. Analysis of the Performance ......................................................................................... 52
   7.1. Search Algorithm ..................................................................................................... 52
   7.2. Testing the Networking Aspects ............................................................................. 53

8. Conclusion ...................................................................................................................... 54
   8.1. Possible Improvements ............................................................................................ 55
   8.3. Conclusion Remark ................................................................................................. 55
9. Bibliography ......................................................................................................................... 56

10. Appendix- Source Code .................................................................................................... 57
1. Abstract

This report is based on an experiment in developing a Kalaha game application in Java where it explores the areas of Artificial Intelligence in two-player games with perfect information, and networking. In AI aspect, the computer player of the program whom a user can choose to play against finds intelligent moves by using minimax algorithm with alpha-beta pruning. In networking aspect, the game is able to play against another human player via a network. Hence, theoretical parts that are applied in implementing the program such as game trees, minimax algorithm, alpha-beta pruning and client-server model are discussed. The design of the program is based on object-oriented programming and there are classes that can be extended for implementing any two-player game. As the conclusion, over all result of the implementation is satisfactory, since the program meets the requirements that have been set.
2. Terminology

This session includes explanations of the terminologies we use in this report.

- **Kalaha**: Kalaha is a two-player board game with perfect information. See the rules at session 6.1.

- **Perfect information**: This means that both players always have a complete overview of the opponent's possible moves, and that nothing shall be settled by coincidences [1, p.19].

- **Board numbering**: Sometimes, there is a reference to the hole-numbers of the Kalaha board. This numbering is not standard and it is according to the implementation of our program. The board on our implementation has the following numbering:

  (5) (4) (3) (2) (1) (0)

  (6)           (13)

  (7) (8) (9) (10) (11) (12)

- **Position**: A position is a game position or game status or game state.

- **Single player**: This is the part of our program, where the computer player serves as the opponent.

- **Multi player**: This is the part of our program, where another player on a network serves as the opponent.

- There is a simple protocol for the communication of the clients and the server. Here is an overview of it:

  - START messages: START1 or START2, are sent from the server to notify the clients their positions on the board and the number of their turns.

  - EXIT messages: EXIT is sent to a player from the server when his/her opponent has exited the game without the game being finished.

  - (Boolean, integer) message: A message of this type can take values like (true,5) or (false,11) and so on. The Boolean part notifies the server who gets the next turn; if true, the client who has send this message get the next turn as well. Otherwise, the other client gets the turn. The integer is always forwarded to the opponent client.

- At the implementation session of the report, there are some “blueprints” of the whole implementation of a particular part of the project. Those “blueprints” have been generated using BlueJ (info and link to blue j and the meaning of the arrows is:
- Describes an inheritance relationship between two classes. The source class inherits the target class.
- This arrow describes a “uses” relation meaning that the source creates or uses objects from the targeting class.
3. Introduction

In this project, the group has implemented a Kalaha game application, which a user can either play against the computer or against a human opponent via a network. The programming language that was used in implementing is Java. The goals of the project are separated into two major areas:

1. Artificial Intelligence (AI)
2. Networking

In AI aspect, our goal was firstly to study AI in two-player games, i.e. the computer player of the program always makes intelligent moves by finding the best move (as quick as possible) for a given game position. Therefore, we have implemented a minimax search algorithm with alpha-beta pruning in a general class, which can be extended for implementing any two-player game with perfect information. Then, in order to illustrate the use of this algorithm, we have implemented a Kalaha game application. The implementation was successful since the “intelligent” computer player is able to make the best move within an acceptable second time.

In the networking part, we have attempted to present the reader, how to change the program in such a way that a human-opponent can replace the computer-opponent. First, in order to achieve the communication between two computers, we had to implement a server program that is able to match opponents and start a communication between them. After the server was implemented, we implemented the client program, which is very similar to the single-player program, but with some tweaks in its classes that enable them to communicate over a network. The communication is achieved by using sockets. Java comes with a couple of classes that can be used to achieve socket-based communication. As a result, we have managed to create a Kalaha game that can be played on-line by human players.

Both in single player and multiplayer parts of the game, a very simple GUI has been created to make the program user-friendlier.

The reader needs to have basic understanding of Java code in order to read and understand the implementation part. The reader may also come across some terminologies that are not explained in this report (such as algorithm or classes etc), which we assume that the reader is familiar with these terms. Furthermore, this report is constructed in a way that the basic theories that were used for achieving the solutions are explained at the beginning. Then, come the explanations of the implementation along with some testing results. There is also a user guide session that can be read on its own for getting information of how to use the program. We have made a clear distinction between the single-player part and the multiplayer part, so that the reader is able to skip one of these parts if he/she has no need for it.

As the conclusion, we can state that over all result of this project is satisfactory. And this report could be helpful for those who would like to experiment with Artificial Intelligence in two-player games or are interested in creating distributed applications or just want to explore the possibilities of Java programming.
4. Problem Analysis

In this section, we will state the requirements we set for our program and explain the theories that are involved in implementation of the program. Furthermore, we will discuss what considerations we have had in connection with the design of our program.

4.1. Specification Requirement

In this session, we will explain the objectives and sub-objectives we have for our program, both in terms of its structure and implementation of this structure. These requirements will be used in the conclusion to assess whether our demands are met.

4.1.1. Functional Requirements

As one of our main purposes is to study AI in any two-player board game with perfect information [1], not only Kalaha, our first goal is to understand and design a program which can be used for developing any two-player board game. From there, we would like to develop a specific game application, namely– Kalaha. The Kalaha board game we are about to develop is akin to a real world physical Kalaha game where two players sitting on opposite sides of the board. However, we will provide the user with two choices— a) Single player where computer player serves as the opponent b) Multiplayer where another human player via a network serves as the opponent. In both cases, we consider the following in our program:

- **Flexibility**  
- **Robustness and Quality**

**Flexibility**

Firstly, we want our program to be platform independent, i.e. the program should be able to run on any operating system. Secondly, in order to provide an opportunity to extend the program for any two-player game with perfect information, it is important that the program apply not only to the game we implement. Therefore, we need to take this into account in our design of the program.

**Robustness and User-Friendliness**

We want a satisfactory level of robustness in the program, considering both the code and the user interface. In the code, this especially means that our handling of exceptions should not influence the use of the program. Moreover, our code requires taking into account of errors, when someone extends our program and develops a new game. In the user interface, robustness consists of two things—the validity of the user’s input check before it is forwarded to the program and the user is given the opportunity to provide a new input if the input is invalid. In addition, our program should be user-friendly, i.e. easy to use and give informative messages to the user when it is necessary.
4.1.1.a) Single Player

In this session, we will spell out the requirements we consider especially for the part of the program where the computer serves as the opponent player.

In addition to the requirements mentioned above, this part of the program should fulfill the following demand:

Artificial Intelligence and Speed

One of our sub-objectives is to achieve a satisfactory performance of our program. We want the computer to play well, i.e. we do not want our computer player to merely choose an arbitrary move, but make intelligent moves instead, just like a relatively good human player. In other words, we want the computer player to think before it makes a move, i.e. it should evaluate the consequences of next possible moves, estimate the opponent’s next move/s and make the best move. In addition, the computer player should make the best move within a reasonable period of time.
4.1.1.b) Multi Player

Considering the networking part in implementation of the program, we will present some specific requirements for the server and the client.

The Server

Connection Issues
It is important that a server a) distinguishes whether a connection with a client is lost and b) informs the other client about the matter, if the connection is lost. Additionally, this has to happen without crashing the server or mixing up the clients by sending messages to the wrong clients.

Multiple Clients Handling with the Smallest Use of Resources
It is important for the server to be able to accept all incoming connections and be able to handle them in parallel. As a game server, it should be able to handle all the games in parallel as well. Regarding the use of resources, the more the server knows about one game, the more resources it uses. Our purpose is to create a server that knows almost nothing about a game, and therefore it would “run” very fast.

Client Matching
We would like the server to be able to match clients together in a game, so the user puts as less effort as possible to find an opponent.

The Client:

Connection and Connection issues
As an additional requirement to the single player version of the game, we would like the client to be able to connect to a server and then play the game online with a human opponent. Of course this can lead to some new problems, such as the opponent’s connection to the server is lost. In such a case, we want to be sure that the user get informed about this matter with an accurate message (meaning that an appropriate message is generated from the client program when it knows about the lost connection).
4.2. Game Strategy

In this section we will explain theoretical parts such as game trees, minimax search algorithm and alpha-beta pruning at a level where they will be involved in implementing the single player part of our program.

4.2.1. Game Trees

Firstly, we would like to introduce the concept of trees, so that it is useful for the reader to follow the concept of "game trees", that are applied in search algorithms such as minimax and alpha-beta pruning.

The simplest type of trees is a binary tree. Each node of a binary tree has up to 2 children nodes. Each of these children can also have up to 2 children and so forth. In figure 1, the top node, namely node 1 is the root node and the tip nodes (or) the leaf nodes are the nodes 8, 9 and 10. A tree with depth two can have maximum 7 nodes, a tree with depth three can have maximum 15 nodes and so on. The structure of a binary tree will thus contain a number of branches, where each node can branch out to 2 branches. In contrast to binary trees, many games trees can contain multiple nodes in each branch. In fact, a game tree can contain as many nodes as they are required—depends on the type of the game and the game position/status.

---

1 The root node of a tree is the node with no parents. There is at most one root node in a rooted tree.
2 A leaf node of a tree is the node with no children.
3 The depth of a node is the length of the path from the root to the node. The set of all nodes at a given depth is sometimes called a level of the tree. The root node is at depth zero. [WIKI]
In tic-tac-toe, there are 9 nodes for the first move, and each of these nodes will contain following 8 children and so on. A tic-tac-toe game tree could look like the following:

![Tic-tac-toe game tree](image)

*Figure 2. A tic-tac-toe game tree [2]*

In Kalaha, each node can contain up to 6 children. It is because there are maximum 6 possible moves that a player can make each turn and hence there are maximum 6 possible game positions resulting from the moves. However, the following children nodes do not necessarily need to contain fewer branches as tic-tac-toe.
4.2.2. Minimax Algorithm

A minimax algorithm is a recursive algorithm for choosing the next move in an n-player game, usually a two-player game [3]. Each position/status of the game is evaluated by using a position evaluation function and the resulted value indicates how good or bad it would be for a player to reach that position. Assume that the game’s participants are two players, namely– Min and Max. Min will make the move that minimizes the maximum possible value and Max will make the move that maximizes the minimum value of the position, resulting from the opponent's possible following moves. If it is a player’s turn to move, that player gives a value to each of his legal moves.

If we depict this strategy in a game tree, each node will represent a (game) position and the children of a given node will represent all of the possible subsequent positions that the parent node has. Therefore, if the parent node is a position occurred by a Max’s move, then each of its children nodes are the possible subsequent positions that could occur by a Min’s move and vice versa. A position evaluation function can assign a value to each node and this value serves as the basic for an assessment of the best successor to the current node. The depth of the tree can be formed depending on the game level and requirements such as how many positions that should be looked ahead and searching time etc.

If a parent node belonged to Max and he were to choose among tip nodes that belonged to Min, he would choose the node with the largest value. Thus, the parent node is assigned a backed-up value, which is the maximum of the evaluations of the tip nodes. On the other hand, if a parent node belonged to Min and he were to choose among tip nodes that belonged to Max, he would choose the node with the smallest value. Thus, the parent node is assigned a backed-up value, which is the minimum of the evaluations of the tip nodes. After the parents of all tip nodes have been assigned backed-up values, the backing up value procedure goes to another level or depth of the tree, assuming that Max would choose the node with the largest backed-up value while Min would choose the node with the smallest backed-up value.

The backing up value procedure continues level by level until the successors of the start node are assigned backed-up values. If it is Max’s turn to move at the start, then the successor node having the largest backed-up value should be his choice as the first move.
Minimax in Tic-Tac-Toe

As tic-tac-toe is much simpler than Kalaha, an example of tic-tac-toe is presented here in order to illustrate how the minimax algorithm works. Suppose that Max marks crosses (X) and Min marks circles (O) and it is Max’s turn to move first. We shall do a breadth-search first and after all of the nodes at desire depth are generated, we can start evaluating each node by using a static evaluation function. Let our evaluation function $e(p)$ of a position $p$:

“If $p$ is a winning position for Max:

$$e(p) = \infty \ (\infty \text{ denotes for a very large positive number})$$

If $p$ is a winning position for Min:

$$e(p) = -\infty \ (-\infty \text{ denotes for a very large positive number})$$

If $p$ is not a winning position:

$$e(p) = (\text{number of complete rows, columns, or diagonals that are still open for Max}) - (\text{number of complete rows, columns, or diagonals that are still open for Min})$$ [4]

Thus, if $p$ is Fig. 3(a), then $e(p) = 5 - 4 = 1$. Note that, all of (a), (b), (c) and (d) are considered to be identical, so that we can employ symmetries in generating successor positions.

---

4 Breadth-first search starts at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal [10].
In figure 5, the position akin to Fig. 4(a) is chosen as the best and first move, since it has the largest backed-up value.

Now, let us suppose that Min replies with Fig. 4(b). Then it yields the search tree shown in Fig. 6. Max makes the best move indicated in Fig. 6 and Min replies with the move that avoids his immediate defeat, yielding Fig. 4(c).
Figure 6. Minimax applied to tic-tac-toe (stage 2) [4]. This figure shows the tree generated by a search to depth two. Static evaluations are shown below the tip nodes and backed-up values are circled.

