Tic Tac Toe

Contents

[1. How to approach Schema Design 2](#_Toc178093521)

[Cardinality 2](#_Toc178093522)

[Steps to approach Schema Design 5](#_Toc178093523)

[MVC (Model, View, Controller) Architecture 6](#_Toc178093524)

[2. Desing Tic-Tac-Toe 8](#_Toc178093525)

[Undo Feature 9](#_Toc178093526)

[Approach 1 9](#_Toc178093527)

[Approach 2 10](#_Toc178093528)

[Approach 3 10](#_Toc178093529)

[Summary: 10](#_Toc178093530)

[Class Diagram 10](#_Toc178093531)

[Design Patterns that will be used 14](#_Toc178093532)

[3. Winning Strategy 15](#_Toc178093533)

[Approach 1: 15](#_Toc178093534)

[Approach 2: 15](#_Toc178093535)

[Approach 3: 16](#_Toc178093536)

[Approach 4: 16](#_Toc178093537)

[MVC Recap 16](#_Toc178093538)

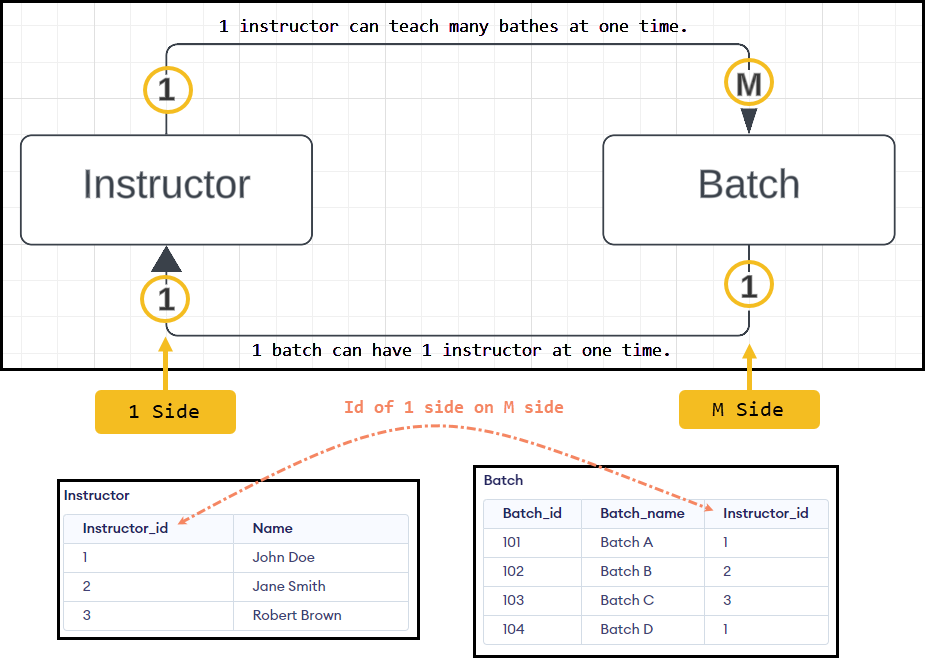
Code and Notes are @ <https://github.com/nishithjain/Design_TicTacToe>

We are not going to use Schema Design in Tic-tac-toe. Before we design Tic-tac-toe, let’s revise schema design. This will help us in the upcoming low level design problems.

