HASAN RAZA
1112333

TicTacToe

Learning focus:
Class Design, Class Diagrams, Activity Diagrams, Code Reuse, Static and Dynamic Polymorphism

Your task is to develop an object oriented version of the game TicTacToe

Marking

Analysis and Design	3 points
Algorithms and Design Decisions	3 points
Implementation and Test	4 points

You need 4 points in order to pass the lab!

Upload the code in Moodle and bring a handwritten solution together with your code to the lab.

Structure will be more important for marking than behavior!

1 Class Relations

1.1 *Requirements*

Your task is to develop a console based version of the boardgame TicTacToe

- The game consists of a 3x3 board.
- On every position, the players place a "x" (player 1) or a "o" (Player 2) stone.
- The first player, who has 3 stones in a row (horizontally, vertically or diagonally) wins the game.

Technical requirements

- Player 1 and 2 can either be human or computer, i.e.
  - Human against Computer
  - Human against human
  - Computer against human
  - Computer against computer are possible combinations.
- The computer may never loose.
- The board will be displayed as a console application.
- Displaying the screen later on in a graphical format shall be easily supported.
- Provide DoxyGen Style comments for all entities and create a website for your project, which shall also be uploaded.

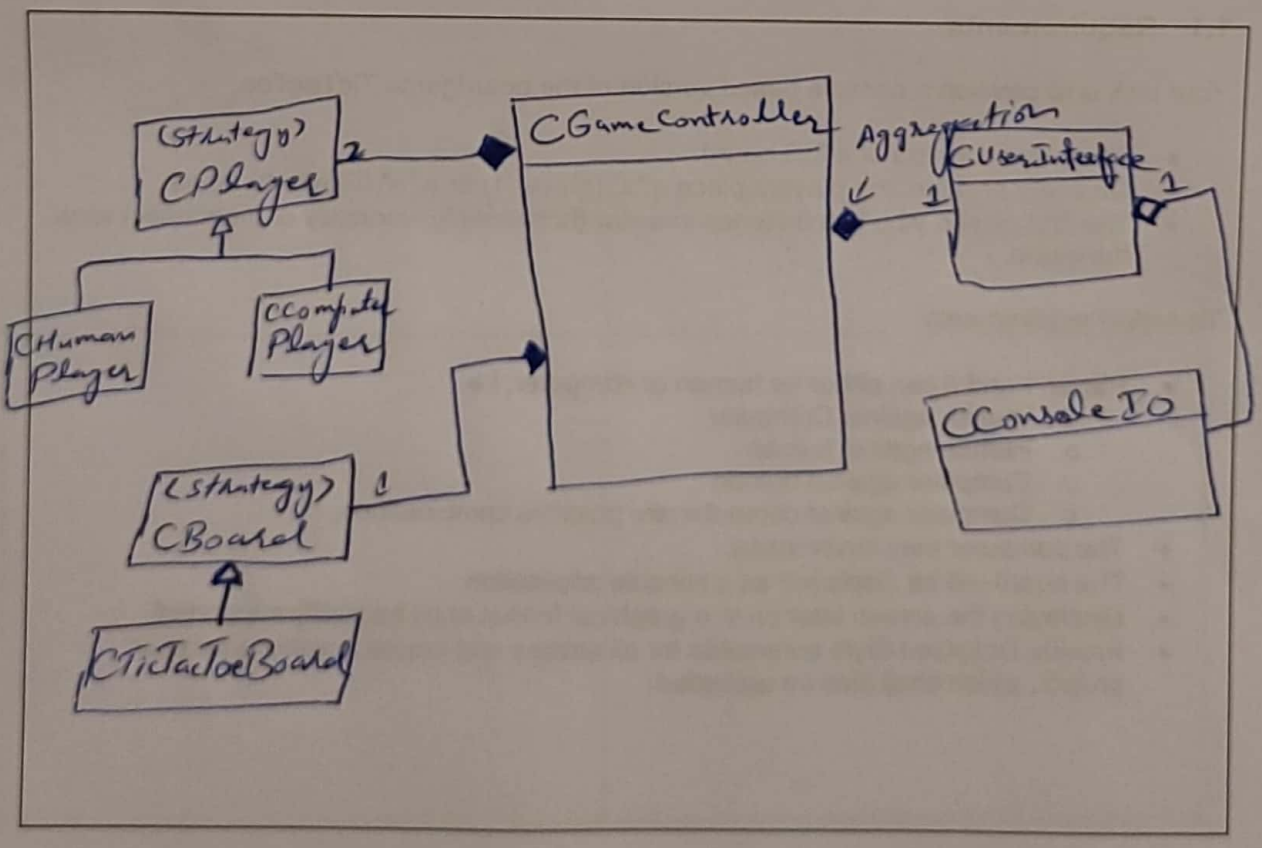
1.2 OO Analysis

From the requirements, identify the mentioned objects and describe them using the concept of OOA (object oriented analysis)

Object	Properties	Capabilities
① Board	2Darray	(i). Represent the board (ii). Validation and Making Move
② Player	Player1, Player2	(i). Represent Player (ii). Decide and Make Move
③ GameController	CurrentTurn	(i). Manage the game flow (ii). Determine the winner or draw
④ UserInterface	ConsoleIO	(i). Display Board and Messages (ii). gets User Move (iii). Show Winner

1.3 OO Design

Create a first rough class design based on the results of the object oriented analysis. You may omit the methods and attributes.



