A Level Computer Science NEA

Analysis

Overview

The aim of this program is to enable users to play Monopoly. They will be able to choose to play against up to three other players with each one either being a human player, playing on the same device; or a computer-controlled player, which will have varying difficulty settings defined when the game is started and so will make smart decisions based upon the current game state and difficulty.

The problem

Monopoly is a “board game in which players engage in simulated property and financial dealings”. The problems with Monopoly include the general complications of board games which include they often take a very long time to play the full game out and almost always require more than one human player; however, the program will have the option to play one human player against three players controlled by the computer which not only allow the game to now be played by a single human player, but also it will reduce the long game time when playing with several computer players as their turn’s will be completed very quickly. Other general problems include when playing on a physical board, the game states could be altered to cheat or even sometimes completely lost by someone moving all the pieces; the program will counter this as the official Monopoly rules will be followed and no cheating, once in game, will be possible.

Unique problems to Monopoly include the complicated and long set up time involving organising and dealing out all of the correct amounts of money as well as having a banker to participate in the running of the game yet never having a chance to win. The final version will have no such big waiting times to start up the game and throughout the run-through of the game will not need a banker as all transactions and property dealing will be done fluently by the computer.

The Game

The game involves a board of forty tiles, as shown on the left, and initially starts with up to four pieces representing each player on the Go tile.

Then each player takes turns rolling two dice and moving the sum of these rolls forward. If they land on an unowned property then they have the choice to purchase that property, if someone else owns it then they pay the relevant owner a fee. There are also some tiles which aren’t properties but have other actions related to them such as the chance or community chest tiles; where you draw a card and it has an action you follow; or tax tiles.

Sometimes a player can land on the go to jail tile where they are moved to the jail tile and can only get out by paying a fee, rolling a double or using a get out of jail card previously drawn. Also, upon passing the go tile that player earns some money and will go around again.

Other aspects on each turn include every time a double is rolled, a player will roll the dice again and take another turn afterwards; and at any point during a player's turn, they can trade with another player in return for properties or money.

A player is forced out the game when they land on an owned tile and don't have the funds to pay the relevant player. The last remaining player is the winner.

Core objectives

1. Game in general
    - 1.1. Defining aspects of Monopoly
        - 1.1.1. The whole game matches that of the board game - similar to an economic simulation with money flow and trading of assets
        - 1.1.2. The software will be able to generate the board with the correct names and values for rent – loading from a modifiable text file allowing for possible customisation
            - 1.1.2.1. The board object will be composed of 40 tile objects demonstrating the strong composition relationship as matches the relationship of the actual board game
        - 1.1.3. Turn based play on a continuous board – players roll the dice and iterate round the board, upon landing on a property either ignore it, purchase it or pay the owner the calculated amount
            - 1.1.3.1. The player objects will be stored in a circular queue to ensure they all get a go and are in order
            - 1.1.3.2. The human and computer player objects will be inherited from an abstract TPlayer class and will have many similar attributes and methods
        - 1.1.4. Property ownership – the players will be able to buy properties that they land on and own them, the program will also allow the properties to be mortgaged and traded between players
            - 1.1.4.1. To organise trading between the players the program will create a record which contains the information involved in the trade
                - 1.1.4.1.1. There should be input validation in the money involved in the trade using regular expression to ensure it is only an integer
                - 1.1.4.1.2. Upon a player receiving a trade, they should have the option to accept or reject the trade. If accept the computer organises the exchange, if reject then the record is cleared and prepared for a future trade
            - 1.1.4.2. To organise the mortgaging of properties, the players will be able to choose properties which they own and mortgage them, however they will no longer receive rent from them
        - 1.1.5. Money management – each player will have a fixed initial balance as defined by the Monopoly rules and will leave the game and lose if they run out

1.1.6. Just visiting, jail and free parking – each tile will have its relevant implementation

1.1.6.1. The two jail tiles will be built in – go to jail will send the player to jail where the player must either roll a double or at the end of three turns pay to get out; the just visiting state will also be on this tile

1.1.6.2. Free parking will collect all the tax money and other money required from the chance/community chest cards; anyone who lands on it will be rewarded with the totalled money and its value will be reset

1.1.7. Chances and community chests – all the cards and functions implemented and use a stack to store them to behave similar to the actual game

1.2. Visually pleasing GUI for user

1.2.1. Original state for user to choose how many people are playing, their name and colour – these will loop round the board and will be located on the correct tile during their turn, indicating to the user where they are

1.2.1.1. If that player is computer controlled and if so, what difficulty they are behaving as

1.2.2. The form in game

1.2.2.1. The Monopoly board - the program will have the normal board as an image with the pieces located on it, also it will indicate what properties are owned and by which colour with an indicator (maybe with a colour)

1.2.2.2. A read-only output box for the user to understand the flow and play of the game- the box will update with any events occurring at that moment in the game and will be easy to read – e.g. 'Player 1 has moved to go' or 'Player 1 has paid £100 to Player 2'

1.2.2.3. All the relevant buttons to perform each action for example when clicking roll dice, the current player's turn will roll, and their piece will move around the board. Then if they land on an unowned property buttons will appear to allow them to buy it or not

2. Difficulty scaling

2.1. Use of artificial intelligence to make suitable decisions relevant to the current game state

2.1.1. The difficulty variable of the computer player will determine what decision it makes

2.1.2. Complete research into the best strategies and decisions to make enabling the hard difficulty to actually be hard to beat for the human player