Then, **Max** searches for the best move again and it yields the tree shown in Fig. 7.

Figure 7. Minimax applied to tic-tac-toe (stage 3) [4]. This figure shows the tree generated by a search to depth two. Static evaluations are shown below the tip nodes and backed-up values are circled.
Some of the backed-up values in Fig. 7 (for instance, the one marked 4) are \(-\infty\), because they represent “wins” for Min. After all of the backed-up values are assigned, Max’s best move is also the one that avoids his immediate defeat.

Minimax algorithm can be written as follow:

```
minimax(player,board)
if(game over in current board position)
    return winner
children = all legal moves for player from this board
if(max's turn)
    return maximal score of calling minimax on all the children
else (min's turn)
    return minimal score of calling minimax on all the children
```

[5]
4.2.3. Alpha-Beta Pruning

Minimax algorithm is not so efficient, namely because it needs to search all of the tip nodes of the tree. Hence, alpha-beta pruning was born.

An Example of Alpha-Beta Pruning with Bags

“This is how it goes: You (your name is Max, what else) and your friend (Min) has a number of bags filled with dollar bills. You will get to pick a bag and then Min will pick one bill from the bag and give it to you. Knowing Min is a greedy bastard you know he will give you the least valuable bill in the bag.

If you only knew about the min-max algorithm you would check every bill in all the bags, and pick the bag, which had the most valuable lowest bill. Now try the alpha-beta way instead.

You start with looking through the first bag. There you find a 100-dollar bill and a 20-dollar bill. Knowing Min will give you the 20-dollar bill if you pick this bag that is what this bag is worth.

You now start looking in the second bag. First you find a 100-dollar bill, and since it is better than the 20-dollar bill in the first bag, the second bag is now the best so far. You then find a 50-dollar bill, it is worse than the 100-dollar bill, but still better then the 20-dollar bill from the first, so you still want to pick the second bag, which is now worth 50 dollars.

So far it works exactly like min-max. Now you move on to the third bag, and you pick up the first bill, which is a 10-dollar bill. Since this bill is worse than the 50-dollar bill from the second bag, you can discard the rest of the contents of the bag without looking at it. No matter what more you find in it, it will not matter since you will get the 10-dollar bill. There could be a 1000-dollar bill in there but you will not get it since Min will give you the 10. There could also be a 1-dollar bill in there, but that does not matter either since you have already discarded the bag as worse than the second bag.

So your final choice will be the second bag, and you saved the hassle of looking at all the bills in the third bag.” [6]
Alpha-Beta in Tic-Tac-Toe

Here again, as tic-tac-toe is much simpler than Kalaha, an example of tic-tac-toe is presented here in order to illustrate how the alpha-beta pruning works. Alpha-beta pruning reduces the amount of search needed, because it performs tip-node evaluations and calculates backed-up values simultaneously with tree generation.

We shall do a depth-first search\(^5\). In Fig. 7, if we suppose every tip node is evaluated as soon as it is generated, then after the node \(A\) is evaluated, there is no need to generate and evaluate nodes \(B, C\) and \(D\). Because \textbf{Min} already has the winning node, namely \(A\) and he would not prefer not winning to \(A\). Thus, we can immediately assign ‘\(-\infty\)’ as \(A\)’s parent node backed-up value and continue the search.

\[\text{Fig. 8. Part of the first stage tic-tac-toe tree [4]}\]

---

\(^5\) Depth-first search starts at the root and explores as far as possible along each branch before backtracking. [10]
backed-up value of the start node may be greater than this PBV, depending on the backed-up values of the other successors of the start node, but cannot be less (because if it is less, we would rather choose the value −1).

Now, suppose the depth-first search proceeds and node B and node C are generated. Since node C has the static value of −1, node B can now be assigned the PBV of −1. At this point, we immediately know that we do not need to proceed searching below node B. Since the value of node B is determined by its successor nodes, the final backed-up value of node B can be less than −1, depending on the static values of the rest of node B’s successors, but it cannot be greater.

According to these constraints, we can make the following rules:

“a) Search can be discontinued below any of Max’s node having a PBV less than or equal to the PBV of any of its ancestors Min’s nodes (including the start node). This Max’s node can then be assigned its PBV as a final backed-up value.

b) Search can be discontinued below any of Min’s node having a PBV greater than or equal to the PBV of any of its ancestors Max’s nodes. This Min’s node can then be assigned its PBV as a final backed-up value.

During search, PBVs are computed as follows:

1. The PBV of a Min’s node (including the start node) is set equal to the currently largest of the final backed-up values of its successors.

2. The PBV of a Max’s node is set equal to the currently smallest of the final backed-up values of its successors.

The PBV of Min’s nodes are usually called alpha values, and the PBV of Max’s nodes are usually called beta values. When search discontinued under rule (a) above, we say that alpha cutoff has occurred, and when search discontinued under rule (b), we say that beta cutoff has occurred.” [6]

The whole process of keeping track of PBVs and making cutoffs when possible is usually called alpha-beta pruning. Alpha-beta pruning always finds the same best move as minimax would do, but alpha-beta finds the best move after usually much less search.
Minimax algorithm with alpha-beta pruning can be written as follow:

```
"alpha-beta(player,board,alpha,beta)
  if(game over in current board position)
    return winner
  children = all legal moves for player from this board
  if(max's turn)
    for each child
      score = alpha-beta(other player,child,alpha,beta)
      if score > alpha then alpha = score (we have found a better best move)
      if alpha >= beta then return alpha (cut off)
    return alpha (this is our best move)
  else (min's turn)
    for each child
      score = alpha-beta(other player,child,alpha,beta)
      if score < beta then beta = score (opponent has found a better worse move)
      if alpha >= beta then return beta (cut off)
    return beta (this is the opponent's best move)"
```

[5]

**The Search Efficiency of Alpha-Beta Pruning**

Since PBVs are based on the static values of the tip nodes, we need to generate at least some part of the search tree to maximum depth in order to perform cutoffs.

The static value of one of the tip nodes becomes the final backed-up value of the start node. If this tip node is reached in a very first depth-first search, then the number of cutoffs will be maximal. If the number of cutoffs is maximal, then we only need to generate and evaluate a minimal number of tip nodes.
Fig. 9. An example illustrating the alpha-beta pruning [4]. This figure shows a search tree with a depth of six and the static value are shown below the tip nodes. Darkened branches indicate the sub tree, which indicates the alpha-beta pruning. The cutoff nodes have ‘X’ drawn through them. We needed to evaluate only 18 of the original 41 tip nodes in order to find the best move.

If a tree has depth $D$, and every node (except tip nodes) has exactly $B$ successors, then this tree will have exactly $B^D$ tip nodes. If an alpha-beta pruning generates successors in the order of their true backed-up values— the highest valued successors first for Max’s nodes and the lowest valued successors first for Min’s nodes, then this order will maximize the number of cutoffs and consequently minimizes the number of tip nodes. If we denote this minimal number of tip nodes by $N_D$, then

$$N_D = \begin{cases} 2B^{D/2} & \text{for even } D \\ B^{(D+1)/2} + B^{(D-1)/2} & \text{for even } D \end{cases}$$

Hence, if the same the number of tip nodes are generated, the optimal alpha-beta search can reach to depth $D$, but minimax alone can only reach to depth $D/2$. It implies that alpha-beta search with perfect successor ordering double the search depth, for the same storage availability.
4.3. Java Networking

In this session, we will present some useful information about the technologies in Java that were used to achieve our goal of implementing a networking version of the game. We will also attempt to give some brief information considering the client-server model.

The most common network architecture is the Client-Server model. The Server-Client model consists of a server that provides some services, and clients that connect to the server and demand these services. A simple example of the use of this model is the Internet surfing; the web-browser is the client, and it connects to a server by typing its web-address and gets its contents, most likely an html file that is translated from the browser to what we perceive as a web page.

The server program usually runs on a remote machine, to which clients can connect and start asking for services. On this machine many server programs can run simultaneously but each one of them is bind to a unique port. Therefore, to connect to the right server program, it is essential to know the port that it “listens” to. The server or the clients can be either “thin” or “fat”. A “fat” server or client means that the program will know most of the services required and will use more resources than it would use if it were a “thin” program. For example, having a “fat” server that knows all about the Kalaha-game makes the clients able to run faster on the users’ machines, but this might cause the “fat” server to have performance issues if many clients connect and start playing on this server. On the other hand, having a “fat” client will reduce the performance on the user’s computer, but the server will be able to send data to multiple users faster.
4.3.1. Server-Client Model in Java

There are two ways to create client-server distributed applications in Java: using sockets or RMI (stands for Remote Method Invocation).

Sockets are objects that use the TCP/IP protocol to establish a connection with another socket (usually located on a server) and they are able to read and write streams to this socket. These streams are usually referred as messages. The way socket-based applications usually work is:

1. The client sends a message to the server.
2. The server processes the message and forwards it (or creates another stream based on the given information) to the correct client.

With this exchange of messages is how the server works with sockets. This way works very good (performance wise) but it can be a bit too slow for some cases. One of these cases can be if the server should handle real-time applications as a shooting game. Then it is pretty obvious that this “mail alike” system would not do the job correctly.

RMI can be a solution to a problem similar to the one mentioned above. RMI enables clients to invoke methods on the server, just as they had the program on their machine. For reasons discussed in the following paragraph, we will not use RMI for our solution and therefore we will not go in more details into the theory behind it. Readers are proposed to read [8] for further information.

In our case, we chose to use sockets to implement our client-server model. The reasons are:

**Complexity**

Using RMI can be rather complicated in the way program is designed. Though using sockets, we are able to present an easy-to-understand and useful solution that is able to fulfill all of our requirements.

**Turn based game**

Our game is a turn based game, which means the exchange of messages will not have a big impact on the game play as it would have been if the game had to be played in “real-time”. Using sockets, we can achieve this goal and avoid performance issues.

**Performance**

A client-server application that uses RMI requires that objects are “alive” on the remote machine, namely the server (fat server). This means that the performance of the server will be decreased, as every game will be played on the server so the server needs to allocate memory for every single one. On the other hand, using sockets can make the server only responsible for sending messages between the clients, which means that the memory requirements are very little (compared to RMI). On the client side, the performance will be reduced, but still in a much more acceptable level, since the user’s machine is only responsible for the current application.
4.3.2 Sockets in Java

A socket is connection between two machines that perform a series of operations. According to Elliot Rusty Harold in [8] a socket can perform seven basic operations:

“1. Connect to a remote machine
2. Send data
3. Receive data
4. Close connection
5. Bind to a port
6. Listen for incoming data
7. Accept connections from remote machines on the bound port.”

There are two kinds of socket classes in Java, the ServerSocket class and the Socket class. The following session will be a brief overview of these two classes and the way they are used in our solution. The reader should refer to [8] or the Java API for additional information.

ServerSocket Sockets and General Guidelines for Creating Server Programs in Java

The ServerSocket class is used to establish a server program on a machine. To construct a ServerSocket object, the port number that the server will listen to has to be provided in the constructor. A ServerSocket object has very limited operations; it can bind to a port and accept connections from remote machines on the bound port [8]. Therefore, the server program should create a Socket class object for every new client that it is connected, so that it is able to communicate with them. This is done by using the accept() method of the ServerSocket class and return its outcome in a Socket object (e.g. Socket example = server.accept(), where server is the ServerSocket object). Though, it is important for the server to be able to handle multiple clients in the same time, otherwise the communication is just point-point with one client (meaning the communication is only between these two). The way to do so is by establishing a new thread for each new connected client. In this thread, it is important to provide the socket and assign stream readers and writers to it, so that it can handle messages. Readers can read more about threads and streams in [8]. It is important as well to create a protocol that the server and the clients will “speak”, so there is an understanding of how to use the sent messages. Last thing that has to be mentioned is the life cycle of the server (or the ServerSocket object). The server runs in an infinite loop and usually has to be stopped by force-quitting it. The typical way that a server program works inside this loop during its life cycle is:

1. Wait for a client to connect
2. Assign the client a Socket object
3. Create a thread that can handle a client separately using the Socket made in 2
4. Back to 1

Of course, some variations can be made such as to wait for 2 clients and supply their sockets in one thread and so on. The idea is to make the server program to be ready to accept a new connection as soon as possible, instead of spending time providing services to one client at the time.
Socket Class in Java

Sockets are objects that are used for the communication between two remote hosts. A Socket can do basically what a ServerSocket cannot do, with emphasis on the reading/writing of data. Socket objects are created both on the server and on the client. The most common way to create a Socket object is by supplying in its constructor the name or ip address of the computer that it should connect to and the port that the connection should be attempted. In the case of Sockets on a server, the ServerSocket object’s accept() method takes care of the connection. For the Socket to start reading and writing data, stream readers and writers (like BufferedReader and PrintWriter) have to be created. This can done by plugging the getInputStream() and getOutputStream() of the Socket class into the readers and writers. For example:

BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));
PrintWriter out = new PrintWriter(new OutputStreamWriter(s.getOutputStream()));

Where s is the Socket object. Then, using the methods from the stream class, the read/write operation can be achieved.

