

CPS2004

-

Assignment

Full name: Miguel Dingli

I.D: 49997M

Course: B.Sc. (Hons) in Computing Science

Table of Contents

1 – Tic-Tac-Toe.....	2
1.1 – UML Diagram.....	2
1.2 – Approach	2
1.3 – Test Cases	3
1.4 – Critical Evaluation and Limitations	4
1.5 – Answer to Additional Questions.....	5
1.5.1 – Extending the Game Engine	5
1.5.2 – Interface for a Player of a Game.....	5
2 – MonOOPoly	6
2.1 – UML Diagram.....	6
2.2 – Approach	6
2.3 – Test Cases	9
2.4 – Critical Evaluation and Limitations	10
3 – Suffix Trie and Tree.....	11
3.1 – UML Diagram.....	11
3.2 – Approach	11
3.3 – Test Cases	13
3.4 – Critical Evaluation and Limitations	15
3.5 – Answer to Additional Questions.....	15
4 – References.....	15

Since a win can take a diagonal, horizontal, or vertical form, the method *calcPriority* was defined to consider one position at a time and check the row, column, and (if applicable) the diagonal on which the position lies. If the other two positions in a row, column, or diagonal are marked with an identical mark, then this means that the position is from medium to high priority. If these marks are the robot's marks, then the robot can win, while if they are the opponent's, then the robot should defend. Otherwise, no mark should be placed at the position. The set of marks that should be played at a position are input into *maxPriority*, which calculates the final priority of playing at the position.

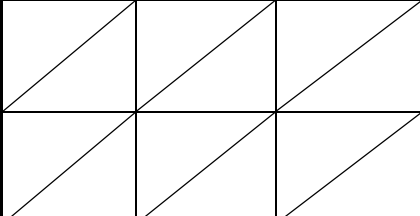
The first high priority empty position that is found is immediately marked, while any medium priority position discovered is noted, so that if no high priority position is found, the mark is placed at the medium priority position. If priority is overall low, the robot instead focuses on the corners and centre.

In general, no new **exceptions** were defined. Asserts were widely used to verify method arguments.

1.3 – Test Cases

Table 1 shows the test cases considered for this task. In the first two tests, '1' and '2' indicate two types of marks that can either be unique or identical. Such test cases were actually performed for all the four possible combinations of O's and X's. In the remaining tests, the robot's mark is assumed to be 'O' and the opponent's mark is assumed to be 'X'.

Table 1 – Test Cases for Task 1

Test 1 (horizontals)			Test 1 cont. (verticals)			Test 2 (diagonals)		
Input Board	Expected Outcome	Actual Outcome	Input Board	Expected Outcome	Actual Outcome	Input Board	Expected Outcome	Actual Outcome
1 1 _ _ _ _ _ _ _	1 1 2 _ _ _ _ _ _	1 1 2 _ _ _ _ _ _	1 _ _ 1 _ _ _ _ _	1 _ _ 1 _ _ 2 _ _	1 _ _ 1 _ _ 2 _ _	1 _ _ _ 1 _ _ _ _	1 _ _ _ 1 _ _ _ 2	1 _ _ _ 1 _ _ _ 2
_ _ _ 1 1 _ _ _ _	_ _ _ 1 1 2 _ _ _	_ _ _ 1 1 2 _ _ _	_ 1 _ _ 1 _ _ _ _	_ 1 _ _ 1 _ _ 2 _	_ 1 _ _ 1 _ _ 2 _	_ _ _ _ 1 _ _ _ 1	2 _ _ _ 1 _ _ _ 1	2 _ _ _ 1 _ _ _ 1
_ _ _ _ _ _ 1 1 _	_ _ _ _ _ _ 1 1 2	_ _ _ _ _ _ 1 1 2	_ _ 1 _ _ 1 _ _ _	_ _ 1 _ _ 1 _ _ 2	_ _ 1 _ _ 1 _ _ 2	1 _ _ _ _ _ _ _ 1	1 _ _ _ 2 _ _ _ 1	1 _ _ _ 2 _ _ _ 1
_ 1 1 _ _ _ _ _ _	2 1 1 _ _ _ _ _ _	2 1 1 _ _ _ _ _ _	_ _ _ 1 _ _ 1 _ _	2 _ _ 1 _ _ 1 _ _	2 _ _ 1 _ _ 1 _ _	_ _ 1 _ 1 _ _ _ _	_ _ 1 _ 1 _ 2 _ _	_ _ 1 _ 1 _ 2 _ _
_ _ _ _ 1 1 _ _ _	_ _ _ 2 1 1 _ _ _	_ _ _ 2 1 1 _ _ _	_ _ _ _ 1 _ _ 1 _	_ 2 _ _ 1 _ _ 1 _	_ 2 _ _ 1 _ _ 1 _	_ _ _ _ 1 _ 1 _ _	_ _ 2 _ 1 _ 1 _ _	_ _ 2 _ 1 _ 1 _ _
_ _ _ _ _ _ _ 1 1	_ _ _ _ _ _ 2 1 1	_ _ _ _ _ _ 2 1 1	_ _ _ _ _ 1 _ _ 1	_ _ 2 _ _ 1 _ _ 1	_ _ 2 _ _ 1 _ _ 1	_ _ 1 _ _ _ 1 _ _	_ _ 1 _ 2 _ 1 _ _	_ _ 1 _ 2 _ 1 _ _
1 _ 1 _ _ _ _ _ _	1 2 1 _ _ _ _ _ _	1 2 1 _ _ _ _ _ _	1 _ _ _ _ _ 1 _ _	1 _ _ 2 _ _ 1 _ _	1 _ _ 2 _ _ 1 _ _			
_ _ _ 1 _ 1 _ _ _	_ _ _ 1 2 1 _ _ _	_ _ _ 1 2 1 _ _ _	_ 1 _ _ _ _ _ 1 _	_ 1 _ 2 _ _ _ 1 _	_ 1 _ 2 _ _ _ 1 _			

