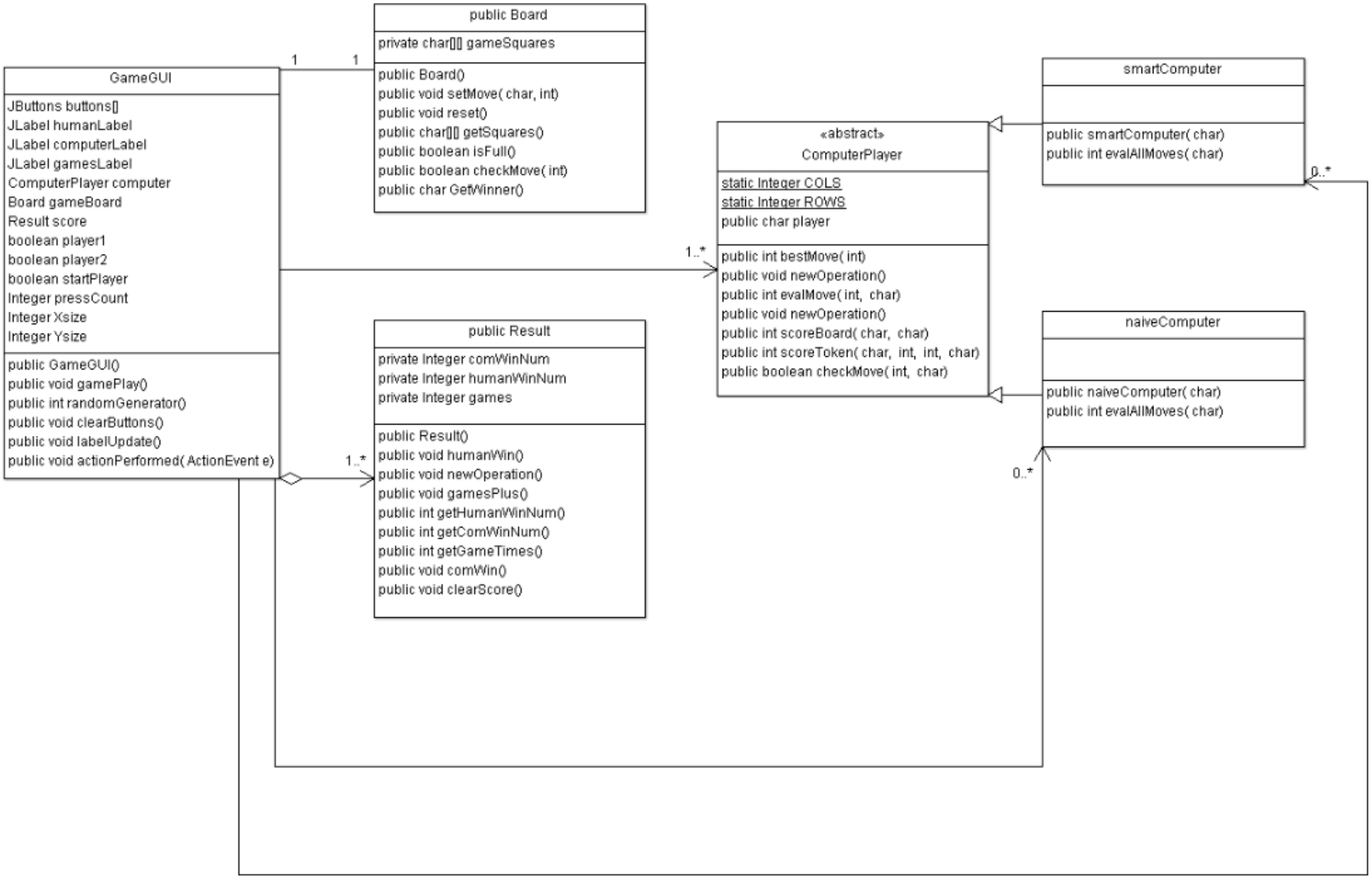
Task No 3a – Class Diagram

You must produce design documentation. This will include a class diagram for the system, a short explanation as to the general purpose of each of the classes you have produced and a justification for any design decisions you have made.


public Board
private char[][] gameSquares
public Board()
public void setMove(char, int)
public void reset()
public char[][] getSquares()
public boolean isFull()
public boolean checkMove(int)
public char GetWinner()
GameGUI
JButtons buttons[]
JLabel humanLabel
JLabel computerLabel
JLabel gamesLabel
ComputerPlayer computer
Board gameBoard
Result score
boolean player1
boolean player2
boolean startPlayer
Integer pressCount
Integer Xsize
Integer Ysize
public GameGUI()
public void gamePlay()
public int randomGenerator()
public void clearButtons()
public void labelUpdate()
public void actionPerformed(ActionEvent e)
1
1
1..*
1..*
«abstract»
ComputerPlayer
static Integer COLS
static Integer ROWS
public char player
public int bestMove(int)
public void newOperation()
public int evalMove(int, char)
public void newOperation()
public int scoreBoard(char, char)
public int scoreToken(char, int, int, char)
public boolean checkMove(int, char)
smartComputer
public smartComputer(char)
public int evalAllMoves(char)
0..*
naiveComputer
public naiveComputer(char)
public int evalAllMoves(char)
0..*
public Result
private Integer comWinNum
private Integer humanWinNum
private Integer games
public Result()
public void humanWin()
public void newOperation()
public void gamesPlus()
public int getHumanWinNum()
public int getComWinNum()
public int getGameTimes()
public void comWin()
public void clearScore()

The application logic aims to follow the key principle of software development and architecture - separation of concerns. This generic principle is also tightly linked to Single Responsibility Principle that supports development of layered application structures and is a natural outcome of the object-oriented programming.
The core feature of this approach is to separate business logic and functionality model from the user interface. The benefits are multilayered and clearly reflecting the object-oriented programming approach: improvement in simplicity, clear boundaries of the responsibilities, clarity when enhancements are developed or maintenance attempted to the existing code.
Furthermore, the flexibility of the design is evaluated in the capability to separate responsibilities for application design and development to different entities, each liable to distinguishable part of the application and not constrained by work done by other entities. This also allows re-use of the code to different environments (e.g. model / logic structure can be implemented to various, unrelated user interfaces). Encountered bugs in the development and post-development stages are easier to track down and correct due to the design modularity and separation of responsibilities between different parts of the application.

The application design set in the class diagram tries to reflect this approach by separating the class/es that provide internal logic of the game against its user interface. With it, the software separation distinguishes between 3 area requirements:

- Deliver interface to the user, detecting events and triggering notifications corresponding to the user interaction with the available user functionalities;
- Handle raw data, allow it to be updated, read from, deleted, etc;
- Manage interaction by receiving input and processing it via set business logic;

The `GameGUI` class encapsulates the design features for the user interface needs. It delivers the methods necessary to create actual design of the game. It also allows

game logic to be interpreted on the game board, while controlling the behavior of the buttons and data display.

The `Board` class delivers the game logic. It manipulates the user input and controls the game results based on the game rationale. It provides the mechanisms by which changes are made to the user display and decides the game outcome (game logic). It also manipulates the game data controlled by `Result` class.

The `Result` class is responsible for managing the game data. While the game requirements did not stipulate the complex list of needs (i.e. solely storing the data for the existing game), it seems suitable to decouple raw game data from other application's parts, so to allow for further enhancements if needed. Additional functionalities that could be attempted in the future (e.g. storing total number of games played, highest win / lose game, longest game played, etc.) could be easily added to the existing `Result` class without the need to address the code structure of the other application parts. In a way, `Result` class takes the role usually assigned to databases and could be extended to encompass further data capturing.

The computer player functionalities are set in the `ComputerPlayer` class. As an abstract class, it contains methods and functionalities common to different capability levels set for the computer player. This responds to the assignment request to deliver abstract `ComputerPlayer` class that extends in two ways to a `NaivePlayer` and `SmartPlayer`. The distinction between two are set in the method controlling evaluation of the suitable move in the game. The derived classes are called based on the game logic and they implement appropriate behavior assigned to them.

Overall, the class separation and game design follows the well-established design pattern that is set as unofficial industry standard. It permits modular design approach and modular manipulation of the code, thus ensuring clear boundaries of responsibilities. While number of similar design patterns can be found in existence today (e.g. MVC model – view – controller approach or its derivation, MVP model – viewer – presenter), they all boil down to the identical aim: data (including how you read and write that data) is separate from the logic, while the display of the result is separate from the rest of the code.