Once the Socket is ready to read and write, the communication can begin. It is important as well to close the socket once we are done using it in order to save resources. The Socket object is closed by using the close() method of the socket class. Another essential thing for the program that uses a Socket object is to be able to catch exceptions for connectivity issues, input & output issues and other. Usually the Socket objects throw IOExceptions.
4.4. Design Considerations

In this session, we will discuss our design consideration related to the implementation of our program.

4.4.1. Flexibility

Firstly, in order to provide the platform independency, we used Java as the programming language for implementing our program.

Secondly, since we wanted to provide an opportunity to extend the program for any two-player game with perfect information and considering object-oriented programming, we will design our program with some general classes that could be used for extending into any specific game, and with some concrete classes that make a user friendly Kalaha game application.

Therefore, the general classes are:

1. Abstract class GamePosition (Objects of this type represents any two-player game position).
2. Abstract class Move (Objects of this type represents a move in any two-player game).

And the concrete classes are:

1. Class KalahaPosition (Objects of this type represents a position in a single player Kalaha game).
2. Class KalahaMove (Objects of this type represents a move in a Kalaha game).
3. Class KalahaGame, which starts a single player Kalaha game and handle the interaction with the user in the game)
4. Class KalahaPositionMultiplayer (Objects of this type represents position in a multi player Kalaha game)
5. Class KalahaGameMultiplayer, which starts a multi player Kalaha game and handle the interaction with the user in the game)
4.4.2. Artificial Intelligence and Speed

Considering AI and speed, we have implemented a method that uses minimax algorithm with alpha-beta pruning to find the best move for the computer player in GamePosition class. In this way, this method can find the best move for any two-player game. Because this method needs all of the next possible moves to find the best move, we have a method that generates next possible moves in this class as well. However, as the ‘moves’ will be different depending on the game, the ‘nextPossibleMoves’ method is abstract and it has to be overwritten in the concrete game class. Furthermore, this method returns a list with all of the next possible moves. We also needed a ‘makeMove’ method to make a move in this class, as the search algorithm needs to make a move before it evaluates a position. And again, as the ‘moves’ will be different depending on the game, this method is abstract and it has to be overwritten in the concrete game class. Furthermore, this method returns a list with all of the next possible moves. We also needed a ‘makeMove’ method to make a move in this class, as the search algorithm needs to make a move before it evaluates a position. And again, as the ‘moves’ will be different depending on the game, this method is abstract and it has to be overwritten in the concrete game class. Then as we need to evaluate a given position, we needed a ‘value’ method. This method is also abstract for the same reason as above. In our ‘KalahaPosition’ class, the value is evaluated as the difference in the number of stones in two players’ Kalaha stores, and each stone is given 10 points. So, if the computer player has 2 stones and the human player has 1 stone, then the value will be:

\[ 10 \times (2 - 1) = 10 \] points. We have also put a ‘unclear’ method for allowing the search algorithm to search further in case current position cannot be evaluated. This method is not really useful for a game like Kalaha, but it will be for more complex game like chess.
4.4.3. Networking

Server Design
As mentioned before, one of our purposes was to implement a server where users can connect and play Kalaha game against each other. Therefore, we needed to make some design considerations that could lead to an accurate implementation. Firstly, we need to assure that the server can handle more than one client simultaneously. Using Java threads can achieve this goal. Another thing we had to consider is how the matchmaking should take place; should the users choose their opponents or should it be with no-user effort. We decided to build the server in such a way that the server generates the matchmaking.

We could then construct a very brief successful use-case of how the server could work:

1. The server program starts by binding on a port and waits for clients to connect.
2. When a client is connected, a thread is created on the server to handle the client.
3. When a second client is connected, a second thread for this client starts and the two threads are matched in a game-table where they can communicate under some protocol.
4. Then the server waits for the next two clients to connect and the same operations happen for every pair of clients until the server program stops running.

From the use case, we could already identify some classes that should exist in the implementation: Game_Server class, which starts the server and waits for clients to connect, A GameClientHandler class that is the “communication tool” between the client and the server, and A Game_Table class that contains two matched clients and establishes a communication between them.

Client Design
For the client part, we would like to keep the same structure as the implementation of the single player Kalaha game, but only change the part where the opponent has to move. Instead of using the alpha-beta search method, the program waits for a move to arrive from the server. Off course, the client should be able to connect to the server and read and write to the server. Thankfully, Java’s Socket class makes this very easy. A brief success use case is the following:

1. The user starts the client and chooses the server that wants to connect.
2. The program waits until an opponent is found and the game starts.
3. Once the game has started, the user can choose either to move first or not.
4. If it is the user’s turn, the user enters a move from the GUI.
5. If it is the opponent’s turn, the program waits until a move is sent from the server.
6. When the game is finished, the user is notified the result and the program closes.
4.5. The GUI

Here are some decisions we have made about the GUI implementation:

• At the beginning of the game, three stones are placed in each of the 12 holes of the board, apart from the Kalaha stores.
• In the single player part, the holes that lie at the lower part of the board belong to the user and the holes that lie at the upper part of the board belong to the computer player. Obviously, the left Kalaha store belongs to the computer player and the right Kalaha store belongs to the user.
• In the multiplayer part, the user has the lower holes if he/she is player 1 and has the upper holes, if he/she is player 2.
• The program will provide a message saying whose turn it is according to the Kalaha rules.
• Pressing on the corresponding holes can make the moves as long as they are legal moves.
• In the single player part, the user will be able to choose the level of difficulty of the game and whether he/she or computer should make the first move.
• In the multiplayer part, the user should be able to decide the server that he/she will connect to.

Note that we tried to keep the GUI separate from the game classes, so it can be always changed or refined.
5. Description of the Program

In this session, we will spell out the implementation of our program, including the classes and their most important methods.

5.1. Single Player General Classes

In this session, we will explain about the general classes in single player part of our program.

5.1.1. Class Move

```java
public abstract class Move {
}
```

This is an abstract class. An object of Move type represents a move in any two-player game. Since this is an abstract class, the concrete subclass of this class, i.e. the class that specifies the move (in our case, class KalahaMove), has to implement the constructor method in order to create objects.

5.1.2. Class GamePosition

```java
public abstract class GamePosition{
    protected abstract int value();
    protected abstract void makeMove(Move m);
    protected abstract GamePosition copy();
    protected abstract List<Move> nextPossibleMoves();
    private Move bestMove;
    public Move getBestMove(int depth){…}
    public boolean maxTurn;
    public boolean unclear(){…}
    protected abstract boolean gameOver();
    public int searchAlphaBeta(int alpha, int beta, int depth){…}
}
```
This is also an abstract class. An object of GamePosition type represents a position in any two-player game. Because this class includes mini-max algorithm with alpha-beta pruning, it can be used to develop any two-player game where the computer uses A.I for finding the best move in the game.

value(), makeMove(Move m), copy(), nextPossibleMoves() and gameOver() are abstract methods. Therefore, any concrete subclass that inherits this class needs to overwrite these methods.

value() is the method which evaluates the value of the current position and returns an integer value of the current position. Complexity of this method will depend on the type of game. For instance, in chess, this method will be quite complex as it is difficult to evaluate a given position precise. In our Kalaha program, it is easier as the value is simply the number of stones in each player’s Kalha store. A position with more stone in his/her own Kalaha store has a better value than a position with less stone.

makeMove(Move m) method makes a given move in a game. This method moves an object of type ‘Move’.

copy() method copies the current position and returns the current position. This method is important for searchAlphaBeta method as this method remembers the current position before it actually makes the move.

nextPossibleMoves() method generates next possible moves from current position depending on whose turn and returns a list with all possible moves. Each move in the list is an object of type ‘Move’.

getBestMove(int depth){…} finds the best move from current position using searchAlphaBeta method. searchAlphaBeta method uses given ‘depth’ as the maximum depth it has to search, Integer.MIN_VALUE as initial alpha value and Integer.MAX_VALUE as initial beta value. This method returns the best move, which is found by the searchAlphaBeta method.

maxTurn makes sure whose turn to move. If maxTurn is true, then max's turn to move, otherwise, the other player (min)'s turn.

unclear(){…} method allows the searchAlphaBeta method to search further in case current position cannot be evaluated. If this method actually has to be used, it has to be overwritten at the specific game position class that inherits this class. This method returns true if current position has been evaluated further, otherwise returns false.

gameOver() checks whether the game is over or not and returns true if the game is over, otherwise false. The actual game over status will depend on the type of the game.

searchAlphaBeta(int alpha, int beta, int depth){…} searches the best move for the computer player, so that it can make intelligent moves. ‘alpha’ is hitherto the best value for max, ‘beta’ is hitherto the best value for min and ‘depth’ is the maximum depth that how many position the computer has to think ahead. The deeper the maximum depth is, the better move this method will find.
First, the algorithm uses `nextPossibleMoves()` method to generate all possible moves, then it evaluates each move by using `value()` method and keeps the best move in the `bestMove` instance field. In this method, the method calls itself recursively in order to evaluate current position at each depth. This method returns the value of given position.
5.2. Single Player Concrete Classes

In this session, we will explain about the concrete classes in single player part of our program.

![Diagram of class relationships]

**Figure 10.** This figure shows the relations between the classes used for the implementation of the single player Kalaha game application. This image has been generated with BlueJ (http://www.bluej.org/). The meaning of the arrows is explained in the terminology session.
5.2.1. Class KalahaMove

```java
public class KalahaMove extends Move{
    protected int fromHole;
    public KalahaMove(int fromHole){
        this.fromHole = fromHole}
}
```

This class is the concrete subclass of the Move class. The constructor of this class constructs a KalahaMove object. fromHole is the hole that the move has to start from.

5.2.2. Class KalahaPosition

```java
public class KalahaPosition extends GamePosition implements Clonable{
    protected int board[] = new int[14];
    protected static int STARTSTONES;
    public KalahaPosition(int initialStones){…}
    public KalahaPosition(){
    protected int value(){…}
    protected void makeMove(Move m){…}
    protected GamePosition copy(){…}
    public boolean unclear(){…}
    protected List<Move> nextPossibleMoves(){…}
    protected boolean gameOver(){…}
    public String toString(){…}
}
```
This class is the concrete subclass of the GamePosition class and it implements the Clonable interface, so that a clone copy of the Kalaha board in current Kalaha position is able to make in copy() method.

board[] is an array of length 14. The board represents a Kalaha game-board. Elements 0-5 in the array are the upper holes of the board and elements 7-12 in the array are the lower holes of the board. Elements 6 and 13 in the array are the Kalaha-stores of the upper and lower player, respectively.

KalahaPosition(int initialStones){…} constructs a KalahaPosition object with given initial stones in each hole (apart from the stores) at the beginning of the game.

KalahaPosition(){ } merely constructs an object of type KalahaPosition.

value(){…} overrides the method from the superclass GamePosition. This method evaluates the current position by subtracting the number of stone in hole 13 from hole 6 and multiplying it by 10, i.e. each stone in the Kalaha stores is given 10 points.

makeMove(Move m){…} also overrides the method from the superclass GamePosition. In this method, the given Move m object is casted/converted to KalahaMove object and since each KalahaMove object has the hole number that the move has to start from, the move is made accordingly depending on which player’s turn. The details are described in the source code.

copy(){…} also overrides the method from the superclass GamePosition. This method copies the board of the current Kalaha position.

unclear(){…} also overrides the method from the superclass GamePosition. This method is for optimizing the horizon of maxTurn; if it is maxTurn, the alpha-beta search further, otherwise not. This method returns true if it is maxTurn.

nextPossibleMoves(){…} also overrides the method from the superclass GamePosition. This method is called by the searchAlphaBeta method and it generates all possible moves depending on which player’s turn, put them in a list object and returns the list.

gameOver(){…} also overrides the method from the superclass GamePosition. This method checks either one player's Kalaha store is more than 6 times of the initial stones OR both players have equal number of stones in each player's Kalaha, then the game is over.
5.2.3. Class KalahaGame

```java
public class KalahaGame {
    private static int SEARCH_DEPTH;
    private static final int INITIALSTONES = 3;
    public static boolean isLegalMove(int[] board, int hole){…}
    public KalahaGame() {…}
    private void gamePlay(){…}
    public void inputHoleChoice(String choice) {…}
    public void inputLevel(String choice) {…}
    public void inputStartFirstChoice(String choice) {…}
    public static void main(String[] args) {…}
}
```

This class is the one, which starts the game and handles the interaction with the user via the GUI.

`SEARCH_DEPTH` is the maximum depth the alpha-beta algorithm has to search and `INITIALSTONE` is the initial stones in each hole at the beginning of the game. Since there are three levels of difficulty to choose, `SEARCH_DEPTH` = 1 for Easy, `SEARCH_DEPTH` = 3 for Medium and `SEARCH_DEPTH` = 12 for Hard.

isLegalMove checks whether the move that the user wants to make is legal or not before the program makes the move.

KalahaGame(){…} is the constructor, which constructs a Kalaha game and calls the gamePlay() method to start the game.

gamePlay(){…} method actually handles the game playing by calling other classes and interacts with the user via GUI.

inputHoleChoice(String choice){…}, inputLevel(String choice){…} and inputStartFirstChoice(String choice){…} are the methods for the GUI, so that the GUI can read the user’s input.
5.3. Multi Player Server Documentation

In this session, we will aim to give an accurate overview of the server that can handle the Kalaha game clients that will request on-line game-play. There are 4 classes and 1 interface that have been created in order to fulfill this task; Game_Server.java, GameClientHandler.java, Table.java, Game_Table.java and Server_Main.java.

