Advanced Tic Tac Toe Game Development Project Report

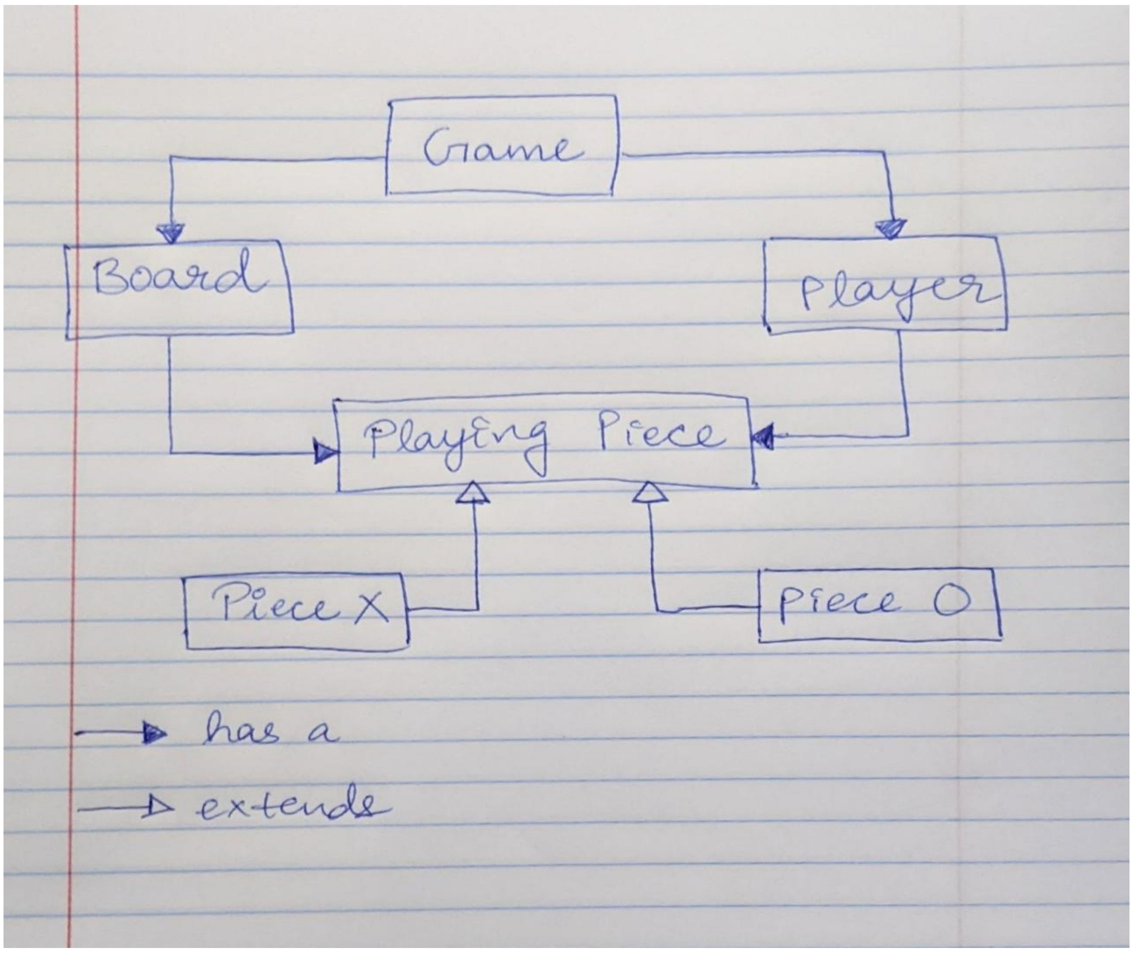
Project Introduction:

Coming up with a complete game with advanced features like undo and redo and what we are going to do is implement the tictactoe game(as taken from, Tic Tac Toe is a paper-and-pencil game for two players,X and O, who take turns marking the spaces in a three-by-three grid. The player who succeeds in getting three of her marks aligned in any row, column, or diagonal is the winner.)and implementing the complete game using design patterns.This new game concept builds on the original Tic Tac Toe game by adding advanced features designed to increase user engagement and strategic gameplay using real-time game development techniques and technologies.This Project Involves Adding a New Gameplay to the Classic Game and an Advanced Graphical User Interface Using JavaFX Algorithm.

