CPS2004
Object-Oriented Programming

Study-Unit Assignment

Compiled by: Marc Ferriggi (286397M)

Lecturer: Dr Jean-Paul Ebejer

Department of Computer Science

University of Malta

Table of Contents

Table of Contents 2

1. Tic-Tac-Toe 3

    1.1 – UML Diagram 3

    1.2 – Description 4

    1.3 – Test Cases 4

    1.4 – Critical Evaluations and Limitations 4

    1.5 – Extending the Game Engine to Support a Class Tournament 5

2. MonOOPoly 6

    2.1 – UML Diagram 6

    2.2 – Description 7

    2.3 - Test Cases 7

    2.4 – Critical Evaluations and Limitations 8

3. Suffix Trie and Tree 9

    3.1 – UML Diagram 9

    3.2 – Description 10

    3.3 – Test Cases 10

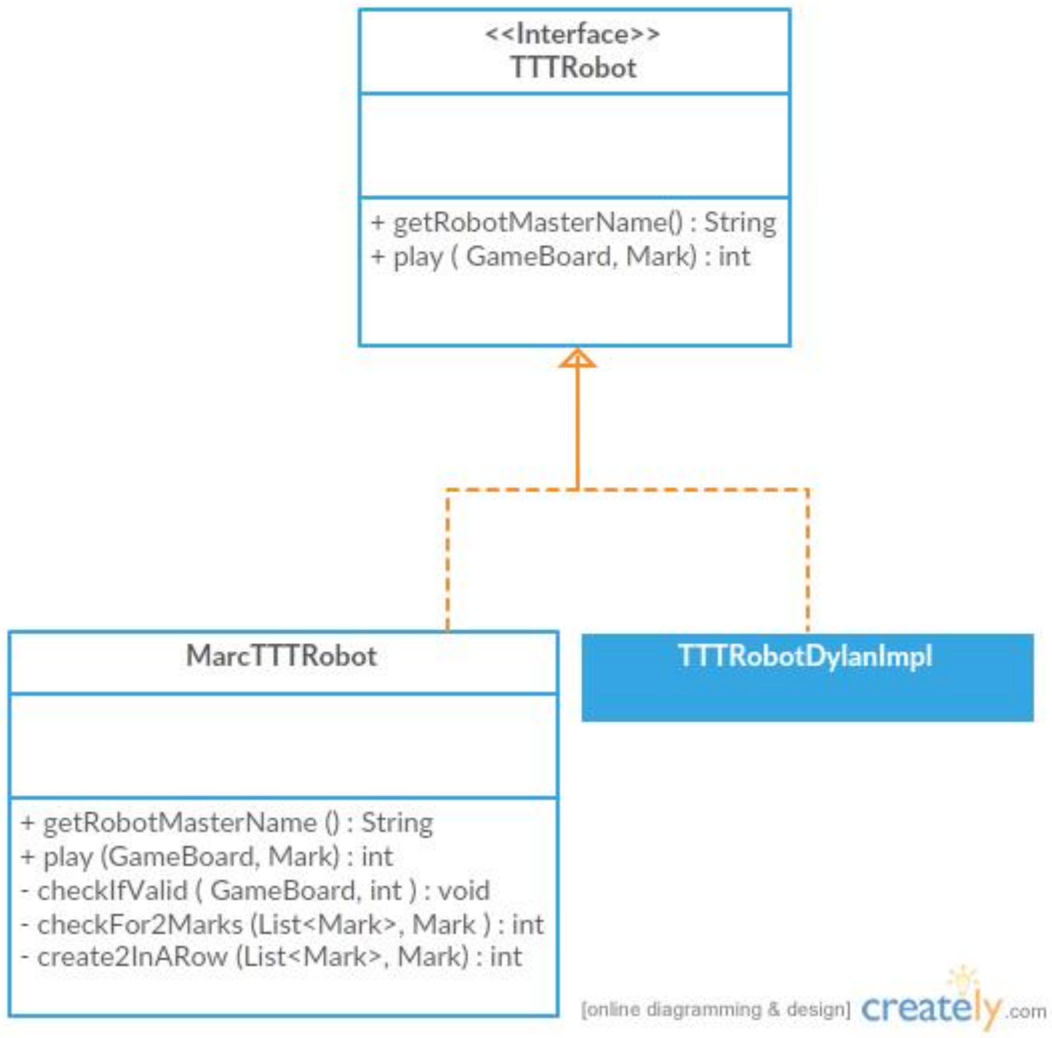
    3.4 – Critical Evaluations and Limitations 11

    3.5 – Making Use of the Suffix Trie (or Tree) Without Tying It to a Particular Implementation 11

References 12

1. Tic-Tac-Toe

1.1 – UML Diagram



<<Interface>>
TTTRobot
+ getRobotMasterName() : String
+ play (GameBoard, Mark) : int
MarcTTTRobot
+ getRobotMasterName () : String
+ play (GameBoard, Mark) : int
- checkIfValid (GameBoard, int) : void
- checkFor2Marks (List<Mark>, Mark) : int
- create2InARow (List<Mark>, Mark) : int
TTTRobotDylanImpl
[online diagramming & design] creately.com

1.2 – Description

On my team, I was tasked with implementing the interface TTTRobot to provide game logic for a Robot to play the Tic-Tac-Toe Game being implemented by my teammate. Since the task at hand was to implement a functioning robot, my only concern was with my created class called MarcTTTRobot and the interface provided (as can be seen in the UML diagram above). Before implementing my class, I also got familiar with all the other classes provided in the package to ensure that I was making the right checks and implementing the class correctly.

My plans for the way I implemented the robot were to make it in such a way that it would analyze the state of the game board and decide on an appropriate move based roughly off of the algorithm of playing a perfect tic-tac-toe game. Rather than being impossible to beat, I decided to try and mimic what Ostermiller calls an “Expert” player [1].

1.3 – Test Cases

For this task, testing was done using dry-runs since at the time of implementing the Robot, I did not have access to the working engine. However, after implementation of the engine, the robot was tested by my teammate and it was reported to me that everything worked well.

1.4 – Critical Evaluations and Limitations

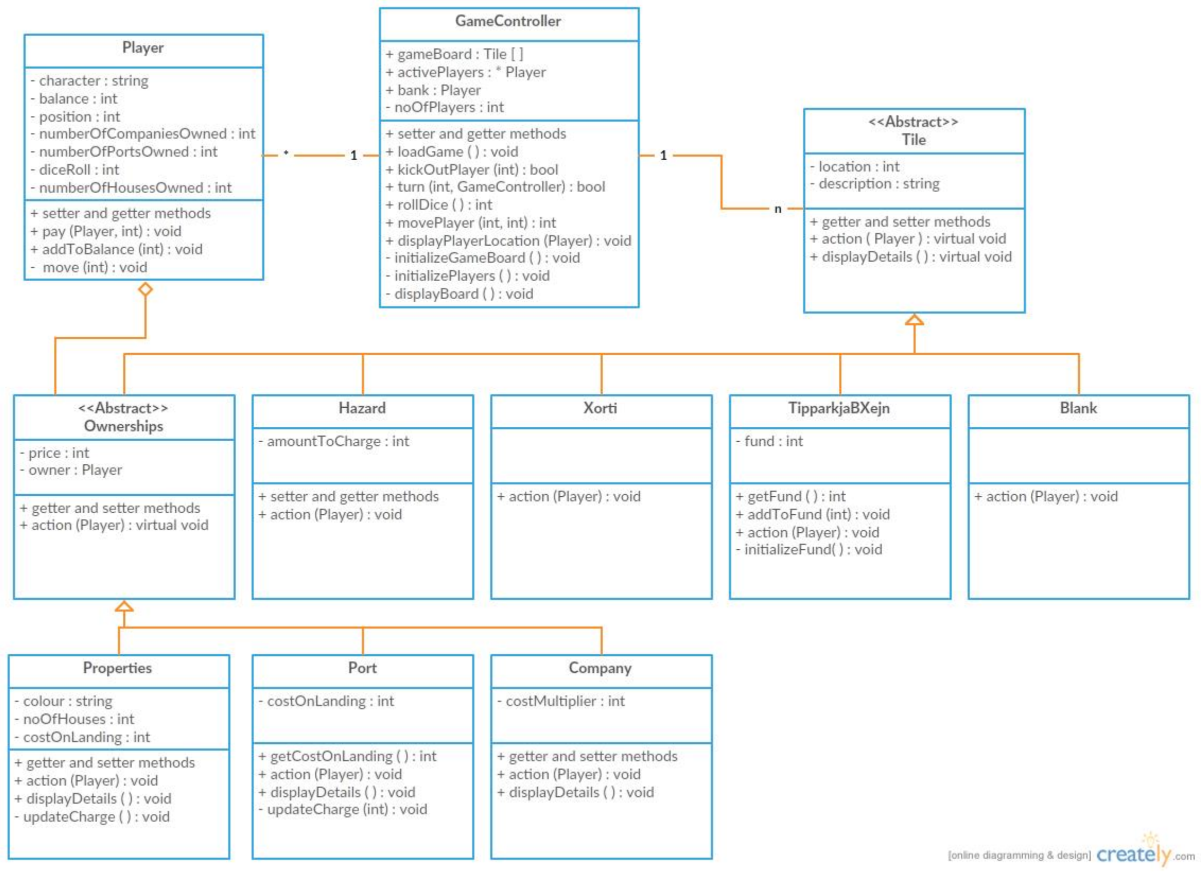
The biggest flaw with the implementation of my robot is that it would only work on a 3x3 table. This is due to the fact that the robot creates exactly 8 lists to mimic all the possible winning states and then checks these lists accordingly. Since all the robot’s logic is built around having the board be a 3x3 board, if we were to extend the board to be any different, then my implementation of the robot would not work.

1.5 – Extending the Game Engine to Support a Class Tournament

In order to extend the game engine to support a class tournament rather than just one game, the first step would be to store a list of TTTPlayers rather than simply having two. Note that the constructor must then also accept a list of TTTRobots. The players should also have a Boolean variable ‘won’ which is initially set to true and would be set to false if that player loses the game. An algorithm would then be implemented to pair up players to compete head to head. If ‘won’ is set to false then that player is removed from the list. The algorithm keeps repeating until only one player is left in the list and that player will be the overall winner. Note that in this case, having an odd number of players and the case of having a tie must be dealt with accordingly.

2. MonOOPoly

2.1 – UML Diagram


GameController
+ gameBoard : Tile [ ]
+ activePlayers : * Player
+ bank : Player
- noOfPlayers : int
+ setter and getter methods
+ loadGame ( ) : void
+ kickOutPlayer (int) : bool
+ turn (int, GameController) : bool
+ rollDice ( ) : int
+ movePlayer (int, int) : int
+ displayPlayerLocation (Player) : void
- initializeGameBoard ( ) : void
- initializePlayers ( ) : void
- displayBoard ( ) : void
Player
- character : string
- balance : int
- position : int
- numberOfCompaniesOwned : int
- numberOfPortsOwned : int
- diceRoll : int
- numberOfHousesOwned : int
+ setter and getter methods
+ pay (Player, int) : void
+ addToBalance (int) : void
- move (int) : void
*
1
1
n
<<Abstract>>
Tile
- location : int
- description : string
+ getter and setter methods
+ action (Player) : virtual void
+ displayDetails ( ) : virtual void
<<Abstract>>
Ownerships
- price : int
- owner : Player
+ getter and setter methods
+ action (Player) : virtual void
Hazard
- amountToCharge : int
+ setter and getter methods
+ action (Player) : void
Xorti
+ action (Player) : void
TipparkjaBXejn
- fund : int
+ getFund ( ) : int
+ addToFund (int) : void
+ action (Player) : void
- initializeFund( ) : void
Blank
+ action (Player) : void
Properties
- colour : string
- noOfHouses : int
- costOnLanding : int
+ getter and setter methods
+ action (Player) : void
+ displayDetails ( ) : void
- updateCharge ( ) : void
Port
- costOnLanding : int
+ getCostOnLanding ( ) : int
+ action (Player) : void
+ displayDetails ( ) : void
- updateCharge (int) : void
Company
- costMultiplier : int
+ getter and setter methods
+ action (Player) : void
+ displayDetails ( ) : void
[online diagramming & design] creately.com

2.2 – Description

Since the main aim of this task was to design and implement an OOP System with smaller working components interacting together, the main idea focused on was to properly design and implement the class diagram and program's structure which can be seen on page 6. My idea was to create a system that can have its features freely be changed around and have the rules of the game changed or bent as needed. Although the final implemented version handed in for the assignment is the traditional game of MonOOPoly as asked for in the assignment sheet, the system is implemented in a way where by simply adding a few methods, the locations of the tiles on the board could be randomized, the amount of players could be changed and certain variables like the player's balance and which players initially own which tiles could also be altered.

When looking at the structure of the player's turn, I split it up into three parts; "the pre-turn phase" where players have the time to buy houses, trade, and check their possessions, "the turn phase" where the dice is rolled, the player is moved to a new position and the player reacts to that new position, and the "post-turn phase" where the game controller checks the player's balance and if its negative, kicks that player out of the game. When the dice is rolled, I also based the algorithm of rolling the dice on the actual probability distribution of rolling two dice so as to give the game a more realistic feel [2].

2.3 - Test Cases

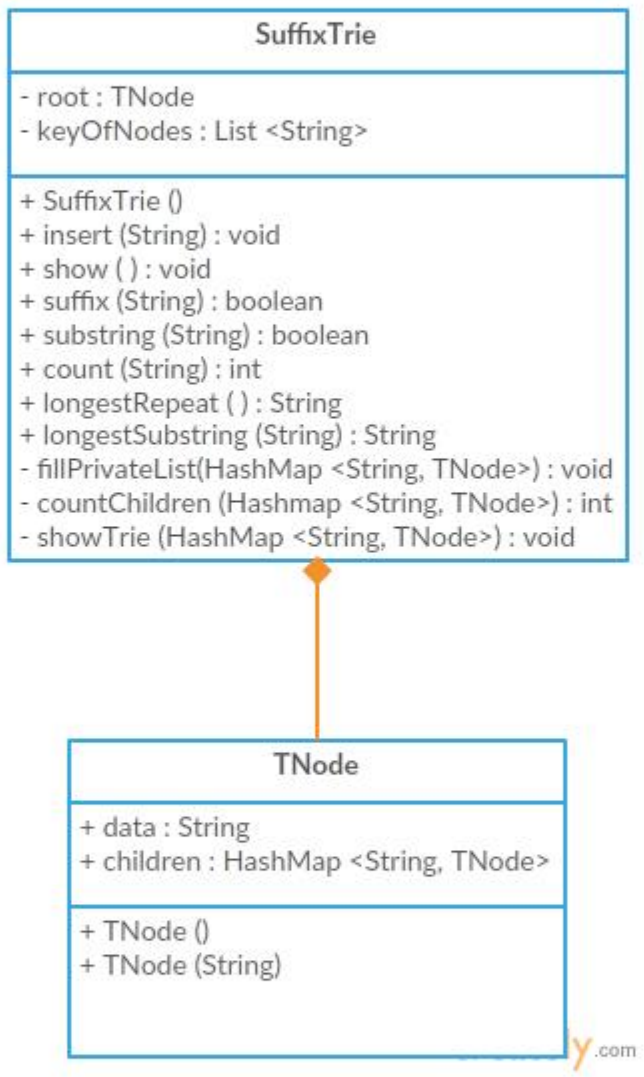
Testing was done to ensure the algorithms were working properly. Certain variables such as the player's balance were altered to ensure games didn't run for too long. A final test was then done where a full game was played between two players. Some noted issues and bugs were listed in section 2.4.

2.4 – Critical Evaluations and Limitations

Due to time constraints, certain features such as buying houses and trading properties have not been implemented. It is also important to note that the try, catch blocks on the user inputs only work when an incorrect value of the correct datatype is entered, entering the incorrect datatype leads to an infinite loop.

3. Suffix Trie and Tree

3.1 – UML Diagram


SuffixTrie
- root : TNode
- keyOfNodes : List <String>
+ SuffixTrie ()
+ insert (String) : void
+ show ( ) : void
+ suffix (String) : boolean
+ substring (String) : boolean
+ count (String) : int
+ longestRepeat ( ) : String
+ longestSubstring (String) : String
- fillPrivateList(HashMap <String, TNode>) : void
- countChildren (Hashmap <String, TNode>) : int
- showTrie (HashMap <String, TNode>) : void
TNode
+ data : String
+ children : HashMap <String, TNode>
+ TNode ()
+ TNode (String)

3.2 – Description

The suffix Trie was implemented as asked in the assignment sheet where each node in the trie contains the value of the spelled out word and a hash-map which contains all of that node's children. I decided to use the hash map data structure since the trie is unordered and has no limiting conditions on its depth. The trie was then searched and traversed using a depth-first recursive algorithm.

I decided to use Java as the programming language of choice for this task since Java has an extensive library of functions to work with Strings and provides a good interface for making use of hash maps. Another advantage of Java is its simplicity when it comes to accepting command line arguments.

3.3 – Test Cases

Initially, all functions were tested as they were being created to ensure that all functions were functioning properly. Once the entire program was working, the following table was created highlighting the test values, expected outputs and actual outputs.

Table 1: Test Table used to test suffix trie

Input	Actual Output	Expected Output
word.txt longestSubstring() haha	haha	haha
word.txt longestRepeat()	ha	ha
word longestRepeat()	Error opening file word	Error opening file word
word.txt substring()	Error, too few arguments have been entered. Terminating program	Error, too few arguments have been entered. Terminating program
word.txt suffix() hehaa	false	false

3.4 – Critical Evaluations and Limitations

Unfortunately, due to the way the suffix trie was implemented, where rather than storing a character at each node, the entire string was stored at each node, I was unable to successfully write an algorithm that optimizes the trie as asked for in the assignment sheet. The erroneous code which I have implemented to do so has still been handed in in the the package.

3.5 – Making Use of the Suffix Trie (or Tree) Without Tying It to a Particular Implementation

In order to make use of the suffix trie (or trie) without tying it to a specific implementation, the first step would be to create an interface for the trie or tree so as to ensure it will be of a certain structure and have the required methods. We can then build a system that will make use of the suffix trie or tree without it knowing what implementation it is using. The implementation of the class suffixTrie may then be changed freely as long as it implements the interface we would have created.

References

[1] (2016, July 20). *Tic-Tac-Toe Strategy*. Available: http://blog.ostermiller.org/tic-tac-toe-strategy.

[2] (2017, January 1). *Image: Probability Distribution for the sum of two six-sided dice*. Available: http://mathinsight.org/image/two_dice_distribution.