Extension objectives

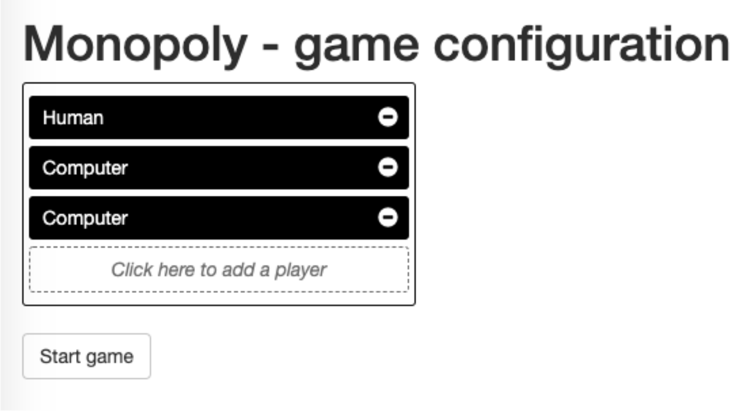
3. Being able to save current game states and loading them from save files in the future

4. Once achieved the initial game and some kind of computer-controlled user, have the game play out without human players and analyse the results to see what strategies are the most effective at winning.

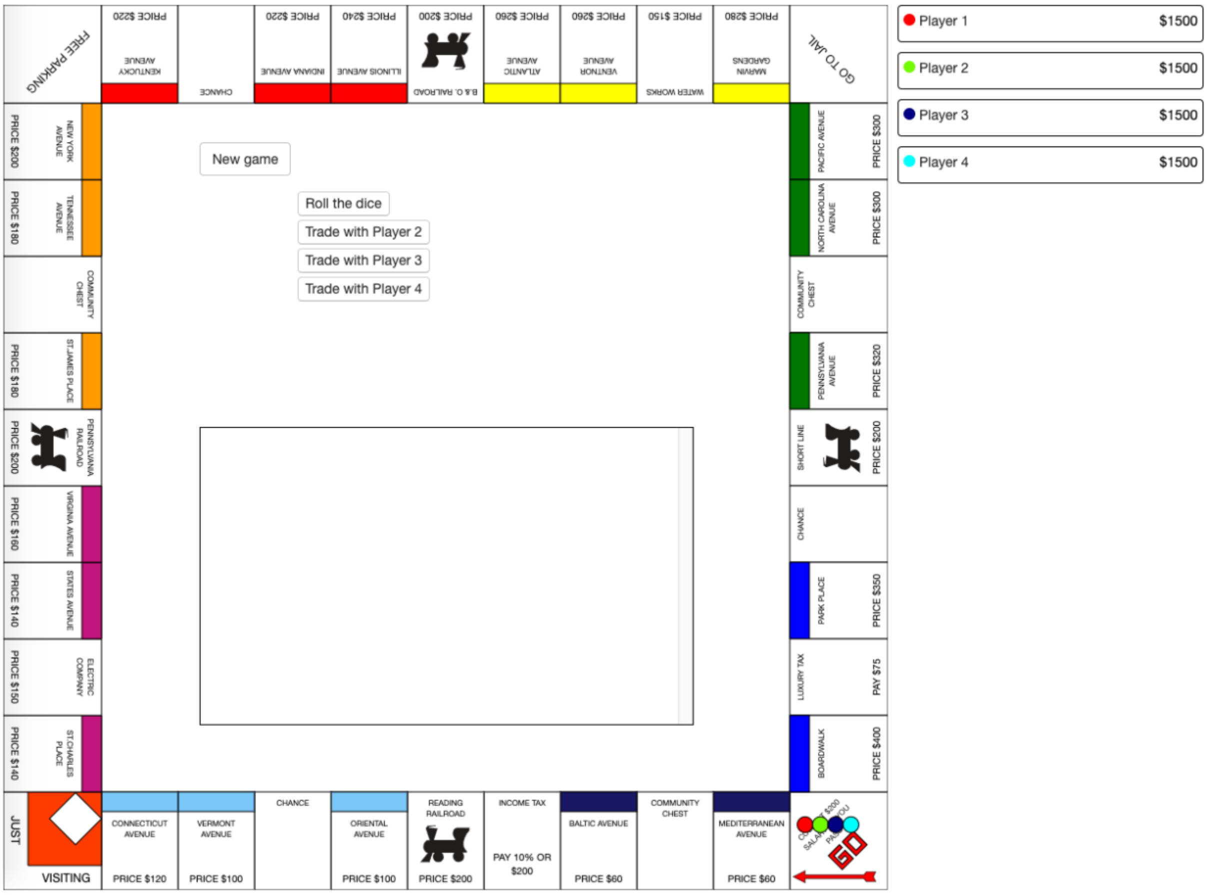
4.1. Create new strategies, derived from experimenting with the program and provide evidence in the form of some sort of output from simulations

The Current System

From: http://francois-roseberry.github.io/Monopoly-js/demo/



The current system found online is a website in which Monopoly can be played against computer players. This is an addition to the normal board game, however, does not have difficulty scaling – meaning a difficulty variable given to the computer-controlled player to determine how they play, generally easy, medium or hard. Some modern computer games which can involve playing against the computer have poor implementation of difficulty as for easy mode often just make the computer make random decisions, and for hard mode give the computer unfair advantages, for example in Monopoly it would be starting with more money than the human players. Both of these ultimately end up in a poor user experience and so should be modified in the program.