The following figure is a “blueprint” of the server part showing the classes that are used and the arrows explain their relation. The “blueprint” was generated with BlueJ.

![Diagram of server classes and their relations]

*Figure 11. This figure shows the relations between the classes used for the implementation of the server. This image has been generated with BlueJ (http://www.bluej.org/). The meaning of the arrows is explained in*
5.3.1. Class Game_Server

```java
public class Game_Server {
    public Game_Server(int tcpport) throws IOException{…}
    public void startServer() throws IOException{…}
}
```

When an object of this class is constructed, a new ServerSocket is initialized to listen to a specified port. Then the startServer() method is called.

startServer() is a method that runs an infinite while loop. Inside the while loop, the server object waits for a client to connect. Once a client is connected, a new socket on the server is created by calling the accept() function of the ServerSocket class. It is important to clarify that the ServerSocket object is only capable of waiting for a client to connect. It cannot achieve any communication with the client and therefore it is vital to create a socket object that is able to read and write streams. Another important matter that a server has to deal with is how to treat each client. In order to treat multiple clients, every new socket has to be placed in a thread. Otherwise, the server can only communicate with a single client. To do this in our case, every new socket is place in a GameClientHandler object (for explanation of the GameClientHandler class look further in the report). The precise way that the startServer() works is that it waits until two clients are connected, initializes a new socket for each client and then places each of them into a GameClientHandler object. Afterwards, these two new GameClientHandler objects are matched into a Game_Table thread, where the actual rules of communication between these two clients is handled. Once two GameClientHandler objects are matched, the loop starts from the top again and the previous steps are repeated until the JVM (Java Virtual Machine) is forced to quit.
5.3.2. Class GameClientHandler

```java
public class GameClientHandler extends Thread {
    public GameClientHandler(Socket soc, String id) {...}
    public String getID() {...}
    public synchronized void sendMessage(String msg) {...}
    public void run() {...}
}
```

This class handles each connected client as a separate thread. To construct an object of this class, a socket object and a string for identification of the thread have to be provided to the constructor. In the constructor, a BufferedReader object and a PrintWriter object are assigned to the thread, so it can read and write socket streams.

getID() method returns the id of the GameClientHandler object as a string.

sendMessage(String msg) method is used to send a message to the client that is connected to this thread (more precisely, to the socket that it is supplied in this thread). This method is synchronized in case that multiple messages are sent in the same time to this client.

run() method has to be implemented, because the GameClientHandler class extends the Thread class. Notice though that the thread never starts.

5.3.3. Class Table (Interface)

```java
public interface Table {
    public void gamePlay();
}
```

The main idea behind this interface is to create an abstraction that describes the way a game is played on a server side. Basically, this abstraction decides how the messages should go between two clients depending on the rules of the game. For example, in chess, a player can only move once and never gets an extra turn, while in Kalaha, there is a case that a player gets one or more consecutive extra turn. This means that the turns that message have to be sent is different from game to game and therefore, it is important to describe this abstraction with the Table class.
5.3.4. Class Game_Table

```
public class Game_Table extends Thread implements Table{
    public Game_Table(GameClientHandler player1, GameClientHandler player2)
        throws IOException{
        ...
    }

    public void gamePlay() {
        ...
    }

    public void run(){
        ...
    }
}
```

In the Game_Table class, the whole “message handling” between two clients is achieved. This class is also the one (on the side of the server) that knows how to handle some special forms of messages (meaning the protocol of the network) that come from the clients. Starting from the constructor, a START1 or START2 message is sent to each client. This is how the server decides who is player one and who is player two (START1 for player one and START2 for player two). On the client side, depending on which START message is got, the user gets the appropriate holes. The reason to send these messages at the constructor is to ensure synchronization between the two clients. If we assume that a client got the message before another client had logged in, then this client will be ready to play as player one with nobody on the other side. But with the use of this technique, we ensure that the clients are hanging on until they receive a START-message. Notice that the objects of the Game_Table class are threads, since the server will have to deal with many of them concurrently. At the end of the constructor, the start() method of the Thread class is called.

gamePlay() is the method where the “exchange” of messages between the two clients is done. There are two if statements that deal with to whom the message should be deliver to. The arguments that these if statements check is an integer called turn, which decides who should talk. If turn is 1 then the server waits for a message to come from player one and sends it to player two. In case this message is null, then it means that the connection with the client is corrupted and the server notifies the other client by sending a message EXIT. When a client receives EXIT, it knows that the other client is left. Otherwise the message consists of two parts, a Boolean variable and an integer. The Boolean variable lets the server know if the next turn is still this player’s turn, and the integer, is the move of the player and it is send forward to the opponent. In case turn is 2, the same idea applies but then player 2 is sending to player 1.

run() of this thread calls the gamePlay() method in order to make the communication start.
This class is the main class that starts the server. It contains nothing more than the main method. The only interesting thing that has to be mentioned in this class is that it has been extended as a JFrame and the reason is that the server runs for ever, and the only way to quit it is to force-quit it. Therefore, making it as a JFrame give us the possibility to EXIT_ON_CLOSE.
5.4. Multi Player Client Documentation

The client consists of 2 classes KalahaGameSimulatorMultiPlayer.java and KalahaPositionMultiplayer.java, but it uses the classes from the single-player part as well. The whole client is considered to come as one Kalaha game that a user can either play against the computer or against another user on the Internet. About the multi player classes, they are similar to KalahaGame.java and KalahaPosition.java but with some extra tweaks, so they can deal with networking.

Here is an overview of he relations between the classes that deal with the multiplayer aspect:

![Diagram showing the relations between classes](Figure 12). This figure shows the relations between the classes used for the implementation of the client that can play the Kalaha game online. This image has been generated with BlueJ (http://www.bluej.org/). The meaning of the arrows is explained in the terminology session.
5.4.1. Class KalahaPositionMultiplayer

```java
public class KalahaPositionMultiplayer extends GamePosition implements Comparable {
    protected int board[] = new int[14];
    protected static int STARTSTONES;
    protected int your_number;

    public KalahaPositionMultiplayer(int initialStones, int player_number) {...}

    public boolean isLegalMove(int fromHole) {...}

    protected void makeMove(Move m) {...}

    protected boolean gameOver() {...}

    public String toString() {...}

    protected List<Move> nextPossibleMoves() {...}

    protected GamePosition copy() {...}

    protected int value() {...}
}
```

For this class, we will not go deep into the description of methods, since the way they work is almost identical to the methods of the KalahaPosition.java. Starting from the constructor, an extra argument is supplied as an integer called player_number. It is this argument that decides which holes belong to a user. If player_number is 1, then the user gets holes 0 to 6, and if it is 2, then the user gets holes 7 to 12. The player_number is used as an if statement argument in the makeMove(Move m) method.

makeMove(Move m) in this class has a major tweak; for the two different values of players_number, different holes are assigned to the user. The opponent’s turn in this case is always described with maxTurn equals to true, similarly to the single player version of the game. In order to show the difference, we need to refer to some code from inside this method:

```java
if (your_number == 1) {
    if (endHole == 13 && !maxTurn) this.maxTurn = !this.maxTurn;
    this.maxTurn = !this.maxTurn;
} else if (your_number == 2)
```
This part of the method checks whether one of the players (user’s turn is described with maxTurn equals to false) has ended on his Kalaha store, which means he gets an extra turn. So, in case the user has player_number equals to 1, it means that he gets holes 7 to 13 that means if the user ends at 13, an extra turn will be granted. Though, it is a different case if the user has player_number equal to 2. In this case, the user will take an extra turn if the last stone is placed at hole number 6. Because it is really important to make this distinction between this two cases, these if arguments have been added to this method. The rest part of the method works the same way as described in the single player, but only with this extra tweak.

5.4.2. Class KalahaGameMultiplayer

```java
public class KalahaGameMultiplayer {
    public KalahaGameMultiplayer() throws UnknownHostException, IOException {

    }

    public void gamePlay() throws UnknownHostException, IOException {
        ...
    }

    public void inputHoleChoice(String choice) {
        ...
    }

    public void inputServerIP(String ip) {
        ...
    }

    public static void main(String[] args) throws UnknownHostException, IOException {
        ...
    }
}
```

In this class is where the whole game is established. The main class constructs a KalahaGameMultiplayer object and the constructor call the gamePlay() method.

gamePlay() method is basically the running game itself. This method involves in different steps. First the GUI is constructed and the program waits until a string with the server’s ip address is provided. After that, the program waits until the server sends a START message (which is send when an opponent is found), so the game can begin. If the message is START1, then this user is player 1, maxTurn is set to false (meaning it is this user’s turn), the user gets holes on the board 7-12 and starts first in the game. If the message is START2, then this user is player 2, maxTurn is set to true (meaning it is the opponent’s turn), the user gets holes 0 to 5 and plays second in the game. Then, there are two blocks of code which depend on whose turn it is. If maxTurn is false then is user’s turn to choose a
hole. After a hole is chosen, then the program checks whether it is a valid choice or not. If not, then the user is asked to make a right choice. When the choice is correct, a move is made and it is send to the opponent. The message consists of two parts (separate by a comma) one Boolean (maxTurn is true OR false) and one integer. The server uses the Boolean to understand whose turn next (note that the server is thin so it knows nothing of the game) and the integer is sent to the opponent. If the next move is the opponent’s move, then the program waits until a message with a move is sent from the server. The message should be an integer which is plugged into the makeMove() method of the KalahaPositionMultiplayer and the move for the opponent (maxTurn equals to true) is made. There is also a case that an EXIT message is sent, which means that the opponent has exited the game and therefore the game is finished at this point. These blocks described above are inside a while loop, which runs as long as the game is not over (meaning that, the game is over due to the rules and not of connection problems). When the game is over, a message with the winner is send and the program closes.

inputHoleChoice(String choice) is used for overwriting the string (name in this program as input) that is used for choosing a hole to move from. This method is called from the GUI whenever someone clicks on the appropriate button.

inputServerIP(String ip) is used for overwriting the string (name in this program as server_ip) that is used providing a socket with the right ip address of the server that it should connect to.

main(String[] args) is the main class that starts the program.
5.5. Graphical User Interface Documentation

The GUI of this program consists of one class called KalahaGameGUI.java. We will not go into the theory of how to create user interfaces in java, so readers can refer to [9] for more information of how the different GUI class in the Java libraries work.

5.5.1. Class KalahaGameGUI

```java
public class KalahaGameGUI extends JFrame{
    public KalahaGameGUI( final KalahaGame kalaha){…}
    public KalahaGameGUI( final KalahaGameSimulatorMultiPlayer kalaha){…}

    private void initComponents(){…}
    private String getStonesFromBoard( int hole_number){…}
    public void addTextToButtons(){…}
    public void refresh(){…}
    public void openLevelDialog(){…}
    public void openChoiceDialog(){…}
    public void closeChoiceDialog(){…}

    class ButtonClick implements MouseListener{…}
}
```

There are two constructors inside the KalahaGameGUI class: One for creating a user interface for the single player and one for the multiplayer. The single player uses more buttons for choosing difficulty level or choosing whether the user or the computer will start first. Inside both constructors the initComponents() method is called.

initComponents() is where the GUI is constructed. Buttons and a text area are added in a JFrame and they get their coordinates on the window. Afterwards each button gets a MouseListener so if they get clicked, different actions can performed.

getStonesFromBoard(int hole_number) is a method from getting the number of stones of a particular hole on the Kalaha board as a string.

addTextToButtons() adds on every button the amount of stones that are on the hole. This is done by calling getStonesFromBoard(int hole_number) for each button (notice that the buttons here, represent the holes of a Kalaha board game).

refresh() basically calls the addTextToButtons() in order to refresh the amount of stones that are in every hole of the board after each turn.
openLevelDialog() reveals the three buttons that make the user able to choose the difficulty level of the single player Kalaha game.

openChoiceDialog() revels the two buttons that make the user able to choose whether him/her or the computer will start first on the single player Kalaha game.

closeChoiceDialog() is closing (making invisible) the buttons for choosing the difficulty level or the “who is playing first” choice.

The inner class is implementing the MouseListener interface, so it can be used to add actions on button clicks. We will not go more into the theory of adding MouseListeners to a GUI. The reader can refer to [9] for additional information.
6. User Guide

This is the user guide for helping the users running the Kalaha game. The players can play it against the computer or against other human opponent over the Internet. Both ways are going to be explained. Also, how to run a server that supports this program is going to be explained. Note that Java should be installed on the users machine in order to run the program.

6.1. Kalaha Rules

Here are the rules of the game:

• Positioning: 7 holes are assigned to each player: 6 holes to move from and one hole to store stones.
• Moving: In of the 12 move-holes (6 for every player) there is an amount of stones. In order to move, the player needs to chose one of his/her move-holes that has stones in it, empties that hole, and moving anti-clock-wise, leave one stone to the other holes until the hand is empty. If the player goes by his store-hole, he/she leaves a stone. If the player goes by the opponents store-hole, then he/she does not leave a stone and continues to the next hole.
• Second Turn: If the last stone in a player’s hand lands in the store hole, then this player gets an extra turn.
• Opposite hole: If the player places the last stone in an empty hole on his/her side and the hole on the opponents side has stones inside, then both that last stone and stones in the opposite hole are captured and placed into the player’s store.
• Winning conditions: The player with more stones in his/her store hole wins. Game is terminated when one of the players’ move-holes have no stones. Then the opponent takes the stone that he has in his/her side, place them into the store hole, and a comparison of the amount of the stones in each store hole should be done.
• As a special winning condition, the game can finish when a player gets more than half of the existing stones.
• Draw: If the players have the same amount of stones in their store holes at the end of the game, then the game is a draw.
6.2. User Guide- The Client

Single Player:
1. To start the program, the user should run the KalahaGame class.
2. In the first screen that appears, the user should choose the difficulty level of the game by clicking one of the three buttons.