# How to approach Schema Design

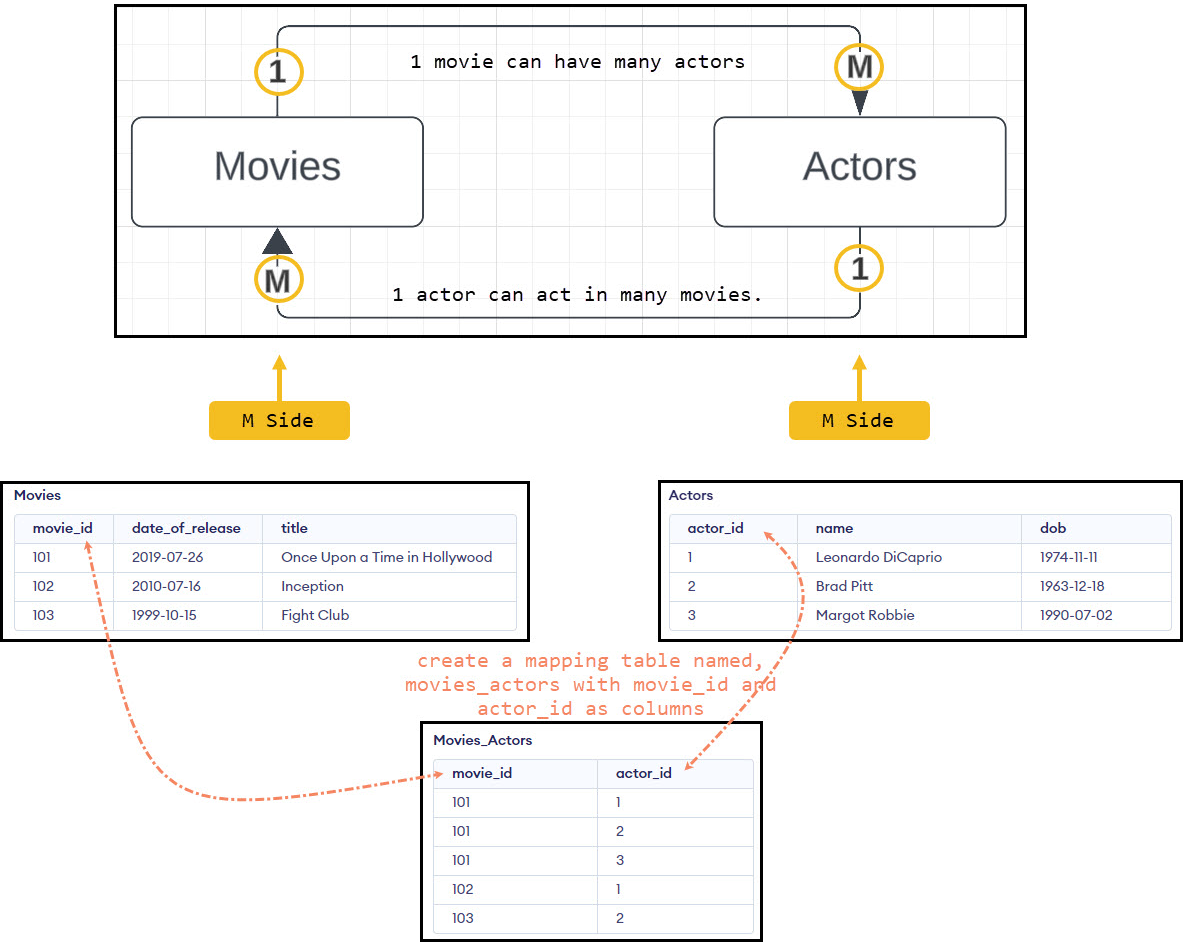
## Cardinality

* Let’s understand what is cardinality before approaching the design.
* *Cardinality* refers to the relationship between two entities. It defines how entities in one table relate to entities in another.
* There are 4 main types of cardinalities in database schema design. Let’s see how we represent the relation in a table:
  + 1:1 -> Id of 1 side on the other side.
  + 1:M Or M:1-> Id of 1 side on “M” side.
    - Example: Instructor and Batch



|  |
| --- |
| -- Create Instructor table without auto-increment on Instructor\_id  CREATE TABLE Instructor (      Instructor\_id INT PRIMARY KEY,  -- Primary key, no auto-increment      Name VARCHAR(100) NOT NULL      -- Example additional column for Instructor name  );  -- Create Batch table without auto-increment on Batch\_id  CREATE TABLE Batch (      Batch\_id INT PRIMARY KEY,         -- Primary key, no auto-increment      Batch\_name VARCHAR(100) NOT NULL, -- Example additional column for Batch name      Instructor\_id INT,                -- Foreign key to Instructor table      FOREIGN KEY (Instructor\_id) REFERENCES Instructor(Instructor\_id)  );  -- Insert values into the Instructor table  INSERT INTO Instructor (Instructor\_id, Name)  VALUES (1, 'John Doe'),         (2, 'Jane Smith'),         (3, 'Robert Brown');  -- Insert values into the Batch table with associated Instructor\_id  INSERT INTO Batch (Batch\_id, Batch\_name, Instructor\_id)  VALUES (101, 'Batch A', 1),  -- Batch A taught by Instructor John Doe (ID 1)         (102, 'Batch B', 2),  -- Batch B taught by Instructor Jane Smith (ID 2)         (103, 'Batch C', 3),  -- Batch C taught by Instructor Robert Brown (ID 3)         (104, 'Batch D', 1);  -- Batch D also taught by Instructor John Doe (ID 1) |

* + M:M -> Mapping table.
    - Let’s take an example of movies and actors.
    - 1 movie can have many actors.
    - 1 actor can act in many movies.