- - -	- - -	- - -	- - 1	- - 1	- - 1			
- - -	- - -	- - -	- - -	- - 2	- - 2			
1 - 1	1 2 1	1 2 1	- - 1	- - 1	- - 1			
Test 3 (horizontal)			Test 3 (vertical)			Test 4 (other)		
Input Board	Expected Outcome	Actual Outcome	Input Board	Expected Outcome	Actual Outcome	Input Board	Expected Outcome	Actual Outcome
0 0 - X X - - - -	0 0 0 X X - - - -	0 0 0 X X - - - -	0 X - 0 X - - - -	0 X - 0 X - 0 - -	0 X - 0 X - 0 - -	- - - - - - - - -	0 - - - - - - - -	0 - - - - - - - -
- - - 0 0 - X X -	- - - 0 0 0 X X -	- - - 0 0 0 X X -	- 0 X - 0 X - - -	- 0 X - 0 X - 0 -	- 0 X - 0 X - 0 -	0 - - - - - - - -	0 - 0 - - - - - -	0 - 0 - - - - - -
X X - - - - 0 0 -	X X - - - - 0 0 0	X X - - - - 0 0 0	X - 0 X - 0 - - -	X - 0 X - 0 - 0 -	X - 0 X - 0 - 0 -	0 X 0 X - X 0 X 0	0 X 0 X 0 X 0 X 0	0 X 0 X 0 X 0 X 0
- 0 0 - X X - - -	0 0 0 - X X - - -	0 0 0 - X X - - -	- - - 0 X - 0 X -	0 - - 0 X - 0 X -	0 - - 0 X - 0 X -	X 0 X 0 - 0 X 0 X	X 0 X 0 0 0 X 0 X	X 0 X 0 0 0 X 0 X
- - - - 0 0 - X X	- - - 0 0 0 - X X	- - - 0 0 0 - X X	- - - - 0 X - 0 X	- 0 - - 0 X - 0 X	- 0 - - 0 X - 0 X			
- X X - - - - 0 0	- X X - - - 0 0 0	- X X - - - 0 0 0	- - - X - 0 X - 0	- - 0 X - 0 X - 0	- - 0 X - 0 X - 0			
0 - 0 X - X - - -	0 0 0 X - X - - -	0 0 0 X - X - - -	0 X - - - - 0 X -	0 X - - 0 - 0 X -	0 X - - 0 - 0 X -			
- - - 0 - 0 X - X	- - - 0 0 0 X - X	- - - 0 0 0 X - X	- 0 X - - - - 0 X	- 0 X - 0 - - 0 X	- 0 X - 0 - - 0 X			
X - X - - - 0 - 0	X - X - - - 0 0 0	X - X - - - 0 0 0	X - 0 - - - X - 0	X - 0 - 0 - X - 0	X - 0 - 0 - X - 0			

1.4 – Critical Evaluation and Limitations

The robot's logic can be described as opportunistic since its main logic is based on waiting for the right opportunity to place a mark at a particular position, rather than planning ahead. This means that an opponent that carefully considers all possibilities before placing a mark would win against this robot.

Secondly, the robot's logic assumes that the board will be made up of nine spaces. Although it still plays on larger or smaller game boards, it does not recognize winning or losing situations on such boards and simply marks the first empty position. A more general algorithm could have been used.

Finally, note that the robot is not designed to handle a board with zero empty positions since in such a case, it attempts to place a mark at the first empty position, which does not exist. The robot also expects the mark and board to never be null and that the game board is not biased (i.e. the difference between counts of marks of the two types placed on the board is at most 1).

1.5 – Answer to Additional Questions

1.5.1 – Extending the Game Engine

To extend the game engine such that a class tournament could be held to find the champion between all the robots developed, one should keep in mind to make use of existing code as much as possible. To achieve this, the only change that must be applied to the existing *TTTWarEngine* is to either add a getter for the winning robot, or have the method *play()* return the robot instance that wins the match.

Next, a *TTTTournament* class could be created. This class would take a list of all participating robots and would have a *startTournament()* method which, either sequentially or concurrently (for faster results), creates instances of a subclass of *TTTWarEngine* with pairs of robots from the list of robots as arguments to their constructors. It then invokes the engines' *play()* method and this same process can be repeated on the collection of winning robot instances, until the ultimate winner robot is found.