![Difficulty Level Buttons]

3. Then the user should choose, by clicking on of two choice buttons that became available, whether him/her should have the first turn or the computer.

![First Turn Buttons]

4. After that, the game starts and the user is assigned to have the lower blue buttons (move-holes) and the computer gets the upper red buttons.

![Game Board]

5. The game goes on according to the Kalaha rules (CHECK KALAH RULES SESSION) until it is over and a winner is announced.
6. For restarting the game, the KalahaGame class should be run again.

Multi Player:
1. To start the program, the user should run the KalahaGameMultiplayer class.
2. Then the user is asked to provide the server’s ip address in order to connect to it and get ready to play. The ip address should be written inside the text field that is available on the GUI and then press the connect button. (Note that the server listens on port 9000)

![Server Connection]

3. When the user is connected and an opponent is matched, the game starts. If the user is player 1, he/she gets the lower blue buttons and has the first turn. If the user is
player 2, then he/she gets the upper red buttons and plays second.

4. The game goes on according to the Kalaha rules (CHECK KALAYHA RULES SESSION) until it is over and a winner is announced.
5. For restarting the game, the KalahaGameMultiplayer class should be run again.

6.3. User Guide- The Server

1. To start the server, the Server_Main class has to be run.
2. The server listens on port 9000. If the user wants to make the server to listen on a different port, the argument at Server_Main(9000) has to be changed to the desired port.
3. To stop the server, the user has to force quit it.
7. Analysis of the Performance

In this session, we will discuss a brief examination of the performance of our program.

7.1. Search Algorithm

In this session, we analyze the search algorithm in our program considering the game tree in aspects of minimax algorithm and alpha-beta pruning.

The following evaluations were conducted on iMac7.1 Intel Core 2 Duo 2.4 GHz with 2GB DDR2 SDRAM. The evaluations are recorded, after the computer player’s first move.

<table>
<thead>
<tr>
<th>Max.Depth</th>
<th>Nodes by minimax</th>
<th>T(s) for minimax</th>
<th>Nodes by alpha-beta</th>
<th>T(s) for alpha-beta</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>0.0010</td>
<td>11</td>
<td>0.0010</td>
</tr>
<tr>
<td>3</td>
<td>776</td>
<td>0.043</td>
<td>356</td>
<td>0.0050</td>
</tr>
<tr>
<td>12</td>
<td>902,109,053</td>
<td>618.236</td>
<td>6,421,680</td>
<td>4.378</td>
</tr>
</tbody>
</table>

*Table 1. The number of nodes created by minimax alone and with alpha-beta pruning*

Observing the table above, we can clearly see that search algorithm with alpha-beta pruning is much more efficient also for our program, especially for the deeper search. Note that the actual searching time is much shorter if the program does not need to count and print out the number of nodes.
7.2. Testing the Networking Aspects

In this session, we will test the networking aspects of our implementation, namely, the different messages that should be passed around through the server and the clients. The following is a white box test using expected results. Several tests have been done to ensure the correctness of our implementations. This is the final test, with one player quitting in the middle of the game:

<table>
<thead>
<tr>
<th>Action</th>
<th>Server Expected received message</th>
<th>Server Actual received message</th>
<th>Client 1 Expected received message</th>
<th>Client 1 received message</th>
<th>Client 2 Expected received message</th>
<th>Client 2 received message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Players connect</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Game Starts</td>
<td>-</td>
<td>-</td>
<td>START1</td>
<td>START1</td>
<td>START2</td>
<td>START2</td>
</tr>
<tr>
<td>Player 1 plays from hole 10</td>
<td>True,10</td>
<td>true,10</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Player 1 plays from hole 12</td>
<td>False,12</td>
<td>false,12</td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Player 2 plays from hole 3</td>
<td>True,3</td>
<td>true,3</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Player 2 plays from hole 4</td>
<td>True,2</td>
<td>true,2</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Player 2 plays from hole 5</td>
<td>False,5</td>
<td>false,5</td>
<td>5</td>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Player 1 plays from hole 11</td>
<td>False,11</td>
<td>false,11</td>
<td>-</td>
<td>-</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Player 2 quits</td>
<td>null</td>
<td>null</td>
<td>EXIT</td>
<td>EXIT</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2. The result of a message exchange between the server and two connected clients during an online game session

As a result of this table, we can see that the generated messages have been delivered as expected.
8. Conclusion

Firstly, we have accomplished implementing a playable Kalaha game with two options, namely, (a) computer as the opponent (b) another human player as the opponent via a network. Next, we would like to make a retrospective evaluation on our project. Hence, we will review the claims we made for the program in the requirement session.

Flexibility

We can say that we have made a satisfactory implementation for this purpose. The program is platform independent, and the general classes give rise to implementing any two-player game.

Robustness and User-Friendliness

Our code is consistent and the GUI is also relatively user-friendly and gives informative messages, when necessary.

Artificial Intelligence and Speed

At every difficulty level we provide in the program, our computer player is able to make intelligent moves depending on how many moves ahead, it is allowed to estimate. The search algorithm is also adequate, so that the computer player can make a move rather fast, considering our measurements in Table 1.

Server:

Connection Issues

Inside the Game_Table class we can see that the server can handle cases when a connection with a client is lost, the server notifies the other client about the problem so cases like one user waiting for a missing opponent to make a move are avoided. Furthermore, since we handle the connected clients and the games as threads, we ensure that a crash on those objects will not cause the server to crash and stop.

Multiple Clients Handling with the Smallest Use of Resources

The server, with the use of threads, is able to handle multiple clients simultaneously. We are ensured that there is no “queue” of clients waiting to get service. Also making the server “thin” we use the minimum of the server-machines resources. Basically, the server side is very “light”, since the only action that is taking place is an exchange of string-messages.

Client Matching

Clients are pair from the server, which makes the users less responsible for finding a game. When the user is connected, is basically ready-to-play.
Client:

**Connection and Connection Issues**

The user needs only to supply the server’s ip address when it is asked in the GUI and then the connection is achieved. In case the connection is broken or the opponent has quitted the game, the user gets the appropriate notifications from the GUI and the exceptions are handled from the program.

**8.1. Possible Improvements**

In this session, we will discuss about the things we consider, in order to make improvements in our program.

**Sorting**

Since alpha-beta pruning can maximize the number of cutoffs in a search, if we sort the next possible moves in the order of their true backed-up value, then the search will be faster.

**Evaluating the Position**

If we could consult a Kalaha expert, we could have improved the evaluation function.

**Smarter Server**

The implemented server can be used for the purposes of the report but in a real life situation, it might be too simple. The problem is the behavior of users on the Internet is random and sometimes it is unacceptable. For example, a user may have “the bad habit” of leaving games without the game being over. It would have been ideal for the server to have some type of a smart ip address recognition and database support to store data about users action. Also the particular implementation of the server has no security precautions and basically, it is easy open door for unwanted attacks.

**GUI**

The GUI has a major flaw. The update of the stone after a move is also instant, meaning that the human eye cannot catch the change. A slower update, with a nice animation, would have made the GUI much more user-friendly. This flaw is very noticeable when playing against the AI that makes a move in matter of ms.

**8.3. Conclusion Remark**

We can conclude that over all result of this project is satisfactory, in spite of the fact that things can always be improved. We have also learned a lot by working on this report, especially on AI in two-player games, networking and Java. Therefore, some of the questions we had in our mind before this project are answered. We feel that developments in Computer Science and AI are quite exciting and unforeseeable.
9. Bibliography


[2] [http://www.akira.ruc.dk/~keld/teaching/DAT_C_e01/Slides/07_Anvendelser_I.ppt](http://www.akira.ruc.dk/~keld/teaching/DAT_C_e01/Slides/07_Anvendelser_I.ppt)


[5] [http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html](http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html)


10. Appendix- Source Code

This session includes the source codes from our program. Please see from the next page.
import java.io.BufferedReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class Game_Server {
    GameClientHandler h1, h2;

    private BufferedReader in1, in2;
    private PrintWriter out1, out2;
    ServerSocket server;

    public Game_Server(int tcpport) throws IOException {
        //Create a ServerSocket object that listens to port 9000 and waits for clients to connect
        server = new ServerSocket(tcpport);
        System.out.println("Server Starts at port: " + tcpport);
        startServer();
    }

    public void startServer() throws IOException {
        while (true) {
            //Once two clients are connected, two threads start and are added to a Game_Table where the communication between them is starting
            //After these two clients are matched, the next two will get matched and so on...
            Socket s1 = server.accept();
            Socket s2 = server.accept();
            h1 = new GameClientHandler(s1, "Player1");
            h2 = new GameClientHandler(s2, "Player2");
            Game_Table newGame = new Game_Table(h1, h2);
        }
    }

} //end of class
import java.io.BufferedReader;
import java.io.IOException;
import java.io.PrintWriter;

public class Game_Table extends Thread implements Table {
    /** *
     * instance fields
     */
    GameClientHandler player1, player2;
    protected BufferedReader in1, in2;
    protected PrintWriter out1, out2;
    int turn, who_is_left;

    /** *
     * this class is intended to control messages sent between
     * the two players.
     * NOTE the this class is constructed to work together
     * with the client (Kalaha game) that was done in this project.
     * other clients may find problems with the protocols and
     * the way it works.
     */
    public Game_Table(GameClientHandler player1, GameClientHandler player2)
        throws IOException {
        this.player1 = player1;
        this.player2 = player2;

        String start1 = "START1";
        String start2 = "START2";
        player1.sendMessage(start1);
        player2.sendMessage(start2);
        start();
    }

    public void gamePlay() {
        turn = 1;
        try {
            while (true) {
                //Messages from player1 to player2
                //The received string is store into recieved.
                if it is null, then it means that the player1 is left, so player2
                has to be notified
                //that the game is over. Therefore EXIT is
                //send. Notice that the messages sent from the clients are control
                //from the gui and therefore
                //the only reason to send a null message is
connection problems.

    //if it is not null, it has the form
    (boolean, integer). The boolean describes if turn should stay 1
    which means the next turn
    //is still of this player, and if false, the
    turn = 2. The integer is the hole number that the was used for the
    player move in the Kalaha.

    if (turn == 1) {
        String received = player1.in.readLine();
        if (received == null) {
            System.out.println(received);
            player2.sendMessage("EXIT");
            System.out.println("Sending EXIT
message to player 2");
            break;
        } else {
            System.out.println(received);
            String[] msg1 = received.split("","", 2);
            player2.sendMessage(msg1[1]);
            if (msg1[0].trim().equals("true")) {
                turn = 1;
            } else turn = 2;
        }
    }

    //Messages from player2 to player1
    //same idea as above but from player 2 to
    player 1

    else if (turn == 2) {
        String received = player2.in.readLine();
        if (received == null) {
            System.out.println(received);
            player1.sendMessage("EXIT");
            System.out.println("Sending EXIT
message to player 1");
            break;
        }
        System.out.println(received);
        String[] msg2 = received.split("","", 2);
        player1.sendMessage(msg2[1]);
        if (msg2[0].trim().equals("true")) {
            turn = 2;
        } else turn = 1;
    }

    } catch (IOException e) {
        System.out.println("The Game Was Terminating
because of connection problems");
        player1.sendMessage("EXIT");
player2.sendMessage("EXIT");

public void run()
{
    gamePlay();
}
public class GameClientHandler extends Thread {

  protected Socket s;
  public String id;
  public BufferedReader in;
  private PrintWriter out;

  public GameClientHandler(Socket soc, String id) {
    this.id = id;
    this.s = soc;
    if (s != null) {
      try {
        in = new BufferedReader(new InputStreamReader(s.getInputStream()));
        out = new PrintWriter(new OutputStreamWriter(s.getOutputStream()));
      }
      catch (Exception e) {
        e.printStackTrace();
      }
    }
  }

  public String getID() {
    return id;
  }

  public synchronized void sendMessage(String msg) {
    if (out != null) {
      out.println(msg);
      out.flush();
    }
  }
}
/**
 * run method for this thread
 */

public void run() {
    try {
        while (true) {
            String message = in.readLine();
        }
    }
    catch (IOException e){}
}
import java.util.List;

/**
 * A general class. An object of GamePosition type represents a
game position in any two-player game.
 * Because this class includes mini-max algorithm with alpha-
beta pruning,
it can be used to develop any two-player game where the
computer uses AI for finding the best move in the game.
 *
 * @author Stephan O'Bryan, December 2009.
 */

public abstract class GamePosition {

   /**
    * Evaluates the value of the current game position/status.
    *
    * @return Returns an integer value of the current game
    * position/status.
    */
   protected abstract int value();

   /**
    * Makes a given move in a game.
    *
    * @param m the move to be made.
    */
   protected abstract void makeMove(Move m);