I did not implement Reversi Boardgame as I am from Communication Major

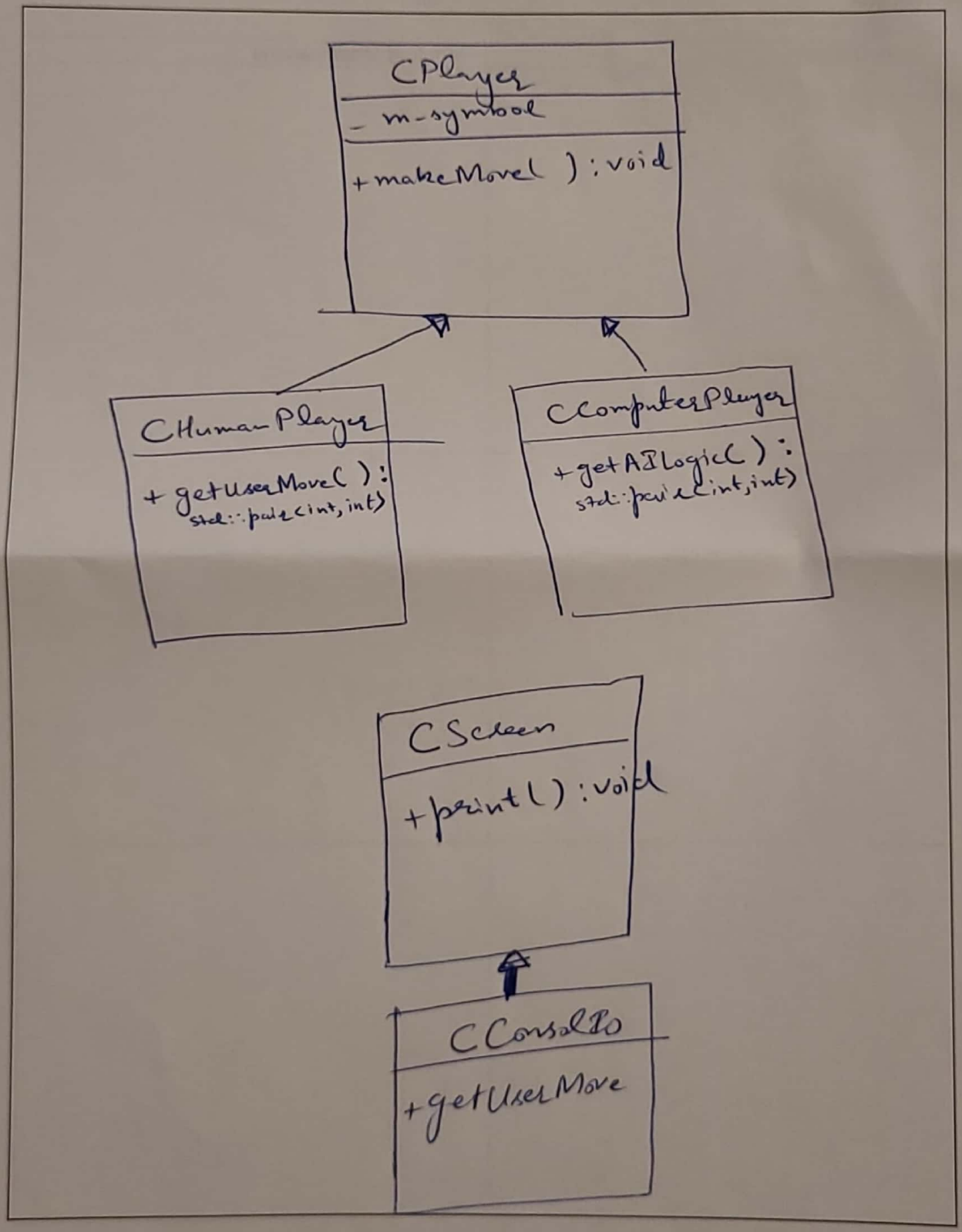
1.4 Reuse

Check the implementation of the Reversi Boardgame and describe which classes can be reused. Provide a class diagram for every class you reuse and describe if changes are required.

Class Diagram	Required Modifications

1.5 Static Polymorphism

Draw a class diagram for a possible implementation of the TicTacToe boardgame using static polymorphism for the Screen and Player partition.