1.5.2 – Interface for a Player of a Game

The game chosen for this question is Checkers. For completeness, an additional class and an enum which would be found in the same package were included in Figure 2 below. The *CheckersRobot* simply defines a method *getCreatorName()* similar to the one seen in *TTTRobot*, which returns the name of the robot's player/creator/master and a method *play(...)* which takes the game board and the colour assigned to the player as an argument and returns an instance of *CheckersMove*. The implementation of the *play(...)* method would include the logic that the robot uses to analyse the current state of the board and to decide which piece to move and where to move it.

CheckersMove is simply a class which stores two pairs of coordinates that specify which piece the robot chose to move, and where the robot wishes to move that piece. Obviously, the move would have to be validated by the engine so as to prevent the robot from performing illegal moves. The *PieceColor* enum simply indicates which set of pieces belong to the robot.

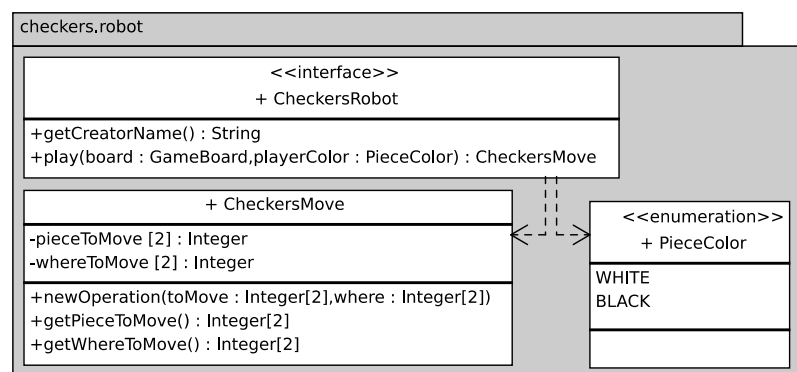


Figure 2 – UML Diagram for Additional Question

2 – MonOOPoly

2.1 – UML Diagram

Kindly refer to Figure 3 overleaf for the UML diagram showing the classes' makeup and interactions.

2.2 – Approach

Throughout this task, the rules defined in [1] were followed. Some elements of the game were, however, not incorporated. To mention some, firstly, the banker and bank's responsibilities are instead a part of the game engine. Secondly, there was no limit set on the number of houses available. "Income Tax" spaces do not offer the player the option to pay 10% of the total worth. Also, traded mortgaged properties do not need to be unmortgaged immediately by the new owner. And lastly, properties with houses built do not lose their houses when the owner gets bankrupt.

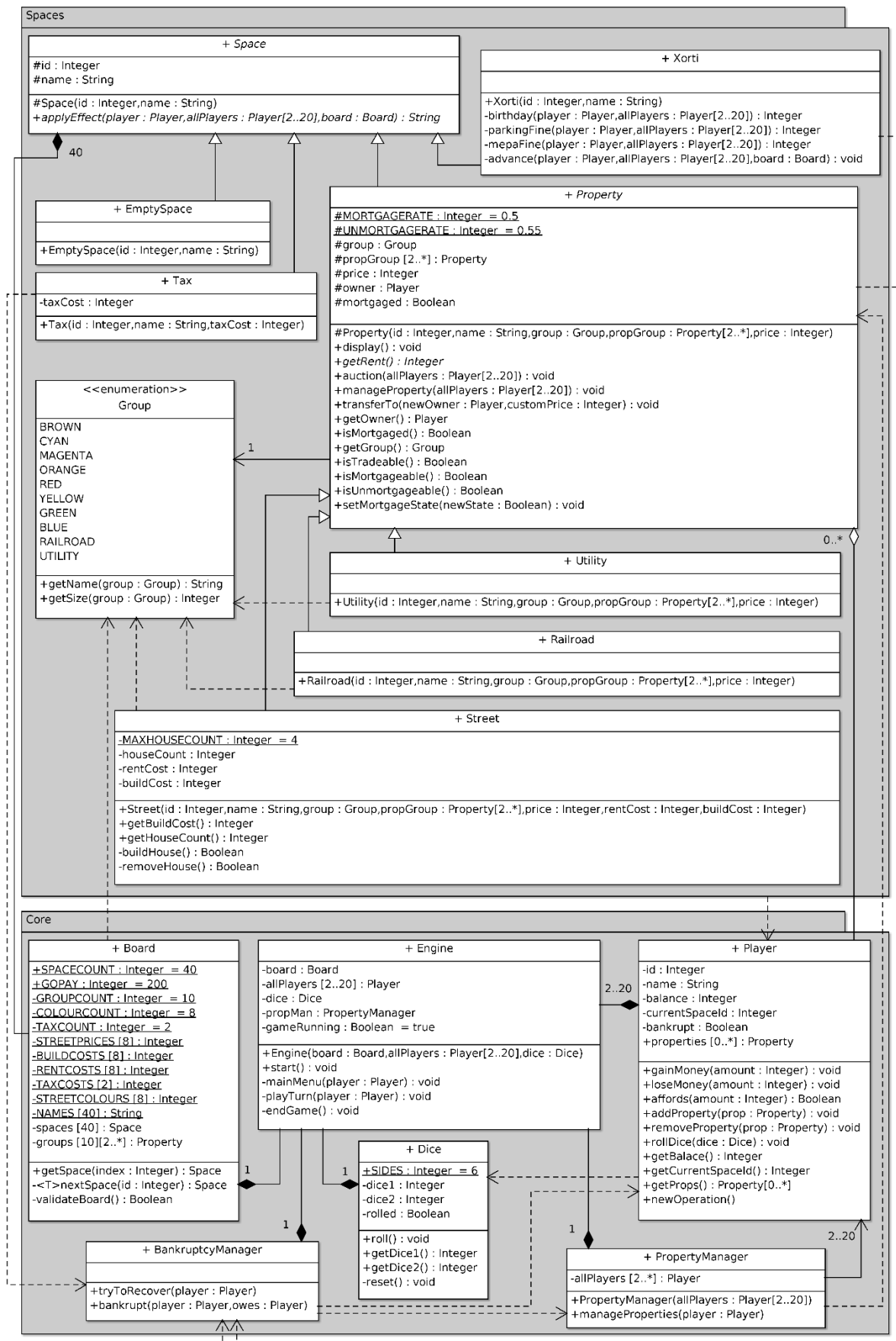
The general approach taken for this task involved firstly creating classes to represent the elements that make up the Monopoly game board. These fall under the general name of "**Space**". The **Board**, **Player**, and **Engine**, were then defined, along with any other convenience classes and functions as seen necessary. Most classes were defined as 'final' since they are not meant to be extended.