   /**
    * Copy the current game position/status.
    * This method is important for searchAlphaBeta method as
    * this method remembers
    * the current game position/status before it actually
    * makes the move.
    *
    * @return Returns the current game position/status.
    */
   protected abstract GamePosition copy();

   /**
    * Generates next possible moves from current game
    * position/status depending on whose turn.
    *
    * @return Returns a list with all possible moves from
    * current game position/status.
    */
   protected abstract List<Move> nextPossibleMoves();
public Move bestMove;

/**
 * Finds the best move from current game position/status using alpha-beta algorithm.
 *
 * @param depth the maximum depth alpha-beta algorithm has to search.
 *
 * @return Returns the best move, which is found by the alpha-beta.
 */

public Move getBestMove(int depth) {
    bestMove = null;
    searchAlphaBeta(Integer.MIN_VALUE, Integer.MAX_VALUE, depth);
    return bestMove;
}

/**
 * Makes sure whose turn to move. If maxTurn is true, then max's turn to move,
 * otherwise, the other player (min)'s turn.
 */

public boolean maxTurn;

/**
 * This method allows the alpha-beta algorithm to search further in case current game position/status cannot be evaluated.
 * If this method actually has to be used, it has to be overwritten at the specific game position class that inherits this class.
 *
 * @return Returns true if current game position/status has be evaluated further, otherwise returns false.
 */

public boolean unclear() {
    return false;
}

/**
 * Checks whether the game is over or not.
 *
 * @return Returns true if the game is over, otherwise false.
 */

protected abstract boolean gameOver();

/**
 * Search-algorithm which searches the best move for the computer so that it can make intelligent moves.

public int searchAlphaBeta(int alpha, int beta, int depth)
{
    List<Move> npmList = nextPossibleMoves();
    if ((depth <= 0 && !unclear()) || npmList == null || npmList.isEmpty()) {
        return value();
    }
    for (Move gm : npmList) {
        GamePosition gp = (GamePosition) copy();
        gp.makeMove(gm);
        int value = gp.searchAlphaBeta(alpha, beta, depth - 1); //Recursive calling.
        if (maxTurn && value > alpha) {
            alpha = value;
            if (alpha >= beta) {
                return alpha;
            }
            bestMove = gm;
        } else if (!maxTurn && value < beta) {
            beta = value;
            if (alpha >= beta) {
                return beta;
            }
            bestMove = gm;
        }
    }
    return maxTurn ? alpha : beta;
}
import java.io.PrintWriter;

/**
 * The class which starts a Kalaha game.
 * @author Stephan O'Bryan, December 2009.
 */
public class KalahaGame {
    /**
     * The maximum depth that alpha-beta has to search.
     */
    private static int SEARCH_DEPTH;

    /**
     * Initial stones to be put in each hole at the beginning of the game.
     */
    private static final int INITIALSTONES = 3;

    /**
     * Strings that can be overwritten by the GUI.
     */
    public String input, level, choice;
    KalahaPosition currentPosition;
    PrintWriter writer;

    /*
     * Checks whether the human player's move is legal or not.
     *
     * @param board the board to be checked.
     * @param hole the hole number to be checked.
     * @return Returns true if it is a legal move.
     */
    public static boolean isLegalMove(int[] board, int hole) {
        return hole >= 7 && hole <= 12 && board[hole] != 0;
    }

    /*
     * The Constructor
     */
    public KalahaGame() {
        this.gamePlay();
    }

    /*
     * Method for playing the game.
     */
    private void gamePlay() {
        currentPosition = new KalahaPosition(INITIALSTONES);
        KalahaGameGUI gui = new KalahaGameGUI(this);
gui.addTextToButtons();

int inputLevel;
gui.openLevelDialog();
level = null;
gui.jTextArea1.append("Choose the level you want to play from the buttons above !\n");
while (level == null) { //Thread for waiting until the user press the button.
    try {
        Thread.sleep(100);
    }
    catch (final InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
switch ((inputLevel = Integer.parseInt(level.trim())))
{
    case 1:
        SEARCH_DEPTH = 1;
        gui.closeChoiceDialog();
        break;
    case 2:
        SEARCH_DEPTH = 3;
        gui.closeChoiceDialog();
        break;
    case 3:
        SEARCH_DEPTH = 12;
        gui.closeChoiceDialog();
        break;
}
gui.jTextArea1.append("Choose if you want to start:
YES or NO?\n");
gui.openChoiceDialog();
choice = null;
while (choice == null) { //Thread for waiting until the user press the button.
    try {
        Thread.sleep(100);
    }
    catch (final InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
currentPosition.maxTurn = !"y".equalsIgnoreCase(choice.trim());
gui.closeChoiceDialog();
gui.jTextArea1.append("\n ^^ THE GAME HAS STARTED
^^ \n");
while (!currentPosition.gameOver()) {
    //Human Turn
    if (!currentPosition.maxTurn)
        //          if (false && !currentPosition.maxTurn) //Use this instead if you want to play Computer Vs. Computer.
    {
        int hole = 0;
        do {
            gui.jTextArea1.append("YOUR TURN
");
            input = null;
            while (input == null) { //Thread for waiting until the user press the button.
                try {
                    Thread.sleep(100);
                } catch (final InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            hole = Integer.parseInt(input);
            if (!isLegalMove(currentPosition.board, hole))
                gui.jTextArea1.append("Illegal move: It was not your hole OR the hole is empty. Try again!\n");
        } while (!isLegalMove(currentPosition.board, hole));
        gui.jTextArea1.append("YOUR TURN!! \n");
        currentPosition.makeMove(new KalahaMove(hole));
    }
    //ComputerTurn
    else {
        gui.jTextArea1.append("My Turn !!! Thinking...
\n");
        final double time1 = System.currentTimeMillis ();
        currentPosition.makeMove (currentPosition.getBestMove (KalahaGame.SEARCH_DEPTH));
        final double time2 = System.currentTimeMillis ();
        System.out.println("SEARCHING TIME: " + (time2 - time1) / 1000 + " seconds");
    }
}
    gui.refresh();
    gui.jTextArea1.append("!!!GAME OVER!!!
");
        gui.jTextArea1.append("You won. Congrats
");
        gui.jTextArea1.append("I won :-) Nobody beats the
");
    else
        gui.jTextArea1.append("DRAW. A rematch?
");

/**
 * This method is used to change the value of the of the
String input.
 * This way, the user can enter the hole him/her wants to
play from the GUI.
 */
    public void inputHoleChoice(String choice) {
        this.input = choice;
    }

/**
 * Similar to the previous method, but for the String
level, which is used to choose the level of difficulty.
 */
    public void inputLevel(String choice) {
        this.level = choice;
    }

/**
 * Similar to the previous two methods, but for choosing
who will start first: Computer or Human ?
 */
    public void inputStartFirstChoice(String choice) {
        this.choice = choice;
    }

/**
 * Main method which starts the program.
 */
    public void main(String[] args) {
        /*
         *   */
    }
public static void main(String[] args) {
    KalahaGame game = new KalahaGame();
}
import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class KalahaGameGUI extends JFrame {
    public KalahaGameGUI(final KalahaGame kalaha) {
        isSinglePlayer = true;
        game = kalaha;
        initComponents();
    }

    public KalahaGameGUI(final KalahaGameMultiplayer kalaha) {
        isSinglePlayer = false;
        game2 = kalaha;
        initComponents();
    }

    // in this method the layout of the window is established
    // by adding and placing buttons, text area, text field etc
    // also mouse listeners are assigned to the buttons
    private void initComponents() {
        jButton1 = new javax.swing.JButton();
        jButton2 = new javax.swing.JButton();
        jButton3 = new javax.swing.JButton();
        jButton4 = new javax.swing.JButton();
        jButton5 = new javax.swing.JButton();
        jButton6 = new javax.swing.JButton();
        jButton7 = new javax.swing.JButton();
        jButton8 = new javax.swing.JButton();
        jButton9 = new javax.swing.JButton();
        jButton10 = new javax.swing.JButton();
        jButton11 = new javax.swing.JButton();
        jButton12 = new javax.swing.JButton();
        jButton13 = new javax.swing.JButton();
        jButton14 = new javax.swing.JButton();
        jButton15 = new javax.swing.JButton();
        jButton16 = new javax.swing.JButton();
        jButton17 = new javax.swing.JButton();
        jButton18 = new javax.swing.JButton();
        jButton19 = new javax.swing.JButton();
        jButton20 = new javax.swing.JButton();
        jScrollPane1 = new javax.swing.JScrollPane();
        jTextArea1 = new javax.swing.JTextArea();
        input = new JTextField();
        input2 = new JTextField();

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
    }
}
jTextArea1.setColumns(20);
jTextArea1.setRows(5);
jsScrollPanel1.setViewportView(jTextArea1);

Container pane = getContentPane();
BoxLayout layout = new BoxLayout(pane, BoxLayout.LINE_AXIS);
pane.setLayout(new BorderLayout());
pane.setSize(950, 700);

jButton1.setLocation(850, 50);
jButton1.setSize(100, 50);
jButton1.setForeground(Color.red);
pane.add(jButton1);

jButton2.setLocation(700, 50);
jButton2.setSize(100, 50);
jButton2.setForeground(Color.red);
pane.add(jButton2);

jButton3.setLocation(550, 50);
jButton3.setSize(100, 50);
jButton3.setForeground(Color.red);
pane.add(jButton3);

jButton4.setLocation(400, 50);
jButton4.setSize(100, 50);
jButton4.setForeground(Color.red);
pane.add(jButton4);

jButton5.setLocation(250, 50);
jButton5.setSize(100, 50);
jButton5.setForeground(Color.red);
pane.add(jButton5);

jButton6.setLocation(100, 50);
jButton6.setSize(100, 50);
jButton6.setForeground(Color.red);
pane.add(jButton6);

jButton7.setLocation(25, 120);
jButton7.setSize(100, 50);
pane.add(jButton7);

jButton8.setLocation(100, 200);
jButton8.setSize(100, 50);
jButton8.setForeground(Color.blue);
pane.add(jButton8);
jButton9.setLocation(250, 200);
jButton9.setSize(100, 50);
jButton9.setForeground(Color.blue);
pane.add(jButton9);

jButton10.setLocation(400, 200);
jButton10.setSize(100, 50);
jButton10.setForeground(Color.blue);
pane.add(jButton10);

jButton11.setLocation(550, 200);
jButton11.setSize(100, 50);
jButton11.setForeground(Color.blue);
pane.add(jButton11);

jButton12.setLocation(700, 200);
jButton12.setSize(100, 50);
jButton12.setForeground(Color.blue);
pane.add(jButton12);

jButton13.setLocation(850, 200);
jButton13.setSize(100, 50);
jButton13.setForeground(Color.blue);
pane.add(jButton13);

jButton14.setLocation(925, 120);
jButton14.setSize(100, 50);
pane.add(jButton14);

jButton15.setLocation(325, 300);
jButton15.setSize(100, 50);
pane.add(jButton15);

jButton16.setLocation(450, 300);
jButton16.setSize(100, 50);
pane.add(jButton16);

jButton17.setLocation(575, 300);
jButton17.setSize(100, 50);
pane.add(jButton17);

jButton18.setLocation(325, 300);
jButton18.setSize(100, 50);
pane.add(jButton18);

jButton19.setLocation(575, 300);
jButton19.setSize(100, 50);
pane.add(jButton19);
jButton20.setLocation(650, 345);
jButton20.setSize(100, 50);
pane.add(jButton20);
pane.add(new JScrollPane(jTextArea1), BorderLayout.SOUTH);
input.setLocation(350, 350);
input.setSize(300, 25);
pane.add(input);
input2.setLocation(350, 350);
input2.setSize(300, 25);
in克服Visible(false);
pane.add(input2);
in克服Visible(false);
jButton15.setVisible(false);
jButton16.setVisible(false);
jButton17.setVisible(false);
jButton18.setVisible(false);
jButton19.setVisible(false);
jButton20.setVisible(false);
closeChoiceDialog();
setVisible(true);
setTitle("Kalaha Game");
this.setMinimumSize(new Dimension(1100, 610));

jButton1.addMouseListener(new ButtonClick() {
    public void mouseClicked(MouseEvent e) {
        if (isSinglePlayer == true) {
            game.inputHoleChoice("0");
        } else if (isSinglePlayer == false) {
            game2.inputHoleChoice("0");
        }
    }
});

jButton2.addMouseListener(new ButtonClick() {
    public void mouseClicked(MouseEvent e) {
        if (isSinglePlayer == true) {
            game.inputHoleChoice("1");
        } else if (isSinglePlayer == false) {
            game2.inputHoleChoice("1");
        }
    }
});
jButton3.addMouseListener(new ButtonClick() {
    public void mouseClicked(MouseEvent e) {
        if (isSinglePlayer == true) {
            game.inputHoleChoice("2");
        } else if (isSinglePlayer == false) {
            game2.inputHoleChoice("2");
        }
    }
});

jButton4.addMouseListener(new ButtonClick() {
    public void mouseClicked(MouseEvent e) {
        if (isSinglePlayer == true) {
            game.inputHoleChoice("3");
        } else if (isSinglePlayer == false) {
            game2.inputHoleChoice("3");
        }
    }
});