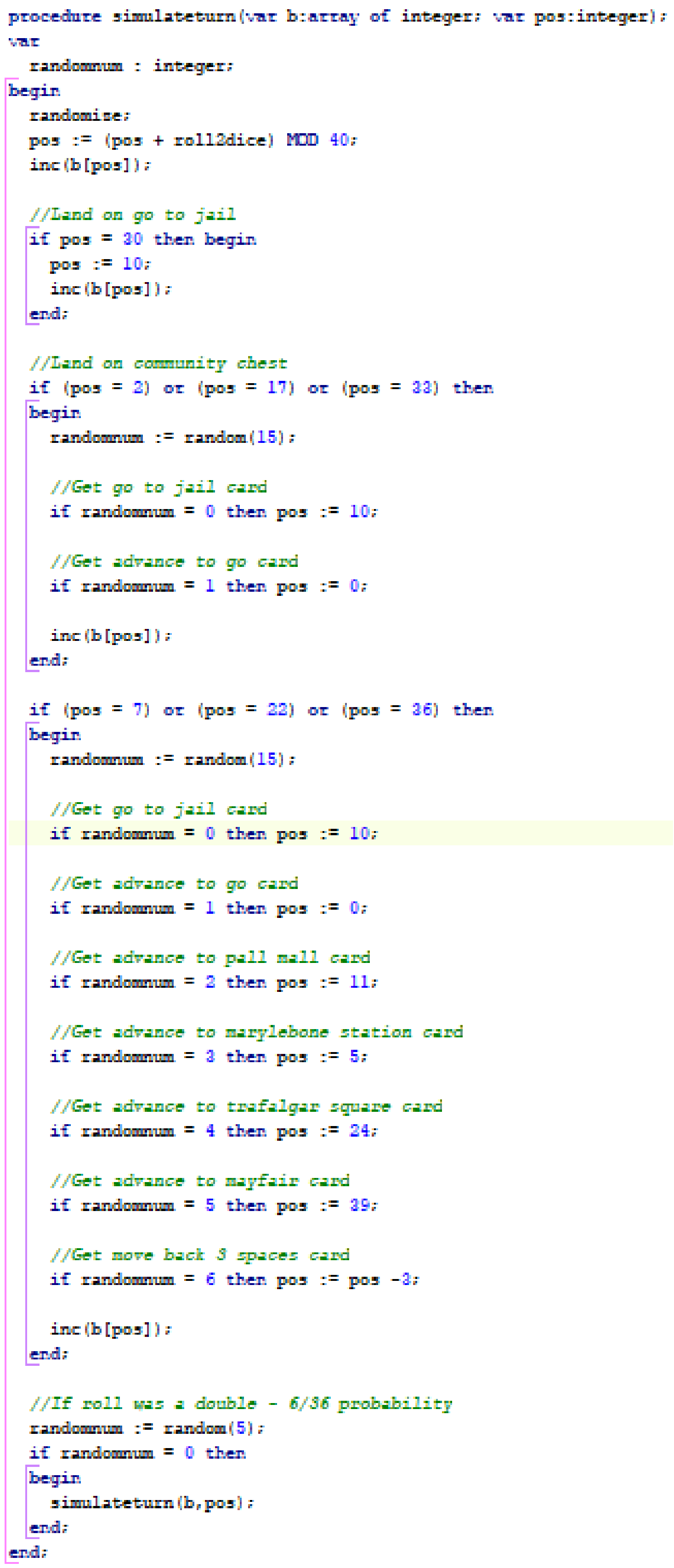

New game
Roll the dice
Trade with Player 2
Trade with Player 3
Trade with Player 4
Player 1 $1500
Player 2 $1500
Player 3 $1500
Player 4 $1500
FREE PARKING
KENTUCKY AVENUE PRICE $220
CHANCE
INDIANA AVENUE PRICE $220
ILLINOIS AVENUE PRICE $240
B & O. RAILROAD PRICE $200
ATLANTIC AVENUE PRICE $260
VENTNOR AVENUE PRICE $260
WATER WORKS PRICE $150
MARVIN GARDENS PRICE $280
GO TO JAIL
PACIFIC AVENUE PRICE $300
NORTH CAROLINA AVENUE PRICE $300
COMMUNITY CHEST
PENNSYLVANIA AVENUE PRICE $320
SHORT LINE PRICE $200
CHANCE
PARK PLACE PRICE $350
LUXURY TAX PAY $75
BOARDWALK PRICE $400
NEW YORK AVENUE PRICE $200
TENNESSEE AVENUE PRICE $180
COMMUNITY CHEST
ST.JAMES PLACE PRICE $180
PENNSYLVANIA RAILROAD PRICE $200
VIRGINIA AVENUE PRICE $160
STATES AVENUE PRICE $140
ELECTRIC COMPANY PRICE $150
ST.CHARLES PLACE PRICE $140
JUST VISITING
CONNECTICUT AVENUE PRICE $120
VERMONT AVENUE PRICE $100
CHANCE
ORIENTAL AVENUE PRICE $100
READING RAILROAD PRICE $200
INCOME TAX PAY 10% OR $200
BALTIC AVENUE PRICE $60
COMMUNITY CHEST
MEDITERRANEAN AVENUE PRICE $60
GO

After using the above current system, the use of a memo box to output the ongoing play inform the user of what the game is expecting them to next do, for example “Player 1’s turn” and “Player 1 has landed on Player 2’s Mayfair and paid them £400”, is quite useful and would be useful to implement into the system which would result in a better user experience as would always know what is going on with visual feedback from the movement of pieces and text feedback.