Objective:

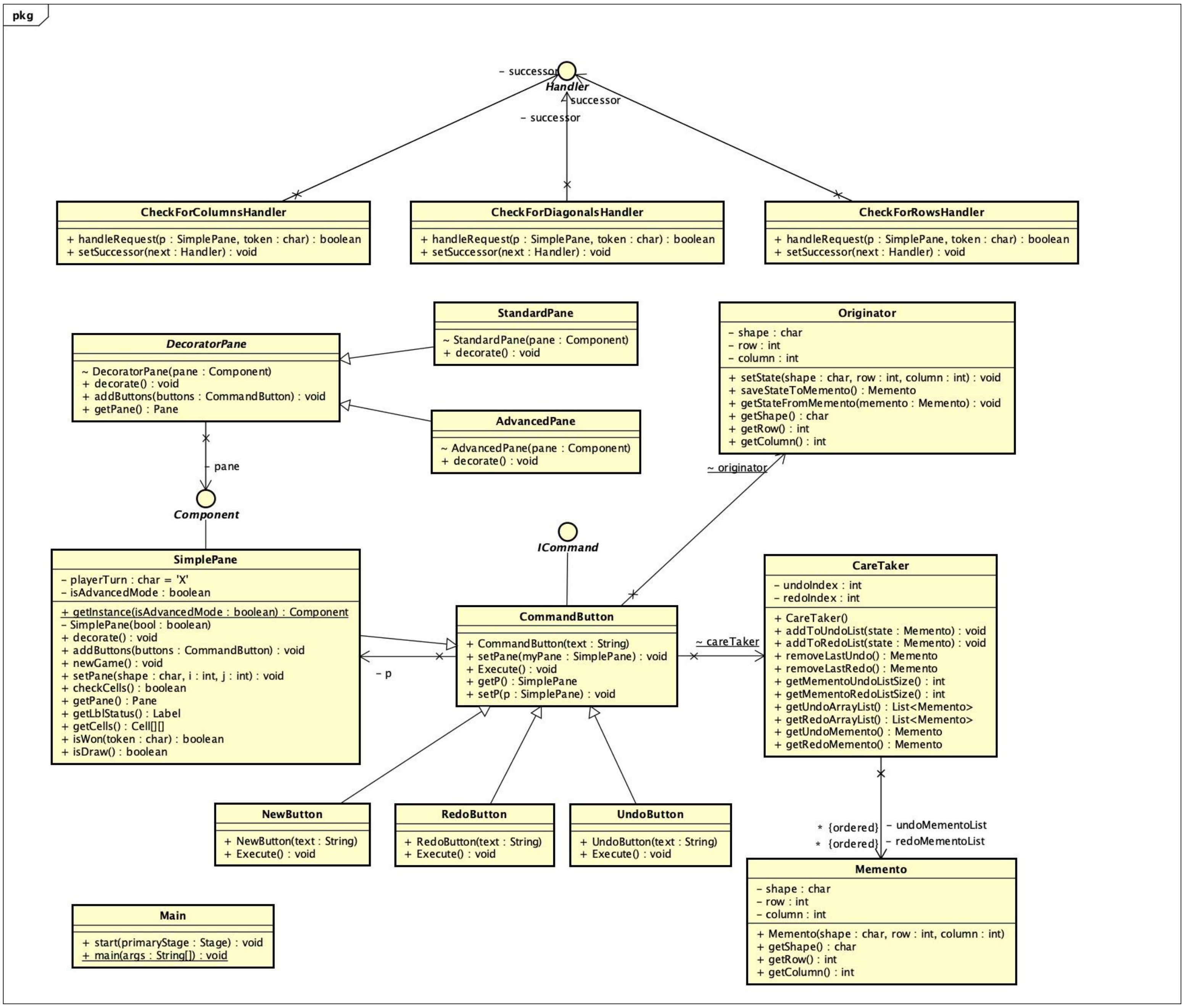
The aim is to create a highly complex Tic Tac Toe where there are move reverting and redoing strategies included, which enables the dynamic movements for the players to act and react accordingly in the course of the gaming. This game is designed for puzzle fans and strategy gamer players of all ages who enjoy to play dynamic puzzle games through appealing widget designs and configuration mechanisms for an easier gameplay.