jButton5.addMouseListener(new ButtonClick() {
    public void mouseClicked(MouseEvent e) {
        if (isSinglePlayer == true) {
            game.inputHoleChoice("4");
        } else if (isSinglePlayer == false) {
            game2.inputHoleChoice("4");
        }
    }
});

jButton6.addMouseListener(new ButtonClick() {
    public void mouseClicked(MouseEvent e) {
        if (isSinglePlayer == true) {
            game.inputHoleChoice("5");
        } else if (isSinglePlayer == false) {
            game2.inputHoleChoice("5");
        }
    }
});

jButton8.addMouseListener(new ButtonClick() {
    public void mouseClicked(MouseEvent e) {
        if (isSinglePlayer == true) {
            game.inputHoleChoice("7");
        } else if (isSinglePlayer == false) {
            game2.inputHoleChoice("7");
        }
    }
});
jButton9.addMouseListener(new ButtonClick() {
    public void mouseClicked(MouseEvent e) {
        if (isSinglePlayer == true) {
            game.inputHoleChoice("8");
        } else if (isSinglePlayer == false) {
            game2.inputHoleChoice("8");
        }
    }
});

jButton10.addMouseListener(new ButtonClick() {
    public void mouseClicked(MouseEvent e) {
        if (isSinglePlayer == true) {
            game.inputHoleChoice("9");
        } else if (isSinglePlayer == false) {
            game2.inputHoleChoice("9");
        }
    }
});

jButton11.addMouseListener(new ButtonClick() {
    public void mouseClicked(MouseEvent e) {
        if (isSinglePlayer == true) {
            game.inputHoleChoice("10");
        } else if (isSinglePlayer == false) {
            game2.inputHoleChoice("10");
        }
    }
});

jButton12.addMouseListener(new ButtonClick() {
    public void mouseClicked(MouseEvent e) {
        if (isSinglePlayer == true) {
            game.inputHoleChoice("11");
        } else if (isSinglePlayer == false) {
            game2.inputHoleChoice("11");
        }
    }
});

jButton13.addMouseListener(new ButtonClick() {
    public void mouseClicked(MouseEvent e) {
        if (isSinglePlayer == true) {
            game.inputHoleChoice("12");
        } else if (isSinglePlayer == false) {
            game2.inputHoleChoice("12");
        }
    }
});
296          });
297          jButton15.addMouseListener(new ButtonClick() {
298                  public void mouseClicked(MouseEvent e) {
299                      game.inputLevel("1");
300                  }
301              });
302          jButton16.addMouseListener(new ButtonClick() {
303                  public void mouseClicked(MouseEvent e) {
304                      game.inputLevel("2");
305                  }
306              });
307          jButton17.addMouseListener(new ButtonClick() {
308                  public void mouseClicked(MouseEvent e) {
309                      game.inputLevel("3");
310                  }
311              });
312          jButton18.addMouseListener(new ButtonClick() {
313                  public void mouseClicked(MouseEvent e) {
314                      game.inputStartFirstChoice("y");
315                  }
316              });
317          jButton19.addMouseListener(new ButtonClick() {
318                  public void mouseClicked(MouseEvent e) {
319                      game.inputStartFirstChoice("n");
320                  }
321              });
322          jButton20.addMouseListener(new ButtonClick() {
323                  public void mouseClicked(MouseEvent e) {
324                      game2.inputServerIP(input.getText());
325                  }
326              });
327          pack();
328      } // </editor-fold>
329
330      /**
331       * it retrieves the stones from a hole on the board.
332       * isSinglePlayer == true means that the game is a KalahaGame.. A
333       * single player.
334       * for the multiplayer game isSinglerPlayer == false
335       * @param hole_number
336       * @return
private String getStonesFromBoard(int hole_number) {
    String stones = null;
    if (isSinglePlayer == true) {
        stones = "" + game.currentPosition.board[hole_number];
    } else if (isSinglePlayer == false) {
        stones = "" + game2.currentPosition.board[hole_number];
    }
    return stones;
}

public void addTextToButtons() {
    jButton1.setText(getStonesFromBoard(0));
    jButton2.setText(getStonesFromBoard(1));
    jButton3.setText(getStonesFromBoard(2));
    jButton4.setText(getStonesFromBoard(3));
    jButton5.setText(getStonesFromBoard(4));
    jButton6.setText(getStonesFromBoard(5));
    jButton7.setText(getStonesFromBoard(6));
    jButton8.setText(getStonesFromBoard(7));
    jButton9.setText(getStonesFromBoard(8));
    jButton10.setText(getStonesFromBoard(9));
    jButton11.setText(getStonesFromBoard(10));
    jButton12.setText(getStonesFromBoard(11));
    jButton13.setText(getStonesFromBoard(12));
    jButton14.setText(getStonesFromBoard(13));
}

/**
 * refreshes the text in the buttons. Used after every
valid move to show the new Position of the board

```java
    public void refresh() {
        addTextToButtons();
    }
```

```java
    /**
     * Shows the buttons for the difficulty choice
     */
    public void openLevelDialog() {
        jButton15.setText("Easy");
        jButton16.setText("Medium");
        jButton17.setText("Hard");
        jButton15.setVisible(true);
        jButton16.setVisible(true);
        jButton17.setVisible(true);
    }

    /**
     * Shows the buttons for the who-will-start first choice
     */
    public void openChoiceDialog() {
        jButton18.setText("YES");
        jButton19.setText("NO");
        jButton18.setVisible(true);
        jButton19.setVisible(true);
    }

    /**
     * Shows the button and text field that are used for
     * inputting the server information
     */
    public void openConnectDialog() {
        input.setVisible(true);
        jButton20.setText("Connect !!!");
        jButton20.setVisible(true);
    }

    /**
     * closes the buttons that are used in the choices and the
     * connection. hides the text field as well
     */
    public void closeChoiceDialog() {
        jButton15.setVisible(false);
        jButton16.setVisible(false);
        jButton17.setVisible(false);
        jButton18.setVisible(false);
        jButton19.setVisible(false);
    }
```
jButton20.setVisible(false);
input.setVisible(false);

private JButton jButton1;
private JButton jButton10;
private JButton jButton11;
private JButton jButton12;
private JButton jButton13;
private JButton jButton14;
private JButton jButton15;
private JButton jButton16;
private JButton jButton17;
private JButton jButton18;
private JButton jButton19;
private JButton jButton2;
private JButton jButton20;
private JButton jButton3;
private JButton jButton4;
private JButton jButton5;
private JButton jButton6;
private JButton jButton7;
private JButton jButton8;
private JButton jButton9;
private JFrame jFrame1;
private JScrollPane jScrollPane1;
public JTextArea jTextArea1;
private JTextField input, input2;
KalahaGame game;
KalahaGameMultiplayer game2;
private boolean isSinglePlayer;

// class for mouse events

class ButtonClick implements MouseListener {
    public void mouseClicked(MouseEvent e) {
    }

    public void mouseEntered(MouseEvent e) {
        // TODO Auto-generated method stub
    }

    public void mouseExited(MouseEvent e) {
        // TODO Auto-generated method stub
    }

    public void mousePressed(MouseEvent e) {
        // TODO Auto-generated method stub
    }
public void mouseReleased(MouseEvent e) {
    // TODO Auto-generated method stub
}
}
import java.io.*;
import java.net.Socket;
import java.net.UnknownHostException;

public class KalahaGameMultiplayer {

    //variable declaration
    public KalahaGameGUI gui;
    public String input, server_ip;
    public KalahaPositionMultiplayer currentPosition;
    BufferedReader socket_in;
    PrintWriter socket_out;
    Socket s;

    //constructor
    public KalahaGameMultiplayer() throws UnknownHostException, IOException, InterruptedException {
        gamePlay();
    }

    public void gamePlay() throws InterruptedException, IOException {
        //creating a GUI with the current object as argument.
        gui = new KalahaGameGUI(this);
        gui.openConnectDialog();

        //the server_ip is set to null and basically the
        program stays here until it is changed. When server_ip gets a value
        //then the program continues. Thread.sleep() is used
        //for making the program to wait.
        server_ip = null;
        gui.jTextArea1.append("Please type the IP_Adress of
        the server and press connect in order to connect and start
        playing !!!\n");

        while (server_ip == null) {
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```
43           gui.closeChoiceDialog();
44
45           // attempts to connect to the server. If not, it shows
46           // an error message and the program quits.
47           try {
48               s = new Socket(server_ip.trim(), 9000);
49               socket_in = new BufferedReader(new
50         InputStreamReader(s.getInputStream()));
51               socket_out = new PrintWriter(new
52         OutputStreamWriter(s.getOutputStream()));
53           } catch (IOException e1) {
54               gui.jTextArea1.append("There is no connection to
55               the desired server. Please try again.");
56               Thread.sleep(3000);
57               System.exit(0);
58           }
59
60           int start_hole = 0;
61           int finish_hole = 0;
62           int your_kalaha = 0;
63           int opponents_kalaha = 0;
64           String turn;
65           int player_number = 0;
66           String message;
67
68           gui.jTextArea1.append("Waiting for the game to start.
69           Please be patient\n69          ");
70           String startgame = socket_in.readLine();
71
72           while (true) {
73
74               // these two if statements check the starting
75               // position of the player according to the type of START message that
76               // the server
77               // will send to the client.
78               //
79               // if (startgame.trim().equals("START1")) {
80                   start_hole = 7;
81                   finish_hole = 12;
82                   your_kalaha = 13;
83                   opponents_kalaha = 6;
84                   currentPosition = new
85                       KalahaPositionMultiplayer(3, 1);
86                   gui.addTextToButtons();
87                   currentPosition.maxTurn = !"y".equalsIgnoreCase("y");
88                   gui.jTextArea1.append("The game will start
```
soon. You have been assigned as player 1

break;

} else if (startgame.trim().equals("START2")) {
    start_hole = 0;
    finish_hole = 5;
    your_kalaha = 6;
    opponents_kalaha = 13;
    currentPosition = new
    KalahaPositionMultiplayer(3, 2);
    currentPosition.maxTurn
        = !"y".equalsIgnoreCase("n");
    gui.addTextToButtons();
    gui.jTextArea1.append("The game will start
soon. You have been assigned as player 2

break;

} while (!currentPosition.gameOver()) {
    //This users Turn. Program waits until input is
    //changed from null and then checks if input is a valid move.
    //if yes, then move is done
    //if no, the user is asked to try again
    if (!currentPosition.maxTurn) {
        BufferedReader console = new BufferedReader
            (new InputStreamReader(System.in));
        int pool = 0;
        do {
            gui.jTextArea1.append("Your Turn, type in
the hole number you want to move

try {
    input = null;
    while (input == null) {
        try { Thread.sleep(50);
            } catch (InterruptedException e) { // TODO Auto-generated catch
block
        e.printStackTrace();
    }
    pool = Integer.parseInt(input);
    if (!currentPosition.isLegalMove
System.out.println("The hole is empty");

if (pool < start_hole || pool > finish_hole)
    gui.jTextArea1.append("Choose hole number" + start_hole + " to " + finish_hole);

    catch (NumberFormatException e) {
        gui.jTextArea1.append("Choose hole number" + start_hole + " to " + finish_hole);
        continue;
    } catch (ArrayIndexOutOfBoundsException e) {
        gui.jTextArea1.append("Choose hole number" + start_hole + " to " + finish_hole);
        continue;
    } while (pool < start_hole || pool > finish_hole || !currentPosition.isLegalMove(pool));

    // if the move is accepted, it makes it on this machine then it makes a message that is send to the server and from there the server sends it further.
    // According to the protocol the message should be a boolean,integer where the boolean indicates if it is again this users turn and the integer the chose move.

    gui.jTextArea1.append("Your Turn \n");
    currentPosition.makeMove(new KalahaMove(pool));

    if (!currentPosition.maxTurn) {
        message = "true" + "," + pool;
    } else message = "false" + "," + pool;

    socket_out.println(message);
    socket_out.flush();

    // Opponents Turn
    else {
        // Waits to recieve a message from opponent. If it is EXIT it means the opponent has left and the program is terminated.
        // Otherwise comes an integer with the opponents move
        gui.jTextArea1.append("Opponent's Turn \n");
try {
    String opponent_move = socket_in.readLine().trim();
    System.out.println(opponent_move);
    if (opponent_move.trim().equals("EXIT")) {
        gui.jTextArea1.append("Your opponent has left the game :((( Good Luck next time. May the Alpha Beta be with You
        
        ");
        Thread.sleep(4000);
        s.close();
        System.exit(0);
    } else {
        int pool2 = Integer.parseInt(opponent_move);
        currentPosition.makeMove(new KalahaMove(pool2));
    }
} catch (IOException e) {
    gui.jTextArea1.append("Your opponent has left the game :((( Good Luck next time. May the Alpha Beta be with You
    
    ");
    Thread.sleep(4000);
    s.close();
    System.exit(0);
}

//refresh the GUI
    gui.refresh();

//shows the result of the game to the user
    gui.jTextArea1.append("Game Over
    
    ");
    if (currentPosition.board[opponents_kalaha] < currentPosition.board[your_kalaha])
        gui.jTextArea1.append("You won :-))
    
    ");
    else if (currentPosition.board[opponents_kalaha] > currentPosition.board[your_kalaha])
        gui.jTextArea1.append("Your opponent has Won !
    
    ");
    else
        gui.jTextArea1.append("DRAW :-) = :-) \n"
    
    ");
    Thread.sleep(4000);
    s.close();
    System.exit(0);
}

//used from the GUI to overwrite the values of input (chosen hole of the user)
    public void inputHoleChoice(String choice) {
        this.input = choice;
// used from the GUI to input the servers ip address

```java
public void inputServerIP(String ip) {
    this.server_ip = ip;
}
```

```java
public static void main(String[] args) throws UnknownHostException, IOException, InterruptedException {
    KalahaGameMultiplayer gs = new KalahaGameMultiplayer();
}
```
** This class represents a move in a Kalaha game.
* @author Stephan O'Bryan, December 2009.

