COMP 1006/1406
Assignment 3 – TicTacToe (Part 1)

Due: Friday, August 4th 2006, before 11h55 pm

In this assignment, you will learn how to use the Subject/Observer design pattern in order to program a TicTacToe game. There will have two types of players: human and computer. The computer will use a recursive AI algorithm in order to decide which is the best place to play.

Part A (The Model) [30 pts]

Create a class called TicTacToe which implements the Subject interface as defined in the course notes. A TicTacToe has three private instance variables:

- **observers**: an ArrayList of observers;
- **squares**: a 3x3 two-dimensional array of **int** representing the game state;
- **currentTeam**: an **int** representing the next team to play.

The TicTacToe class also has three public static final variables, all of type **int**:

- NO_TEAM
- X
- O

each having a different value. These constants will be used to specify which team has used which square on the game board.

You have to:

- Create a constructor for the TicTacToe class which randomly decides which team will start the game. If the Xs start the game, then **currentTeam** is initialized with value X, if the Os start the game, then **currentTeam** is initialized with value O. All values in the **squares** array must be initialized with NO_TEAM. Once the game is done constructing, it should update its observers.
- Create getCurrentTeam() and getTeamAt(int **i**, int **j**) methods which respectively return the next team to play and the team which played on the square **i**, **j**. The return type of these methods should be **int**.
- Create a play(int **i**, int **j**) method which allows the current team to play on the square located on the i^{th} row and j^{th} column. The current team is then switched and the observers are updated.
- Create a **private** method isTeamWinning(int **team**) which returns a **boolean** indicating whether the team **team** has win. A team has win if it has played on three squares which are either vertically aligned, horizontally aligned, or form a diagonal.
- Create a **public** getWinner() method which return type is **int**. If the Xs are winning, then it returns X, if the Os are winning, then it returns Os, otherwise it returns NO_TEAM.
- Create a public isFinished() method which returns a **boolean** indicating whether the game is finished. The game is finished if either the Xs or the Os have win, **OR** there is no more free square on which to play.
- Finally, implement all of the methods you need in order to implement the Subject interface properly.
- Although it is not mandatory to provide test code, it is **strongly recommended** to do it at this point. If you do not provide test code and the rest of your assignment does not work perfectly, **you will get zero the whole assignment including this part**. You should provide test code that will convince the TA that your class is working properly.

Part B (The View) [15 pts]

Create a class called TicTacToePanel which extends JPanel and implements the Observer class as defined in the course notes as well as the ActionListener interface (for the Controller part). For now, a TicTacToePanel will have two private instance variables:

- **game**: of type TicTacToe;
- **buttons**: a 3x3 two-dimensional array of JButtons displaying the game state and allowing a human player to play.

You have to:

- Create a zero argument constructor which creates the 9 JButtons and dispose them appropriately. The TicTacToePanel should listen on every ActionEvent generated by the JButtons.
- In the update() method, set the text of every JButtons in order for it to display the appropriate team. For example, if the Xs have played in the top-left corner, then the text on the button which is at the top-left corner should be “X”.

Create a class called TicTacToeFrame which extends JFrame. A TicTacToeFrame contains a TicTacToePanel.

You have to:

- Create a zero-argument constructor for the TicTacToeFrame class.
- Create a main method which constructs a TicTacToeFrame and displays it.

Part C (The Players) [15 pts]

Create an abstract class called Player which implements Observer. The update() method will be defined in the sub-classes. A Player has two private instance variables:

- **team**: of type **int**, representing the Player's team;
- **game**: of type TicTacToe, representing the game on which the player plays. We say that the player is **bounded** to that game.

You have to:

- Create the getters for each instance variables.
- Create a constructor which takes a team (of type **int**) and a game (of type TicTacToe) as arguments.
- Define an abstract void method handleClick(int i, int j, TicTacToe game) which will tell the player that a button has been pressed. Since the method is abstract, you do not define any behavior for it now. That method will be implemented differently by each of the subclasses.

Create a class called HumanPlayer which extends Player. You have to:

- Create a constructor which takes a team (of type **int**) and a game (of type TicTacToe) in argument. This constructor will make a call to the superclass constructor.
- Implement the handleClick() method.
    - If the game given as argument is not the one to which the player is bounded, then the method just returns.
    - If the current team is not the same as the player's one, then the method just returns.
    - If the game is finished, then the method just returns.
    - If the square that has been clicked already has a team on it, then the method just returns.
    - Otherwise, the player plays on the square that has just been clicked.
- The update() method does nothing.

Create a class called AIPlayer which extends Player. You have to:

- Create a constructor which takes a team (of type **int**) and a game (of type TicTacToe) in argument. This constructor will make a call to the superclass constructor.
- Create a private void play() method which seeks for the next available square and plays on that square. Later on (in Part E), that algorithm will be replaced by something more "intelligent".
- Implement the update() method.
    - If the current team is not the same as the player's one, then the method just returns.
    - If the game is finished, then the method just returns.
    - Otherwise, is calls the play() method.
- The handleClick() method does nothing.

Part D (The Controller) [20 pts]

We will now glue the players, the game and the view together. In order to do so, we will add code to our previously created TicTacToePanel class. You have to:

- Add two more instance variables to the TicTacToePanel class: xPlayer and oPlayer, both of type Player.
- Create a private method createPlayer(int **team**) which returns a Player. By mean of the showInputDialog method of JOptionPane, the method will prompt the user for which kind of player he wants for the team **team** (human or computer). The user should make his choice by mean of a combo box. Your program should force the user to make a choice. For example, if he clicks the close button in the top right corner, he should be prompted again until he makes a choice. Once the user has made his choice, the appropriate player should be created. Its team should be **team** and its game should be the one contained in the TicTacToePanel. Also, the player should be registered as an observer to the game. Once all this is done, the player is returned.
- Create a private void method called newGame(). This method should first initialize the **game** variable of the TicTacToePanel. Then, the xPlayer and oPlayer variables should be initialized using the createPlayer() method. The TicTacToePanel should be registered as an observer to the game. Remember that the starting team is randomly decided. The user should be informed of the starting team using the showMessageDialog() of the JOptionPane class. Once all this is done, make an explicit call to the updateObservers() method of the game object. This will start the game.
- Add a call to newGame() at the end of the TicTacToePanel constructor.
- In the ActionPerformed() method, make a call to the handleClick() methods of both the xPlayer and the oPlayer. The **i**, **j** values should be the row and the column of the button which generated the event. Remember that you can use the getSource() method on the event to figure out which button generated the event. The game argument should be the game which is contained in the TicTacToePanel.
- In the update() method, after you have set the text of the JButtons, if the game is finished, announce the winner to the user. You have to do it by using the showMessageDialog() of the JOptionPane class. Then, make a call to newGame().
- You should now have a fully working TicTacToe game! The only problem is that the computer player may be a little dumb... this is what we are going to fix right now!

Part E (The AI Player) [20 pts]

The strategy we will use is the following: for each possible move the player could do, we will count the number of winning scenarios this could lead to. For example, if a given move directly leads to a winning configuration, then the number of winning scenarios is 1. If it leads to a draw configuration, or a configuration in which the opponent wins, then the number of winning scenarios is 0. Otherwise, for every other available move, we will recursively count the number of winning scenarios and add them up. The AI player will then chose the move which gives him the maximum of winning scenarios.

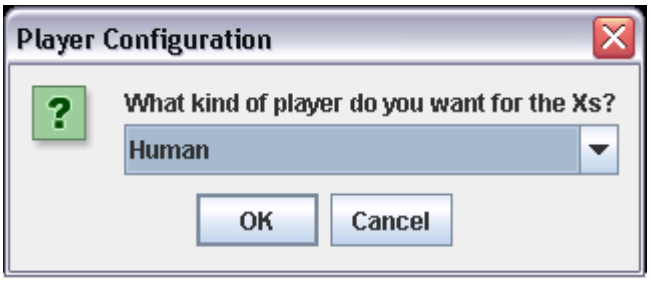
You have to:

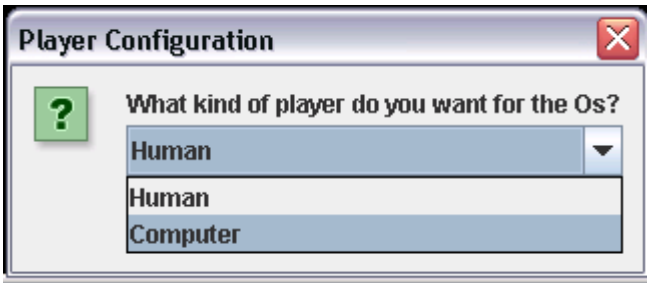
- In the AIPlayer class, create a private method called countWinning(TicTacToe game) which returns an integer. The algorithm you have to implement is the following:
    - if the game is finihed, then
        - if the winner is the AI player, return 1
        - otherwise return zero
    - otherwise
        - for each available position
            - make a copy of the game
            - play on that position
            - count the number of winning positions
        - return the sum of the number of winning positions.
- In order to implement this method, you will have to create a copy() method in the TicTacToe class. This method will create a new TicTacToe game, set the currentPlayer attribute to the copied game currentPlayer attribute, and copy each value of the squares array. **Warning: if you simply copy the whole array, your code will not work.**
- Modify the play() method in order for it to seek for the move which leads to the maximum number of winning scenarios. Do not forget to make a copy of the game each time you call the countWinning() method!

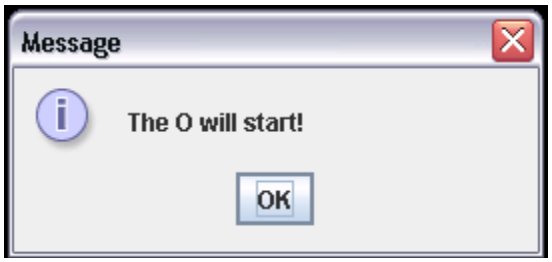
Bonus [20 pts]

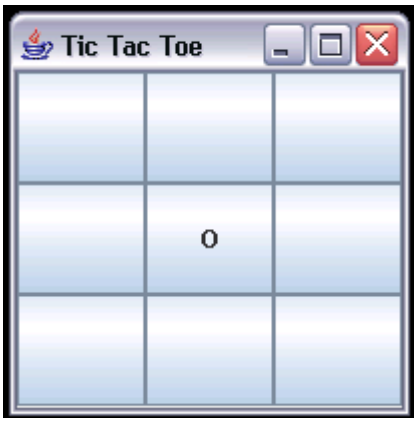
As you can see in Appendix 1, in certain cases, our AI player may not be that intelligent... can you say why? I will give you a hint: instead of counting the **maximum** number of **winning** scenarios, maybe we should have counted the **minimum** number of... something else! Rename the AIPlayer class to MAXPlayer and create a new one called one called MINPlayer, which implements the other strategy. Which one works the best?

Appendix 1: example



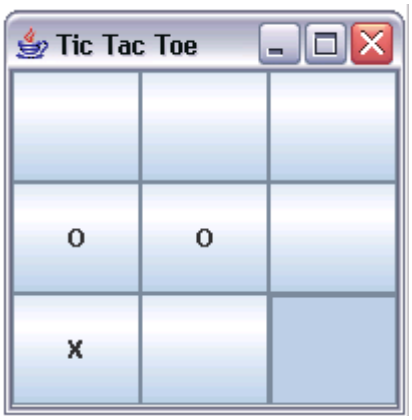




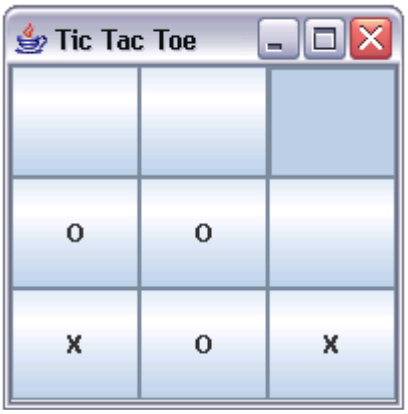



Tic Tac Toe
O


Tic Tac Toe
O
O
X


Tic Tac Toe
O
O
X

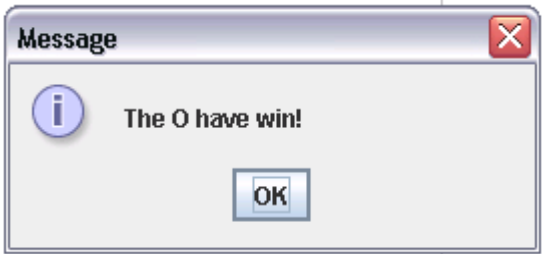

Tic Tac Toe
O
O
X
O
X

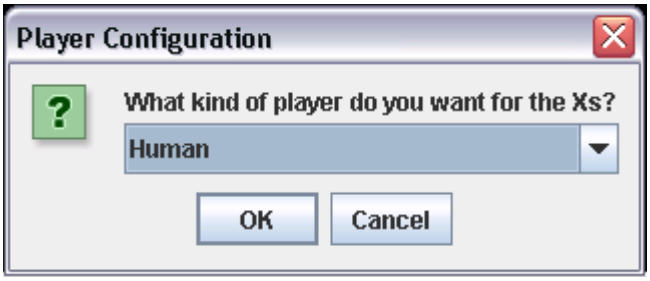

Tic Tac Toe
O O
X O X


Tic Tac Toe
O X
O O
X O X


Tic Tac Toe
O X
O O
X O X


Tic Tac Toe
O X X
O O O
X O X


Message
The O have win!
OK


Player Configuration
What kind of player do you want for the Xs?
Human
OK
Cancel

Submission

As usual, you will submit to the Raven system. You should submit an entire directory that contains all of your java files, your class files and any testing/output files that you created. Also, have a readme.txt file that clearly indicates which files are which.

Good Programming and Good Practice Requirements.

These requirements pertain regardless of what your application is supposed to do (i.e. regardless of the design requirements). These requirements are to ensure that your code is readable and maintainable by other programmers (or TA's in our case), and that your program is robust (It does not crash from bad object references). You will loose 5 marks from your total assignment mark for each of the following requirements that are not satisfied. If you do not satisfy requirement R0 you will get nothing for the assignment.

R0) IMPORTANT Uniqueness Requirement. The solution and code you submit MUST be unique. That is, it cannot be a copy, or be too similar, to someone else's assignment, or other code found elsewhere. A mark of 0 will be assigned to any assignment that is judged by us or the TA's not to be unique.

R1) All of your variables, methods and classes should have meaningful names that reflect their purpose. Do not follow the convention common in math courses where they say things like: "let x be the number of customers and let y be the number of products...". Instead call your variables numberOfCustomers or numberOfProducts. Your program should not have any variables called "x" unless there is a good reason for them to be called "x". (One exception: It's OK to call simple for-loop counters i, j and k etc. when the context is clear.)

R2) All variables in your classes should be private, unless a specific design requirements asks for them to be public (which is unlikely). We will design objects that provide services to others through their public methods. How they store their variables is their own private business.

R3) Robustness Requirements: Your program should never crash because of a "null pointer exception". This exception means that you are using a variable that does not actually refer to an object. We instruct the TA's to try and crash your program for this reason so guard against it.

R4) Comments in your code must coincide with what the code actually does. It is a very common bug in industry for people to modify code and forget to modify the comments and so you end up with comments that say one thing and code that actually does another. (By the way, try not to over-comment your code; instead choose good variable names and method names which make the code more "self commenting".)

R5) Java 1.5 Requirement: Your code should compile with the Java 1.5SDK without warnings.