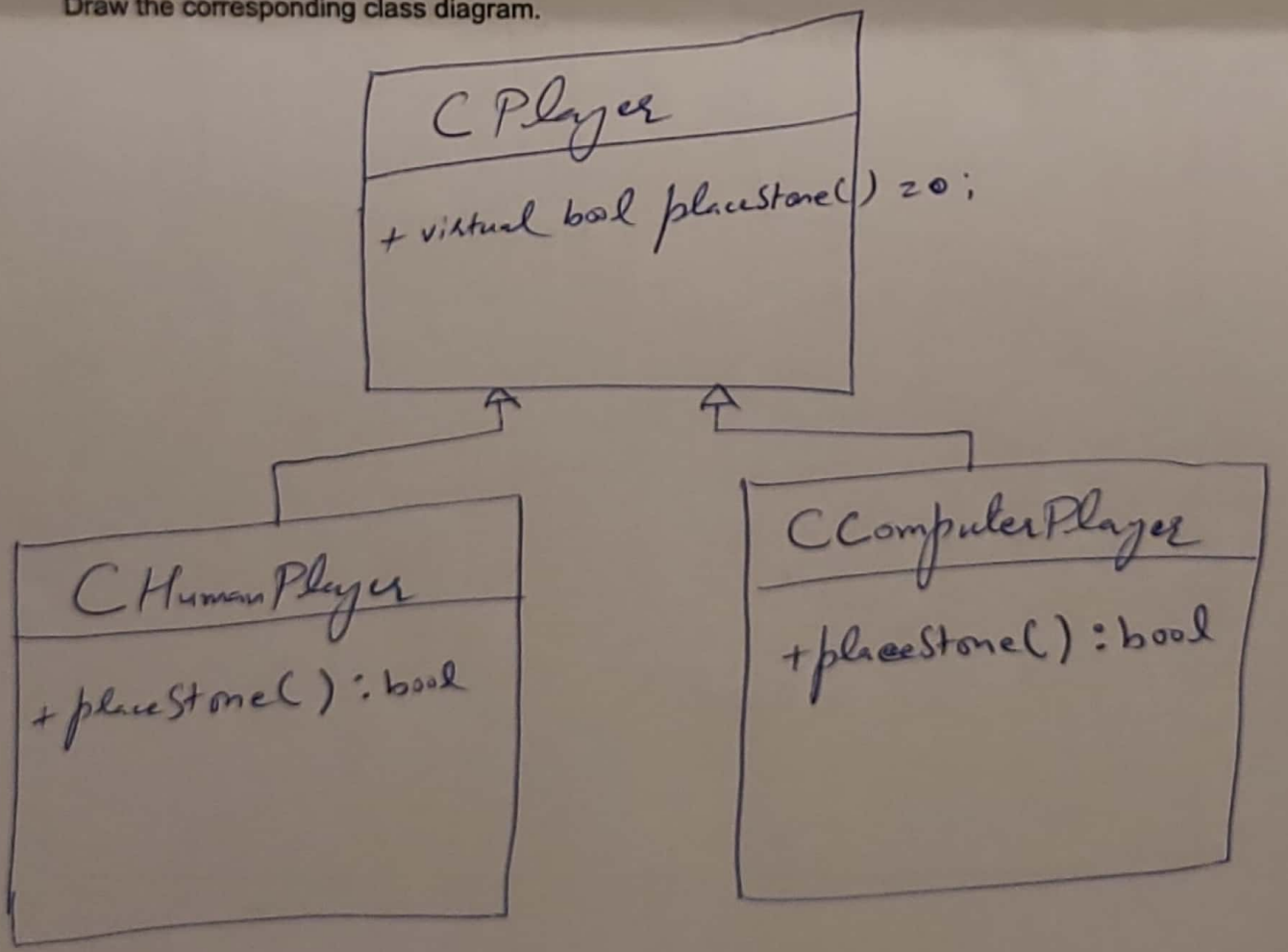


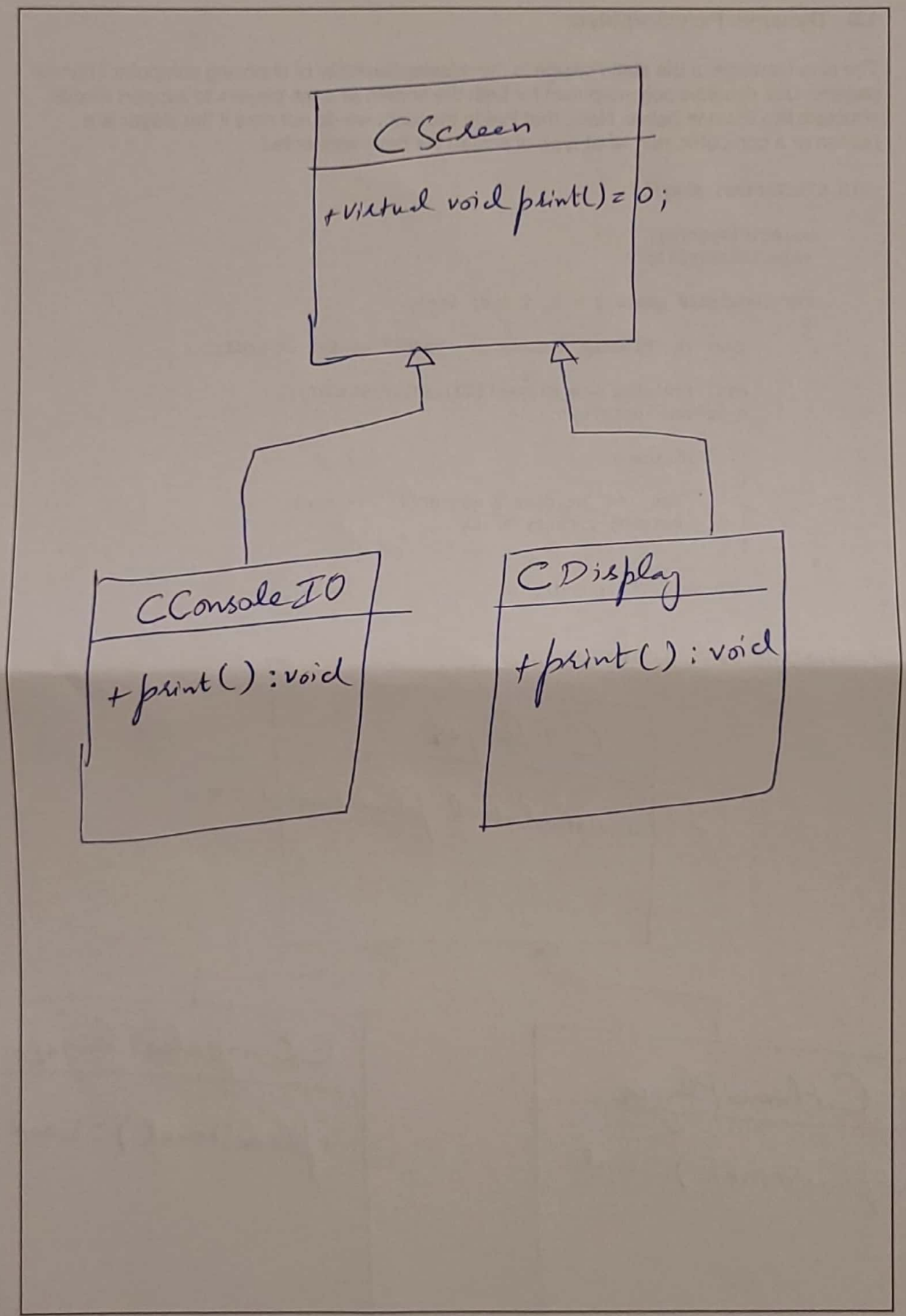
1.6 Dynamic Polymorphism

The disadvantage of the static design is the missing flexibility of choosing computer / human players. Use dynamic polymorphism for both the screen and the players to support a code structure like the one below. Note, that inside the loop, we do not care if the player is a human or a computer, nor, what type of screen we have connected.

```
void CTicTacToe::play()
{
	selectPlayer(0);
	selectPlayer(1);

	for (unsigned short i = 0; i < 9; i++)
	{
		cout << "Placing a stone in round " << i+1 << endl;

		bool finished = m_player[i%2]->placeStone();
		m_screen->print();

		if (finished)
		{
			cout << "We have a winner!!!" << endl;
			return; //hacky break
		}
	}
	cout << "Draw...." << endl;
}
```

Draw the corresponding class diagram.