```java
public class KalahaMove extends Move {
    int fromHole;
    
    /**
     * Constructs a KalahaMove object.
     * @param fromHole the hole that the move has to start from.
     */
    public KalahaMove(int fromHole) {
        this.fromHole = fromHole;
    }
    
    /**
     * Method for printing out fromHole object nicely(if needed).
     */
    public String toString() {
        return "" + fromHole;
    }
}
```
import java.util.ArrayList;

import java.util.List;

/**
 * KalahaPosition defines a GamePosition. This class inherits the GamePosition class.
 * A KalahaPosition consists of a kalaha game-board and methods to play a kalaha game.
 *
 * @author Stephan O'Bryan, December 2009.
 */

public class KalahaPosition extends GamePosition implements Cloneable {

    /**
     * The board represents a kalaha game-board. Elements 0-5 in the array are the upper holes of the board.
     * Elements 7-12 in the array are the lower holes of the board.
     * Elements 6 and 13 in the array are the kalaha-stores of the upper and lower player, respectively.
     */
    protected int[] board = new int[14];

    /**
     * Initial stones to be put in each hole at the beginning of the game.
     */
    protected static int STARTSTONES;

    /**
     * Constructs a KalahaPosition object with given initial stones in each hole (apart from the stores).
     *
     * @param initialStones the Initial stones to be put in each hole at the beginning of the game.
     */
    public KalahaPosition(int initialStones) {
        STARTSTONES = initialStones;
        for (int i = 0; i < 6; i++) {
            board[i] = STARTSTONES;
        }
        for (int i = 7; i < 13; i++) {
            board[i] = STARTSTONES;
        }
    }

    // static int count; //For counting the nodes.
    /**
     * Constructs a KalahaPositon object.
     */
**/  
public KalahaPosition() {  
    //      count++;  
    //      System.out.println(count);  
}  

@Override  
protected int value() {  
    int currentValue = 0;  
    return currentValue;  
}  

/**  
* Makes a move from the given hole number.  
*/  
protected void makeMove(int hole) {  
    makeMove(new KalahaMove(hole));  
}  

@Override  
protected void makeMove(Move m) {  
    KalahaMove km = (KalahaMove) m;  
    int fromHole = km.fromHole;  
    int stones = board[fromHole];  
    board[fromHole] = 0;  
    int endHole = 0;  
    for (int i = stones; i > 0; i--) {  
        fromHole++;  
        if (fromHole > 13)  
            fromHole = 0;  
        if (maxTurn && fromHole == 13)  
            fromHole = 0;  
        if (!maxTurn && fromHole == 6)  
            fromHole = 7;  
        endHole = fromHole;  
        this.board[fromHole] += 1;  
    }  
    // This one checks whether the last stone lands in an  
    // empty hole owned by the player, and the opposite hole contains  
    // seeds.  
    // If so, both the last stone and the stones in the  
    // opposite hole are captured and placed into the player's store.  
    if (this.board[endHole] == 1 && endHole != 6 &&  
        endHole != 13 && this.board[-1 * (endHole - 6) + 6] != 0) {  
        int oppositePool = -1 * (endHole - 6) + 6;  
        if (maxTurn && endHole < 6) {  
            this.board[6] += this.board[oppositePool] +  
            this.board[endHole];  
        }  
        this.board[oppositePool] = 0;  
    }  
}
this.board[endHole] = 0;
if (!maxTurn && endHole > 6 && endHole < 13) {
    this.board[13] += this.board[oppositePool] +
    this.board[endHole];
    this.board[oppositePool] = 0;
    this.board[endHole] = 0;
}

// When one player no longer has stone in any of his
// holes, the game ends.
// The other player moves all remaining stones to his
// store, and the player with the most stones in his store wins.
    for (int i = 7; i < 13; i++) {
        int addStones = board[i];
        board[i] = 0;
        if (maxTurn)
            board[13] += addStones;
        else
            board[6] += addStones;
    }
        for (int i = 0; i < 6; i++) {
            int addStones = board[i];
            board[i] = 0;
            if (!maxTurn)
                board[6] += addStones;
            else
                board[13] += addStones;
        }
        // If the last stone lands in the player's hole, the
        // player gets an additional move.
        if (endHole == 6 && maxTurn) this.maxTurn = !
        this.maxTurn;
        if (endHole == 13 && !maxTurn) this.maxTurn = !
        this.maxTurn;
        this.maxTurn = !this.maxTurn;
    }
    @Override
    protected GamePosition copy() {
        KalahaPosition kp = new KalahaPosition();
        kp.maxTurn = maxTurn;
        kp.board = board.clone();
        return kp;
Use it for optimizing the horizon of maxTurn:
if it is maxTurn, the alpha-beta search further,
otherwise not.

@return Returns true if it is maxTurn.

@override public boolean unclear() {
    return maxTurn;
}

@Override protected List<Move> nextPossibleMoves() {
    List<Move> npmList = new ArrayList<Move>();
    if (!gameOver()) {
        if (maxTurn)
            for (int holeNumber = 0; holeNumber <= 5; holeNumber++)
                if (board[holeNumber] != 0)
                    npmList.add(new KalahaMove(holeNumber));
        else {
            for (int holeNumber = 7; holeNumber <= 12; holeNumber++)
                if (board[holeNumber] != 0)
                    npmList.add(new KalahaMove(holeNumber));
        }
    }
    return npmList;
}

@Override
/**
 * Checks either one player's Kalaha store is more than 6
 * times of the initial stones OR
 * both players have equal number of stone in each
 * player's Kalaha, then the game is over.
 */
protected boolean gameOver() {
}
* Method (for GUI in console, if needed) which prints out the current KalahaPosition nicely.

```java
public String toString() {
    String s = "\n
    5  4  3  2  1  0\n
    ";
    for (int i = 5; i > -1; i--) {
        s += "(" + board[i] + ")";
    }
    s += "\n
    6 (" + board[6] + ")" + " +
    '(' + board[13] + ")") 13" + '\n' + "
    ";
    for (int i = 7; i < 13; i++) {
        s += "(" + board[i] + ")";
    }
    s += "\n
    7  8  9 10 11 12";
    return s;
}
```
```java
import java.util.List;

public class KalahaPositionMultiplayer extends GamePosition {
    protected int board[] = new int[14];
    protected static int STARTSTONES;
    protected int your_number;

    //for extra comments, check KalahaPosition
    public KalahaPositionMultiplayer(int initialStones, int player_number) {
        STARTSTONES = initialStones;
        for (int i = 0; i < 6; i++) {
            board[i] = STARTSTONES;
        }
        for (int i = 7; i < 13; i++) {
            board[i] = STARTSTONES;
        }
        this.your_number = player_number;
    }

    public boolean isLegalMove(int fromHole) {
        return board[fromHole] != 0;
    }

    protected void makeMove(Move m) {
        KalahaMove km = (KalahaMove) m;
        int fromHole = km.fromHole;

        int marble = this.board[fromHole];
        this.board[fromHole] = 0;
        int endHole = 0;
        for (int i = marble; i > 0; i--) {
            if (your_number == 1) {
                fromHole++;
                if (fromHole > 13)
                    fromHole = 0;
                if (maxTurn && fromHole == 13)
                    fromHole = 0;
                if (!maxTurn && fromHole == 6)
                    fromHole = 7;
                endHole = fromHole;
            } else {
                fromHole--;
                if (fromHole == 0)
                    fromHole = 13;
                if (maxTurn && fromHole == 13)
                    fromHole = 0;
                if (!maxTurn && fromHole == 6)
                    fromHole = 0;
                endHole = fromHole;
            }
            this.board[endHole] += 1;
        }
    }
}
```
} else if (your_number == 2) {
    fromHole++;  
    if (fromHole > 13)  
        fromHole = 0;  
    if (!maxTurn && fromHole == 13)  
        fromHole = 0;  
    if (maxTurn && fromHole == 6)  
        fromHole = 7;  
    endHole = fromHole;  
    this.board[fromHole] += 1;  
}  
}  

// This one checks whether the last stone lands in an empty hole owned by the player, and the opposite hole contains seeds.  
// If so, both the last stone and the stones in the opposite hole are captured and placed into the player's store.  
if (this.board[endHole] == 1 && endHole != 6 && endHole != 13 && this.board[-1 * (endHole - 6) + 6] != 0) {
    int oppositePool = -1 * (endHole - 6) + 6;  
    if (your_number == 1) {
        if (maxTurn && endHole < 6) {
            this.board[6] += this.board[oppositePool] + this.board[endHole];  
            this.board[oppositePool] = 0;  
            this.board[endHole] = 0;  
        }  
        if (!maxTurn && endHole > 6 && endHole < 13) {
            this.board[13] += this.board[oppositePool] + this.board[endHole];  
            this.board[oppositePool] = 0;  
            this.board[endHole] = 0;  
        }  
    }  
} else if (your_number == 2) {
    if (!maxTurn && endHole < 6) {
        this.board[6] += this.board[oppositePool] + this.board[endHole];  
        this.board[oppositePool] = 0;  
        this.board[endHole] = 0;  
    }  
    if (maxTurn && endHole > 6 && endHole < 13) {
        this.board[13] += this.board[oppositePool] + this.board[endHole];  
        this.board[oppositePool] = 0;  
        this.board[endHole] = 0;  
    }  
}  
   // When one player no longer has stone in any of his holes, the game ends.
// The other player moves all remaining stones to his store, and the player with the most stones in his store wins.
    if (your_number == 1) {
        for (int i = 7; i < 13; i++) {
            int addMarbles = board[i];
            board[i] = 0;
            if (maxTurn)
                board[13] += addMarbles;
            else
                board[6] += addMarbles;
        }
    } else if (your_number == 2) {
        for (int i = 7; i < 13; i++) {
            int addMarbles = board[i];
            board[i] = 0;
            if (!maxTurn)
                board[13] += addMarbles;
            else
                board[6] += addMarbles;
        }
    }
}

    if (your_number == 1) {
        for (int i = 0; i < 6; i++) {
            int addMarbles = board[i];
            board[i] = 0;
            if (!maxTurn)
                board[6] += addMarbles;
            else
                board[13] += addMarbles;
        }
    } else if (your_number == 2) {
        for (int i = 0; i < 6; i++) {
            int addMarbles = board[i];
            board[i] = 0;
            if (maxTurn)
                board[6] += addMarbles;
            else
                board[13] += addMarbles;
        }
    }
}

// If the last stone lands in the player's hole, the player gets an additional move.
if (your_number == 1) {
    // if (endHole == 6 && maxTurn) this.maxTurn = !
134         if (endHole == 13 && !maxTurn) this.maxTurn = !this.maxTurn;
135         this.maxTurn = !this.maxTurn;
136     } else if (your_number == 2) {
137         if (endHole == 6 && !maxTurn) this.maxTurn = !this.maxTurn;
138         // if (endHole == 13 && maxTurn) this.maxTurn = !this.maxTurn;
139         this.maxTurn = !this.maxTurn;
140     }
141 }
142 }
143
144 protected boolean gameOver() {
146 }
147
148 /**
149     * Method (for GUI in console, if needed) which prints out the current KalahaPosition nicely.
150     */
151     public String toString() {
152         String s = "\n
5  4  3  2  1  0\n
";
153         for (int i = 5; i > -1; i--) {
154             s += "(" + board[i] + ")";
155         }
156         s += "\n";
157         s += "6 (" + board[6] + ")" + "
";
158         for (int i = 7; i < 13; i++) {
159             s += "(" + board[i] + ")";
160         }
161         s += "\n"
162         s += "
7  8  9 10 11 12"
163         s += "\n"
164         return s;
165     }
166 }
167 @Override
168     protected List<Move> nextPossibleMoves() {
169         // TODO Auto-generated method stub
170         return null;
171     }
172 }
173 @Override
174     protected GamePosition copy() {
175         // TODO Auto-generated method stub
176         return null;
177     }
return null;

@Override
protected int value() {
    return 0;
}
}
Move.java

```java
/**
 * A general class. An object of Move type represents a move in any two-player game.
 *
 * @author Stephan O'Bryan, December 2009.
 */

public abstract class Move {
}
```
Server_Main.java

import javax.swing.*;
import java.io.IOException;

public class Server_Main extends JFrame {

    public Server_Main(int tcpport) throws IOException {
        Game_Server server = new Game_Server(tcpport);
    }

    public static void main(String[] args) throws IOException {
        Server_Main gms = new Server_Main(9000);
        gms.setSize(400, 400);
        gms.setTitle("Server");
        gms.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        gms.setVisible(true);
    }

}
```java
public interface Table {

    /**
     * This method describes the rules of how the communication between two clients should go.
     * Should it be turned based, between two players? should it be real time? etc etc.
     */

    public void gamePlay();
}
```