|  |
| --- |
| -- Create Actors table without auto-increment on actor\_id  CREATE TABLE Actors (      actor\_id INT PRIMARY KEY,    -- Primary key, no auto-increment      name VARCHAR(100) NOT NULL,  -- Name of the actor      dob DATE                     -- Date of birth of the actor  );  -- Create Movies table without auto-increment on movie\_id  CREATE TABLE Movies (      movie\_id INT PRIMARY KEY,          -- Primary key, no auto-increment      date\_of\_release DATE,              -- Release date of the movie      title VARCHAR(150) NOT NULL        -- Title of the movie  );  -- Create the mapping table to represent M:M relationship between Actors and Movies  CREATE TABLE Movies\_Actors (      movie\_id INT,                       -- Foreign key referencing Movies table      actor\_id INT,                       -- Foreign key referencing Actors table      PRIMARY KEY (movie\_id, actor\_id),   -- Composite primary key      FOREIGN KEY (movie\_id) REFERENCES Movies(movie\_id),      FOREIGN KEY (actor\_id) REFERENCES Actors(actor\_id)  );  -- Insert values into the Actors table  INSERT INTO Actors (actor\_id, name, dob)  VALUES (1, 'Leonardo DiCaprio', '1974-11-11'),         (2, 'Brad Pitt', '1963-12-18'),         (3, 'Margot Robbie', '1990-07-02');  -- Insert values into the Movies table  INSERT INTO Movies (movie\_id, date\_of\_release, title)  VALUES (101, '2019-07-26', 'Once Upon a Time in Hollywood'),         (102, '2010-07-16', 'Inception'),         (103, '1999-10-15', 'Fight Club');  -- Insert mappings between actors and movies in the Movies\_Actors table  INSERT INTO Movies\_Actors (movie\_id, actor\_id)  VALUES (101, 1),  -- Leonardo DiCaprio in 'Once Upon a Time in Hollywood'         (101, 2),  -- Brad Pitt in 'Once Upon a Time in Hollywood'         (101, 3),  -- Margot Robbie in 'Once Upon a Time in Hollywood'         (102, 1),  -- Leonardo DiCaprio in 'Inception'         (103, 2);  -- Brad Pitt in 'Fight Club' |

## Steps to approach Schema Design

Step 1: For each class in the class diagram, we will create a table. (Class can be called as entity/model)

Step 2: For each primitive attribute (int, bool, float, string) of a class, create a column in the respective table.

Step 3: For non-primitive attributes, find the cardinality of the relation and then apply its rule.

|  |
| --- |
| class Actor {  int id\_; // Primitive attribute.  string name\_; // Primitive attribute.  string dob\_; // Primitive attribute.  };  class Movie  {  int id\_; // Primitive attribute.  string date\_; // Primitive attribute.  string title\_; // Primitive attribute.  vector<Actor> actors\_; // Non-primitive attribute.  }; |

* + Create 2 tables Movies and Actor.
  + In the above case, it is M:M relation. Hence create a mapping table…

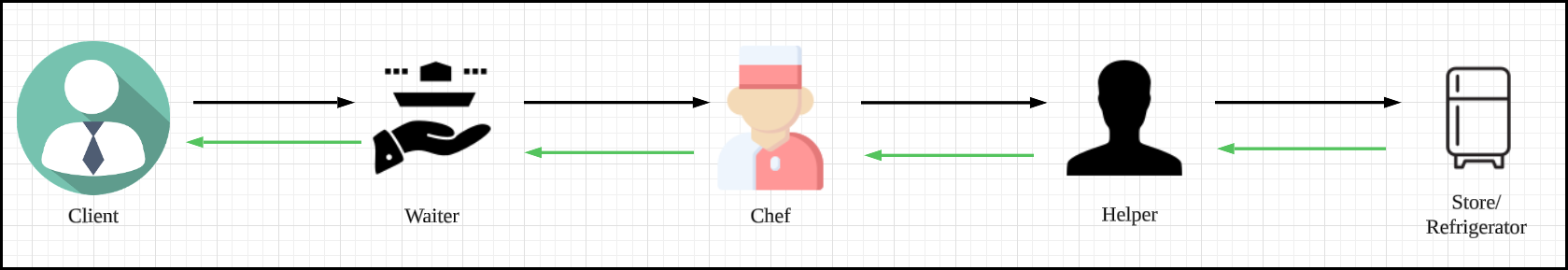
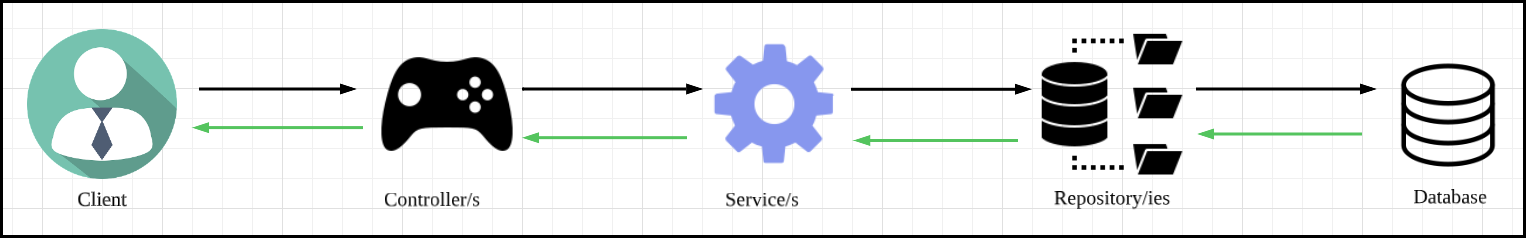
Step 4: How to write code?