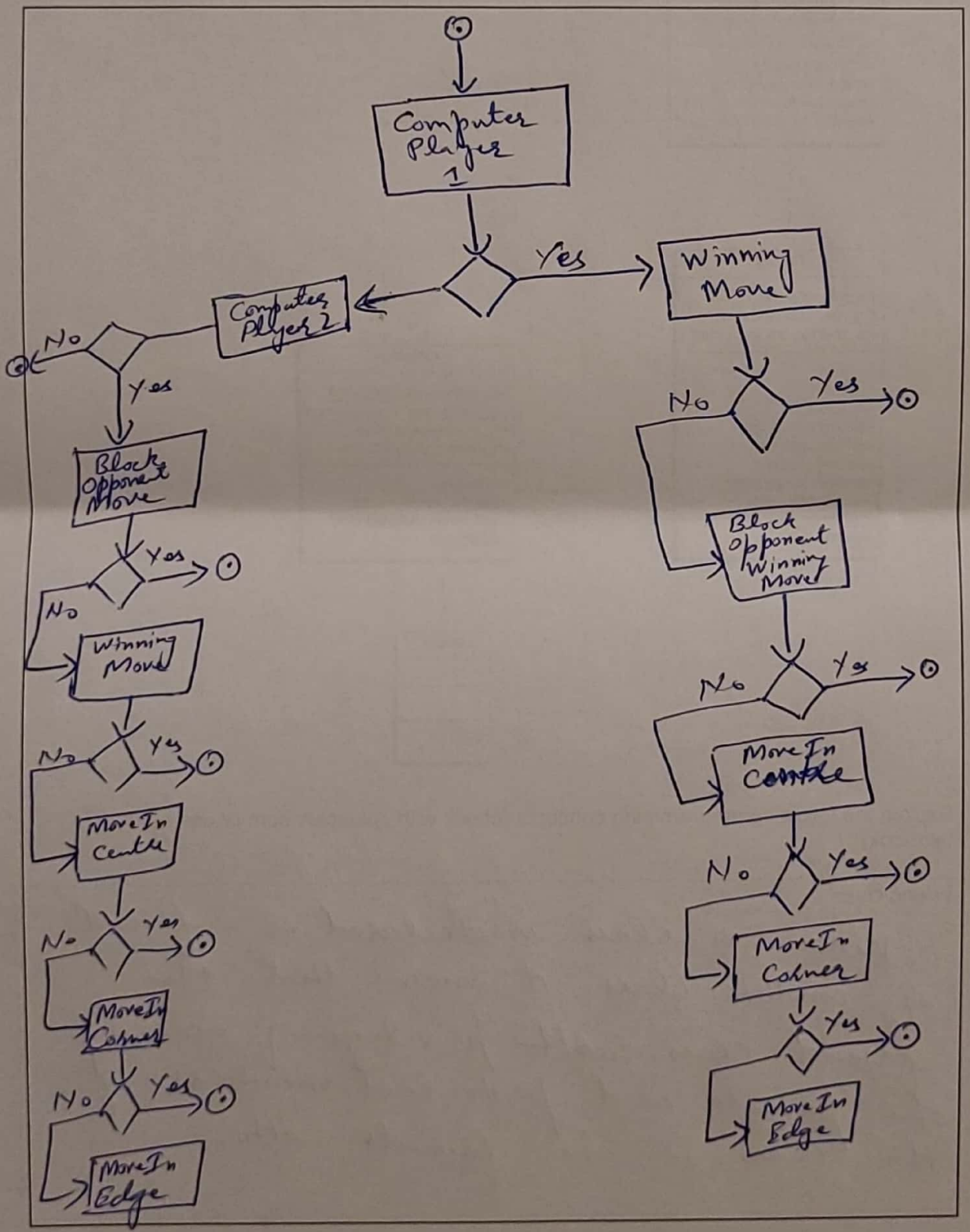



CScreen
+virtual void print() = 0;
CConsoleIO
+print() : void
CDisplay
+print() : void