Starting from the most general space, the abstract class **Space** defines a general location that a player can land on after rolling the dice. The abstract method *applyEffect(...)* is the method that subclasses implement to define what happens when a player lands on the space. The method takes the list of all players since some spaces allow the player to interact with other players through trading, auctions, fines. The board is also passed so that a space has information about the other spaces. The string returned by the method has the purpose of describing the effect that was applied on the player.

The first subclass of Space, **EmptySpace**, caters for the board's corners. Since these do not affect the player, the *applyEffect(...)* simply returns a string stating that the space had no effect on the player. Next is **Tax**, which simply applies a tax fine to the player that landed on it. Since this class has the potential to bankrupt the player, it uses the bankruptcy management, discussed later on. The **Xorti** class caters for the original game's "chance" spaces. The *applyEffect(...)* for this class picks a random 'xorti' and applies it to the player. This class also has the potential to bankrupt the player, but in this case a player may end up owing another player money. The *advance(...)* method for moving the player to a random space takes the board, so that the new space's *applyEffect(...)* method can be called.

One of the more important subclass of Space is the abstract class **Property**, which represents purchasable spaces. A property has a group, price, owner, and a mortgaged state. The properties in *propGroup* are properties that belong to the same group. This collection is used when the amount of properties owned from the group is required, usually to check whether the whole group is owned. The abstract method *getRent()* allows subclasses to define a way to calculate the rent that a player has to pay when landing on a property. Property management includes mortgaging and trading but can be extended by subclasses. The *applyEffect(...)* function allows the property to be bought if it has no owner, or charges the player rent if it is owned and unmortgaged, unless the player is the owner. Unbought properties get auctioned off. The *auction()* function is also used in bankruptcy management.

Two subclasses of Property are **Utility** and **Railroad**. These override *display()* and implement *getRent()*. The railroad's rent is based on the amount of railroads owned from the group, while that of Utility, in the original Monopoly, is based on the latest dice values rolled by the player. However, in this version, a pair of dice rolls is simulated by generating two random numbers.



The **Street** class inherits from the **Property** class to cater for properties that have a colour group and on which a player can build houses, with a maximum of four houses. This class overrides multiple functions of **Property**, especially when it comes to rent, mortgage, and trading. The functions for building and removing houses take into consideration the amount of houses built on the properties in the property group, if the whole group is owned by a single player. The *getHouseCount()*, as seen in the UML diagram, was moved to **Property** and given a default return value of zero. This was mainly done so that no cast to a street is necessary when this count is required. For the calculation of the rent, the function *getRent()* multiplies the base rent by fixed values based on the number of houses.

The **Core** namespace will now be discussed, starting from the **GameBoard** (seen as **Board** in the UML). This class has the responsibility of creating the spaces on the board and placing them in an array of forty spaces. The strict assumptions made by this class take the form of constant class values. The generic *nextSpace(...)* function returns a new space based on the type. This class also maintains a collection of ten property groups to which properties are added by *nextSpace(...)* after instantiation. A property is passed a reference to its property group, the implications of which were discussed earlier.

The **Player** class includes an ID, name, balance, current space, bankruptcy state, and a collection of owned properties. The defined behaviour mostly has to do with getters and setters. The reason behind storing a space ID rather than a space reference is that generally, a player does not directly manipulate the current space; it is the space that affects the player, via the *applyEffect(...)* function. However, since a player is able to manage owned properties, pointers to them are kept. An interesting function is the *rollDice(...)* which uses the **Dice** class and is used when a player is expected to roll.

Moving on, the class **Engine** joins most of the components in the namespace to make the game work. It stores a reference to the board, players, and a dice. The engine keeps track of whose turn it is, and gives that player the turn. A turn gives the player the ability to manage properties and to roll the dice. The **Dice** class has the engine set as its friend so that only the engine may reset the dice. The space a player is moved to depends on the dice value. The *applyEffect(...)* function of the space that the player lands on is called, the return string value of which is output. If a player rolls two identical values, the player is asked to roll again after moving. Bankrupt players are removed from the players list by the engine as soon as possible and a winner is declared when all players except for one are bankrupt.

The final two components to consider are the **BankruptcyManager** and the **PropertyManager**. As opposed to what is indicated by the UML diagram, these two were not implemented as classes, but as publicly accessible functions. Bankruptcy management includes a function to allow the player to try to recover using property management and two other functions, which are called when the player did not recover, based on who the player owns; the bank or a player. Property management has a function which allows a player to manage owned properties using the property's *manageProperty(...)*.

In general, the only **exceptions** that can be thrown by the program are related to the initialization of a board, thrown if the board does not follow strict guidelines, and to a player losing money, thrown if the player's *loseMoney(...)* is used with a value greater than the balance. Since the *affords(...)* function is meant to be used as prevention, no attempts were made to catch the exception. Board initialization exceptions are also not checked since the program is not meant to proceed if the board is invalid.

Lastly, a collection of **input and output functions**, not shown in the UML, were defined under the **io** namespace. These are meant to group most input and output under one namespace for convenience.

2.3 – Test Cases

Table 2 is a table of test cases considered for this task, with results. Note that these test cases do not cover the whole functionality and there are still bound to be many undetected issues.

Table 2 – Test Cases for Task 3

Player Test		
Test	Expected Outcome	Actual Outcome
affords(balance)	True	True
affords(balance + 1)	False	False
gainMoney(balance)	Balance doubles.	Balance doubled.
loseMoney(balance)	Balance becomes zero.	Balance becomes zero.
setToBankrupt() <u>and</u> isBankrupt()	False before player set to bankrupt and True afterwards.	False before player set to bankrupt and True afterwards.
advanceTo(board.getSpace(5))	Player's space ID changes to 5.	Player's space ID changed to 5.
player == player	True	True
player1 == player2	False	False
Property General Test		
Test	Expected Outcome	Actual Outcome
transferTo(player1, balance/2) <u>and</u> isOwnedBy(player1)	Player1's balance halves and property now owned by Player1.	Player1's balance halved and property now owned by Player1.
transferTo(player2, balance/2) <u>and</u> isOwnedBy(player2) <u>and</u> getProps()	New owner is player2. Player1's balance increases, Player2's halves.	New owner is player2. Player1's balance increased, Player2's halved.
resetOwner()	Property's owner reset	Property's owner reset
Property Auction Test		
Test	Expected Outcome	Actual Outcome
auction(allPlayers) with 4 players	Winner purchases the property	Winner purchases the property
auction(allPlayers) with some players bankrupt	Bankrupt players are not asked whether they want to participate.	Bankrupt players were not asked whether they want to participate.
auction(allPlayers) with all players bankrupt	Property remains unowned.	Property remains unowned.
Property Management Test		
Test	Expected Outcome	Actual Outcome
manageProperties(player, allPlayers)	Property manager opens, allowing player to manage property.	Property manager opened, allowing player to manage property.
Bankruptcy Management Test		
Test	Expected Outcome	Actual Outcome
tryRecover(player, allPlayers, 0, "TEST")	Recovery ends immediately since the player needs to recover zero.	Recovery ended immediately since the player needed to recover zero.
tryRecover(player, allPlayers, 1000, "TEST")	Recovery menu opens, allowing player to try to recover.	Recovery menu opened, allowing player to try to recover.
bankrupt_owesBank(player, allPlayers)	Bankruptcy management places the player's properties on auction and removes the player's money.	Bankruptcy management places the player's properties on auction and removed the player's money.
Bankrupt_owesPlayer(player1, player2)	Player1's properties and money are transferred to player2. Player1's balance becomes zero and player2's now has player1's added to it.	Player1's properties and money were transferred to player2. Player1's balance became zero and player2's now had player1's added to it.

2.4 – Critical Evaluation and Limitations

To start with, error handling could have been much better thought out throughout the program. Any unconsidered issue that may be hidden in the code can easily cause an error. It is however important to note that in some cases, boolean methods (such as *buildHouse()*) were implemented in a way as to return a truth value which indicates whether the function was successful, rather than throwing exceptions when the function is unsuccessful.

Moving on to outputs, these could have been more consistent. For example, the player is sometimes not asked to confirm a decision, meaning that the player must be sure when deciding. The text-output intensive program is in some cases not very user friendly and may become boring for the user.

When it comes to the spaces that make up the board, there are a lot of assumptions made, most of which take the form of constant values in the GameBoard class. An effort could have been made to make the program more extendable. Also, due to the hard-coded nature of the program, it is not customizable when it comes to starting values, space names, costs, fines, etc. A configuration file could have easily been included to be able to customize the experience.

Design-wise, a few functions (such as *getHouseCount()*) that should have been implemented in a particular class were instead implemented in superclasses, just so that casting is avoided. With a better design, this could have been avoided. Considering the EmptySpace class, it would have been more ideal to have it be the subclass of a more general space that does not have the *applyEffect(...)* method. However, the class still uses the function to indicate that no effect was applied on the player.

A small but annoying limitation is that a player cannot forfeit at any time they wish. Players are only removed from the game once they are bankrupt. This could have easily been solved by adding another option to the main menu, allowing the player to quit.

3 – Suffix Trie and Tree

3.1 – UML Diagram

The following is a UML diagram showing the classes' makeup and interactions:

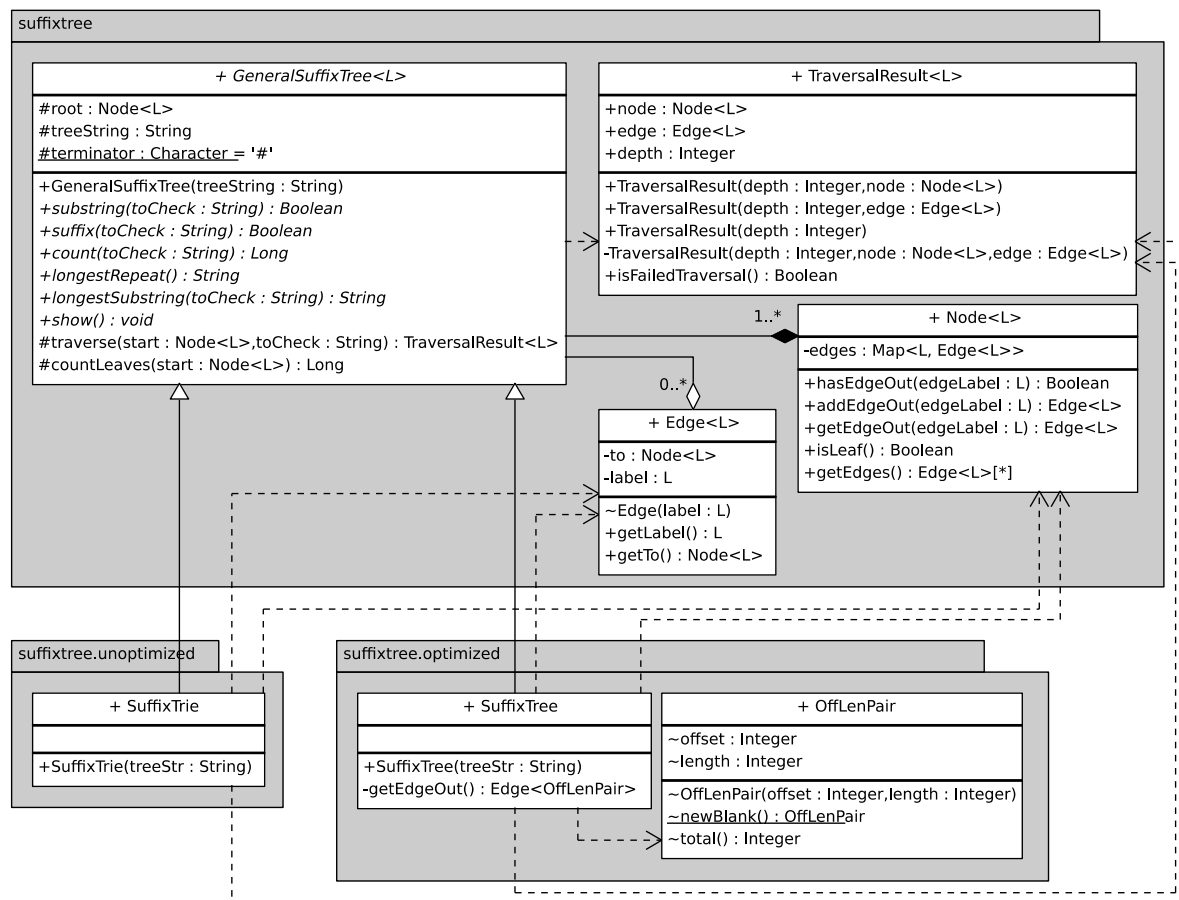


Figure 4 – UML Diagram for Task 3

3.2 – Approach

To start with, it is worth mentioning that the programming language chosen for this task is Java. Besides the fact that I am more comfortable using Java, there are a few other reasons why I chose this language. Firstly, since Java has much better memory management, there is a lesser chance of errors such as memory leaks occurring. This is important due to the high memory usage of suffix tries. There is also no need to perform *deletes* since memory that is no longer in use is handled by the garbage collector. A second reason is that, given that suffix tries and trees are based around strings, Java seems to be the better choice due to strings almost being given a primitive status in Java.

Note that it was assumed that **suffix tries** fall under the general category of **suffix trees**. Also note that steps required to build a system that makes use of both implementation is discussed in section 3.5.

For this task, the approach taken was to first define a set of generic classes and a main abstract class in the package *suffixtree* with the purpose of defining attributes and behaviour that a general suffix tree would have. This includes a generic unlabelled `Node<L>` and labelled `Edge<L>`, where type *L* refers to the type of the edge labels. Some techniques mentioned in [2] were used as a general guide.