* Imagine we have come up with 8-10 requirements. It is impossible to code all the requirements in the interview. The metric on which we will be evaluated is…
  + Project Structure: Our codebase should have proper structure, i.e. different folders for different type of files.

|  |
| --- |
| Example:  ├───src  │ ├───controllers # Handles user input and request processing  │ ├───models # Data structures, classes, and logic  │ ├───repositories # Database access or file management  │ ├───services # Core business logic  │ └───utils # Utility functions or helper code  │  ├───include # Header files (.h/.hpp) for public interfaces  ├───tests # Unit and integration tests  ├───lib # External libraries (optional)  ├───build # Compiled binaries (optional, usually ignored in version control)  └───CMakeLists.txt # Build system config (for CMake) |

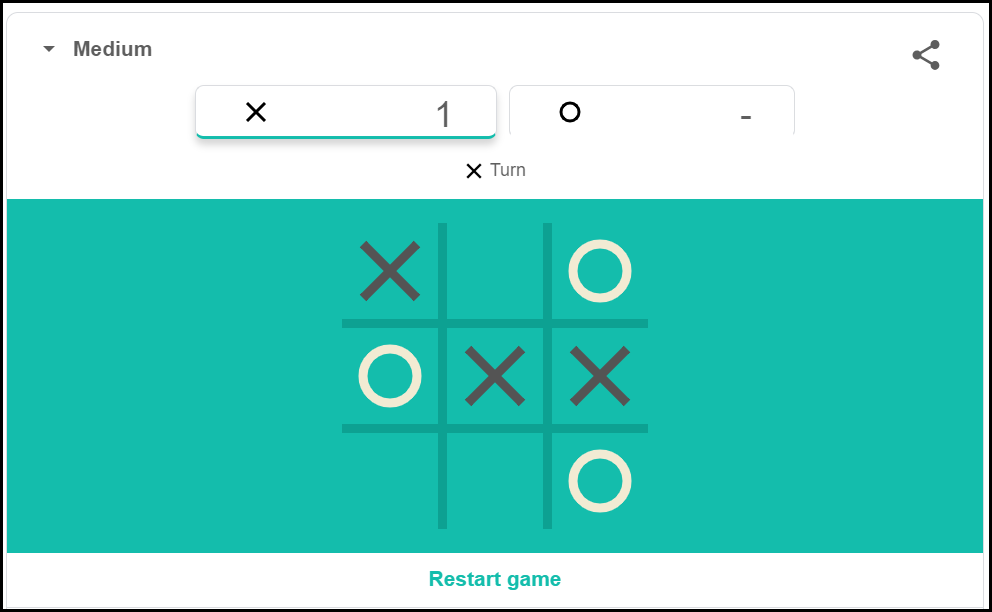
* + Instead of implementing all the requirements, implement some of the requirements and it should be in the working state.
    - Code all the models (classes that we came up with in the class diagram).
    - Start coding requirement by requirement.
  + Next up is coding controllers, services, etc. For this we need to understand MVC architecture.

### MVC (Model, View, Controller) Architecture

* Views will not be studied as this is related to UI.
* Models are Entities (Classes that we have come up with in the class diagram).
* To understand MVC, let's understand with an example.
  + If we go to restaurant, we give the order to waiter. We don't give the order to chef directly.
  + Client gives order to Waiter.
  + Waiter takes the order to Chef.
  + Chef takes the help of Helper to get the items from store and prepares the order.
  + Chef is the one who is doing the hard work.
* If we think, the restaurant is the system, then waiter is the first point of contact with the system. This is called as controller.
* Controller is the first point of contact with the client for the system.
* Sometimes, the controllers may be UI, backend, can be anything.
* Controller is not doing any hard work.
* Controller is the interface through which clients will interacts with our system. Controllers are very lightweight.
* It takes the input from the client and passes on to the person who is doing the hard work (Service/Business logic/Algorithms).
* The service will interact with the helper to process the request. The helper is called as repository. Basically, repositories are the classes that will be used to interact with database. They are also called as **DAO** (Data Access Object).
* What are models? Models are nothing but tables in the database.
* Basically, for every object, a row in the database is created. For example, if there is a Movie class in the code, and there is a corresponding Movies table in the database, for every movie, there will be a movie object and corresponding row in the Movies table.
* There will be multiple controllers, multiple services and multiple repositories.
  + For example, to interact with User, we have user controller, we have user service and user repository. i.e. for every entity, we have respective controller, respective service and respective repository.