2 Algorithms and Design Decisions

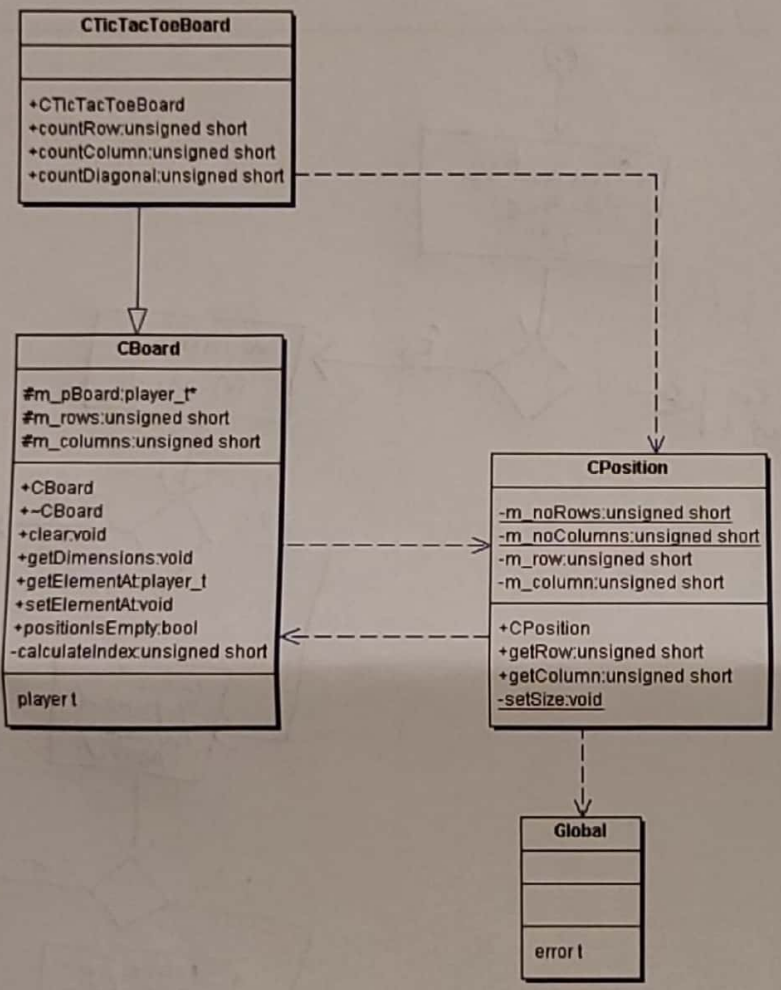
2.1 Activity Diagram

Draw an activity diagram describing the top level placeStone-Strategy for the computer player.



2.2 Spooky Hierarchy

After spending a frustratingly large number of hours for creating the first design, you decide to hack the notebook from Nelly, the Nerd to check for his solution. You find the following design snippet:



Explain the following programming concepts (check with cplusplus.com or use a good textbook)

Friend class

When a class is declared as a friend of another class, it means that the friend class has the privilege of accessing the private and protected members of the class it is friends with.

Static attribute

A static attribute is a class member variable that belongs to the class rather than to instances of the class. This means there is only one copy of the static attribute shared by all instances of the class it is declared using the 'static' keyword

Static method

A static method is a class member function that belongs to the class rather than to the instances of the class. Static methods do not operate on an instance of the class and can be called using the class name without creating an object.

Throwing and catching exceptions

Exception handling is a mechanism that allows to handle errors or exceptional conditions in the code. This involves throwing exceptions when an error occurs and catching those exceptions at an appropriate place to handle the error

Using these concepts, describe how you can implement the following feature:

- A CPosition object describes a specific position on the boardgame.
- The CPosition Object knows the size of the board (number of rows and columns).
- The size attribute may only be set by the CBoard class, while the board is being constructed.

- If a CPosition object is constructed, whose value is out of the range of the board, the user will be informed so that he can initiate a proper error handling.