Architecture Overview:



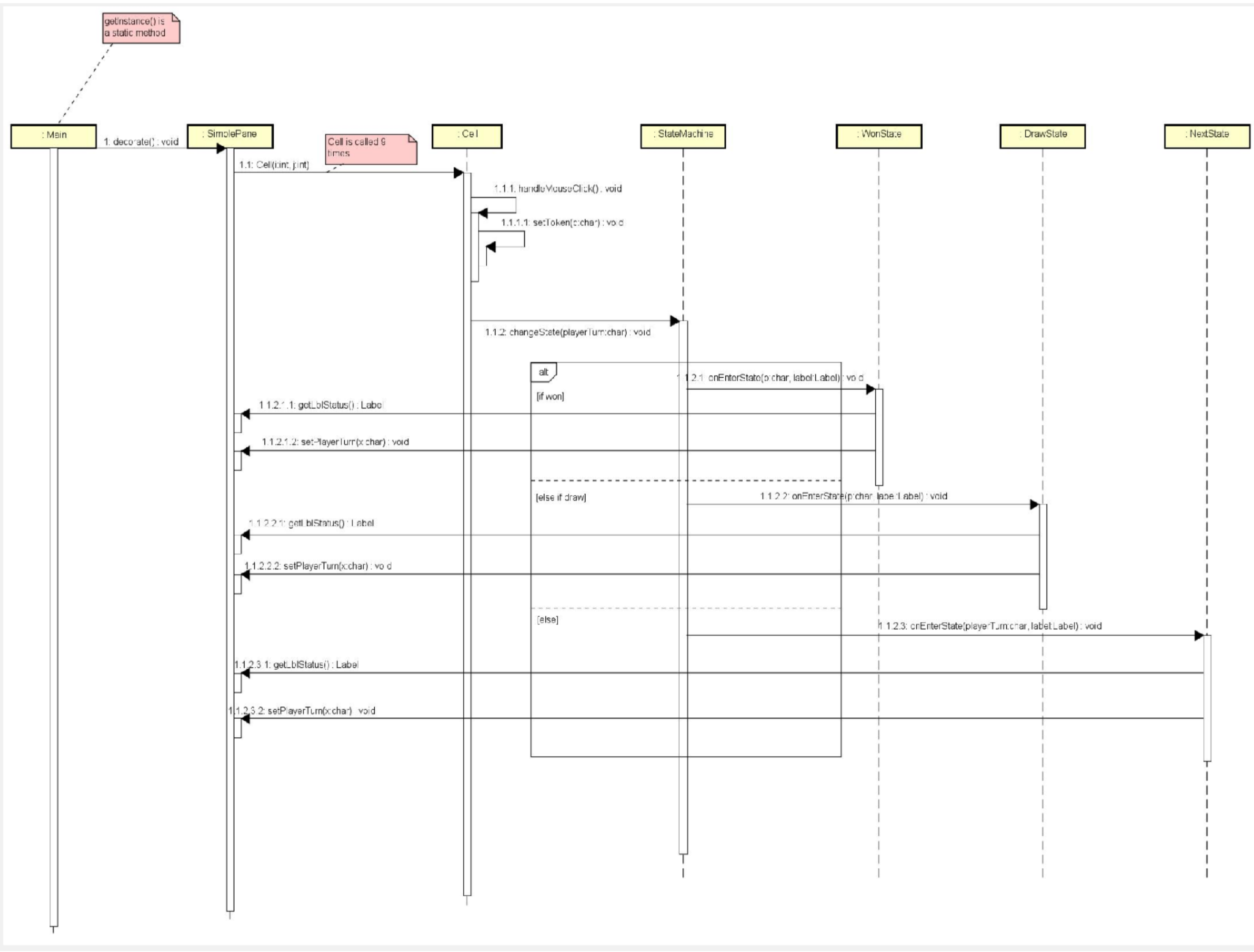
The architecture's focus for the Advanced Tic Tac Toe game development project is centered on the following listed core components of the game. The Game represents the core of the game feature, responsible for managing all the states of the game and steps of the players' moves. The Player component represent the player that will play the game, marking it respectively with an assigned marker and a respective game strategy. The Board represents the scene in 2D grid where the actual game will be played, representivising the state of the board cells in the given grid. The Playing Pieces reflects the players' markers in the game by differentiating between two types of playing pieces: Piece X and Piece O. The technology components to build the project has a strong enough technology stack for the interactive and dynamic working of the game . The Front End which is always the part of the system that a user interacts with via interface , including input devices like keyboards, mice or touchscreens .This is a platform that uses JavaFX which is very powerful toolkit to build a more compelling graphical user interface thats responds to user actions and provides rich visual classes could be regarded as a good fit for user interaction with the context, and secondly to help with some animation that is done to spice up the application . i Chose J Eve because it is one liberal and unlike its older sibling,Java ,this language a little less strict which means that you’re free to be more creative also allows for writing of code that is dynamic and fluid and provide support for animation and end up with an application that puts a smile. The back end also uses java to make sure that the game logic and state is handling what needs to be handled with a strong foundation The Back End not only has superior capability to perform such actions but is also a format that tailors to create a back end that supports complex and tedious game logic and state handling game. Reason being that as game gets more involved with multiple and complex rules also requires more extensive features of game play like the use of real time game state updates.