# Desing Tic-Tac-Toe

* In any design problem, we will have a situation where…
  + We know the system.
  + We don't know the system.
* If we know the system, align with the requirements.
* If we don’t know the system, gather the requirements and clarify the requirements.
* Let’s gather the requirements for Tic-Tac-Toe…

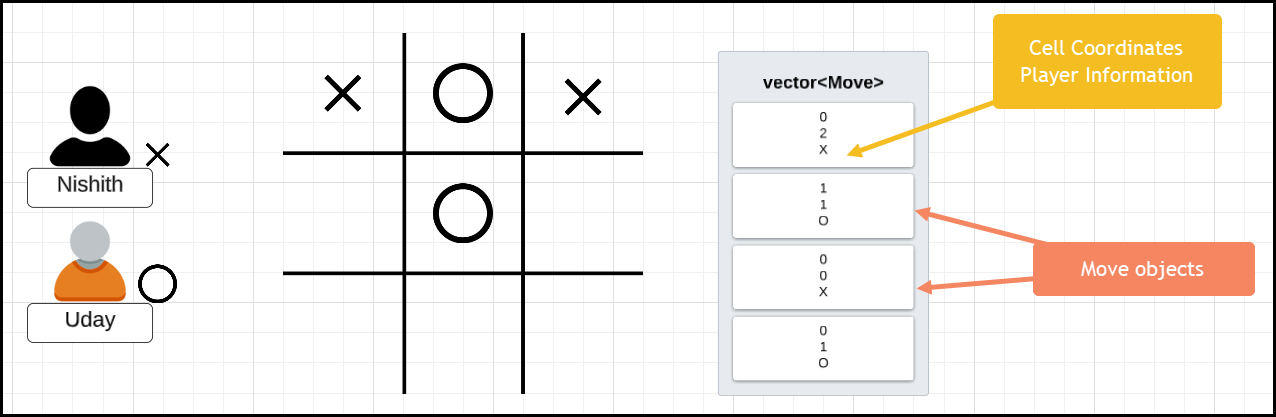


* 1. Size of the board is N x N (N>=3)
  2. Number of players will be N-1.
  3. Player can choose their symbol, but no 2 players are allowed to choose the same symbol.
  4. Will there be any bot?
     + Yes. If yes,
     + How many bots? Only 1 bot is allowed per game.
  5. Bot can have different difficulty level (Easy, Medium, Hard)
  6. Who will start the game?
     + At the start of the game, we will randomize the list of players and players will play in that order.
  7. How to check the winner?
     + A player wins the game if he has his symbol across a row/column/diagonal.
  8. When the game will end?
     + Someone wins the game
     + Draw
  9. Quit/Reset feature is not available.
  10. No timer is allowed.
  11. Leaderboard feature is not available.
  12. Undo - Most recent move will be undone.

## Undo Feature

* Undo feature requirements
  + The Undo will be a global button.
  + Any player can press this button any number of times.
  + When Undo button is pressed, the most recent move will be undone.
* Analysis:
  + If we have to undo the move, we have to store the move.
  + How to store them?
    - We can use list to store the moves. The list acts as stack.

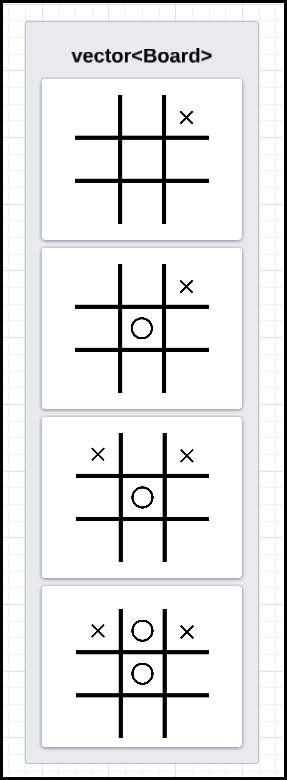
vector<Move> moves;

* + If we are making a move, what all the information is required? We need...
    - Cell Information on which the move is made.
    - Player who has made the move.
  + There are multiple ways to implement undo feature...

### Approach 1

* If we want to undo the most recent move, we have to remove the last entry from the list (vector<Move> moves;).
  + Remove the latest move from the list.
  + Remove the symbol from the cell as well. (Undo the state of the board)
  + Undoing in the game like chess is not easy.
  + This is not the best way to approach.

### Approach 2

* Create an empty board.
* Redo all the n-1 moves. (All the moves except the last move)
  + Time complexity is higher… (Order of number of moves)
* This is the easiest way to approach.