```
#include <iostream>
class CBoard; // Forward declaration of
              // CBoard

class CPosition {
  private:
    int row;
    int col;
    int boardsize;
  CPosition(int r, int c, int size) : row(r), col(c), boardsize(size){}

  public:
  static CPosition createPosition(int r, int c, const CBoard&
                                  board);
  // Getter methods
  int getRow() const { return row; }
  int getCol() const { return col; }
  bool isValid() const {
    return row >= 0 && row < boardsize && col >=
    && col < boardsize;
};

class CBoard {
  private:
    int size;
  public:
    CBoard(int boardsize) : size(boardsize) { }
    int getsize() const { return size; }
};

CPosition CPosition::createPosition(int r, int c,
                                    const CBoard& board)
{ if (r<0 || r>= board.getsize() || c<0 ||
      c>=board.getsize()) {
    std::cerr<<"Invalid Position." << std::endl;
    return CPosition(-1, -1, -1);
  } return CPosition(r, c, board.getsize());
}
```

3 Implementation and Test

Implement the polymorph design and test the code.

Alternatively, you may also implement the static design, which however reduces the implementation points to 2 max.

The output should look similar to (Debug message are optional):

```
Select player: 1 (1 - human, 2 - computer) : 2
Computer  Player selected

Select player: 2 (1 - human, 2 - computer) : 2
Computer  Player selected

Placing a stone in round 1
 . . .
 . x .
 . . .

Placing a stone in round 2
 o . .
 . x .
 . . .

Placing a stone in round 3
 o . .
 . x .
 x . .

Placing a stone in round 4
Found defense diagonal: (0, 2)
 o . o
 . x .
 x . .

Placing a stone in round 5
Found defense row: (0, 1)
 o x o
 . x .
 x . .

Placing a stone in round 6
Found defense column: (2, 1)
 o x o
 . x .
 x o .

Placing a stone in round 7
 o x o
 . x .
 x o x

Placing a stone in round 8
 o x o
 o x .
 x o x

Placing a stone in round 9
 o x o
 o x x
 x o x

Draw....
```