Class Diagram:


pkg
- successor
Handler
successor
- successor
CheckForColumnsHandler
+ handleRequest(p : SimplePane, token : char) : boolean
+ setSuccessor(next : Handler) : void
CheckForDiagonalsHandler
+ handleRequest(p : SimplePane, token : char) : boolean
+ setSuccessor(next : Handler) : void
CheckForRowsHandler
+ handleRequest(p : SimplePane, token : char) : boolean
+ setSuccessor(next : Handler) : void
StandardPane
~ StandardPane(pane : Component)
+ decorate() : void
DecoratorPane
~ DecoratorPane(pane : Component)
+ decorate() : void
+ addButtons(buttons : CommandButton) : void
+ getPane() : Pane
AdvancedPane
~ AdvancedPane(pane : Component)
+ decorate() : void
Originator
- shape : char
- row : int
- column : int
+ setState(shape : char, row : int, column : int) : void
+ saveStateToMemento() : Memento
+ getStateFromMemento(memento : Memento) : void
+ getShape() : char
+ getRow() : int
+ getColumn() : int
- pane
~ originator
Component
ICommand
SimplePane
- playerTurn : char = 'X'
- isAdvancedMode : boolean
+ getInstance(isAdvancedMode : boolean) : Component
- SimplePane(bool : boolean)
+ decorate() : void
+ addButtons(buttons : CommandButton) : void
+ newGame() : void
+ setPane(shape : char, i : int, j : int) : void
+ checkCells() : boolean
+ getPane() : Pane
+ getLblStatus() : Label
+ getCells() : Cell[][]
+ isWon(token : char) : boolean
+ isDraw() : boolean
CommandButton
+ CommandButton(text : String)
+ setPane(myPane : SimplePane) : void
+ Execute() : void
+ getP() : SimplePane
+ setP(p : SimplePane) : void
- p
~ careTaker
CareTaker
- undoIndex : int
- redoIndex : int
+ CareTaker()
+ addToUndoList(state : Memento) : void
+ addToRedoList(state : Memento) : void
+ removeLastUndo() : Memento
+ removeLastRedo() : Memento
+ getMementoUndoListSize() : int
+ getMementoRedoListSize() : int
+ getUndoArrayList() : List<Memento>
+ getRedoArrayList() : List<Memento>
+ getUndoMemento() : Memento
+ getRedoMemento() : Memento
NewButton
+ NewButton(text : String)
+ Execute() : void
RedoButton
+ RedoButton(text : String)
+ Execute() : void
UndoButton
+ UndoButton(text : String)
+ Execute() : void
* {ordered}
- undoMementoList
* {ordered}
- redoMementoList
Memento
- shape : char
- row : int
- column : int
+ Memento(shape : char, row : int, column : int)
+ getShape() : char
+ getRow() : int
+ getColumn() : int
Main
+ start(primaryStage : Stage) : void
+ main(args : String[]) : void