### Approach 3

* Resetting the state of the board is difficult in approach 1.
* We can also create a list which has state of the board…
* Here space complexity is higher…

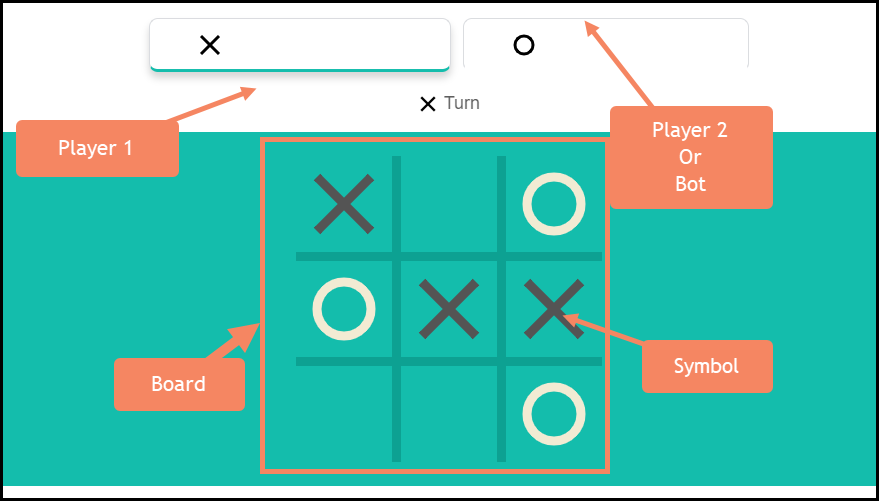
### Summary:

* If undoing the move from the board is easy, go for approach 1.
  + Approach 1:
    - Store list of moves
    - Remove the last move from the list of moves.
    - Remove the last move from board also.
* If reversing is not easy like in the game of chess, go with approach 2.
  + Approach 2:
    - Store list of moves.
    - Create an empty board.
    - Redo all the n-1 moves.
* Approach 3:
  + Store list of moves as well as list of board status for each move.
  + Remove the latest move and the latest board status from the list.

## Class Diagram

1. Find the nouns .
2. Visualize the system.

* In class diagrams, we represent entities, interfaces, enums, etc.
* We also have to represent design patterns.
* Let’s visualize the game…
* We will go from outer side to inside…



|  |
| --- |
| class TicTacToeGame  {  GameBoard board\_;  std::vector<Player> players\_;  std::vector<Move> moves\_;  int next\_player\_index\_;  GameStatus game\_status\_;  Player winner\_;  };  enum class GameStatus  {  IN\_PROGRESS,  ENDED,  DRAW,  }; |

* All the entities shown above are part of game. Hence there will be a TicTacToeGame class.
  + Next, inside the game, there will be Board, hence there will be GameBoard class.
  + Next, it will have many players. So, vector<Player>.
  + Game will store list of moves. So, vector<Move>.
  + Game should maintain next player turn.
  + Game should also have game status. It can be an enum (IN\_PROGRESSS, ENDED, DRAW).
  + Game can also have who is the winner.
* Next, let’s see what will be there in GameBoard class.
  + This board is nothing but it is a 2D matrix.
  + It can be std::vector<std::vector<char>> board\_;. But, just storing a char is not good idea.
  + Suppose in the future, we want to know if the cell state (Whether cell is empty, blocked, etc.), it is not possible. Hence create a separate class Cell and store std::vector<std::vector<Cell>> board\_;.

|  |
| --- |
| class GameBoard  {  std::vector<std::vector<char>> board\_;  size\_t size\_; // Size of the Board.  }; |

* Let’s see what will be there in Cell class.
  + The row number of the cell.
  + Column number of the cell.
  + Instead of storing the just symbol (X or O or #, etc..) , its better to store the Player object. It will be difficult to know who is the Player in that cell if we just put the symbol.
  + We need to store the status of the Cell, whether it is empty or filled.

|  |  |
| --- | --- |
| class Cell  {  int row\_;  int col\_;  Player player\_;  CellState cell\_state\_;  }; | enum class CellState  {  EMPTY,  FILLED,  }; |

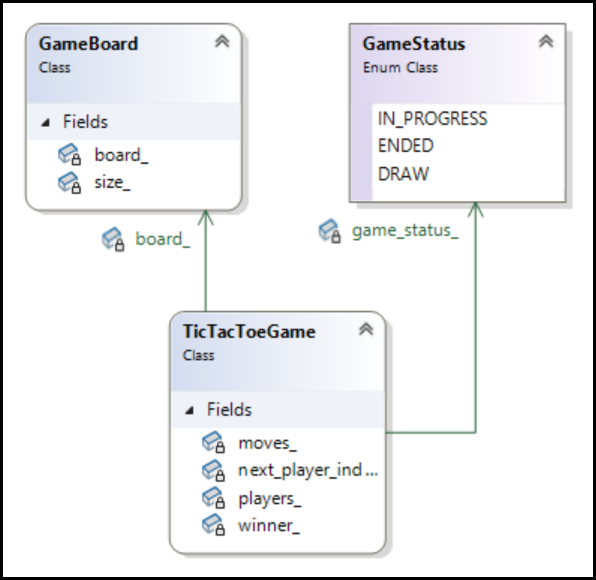
|  |
| --- |
| class Player  {  std::string name\_;  char symbol\_;  int id\_;  PlayerType player\_type\_;  };  enum class PlayerType  {  HUMAN,  BOT,  }; |