In the **Node<L>** class, outward edges are stored in a map structure which maps labels onto edges. This class also has methods to get or add edges, to get all edges, and to check whether an edge exists, by label. A convenience method to check whether the node is a leaf was also added. This works by checking whether the node has any edges out. A method not shown in the UML is *replaceEdge()*, which swaps an edge for another but keeps nodes in place. The *hasEdgeOut(...)* is meant to be used to prevent *getEdgeOut(...)* or *addEdgeOut(...)* from throwing exceptions having to do with edge existence.

The **Edge<L>** class includes getters for the label and the node that it leads to and a setter (not shown in the UML) for the label. The constructor only includes a label parameter since it is the edge's job to create a new node that it leads to, meaning that there is virtually never an edge that does not lead to a node. The label setter is used in suffix tree construction by the node class' *replaceEdge()* method.

The second pair of generic classes are **GenericSuffixTree<L>** and **TraversalResult<L>**, both of which have a generic type with the same purpose as that of the node and edge, i.e. the edge labels' type.

The **GeneralSuffixTree<L>** class defines abstract methods for operations that the suffix trie and suffix tree will be able to perform but also defines the signature for abstract methods *traverse(...)* and *countleaves(...)*, intended for convenience. These turned out to be so important that some methods in this class were made concrete, contrary to what was planned, since most of the procedures, namely *substring(...)*, *suffix(...)*, *count(...)*, and *longestSubstring(...)*, could be expressed in terms of these abstract methods, the implementation for which would be provided by the concrete classes. The *show()* method was also made concrete. This only leaves *longestRepeat()* to be implemented. The class also defines three attributes of a suffix tree; the root node, the string that the tree is based on, and the fixed terminator which is added to the end of the tree's string to indicate the end of suffixes.

The last generic class is **TraversalResult<L>**. An instance of this class is returned when a traversal is performed. Generally, a traversal of a tree along a path defined by a string input has three possible endpoints. Traditionally, a traversal ends either at a node (when a string's characters get exhausted), or fails (when a path along the string's characters does not exist). However, in a tree where edge labels represent more than one character, a traversal may also end mid-way across an edge. For these three cases, the class was given three constructors. The depth is set in all three cases, but if an edge, a node, or both are not supplied, these are privately set to null to indicate where the traversal ended. Since this class does not need too much encapsulation, the (unchangeable) results were made public.

The concrete implementations will now be discussed. In general, these implement the parent class' abstract methods and use unique procedures to generate themselves based on the given string.

In the *suffixtree.unoptimized* package, **SuffixTrie** can be found. This class extends the general suffix tree class, setting **L** to *Character* for character edge labels. This tree is generated by taking suffixes from largest to smallest, and tracing paths along their characters, creating edges as required. A method not shown in the UML diagram is the private *longestRepeat(...)* which is a helper method for *longestRepeat()* and which takes a starting node and a string, starting from a blank string, which is recursively used as an accumulator so that the longest string can be found as the tree is traversed.

In the *suffixtrie.optimized* package, the concrete **SuffixTree** can be found. This class extends the general suffix tree class, setting **L** to **OffLenPair**, which is used so that the two values, offset and length, can be stored in each edge label of the suffix tree. Similar to the *TraversalResult* class, the (fixed) values in *OffLenPair* were not encapsulated and were given package access, for convenience.

The tree creation follows a procedure similar to that explained in [3] which works by considering suffixes of the string from smallest to largest and traversing the tree along the suffix's characters. Whenever a label is exhausted, it is split along the longest common prefix (between the remainder of the suffix being considered and the string represented by the label). If no edge to traverse is found, the remainder of the suffix is added all at once using a single new edge. Similar to the `SuffixTrie` class, the `SuffixTree` class also has a helper method for `longestRepeat()` but instead of using a blank string, it uses an `OffLenPair` to represent a blank string by creating an instance using its `newBlank()` method.

In general, only **exceptions** related to the existence of edges and illegal argument exceptions were explicitly thrown. Since these exceptions imply a bug in the program, no attempt was made to catch these exceptions. For private methods, asserts were widely used to verify arguments.

3.3 – Test Cases

Table 3 is a table of test cases considered for this task and performed using `Launcher.java` through `run.sh`, while Table 4 is a table of cases performed using `Tester.java`, and which is run through `run.sh`. For test 4 in Table 4, a random string generator [4] was used.

Table 3 – `run.sh` Test Cases for Task 2

run.sh Test		
Program Arguments	Expected Outcome	Actual Outcome
Test_hahaa.txt suffix	Yes	Yes
Test_hahaa.txt suffix h	No	No
Test_hahaa.txt suffix ha	No	No
Test_hahaa.txt suffix hah	No	No
Test_hahaa.txt suffix haha	No	No
Test_hahaa.txt suffix hahaa	Yes	Yes
Test_hahaa.txt suffix ahaa	Yes	Yes
Test_hahaa.txt suffix haa	Yes	Yes
Test_hahaa.txt suffix aa	Yes	Yes
Test_hahaa.txt suffix a	Yes	Yes
Test_hahaa.txt suffix aha	No	No
Test_hahaa.txt suffix bahaha	No	No
Test_hahaa.txt substr h	Yes	Yes
Test_hahaa.txt substr ha	Yes	Yes
Test_hahaa.txt substr hah	Yes	Yes
Test_hahaa.txt substr haha	Yes	Yes
Test_hahaa.txt substr hahaa	Yes	Yes
Test_hahaa.txt substr ahaa	Yes	Yes
Test_hahaa.txt substr haa	Yes	Yes
Test_hahaa.txt substr aa	Yes	Yes
Test_hahaa.txt substr a	Yes	Yes
Test_hahaa.txt substr aha	Yes	Yes
Test_hahaa.txt substr bahaha	No	No
Test_hahaa.txt count	6	6
Test_hahaa.txt count h	2	2
Test_hahaa.txt count ha	2	2
Test_hahaa.txt count hah	1	1
Test_hahaa.txt count haha	1	1
Test_hahaa.txt count hahaa	1	1