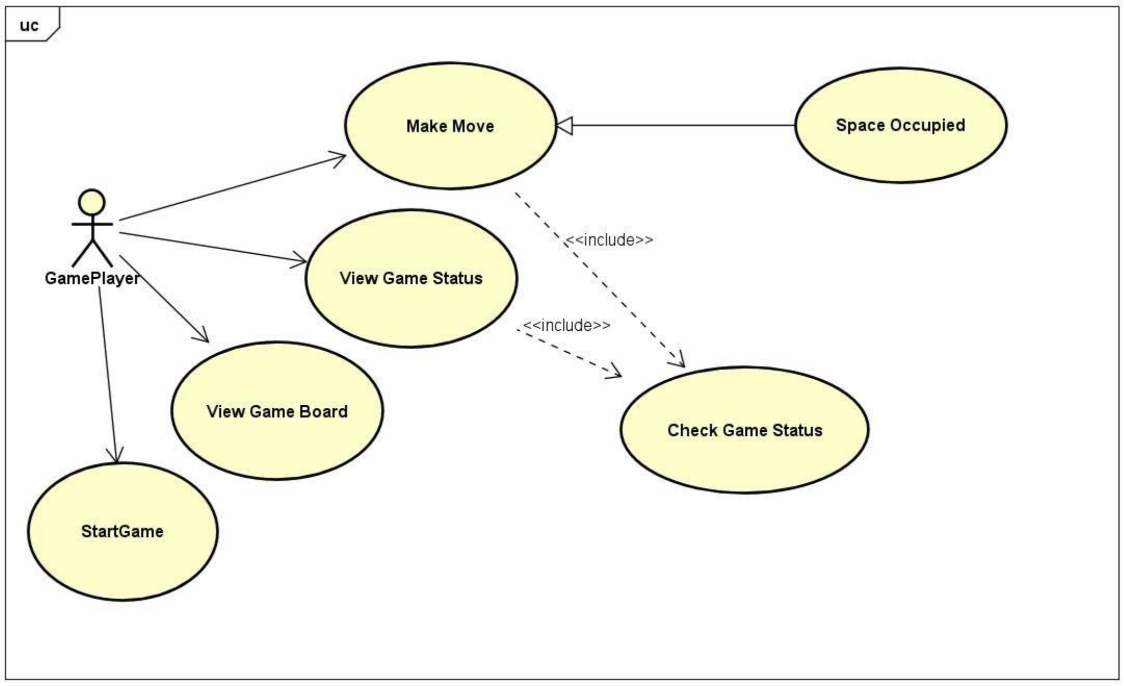
This class diagram depicts the design of a software application that uses multiple design patterns to improve the functionality and maintainability of the software. There is use of Chain of Responsibility pattern via handlers CheckForColumnsHandler, CheckForRowsHandler and CheckForDiagonalsHandler and HandlerCommand for processing of different kind of requests. Decorator and Composite patterns by Component and DecoratorPane can dynamically add or remove responsibilities on GUI elements, while Command pattern via ICommand and CommandButton can control the actions and encapsulate them into objects to achieve the actions, hence making addition and removal of commands easier without depending on its knowledge. Memento pattern by Originator, Memento and CareTaker can manage states for undo n redo. The pivot of everything is Showing and SimplePane, that links everything thus indicating its significance in GUI management and maybe the logic associated with the game/app With this thoughtful design in place, the architecture becomes flexible and extensible. Most probably this can be a game or any interactive application that needs complex interactions to perform an action, along with managing states of different objects. The GUI needs to be defined with extending other objects or inbuilt components customised dynamically.

Sequence Diagrams:

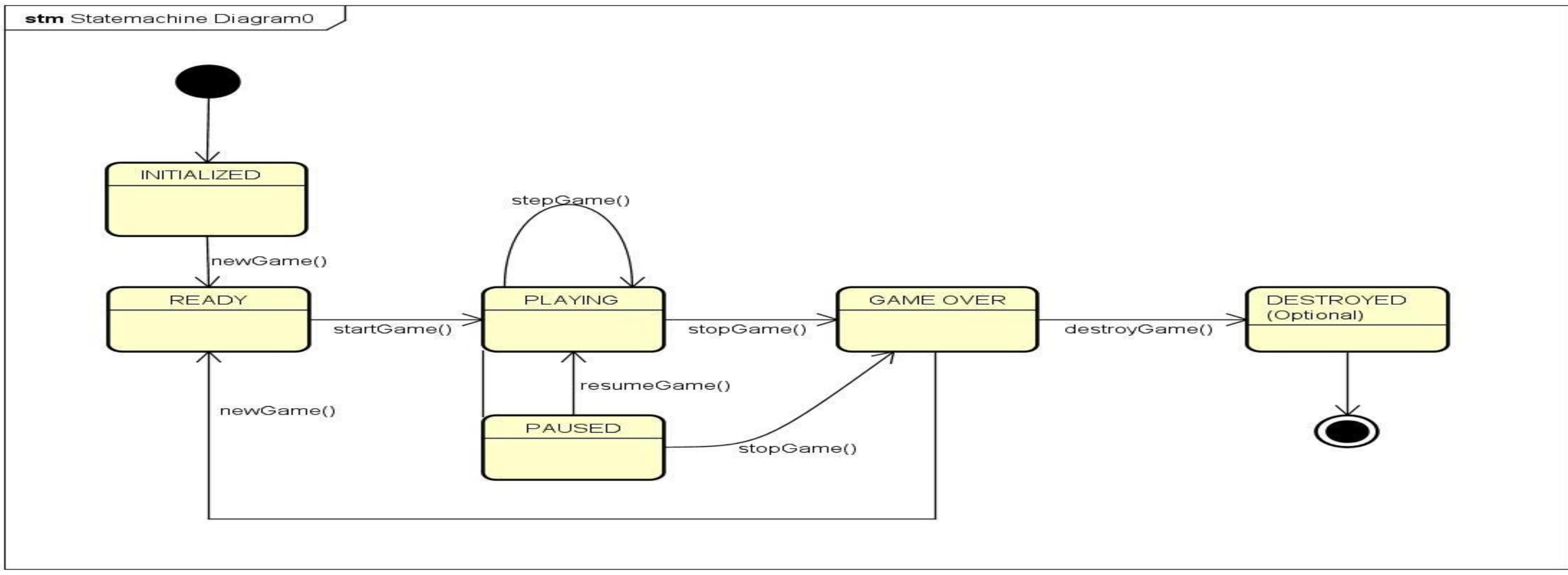


getInstance() is a static method
: Main
: SimplePane
: Cell
: StateMachine
: WonState
: DrawState
: NextState
1: decorate() : void
Cell is called 9 times
1.1: Cell(i:int, j:int)
1.1.1: handleMouseClick() : void
1.1.1.1: setToken(c:char) : void
1.1.2: changeState(playerTurn:char) : void
alt
[if won]
1.1.2.1: onEnterState(p:char, label:Label) : void
1.1.2.1.1: getLblStatus() : Label
1.1.2.1.2: setPlayerTurn(x:char) : void
[else if draw]
1.1.2.2: onEnterState(p:char, label:Label) : void
1.1.2.2.1: getLblStatus() : Label
1.1.2.2.2: setPlayerTurn(x:char) : void
[else]
1.1.2.3: onEnterState(playerTurn:char, label:Label) : void
1.1.2.3.1: getLblStatus() : Label
1.1.2.3.2: setPlayerTurn(x:char) : void

UseCase Diagram:



State Diagram:



Design Patterns used in the project:

Strategy Pattern
 Behavioural design patternstrategy: helps you to define a family of algorithms, encapsulate each one, and make them interchangeable via their objects.

Decorator: it is used to add responsibilities to a object in a dynamic way. In this project useing Decorator concept for displaying 2 types of game modes, Standard Mode and Advance Mode. Playing Standard mode: tic tacs to play the original three by three game with only a option to reset game .Playing Advance mode: replacing tic tac toe counters to undo ,redo with game play with features of standard mode.

Chain of responsibility Chain of responsibility is a design pattern applied to avoid coupling the sender of a request to its receiver by giving more than one object a shot at handling the request. The receiving objects chain up and the request is passed among the objects until an object handles it.Using this pattern for Checking the winner of the game in this project. So we have three handlers that one of them check if sequence of X's or O's are there in any row, then if not check if there is sequence of X's or O's in any column and if not check if there is sequence of X's or O's in any of the diagonals or anti diagonals.