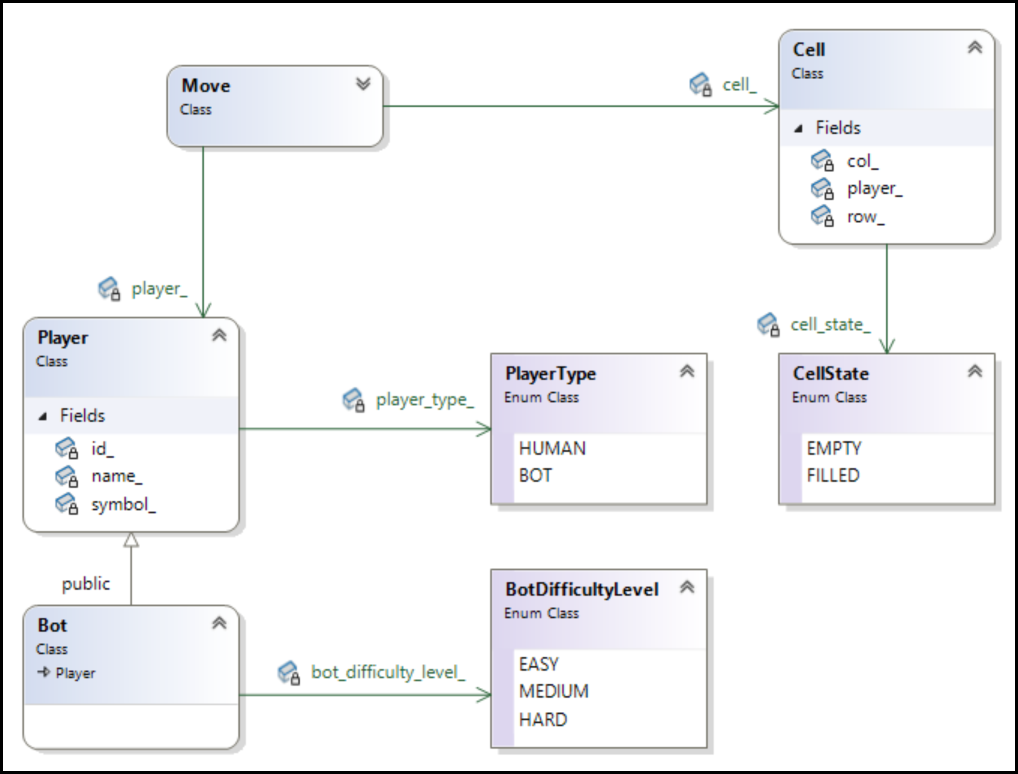
* Let’s see the next important class, Player.
  + Name of the player.
  + Symbol of the player.
  + ID of the player.
  + Whether it is a bot player or not.
* What about bot?
  + Additional to Player attributes, Bot can have difficulty level. Hence Bot class can extend the Player class.

|  |  |
| --- | --- |
| class Bot : public Player  {  BotDifficultyLevel bot\_difficulty\_level\_;  }; | enum class BotDifficultyLevel  {  EASY,  MEDIUM,  HARD,  }; |

* The only thing that is remaining is Move class.
  + We can have an object of Cell.
  + We can have an object of Player. (Since Cell itself is having the player object, is it required here or not, let’s see).

|  |
| --- |
| class Move  {  Cell cell\_;  Player player\_; // Let’s see, if this is required or not.  }; |





## Design Patterns that will be used

* Undo can be implemented in 3 different ways. If required, Undo can be implemented using **Strategy Design Pattern**.
* To create the game object, we can use **Builder Design Pattern**. This is because,
  + It has lot have lot of attributes.
  + Validations:
    - No 2 players should have the same symbol.
    - Number of players = Dimension – 1.
  + For Game, we can use GameBuilder Design Pattern.
* Bot, there can be 3 different types of Bots, EasyBot, MediumBot and HardBot, these will be making move in different ways. Hence, we can use **Strategy Design Pattern**.
  + We can implement BotPlayingStrategy.
  + Based on user input we decide the type of Bot. Hence, we can use **Factory Design Pattern**.
* There are 3 ways a player can win. Hence, we can use **Strategy Design Pattern**…
  + Row winning strategy.
  + Column winning strategy.
  + Diagonal winning strategy.

# Winning Strategy

* How to check if a Player has won the game…
* A Player is considered to be a winner if it has same symbol across…
  + Any row
  + Any column
  + Any diagonal
* Algorithm to check the winner.
  + First, we will see O(N3) -> O(N2) -> O(N) -> O(1).

## Approach 1:

* For each player, check all the rows, all the columns and both the diagonals.

|  |
| --- |
| for (auto p : players) // N-1 Players  {  // If there is a winner in the row...  bool found = true;  for (i = 0; i < N; i++) { // N rows  for (j = 0; j < N; j++) { // N Columns  if (mat[i][j] != p.symbol) {  break;  }  }  if (j == N) return true;  } // N\*N Complexity  // If there is a winner in the column...  {  } // N\*N Complexity  // If there is a winner in the diagonal...  {  } // 2N Complexity  } |

* Total Time Complexity:

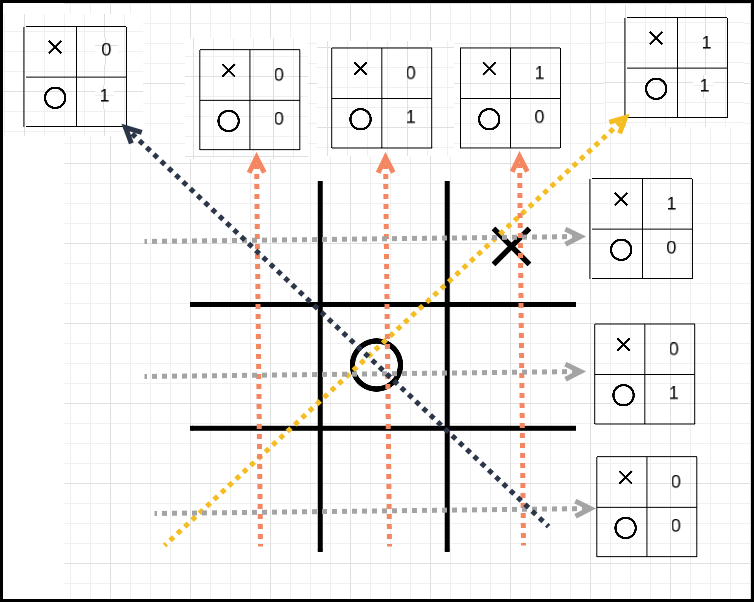
## Approach 2:

* If a player is playing and it is their turn, then they have a chance to win. No other player can win since it is not their turn.
* So, for current player P,
  + Check all the rows.
  + Check all the columns.
  + Check both the diagonals
* Total Time Complexity:

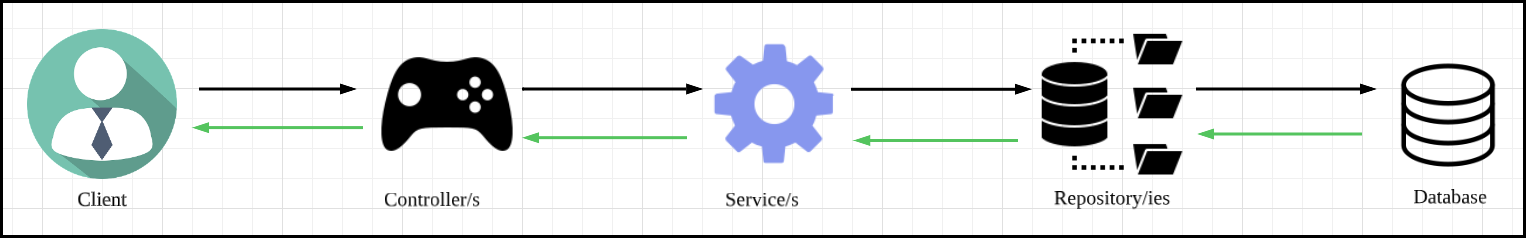
## Approach 3:

* A win can only occur in the row, column, or diagonal that includes the current cell.
* For every move …
  + Check the current row.
  + Check the current column.
  + Check the diagonals for which the current index is a part of.
* Total Time Complexity:

## Approach 4:

* For every row, we can maintain a map to store the count/occurrence of symbol.
* For every column, we can maintain a map to store the count/occurrence of symbol.
* For every diagonal, we can maintain a map to store the count/occurrence of symbol.
* Winning will happen if the count of the symbol of current player == N.
* Steps:
  + Increment the count in row map.
  + Increment the count in the column map.
  + Increment the count in the diagonal map if the current index is a part of any diagonal.
  + Check the map for the count.
* Total Time Complexity:
* Total Space Complexity:

## MVC Recap

* Model -> nothing but entities/classes. Classes that we described in the class diagram.
* Controllers -> Controller is the interface through which clients will interacts with system.
* Services -> Actual hard worker where business logic/algorithm comes.
* Repositories -> DAO (Data Access Object) repositories are the classes that will be used to interact with database.
* To implement tic-tac-toe, we don’t need database, hence ~~Repositories~~ won’t be there.
* All the logic can be incorporated in the game class itself, hence ~~Services~~ won’t be there.