Test_hahaa.txt count ahaa	1	1
Test_hahaa.txt count haa	1	1
Test_hahaa.txt count aa	1	1
Test_hahaa.txt count a	3	3
Test_hahaa.txt count aha	1	1
Test_hahaa.txt count bahaha	0	0
Test_hahaa.txt lonrep	ha	ha
Test_hahaa.txt lonsub	(*blank string*)	(*blank string*)
Test_hahaa.txt lonsub bahaha	haha	haha
Test_hahaa.txt show	Correct representation	Correct representation
Test_blank.txt show	Correct representation	Correct representation
Test_endsWith#.txt show	Correct representation	Correct representation
(*blank string*)	Invalid number of inputs	Invalid number of inputs
Test_hahaa.txt	Invalid number of inputs	Invalid number of inputs
Test_invalidFile.txt show	File not found	File not found
Test_hahaa.txt display	Invalid function	Invalid function

Table 4 - Tester.java Test Cases for Task 2

Test 1 (*blank string*)		
Function	Expected Outcome	Actual Outcome
substring("")	True	True
suffix("")	True	True
count("")	1	1
longestRepeat()	(*blank string*)	(*blank string*)
longestSubstring(abcd.....wxyz)	(*blank string*)	(*blank string*)
show()	Correct representation	Correct representation
Test 2 (hahaa)		
Test cases and results identical to those in Table 3		
Test 3 (abcd.....wxyz)		
longestRepeat()	(*blank string*)	(*blank string*)
longestSubstring(zyxw)	z	Z
show()	Correct representation	Correct representation
Test 4 (abcdLONGESTREPEATefghLONGESTREPEATijkl)		
longestRepeat()	LONGESTREPEAT	LONGESTREPEAT
longestSubstring(ijklLONGESTREPEATabcd)	LONGESTREPEAT	LONGESTREPEAT
show()	Correct representation	Correct representation
Test 5 (8W68QV78nV.....L4AHMlpHgF)		
Multiple suffix(...) tests performed based on parts of the long string.		
Multiple substring(...) tests performed based on parts of the long string.		
Multiple count(...) tests performed based on parts of the long string.		
longestRepeat()	fM7xB...nl3MV	fM7xB...nl3MV
Single longestSubstring(...) test performed based on parts of the long string.		

3.4 – Critical Evaluation and Limitations

Many of the techniques used in this task felt somewhat controversial. To start with, although having an abstract class instead of an interface means that a lot of code repetition was avoided, the functionality implemented might not suite all types of suffix trees that may extend this abstract class. Obviously, such classes could override the methods, but not doing so may produce erroneous results.

Moving on to the creation of the suffix tries and trees, these could have been improved with the use of suffix links that link a leaf node to the leaf node representing a suffix smaller by one character. Currently, all suffixes are traversed character-by-character. This leads to suboptimal performance.

While the creation inefficiencies may be overlooked since creation only happens once, the *longest-Substring(...)* method uses a similar process of going through all suffixes, meaning that the method takes a long time to give a result for large strings. Suffix links could also have been used in this case.

The last critical evaluation is about the Edge and Node classes which contain methods *setLabel(...)* and *replaceEdge(...)*, respectively. These methods used only by the optimized suffix tree implementation and could have been placed in new Edge and Node subclasses with less focus on a simple OO design.

3.5 – Answer to Additional Questions

To build a system that makes use of the trie (or tree) functionality without tying itself to a particular implementation, the most important element to have is an abstract class (alternatively, an interface) that defines the general functionality that a tree would provide, without going into concrete ideas.

The system can then be modelled around this abstract class and be guaranteed that the tree instance that it will be working with will at least have the functionality indicated by the abstract class. The actual implementation details of the tree would not be important for the system, as long as the tree satisfies the performance requirements of the system.

4 – References

- [1] MONOPOLY, <https://goo.gl/36H8u>. Last accessed: January 12th, 2017.
- [2] Carl Kingsford, Suffix Trees Lecture Notes, Carnegie Mellon University, <https://goo.gl/z6MWkz>, Last Accessed: January 12th, 2017
- [3] Christa McCarthy (christamcc). (2012, May 9). Creating the Suffix Tree - Conceptually. [YouTube video]. Available: <https://goo.gl/GHazRF>. Last accessed: January 12th, 2017.
- [4] Tool: Random String Generator, <https://goo.gl/ssa4d4>. Last accessed: January 12th, 2017.