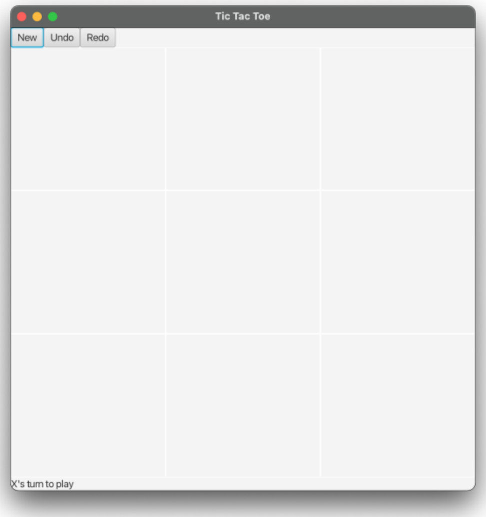
Command Pattern Indeed, most turn-based games feature an ‘undo’ button – often because of clumsy touch recognition, which is particularly prevalent in mobile game development. When you are designing an action to be undertaken by the human player, it’s more efficient for them to get things wrong every now and then, and be able to easily reverse a move, than to ask them ‘are you sure you want to perform action X?’ every time they do anything. So, i have implemented 3 buttons for this game. New Game button Undo Button Redo Button This Command Pattern comes into play when the player clicks on any of the above buttons to make the system to perform the appropriate logic. Memento Memento was used to encapsulate and materialize the state of the object from the inside to be later restored from the outside. In this project, useing both Command and Memento pattern to rollback the last moves of Tic Tac Toe. We can Undo() and Redo(the move multiple times here, and even if i do it a lot, i won’t ever lose what state the object is in. recording its state every time a move is made, because now we have the last known state of it so we can now move back.

State Pattern State makes it possible for an object to behave differently when its state changed. The object will look like it no longer belongs to the class that it once belonged to.For the Tic-Tac-Toe game every move can go into one of 3 states. Draw Win Next Move - Continue The states keep running until game is over. We inspect the current state and the state of the machine by the game check every turn.

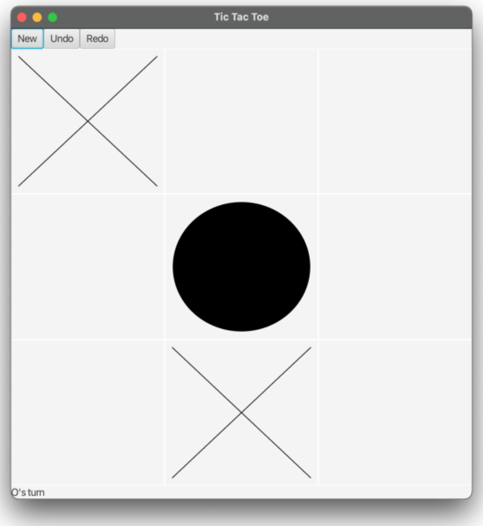
Singleton Make sure there’s only one instance of a class, and provide a global place to access it. As our object is not created multiple time, Singleton has a private constructor. retrieving the global access of our object with a static method, which checks if its creation has been done and return the actual state of the object. There is only one instance of object.

Game screenshots:

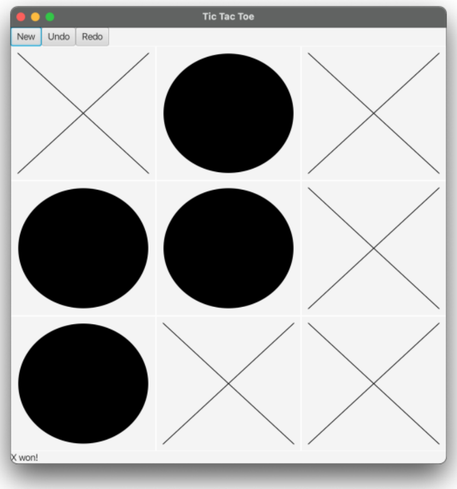
Advanced Mode:



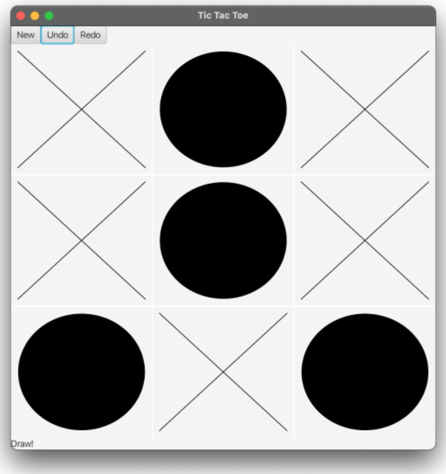
X's turn to play



O's turn



X won!



Draw!

References:

1. Bud, J. (2002). Applying Patterns to Develop Extensible and Maintainable Game Engines. IEEE Software.
2. Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley Professional.
3. Riehle, D. (1996). The Event Notification Pattern – Integrating Implicit Invocation with Object-Oriented Systems. Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).
4. Nystrom, Robert. (2014). Game Programming Patterns.
5. Krasner, G. E., & Pope, S. T. (1988). A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming.