**IFQ563**

**OBJECT ORIENTATED DESIGN**

**ASSIGNMENT 2:**

**DESIGN REPORT AND IMPLEMENTATION**

SELKIE MACKAY

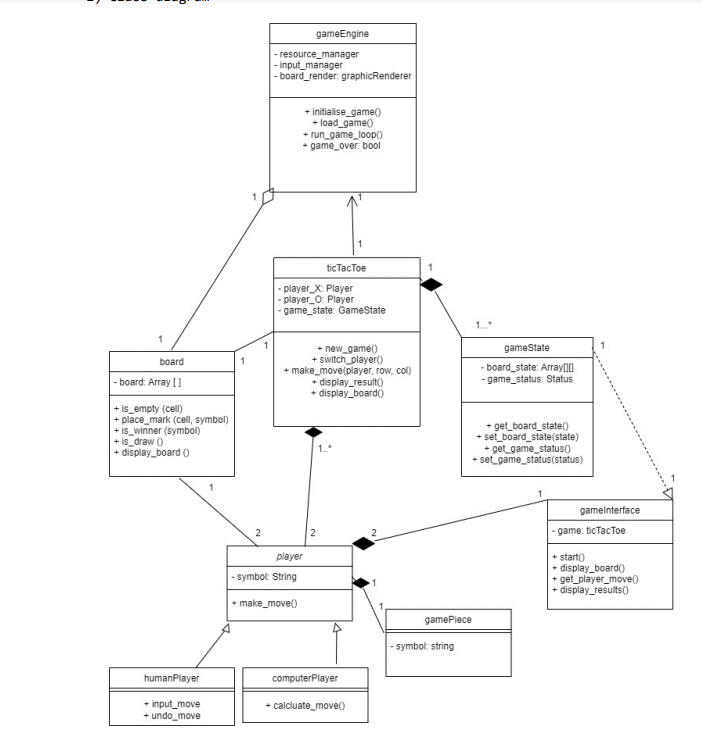
N9660640

**STATEMENT OF COMPLETION**

* Reusable and extensible framework
* Caters for different modes of play- human vs human
* Caters for different modes of play- Computer vs human
* With human player-System checks validity of moves as entered
* With computer player- user is able to select range of possible strategies and/or levels of difficulty
* Implemented Random selection of legal moves
* Implement Alpha-beta algorithm
* Games should be able to be saved and restored from any state of play
* Moves should be undoable and re-doable
* A game should be re-playable from any position
* A primitive online help system to assist users with the available commands.

**OVERVIEW OF FINAL DESIGN**

The preliminary design from assignment 1 did not clearly demonstrate how design patterns are applied or demonstrate how the framework is built with extensibility and reusability principles. Furthermore, the initial design could not be implemented to quickly build other specific games because neglected to reflect the concepts of reusability and extensibility through the factory method where ‘gameEngine’ class should be responsible for creating instances of different games. If the initial design was to be implemented as code, game creation logic would be duplicated for different games and would not be extendable to accommodate other games. Referencing the initial class diagram pictured below, extendibility is not reflected. As observed, this diagram narrowly represents one game (tic tac toe).



As depicted, only ‘gameBoard’ and ‘ticTacToe’ are inherited from the game engine. To enhance reusability, classes were remodelled to be less specific to the tic tac toe game. The improved design was orientated towards a game framework with classes that could be reused to implement other board games and added into a class library.

‘gameBoard’, ‘players’, ‘isWinner’, ‘standardMessages’ and ‘program’ classes were redesigned with direct relationships to ‘gameEngine’. ‘gameEngine’ holds common game logic and responsibility for assigning markers and making moves. These methods apply to not only tic tac to but other fulfil other game requirements. In the preliminary design, responsibilities of ‘gameEngine’ were minimised and distributed to ‘isWinner’ and ‘game’ class to organise code.

A new class entitled ‘game’ was created to employ reusable and extendable design. This class acts as a selection menu, allowing the user to select different games to be run on the framework. In this case, only tic tac toe is implemented, however Reversi and Morris game would be able to utilise and extend the improved the framework.

The ‘gameState’ class was decomposed and its responsibilities were shared with ‘gameBoard’, ‘gameEngine’ for efficient management. This change allowed player class to access information efficiently when game strategy was enabled for ‘alphaBetaComputer’ class. If ‘gameState’ class remained, it was delay the communication needed to inform the algorithm.

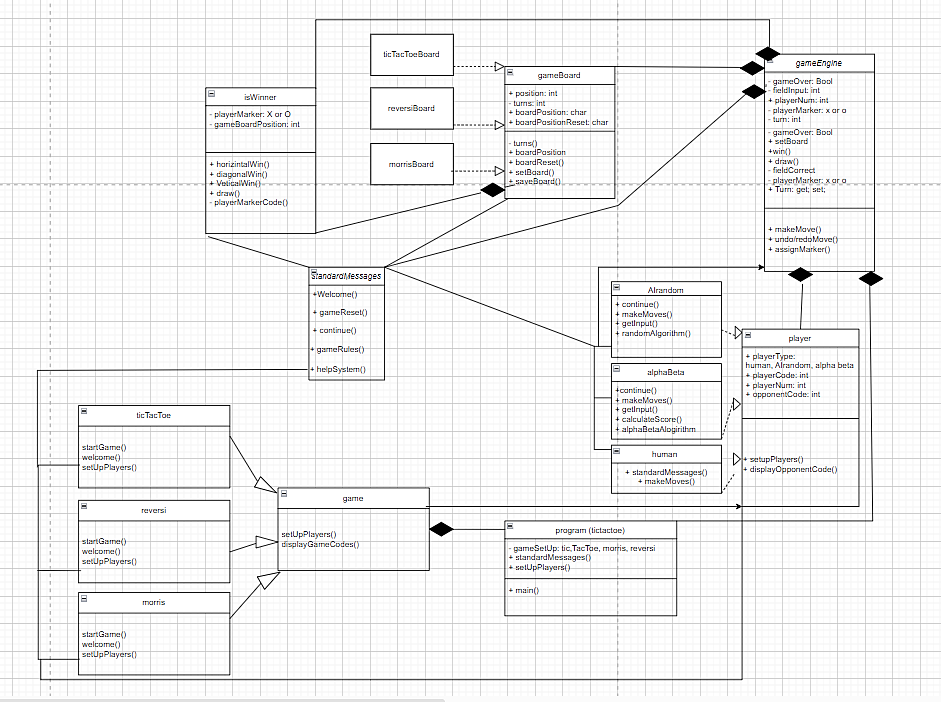
In the original design, two child classes were inherited from player. To satisfy the project requirements, ‘alphaBetaComputer’ class was added to allow players to choose between two computer opponent options.

An ‘isWinner’ class was created to handle the logic of performing different types of wins which, in turn, informs ‘gameBoard’ and ‘gameEngine’ class of game specific rules. Creating this class decomposes the properties originally set out in ‘gameBoard’ class effectively manage responsibilities.

‘standardMessages’ class was created to provide a means of communication between program and user. Creating this class disperses and minimises communication related methods within other classes. This class can be called whenever the user needs instructions or assistance.

The final design allows for the user to choose a particular game to be selected. Ideally, to enhance the reusability and extendibility of the framework, the selected game should be able to pass its specific attributes and properties to the existing classes to implement specific game logic. However, in this design, tic tac toe is executed within the framework. To run either Morris or Reversi, changes to the follow code would need to be made:

1. ‘gameBoard’ class setBoard() and resetBoard() methods altered to accommodate new game board. However, the final design class diagram illistruates the realisation of different game boards that could be implemented to meet game requirements.
2. ‘isWinner’ class logic would need altering to accommodate new game logic to win.
3. ‘gameEngine’ class makeMoves() and assignMarker() would need altering to accommodate for specific game markers and allowed moves.

**UPDATED CLASS DIAGRAMS**

**UPDATED SEQUENCE DIAGRAMS**

**A diagram of a computer program

Description automatically generated**

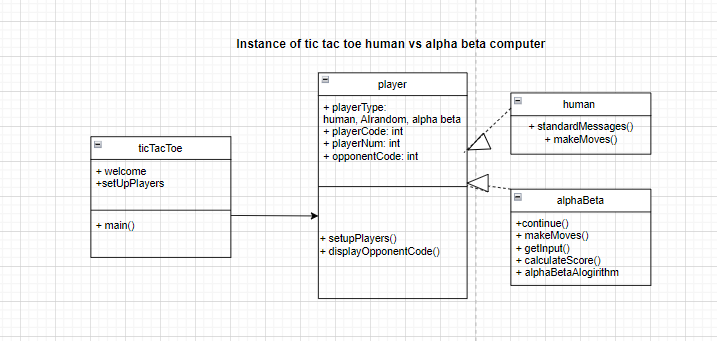
A diagram of a computer program

Description automatically generated

**UPDATED OBJECT DIAGRAMS**

A diagram of a computer game

Description automatically generated



**DESIGNED PRINCIPLES AND PATTERNS**

The Single Responsibility Principle (SRP) states that a class should only have one reason to change and should encapsulate only one aspect of a system’s functionality (Hall, 2014). The ‘isWinner’ class exclusively handles logic to determine the game winner considering the rules of the implemented game. In this case, tic tac toe was the chosen game. Regarding SRP, implementing different game rules to determine the winner would be the only reason to make changes to this class. Changing the logic of Win() methods and draw() to suit specific games would require alternations. Due to all three games requiring a winner with different methods to obtain a win, the class would not require extensive restructuring to cater to specific game logic. This, in turn, promotes extensibility and reusability of design (Hall, 2014).

The same can be applied to the parent ‘gameBoard’ class. The single responsibility of the class is to efficiently set up the board boundaries and parameters. The only reason this class requires changes is when an implemented game has a specific board.

The structure of the ‘player’ class uses inheritance to create the new classes of Human player, randomAI player and alpha beta player. These three child classes inherit properties and methods of the player parent class which acts as the base class. This inheritance supports code reusability as common functionality and data are defined in the parent class, reducing duplication and promoting consistency across player types (Hales, Karpavicius & Viegas, 2022). Therefore, each child class can then implement specific behaviour while inheriting common attributes and methods from parent/base class. This class also demonstrates elements of polymorphism with method overriding and treating different player types in a uniform way based on their individual behaviours (Hales, Karpavicius & Viegas, 2022).  This contributes to a flexible and reusable design as core logic of the framework can remain untouched if the setUpPlayer method needed to be altered to accommodate a different game.

In regards to polymorphism, the class ‘standardMessages’ provides a common interface for displaying game related messages and handling user interactions with methods like ‘welcome()’, ‘gameReset()’ and ‘continue()’. This class interacts with ‘gameBoard’ and ‘gameEngine’ by resetting the board, initiating moves and providing user information. This design allows message displays and user interactions uniformly throughout the game and contains broad methods, pertaining multiple games, promoting code reusability (Hales, Karpavicius & Viegas, 2022). The Open-closed Principle (OCP) can be discussed with reference to this class and player class which define broad methods that can be applied to multiple board games. The non-specific elements of these classes support reusability as they can operate with different games types and also extendibility as they can made to accommodate other programs without modification (Hall, 2014)

Encapsulation is applied to private access modifiers for methods in classes such as gameEngine, gameboard, isWinner and standardMessages. These access modifiers hide internal implementation details of classes. Separating public and private members permits changes of internal code without altering external code. Encapsulation aligns with the principles of reusability and extendibility by creating self-contained classes that can be reused for different programs (Hales, Karpavicius & Viegas, 2022).

The factory method pattern is considered in the constructing of ‘gameBoard’ which provides standard method for setting and resetting a board as an object. These methods can be applied to create different board games with different cells and boundaries which promotes flexibility and encapsulation (Nesteruk, 2022). Additionally, the factory method is observed in ‘gameEngine’ where new implementation of the interface can be created for games by maintaining fundamental logic and responsibilities of the class and altering the referenced ‘IsWinner’ and ‘gameBoard’ specificities to accommodate game logic. The factory method encourages code reusability by providing a common way to create objects and encapsulates the responsibilities of the objects within the class, localising the logic (Sarcar, 2020).

**EXECUTION INSTRUCTIONS: HOW PROGRAM CAN BE EXECUTED**

1. Human player starts program to view command prompt
2. When program is loaded, user is prompted to select game to play
3. User receives initial message “Welcome to a game of Tic Tac Toe!”
4. User receives simple instructions for tic tac toe “In this game, players are represented with a unique markers: X or O

The first player to score three same markers in a horizontal, vertical or diagonal row is the winner!

1. Press any key to proceed
2. User displayed list of possible opponents

* 1: other human- this option would allow two human users to play and turns would be alternating between two human players
* 2: computer random- user can opt to play against computer that makes random moves on board
* 3: computer alpha beta- user can opt to play against a challenging opponent using the alpha beta pruning strategy

1. User selects opponent code (1, 2 or 3)
2. User is then allocated marker
3. Game board is displayed and user is prompted to choose number 1-9 which corresponds with depicted game squares
4. Once player has made move, opponent makes move
5. This loop is repeated until a winner is determined
6. Once winner is determined or draw is determined, winner is displayed and congratulated.
7. Turned taken is displayed
8. User can opt to reset for new game

**SUMMARY OF CLASSES**

Collections library was not utilised in the making of this program however the following classes could be added to collections library to be reused for other game programs:

* AIrandom algorithm (references ‘gameBoard’ therefore gameboard would need to reflect implemented game’s board)
* Alpha Beta Algorithm (references ‘gameBoard’ therefore gameboard would need to reflect implemented game’s board)
* Human player
* Player
* Standard messages
* Game
* Game engine (references ‘IsWinner’ and ‘gameBoard’ which would need methods changed to reflect implemented game)
* Is winner (winner logic may need altering to accommodate different game rules)
* Game board (game board squares need altering to accommodate different game board requirements)

**REFERENCES**

Nesteruk, D. (2022). *Design Patterns in .NET 6 : Reusable Approaches in C# and F# for Object-Oriented Software Design* (Third edition.). Apress Media LLC. <https://doi.org/10.1007/978-1-4842-8245-8>

Sarcar, V. (2020). *Design patterns in C# : a hands-on guide with real -world examples* (Second edition.). Apress. <https://doi.org/10.1007/978-1-4842-6062-3>

Hales, J., Karpavicius, A., & Viegas, M. (2022). *The C# Workshop : Kickstart Your Career As a Software Developer with C#.* ([First edition].). Packt Publishing, Limited.

Hall, G. M. (2014). *Adaptive code via C# Agile coding with design patterns and SOLID principles* (1st edition). Microsoft Press.