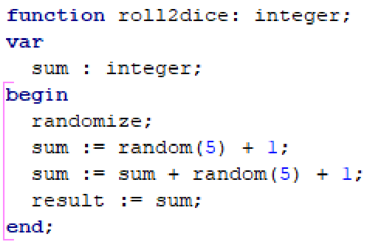
Research

For the hard difficulty settings of the computer controlled player, in order to avoid unfair advantages which a human player would not be able to do, research into the probability of landing on each tile ought to be done as this would reflect on the best tile to purchase when wanting to win. The hard difficulty would then prioritise buying these properties over others when given the opportunity, meaning it would be harder for the human players to win.

The best way to produce realistic data was to simulate a minimalistic Monopoly game with only the factors which effect the position of the counters implemented. As the final program would be coded in Delphi, the prototyping and research projects would be best written also in Delphi.



```
procedure simulateturn(var b:array of integer; var pos:integer);
var
  randomnum : integer;
begin
  randomise;
  pos := (pos + roll2dice) MOD 40;
  inc(b[pos]);

  //Land on go to jail
  if pos = 30 then begin
    pos := 10;
    inc(b[pos]);
  end;

  //Land on community chest
  if (pos = 2) or (pos = 17) or (pos = 33) then
  begin
    randomnum := random(15);

    //Get go to jail card
    if randomnum = 0 then pos := 10;

    //Get advance to go card
    if randomnum = 1 then pos := 0;

    inc(b[pos]);
  end;

  if (pos = 7) or (pos = 22) or (pos = 36) then
  begin
    randomnum := random(15);

    //Get go to jail card
    if randomnum = 0 then pos := 10;

    //Get advance to go card
    if randomnum = 1 then pos := 0;

    //Get advance to pall mall card
    if randomnum = 2 then pos := 11;

    //Get advance to marylebone station card
    if randomnum = 3 then pos := 5;

    //Get advance to trafalgar square card
    if randomnum = 4 then pos := 24;

    //Get advance to mayfair card
    if randomnum = 5 then pos := 39;

    //Get move back 3 spaces card
    if randomnum = 6 then pos := pos -3;

    inc(b[pos]);
  end;

  //If roll was a double - 6/36 probability
  randomnum := random(5);
  if randomnum = 0 then
  begin
    simulateturn(b,pos);
  end;
end;
```



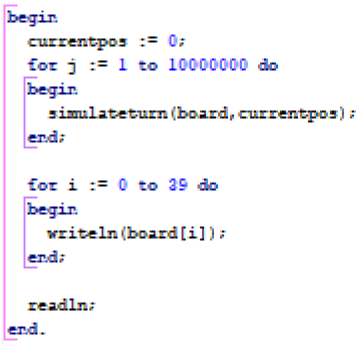
```
function roll2dice: integer;
var
  sum : integer;
begin
  randomize;
  sum := random(5) + 1;
  sum := sum + random(5) + 1;
  result := sum;
end;
```

The unit began with a function which rolled two dice and returned the sum of the two, and then the main bulky procedure which simulated a single turn on a counter. It took in an array and position as parameters. The array was the board and, in each element, contained the number of times that space had been visited. The position was the current position of the counter as it went around the board. Firstly, the dice were rolled and counter, representing a player, was moved that many spaces so value inside the element at that position was incremented.

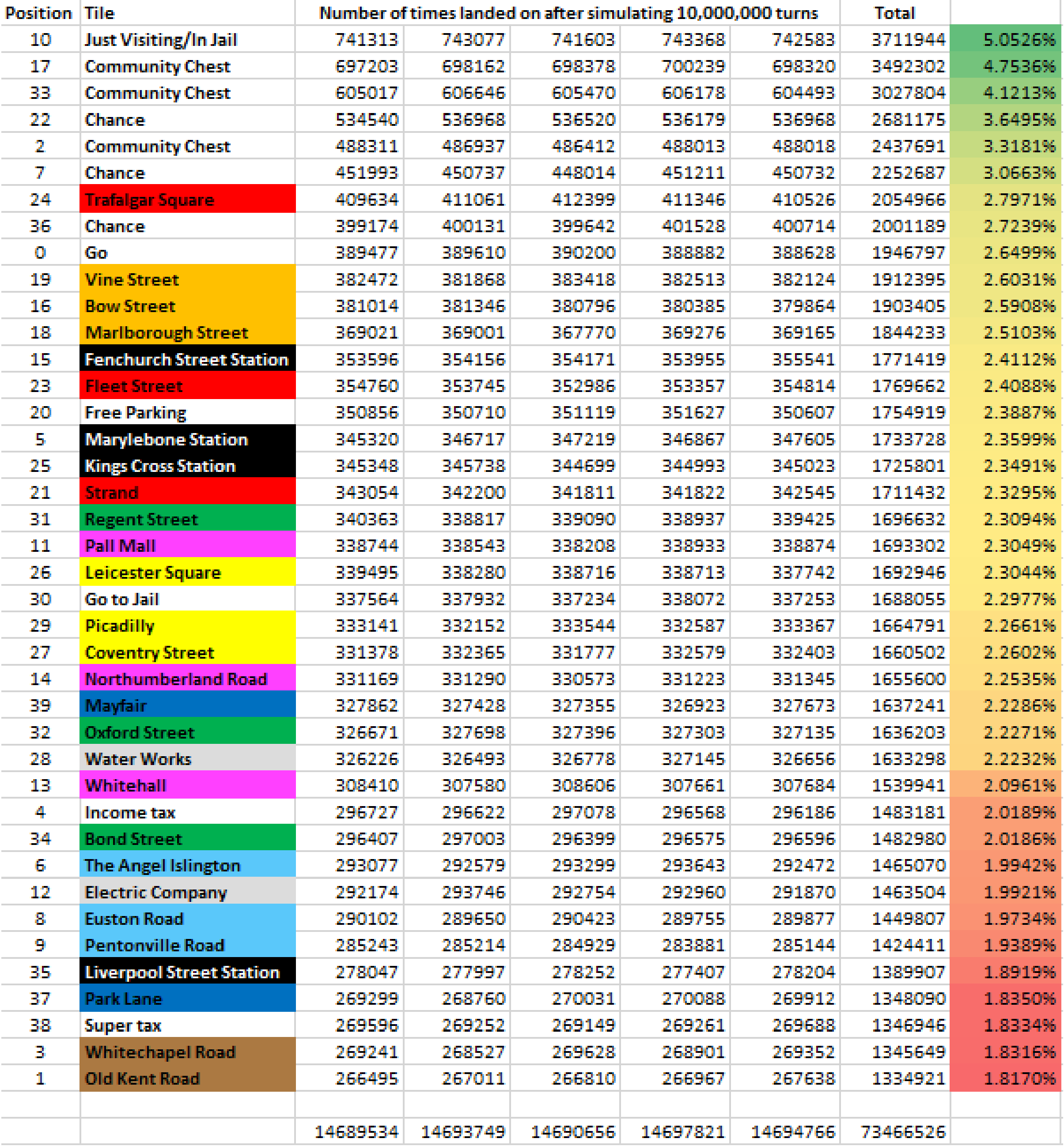
As there are other factors which determine where a counter will be, these also should be implemented. For community chest and chance cards, there was a 1/16 chance of drawing a specific movement card and so if that was drawn, then it would be executed.

Finally, there was an if clause which incorporated recursion to determine if a turn was a double, as in the rules of Monopoly, if a double is rolled then that person has their go again straight after.

As the majority of the program was inside functions, the main part was simple and just executed 10,000,000 turns. Then the values in the array were outputted enabling further analysis of the values could be done, ultimately producing a clear result.

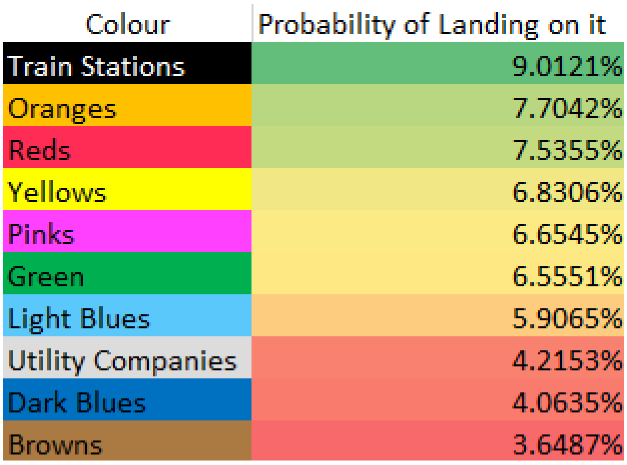


```
begin
  currentpos := 0;
  for j := 1 to 10000000 do
  begin
    simulateturn(board,currentpos);
  end;

  for i := 0 to 39 do
  begin
    writeln(board[i]);
  end;

  readln;
end.
```



Position	Tile	Number of times landed on after simulating 10,000,000 turns					Total	
10	Just Visiting/In Jail	741313	743077	741603	743368	742583	3711944	5.0526%
17	Community Chest	697203	698162	698378	700239	698320	3492302	4.7536%
33	Community Chest	605017	606646	605470	606178	604493	3027804	4.1213%
22	Chance	534540	536968	536520	536179	536968	2681175	3.6495%
2	Community Chest	488311	486937	486412	488013	488018	2437691	3.3181%
7	Chance	451993	450737	448014	451211	450732	2252687	3.0663%
24	Trafalgar Square	409634	411061	412399	411346	410526	2054966	2.7971%
36	Chance	399174	400131	399642	401528	400714	2001189	2.7239%
0	Go	389477	389610	390200	388882	388628	1946797	2.6499%
19	Vine Street	382472	381868	383418	382513	382124	1912395	2.6031%
16	Bow Street	381014	381346	380796	380385	379864	1903405	2.5908%
18	Marlborough Street	369021	369001	367770	369276	369165	1844233	2.5103%
15	Fenchurch Street Station	353596	354156	354171	353955	355541	1771419	2.4112%
23	Fleet Street	354760	353745	352986	353357	354814	1769662	2.4088%
20	Free Parking	350856	350710	351119	351627	350607	1754919	2.3887%
5	Marylebone Station	345320	346717	347219	346867	347605	1733728	2.3599%
25	Kings Cross Station	345348	345738	344699	344993	345023	1725801	2.3491%
21	Strand	343054	342200	341811	341822	342545	1711432	2.3295%
31	Regent Street	340363	338817	339090	338937	339425	1696632	2.3094%
11	Pall Mall	338744	338543	338208	338933	338874	1693302	2.3049%
26	Leicester Square	339495	338280	338716	338713	337742	1692946	2.3044%
30	Go to Jail	337564	337932	337234	338072	337253	1688055	2.2977%
29	Picadilly	333141	332152	333544	332587	333367	1664791	2.2661%
27	Coventry Street	331378	332365	331777	332579	332403	1660502	2.2602%
14	Northumberland Road	331169	331290	330573	331223	331345	1655600	2.2535%
39	Mayfair	327862	327428	327355	326923	327673	1637241	2.2286%
32	Oxford Street	326671	327698	327396	327303	327135	1636203	2.2271%
28	Water Works	326226	326493	326778	327145	326656	1633298	2.2232%
13	Whitehall	308410	307580	308606	307661	307684	1539941	2.0961%
4	Income tax	296727	296622	297078	296568	296186	1483181	2.0189%
34	Bond Street	296407	297003	296399	296575	296596	1482980	2.0186%
6	The Angel Islington	293077	292579	293299	293643	292472	1465070	1.9942%
12	Electric Company	292174	293746	292754	292960	291870	1463504	1.9921%
8	Euston Road	290102	289650	290423	289755	289877	1449807	1.9734%
9	Pentonville Road	285243	285214	284929	283881	285144	1424411	1.9389%
35	Liverpool Street Station	278047	277997	278252	277407	278204	1389907	1.8919%
37	Park Lane	269299	268760	270031	270088	269912	1348090	1.8350%
38	Super tax	269596	269252	269149	269261	269688	1346946	1.8334%
3	Whitechapel Road	269241	268527	269628	268901	269352	1345649	1.8316%
1	Old Kent Road	266495	267011	266810	266967	267638	1334921	1.8170%
		14689534	14693749	14690656	14697821	14694766	73466526	

Above is the sorted data gained from running the simulation. To ensure the data was reliable and not just an anomaly, the simulation was ran five times and the sum of numbers were taken to minimise the impact of any skewed results. Then the percentage chance of that tile being landed on were worked out and sorted by this to produce an order of tiles most likely to be landed on.



Colour	Probability of Landing on it
Train Stations	9.0121%
Oranges	7.7042%
Reds	7.5355%
Yellows	6.8306%
Pinks	6.6545%
Green	6.5551%
Light Blues	5.9065%
Utility Companies	4.2153%
Dark Blues	4.0635%
Browns	3.6487%

Some of the results can be explained such as the highest being the jail tile as there are three ways to end up on that tile – being roll there, get a chance/community chest, which sends you there, or landed on the go to jail tile. Then, as this is the most visited, the second most visited is the community chest tile which is seven spaces after this as seven is the most likely rolled number on two dice; the community chest tile is surrounded by the oranges explaining that figure. Trafalgar square is high up as can be rolled or drawn from a chance to go there. Furthermore, the results can be compiled and sorted by property colour as would give me a good indicator of what properties to prioritise when buying in Monopoly, when aiming to win. From this you can see that giving the stations, oranges and reds the highest chance of buying would be best as are most likely to be landed on. The trains are the highest by over 1% as there are four tiles which are stations, also the stations often don’t render an opponent bankrupt as the fee for landing on it when owned can go no higher that two-hundred pounds, whereas for oranges, Vine Street can reach one-thousand pounds with a hotel.

Overall it is clear that the in hard mode, the chance of buying properties should match the order above, however, will make oranges almost a one-hundred percent chance of buying them. If the extension objectives are also completed, then it will be possible to complete simulations of the hard mode player and analyse some of the results of it winning, as to create better and more effective strategies as winning, ultimately making hard mode more difficult to beat.

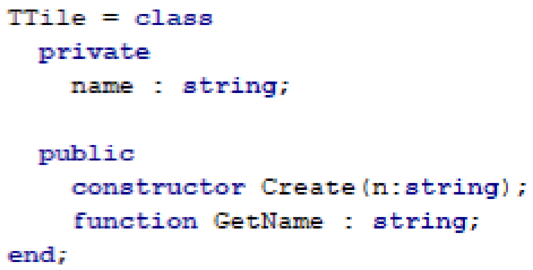
End Users

As Monopoly is a popular game, many people know how to play the game and so having one specific end user would limit the functionality of the final program. Therefore, the Monopoly game may be introduced to multiple possible users and incorporate all of their ideas, if realistic, into the final program to produce a fully functional and fun game.

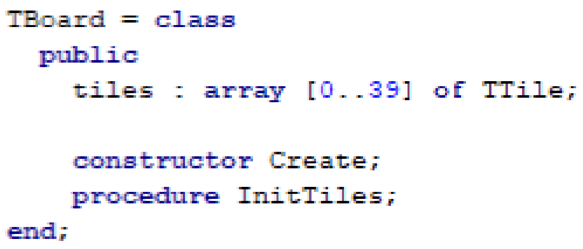
Upon speaking to Lucas Hockley; a computer science A-Level student with knowledge in how to play Monopoly and someone with desire to play the full game when completed; he informed me that on top of the normal Monopoly rules, he would also like to have a visual icon representing the piece movement to show where each player is, and an indicator on the properties to show if they are owned and who by. To stay with the normal Monopoly style, the software will use the pieces from the original game such as the iron, at the beginning, the players will choose a piece and as the game is played, their relevant icon will also move around and be on the same tile as the actual player. A possible secondary objective which was also suggested is that on top of the in-built icons, there could be an option to upload your own image to represent you and be more personalised.

Prototyping

When programming the project, it ought to be mainly based on an object-oriented structure, for example all of the properties and players and board will all be objects with some relationship to each other. The main two objects will be the board and tiles as they will be called and used on almost every turn, so prototyping in creating these two and determining the relationship and using the functions of each object would be useful.

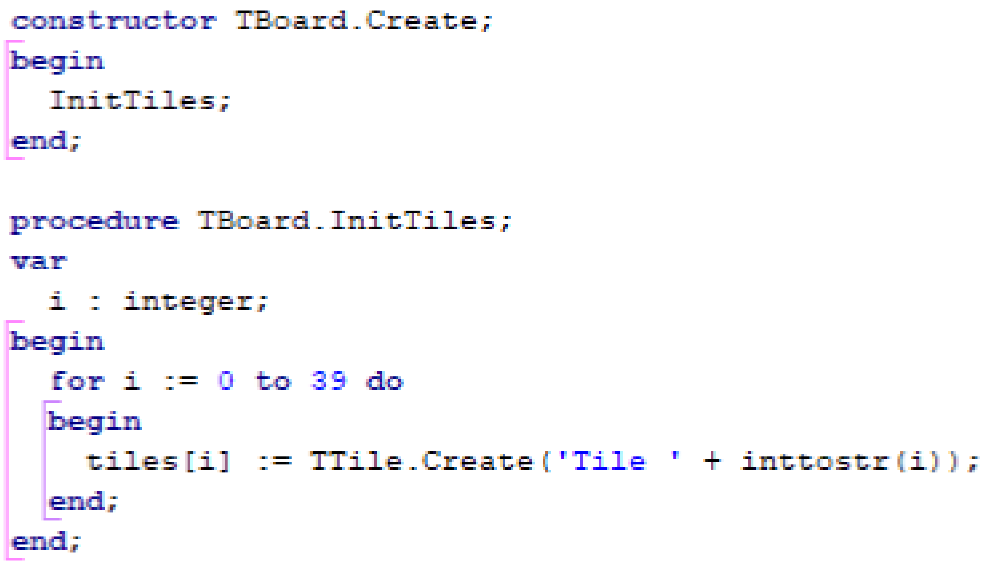


```
TTile = class
  private
    name : string;

  public
    constructor Create(n:string);
    function GetName : string;
end;
```



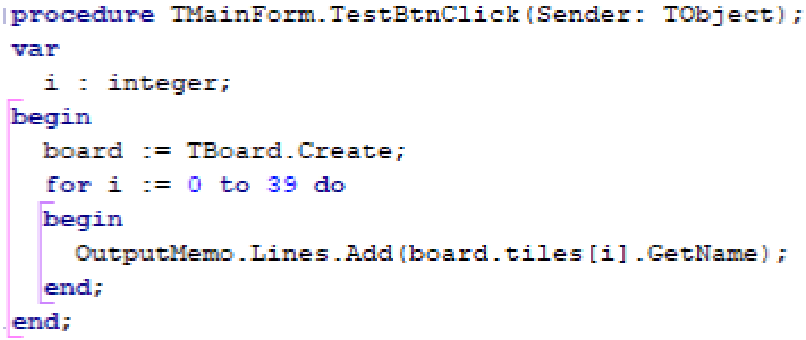
```
TBoard = class
  public
    tiles : array [0..39] of TTile;

    constructor Create;
    procedure InitTiles;
end;
```

Above are the two definitions for objects used in prototyping. One is a tile which will represent all the tiles on the board. Upon abstracting down to only have programmed the name of the tile and the tiles on the board it is possible to test the ownership and relationship between the two so other methods and fields are not needed.



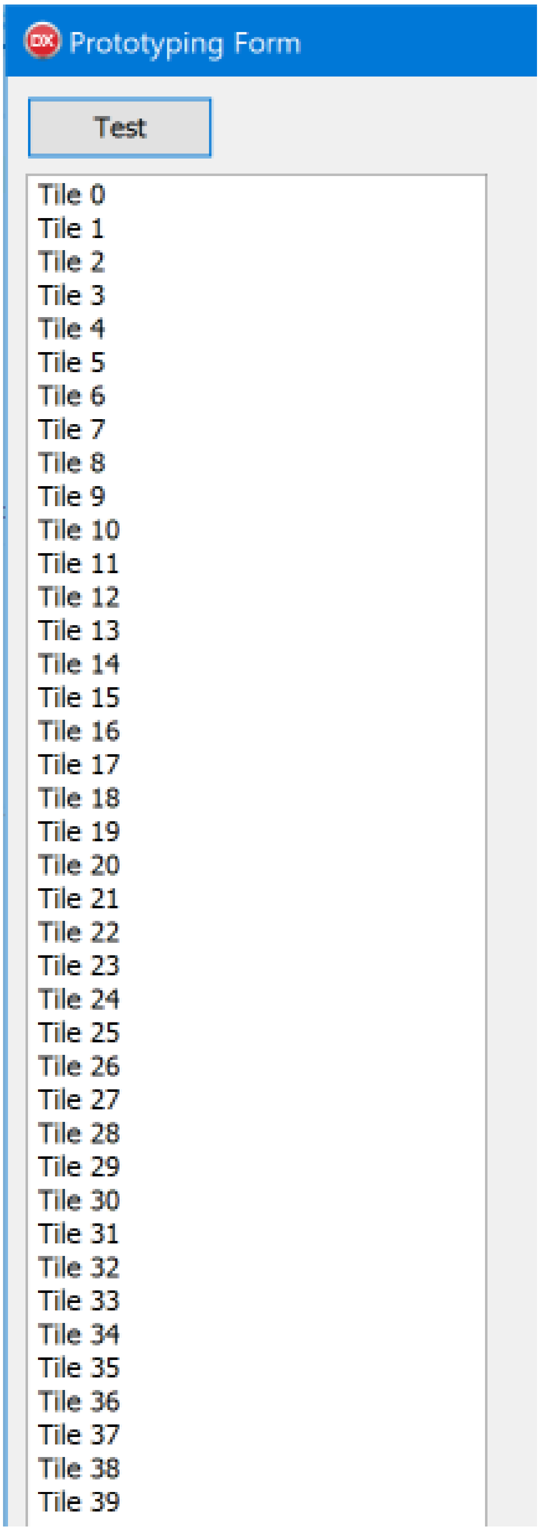
```
constructor TBoard.Create;
begin
  InitTiles;
end;

procedure TBoard.InitTiles;
var
  i : integer;
begin
  for i := 0 to 39 do
  begin
    tiles[i] := TTile.Create('Tile ' + inttostr(i));
  end;
end;
```

Here are the methods defined in the TBoard type. By having an array which contains 40 tiles in total, it demonstrates the relationship being a composition of tiles which is owned, created and destroyed by the board – the tiles cannot exist independently of the board. For testing purposes, the tiles are given a name which is configured when created, in order to see if the program ultimately works.



```
procedure TMainForm.TestBtnClick(Sender: TObject);
var
  i : integer;
begin
  board := TBoard.Create;
  for i := 0 to 39 do
  begin
    OutputMemo.Lines.Add(board.tiles[i].GetName);
  end;
end;
```

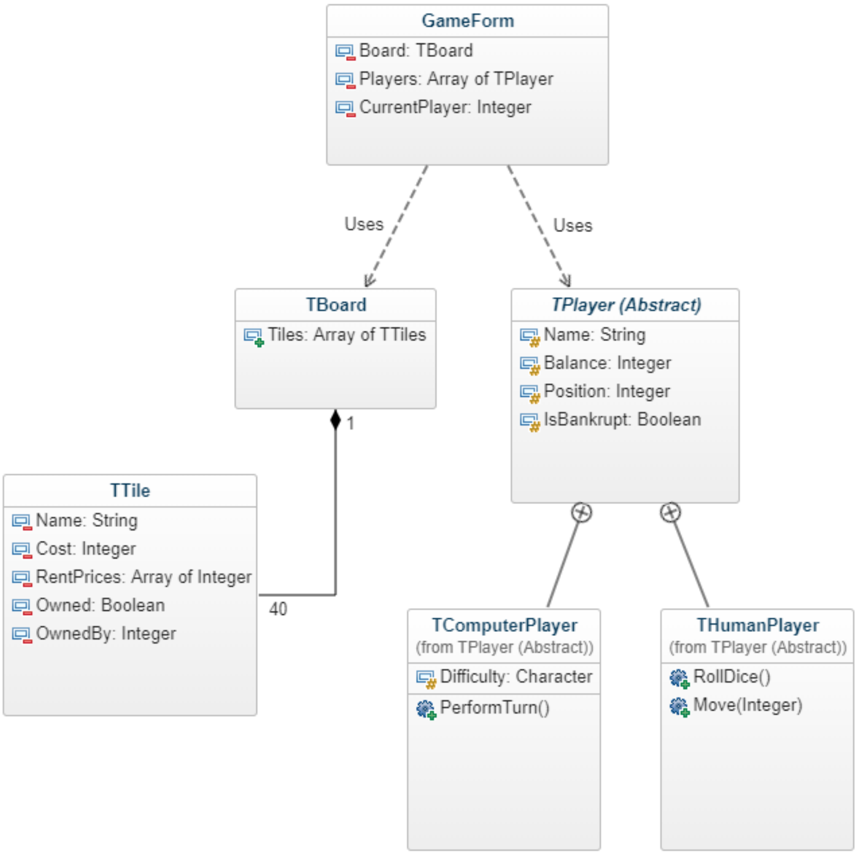
Above is the code for when the test button on the form is clicked to further show the composition relationship as the tiles are called through the board object.
The right shows the output of this prototype which demonstrates that it works. It lists all the different tile names as can indicate that each tile can be independently called and operated on – as it will be used in the final program.



```
Tile 0
Tile 1
Tile 2
Tile 3
Tile 4
Tile 5
Tile 6
Tile 7
Tile 8
Tile 9
Tile 10
Tile 11
Tile 12
Tile 13
Tile 14
Tile 15
Tile 16
Tile 17
Tile 18
Tile 19
Tile 20
Tile 21
Tile 22
Tile 23
Tile 24
Tile 25
Tile 26
Tile 27
Tile 28
Tile 29
Tile 30
Tile 31
Tile 32
Tile 33
Tile 34
Tile 35
Tile 36
Tile 37
Tile 38
Tile 39
```

Entity relationship diagram

Upon further research and thought into the program a clear visual representation of the objects and how they interact with each other would be useful for when it comes to design and programming. Below is a flexible but suitable entity relationship diagram in UML to indicate the relationships between the objects and added are some important variables needed for the program to operate as expected.

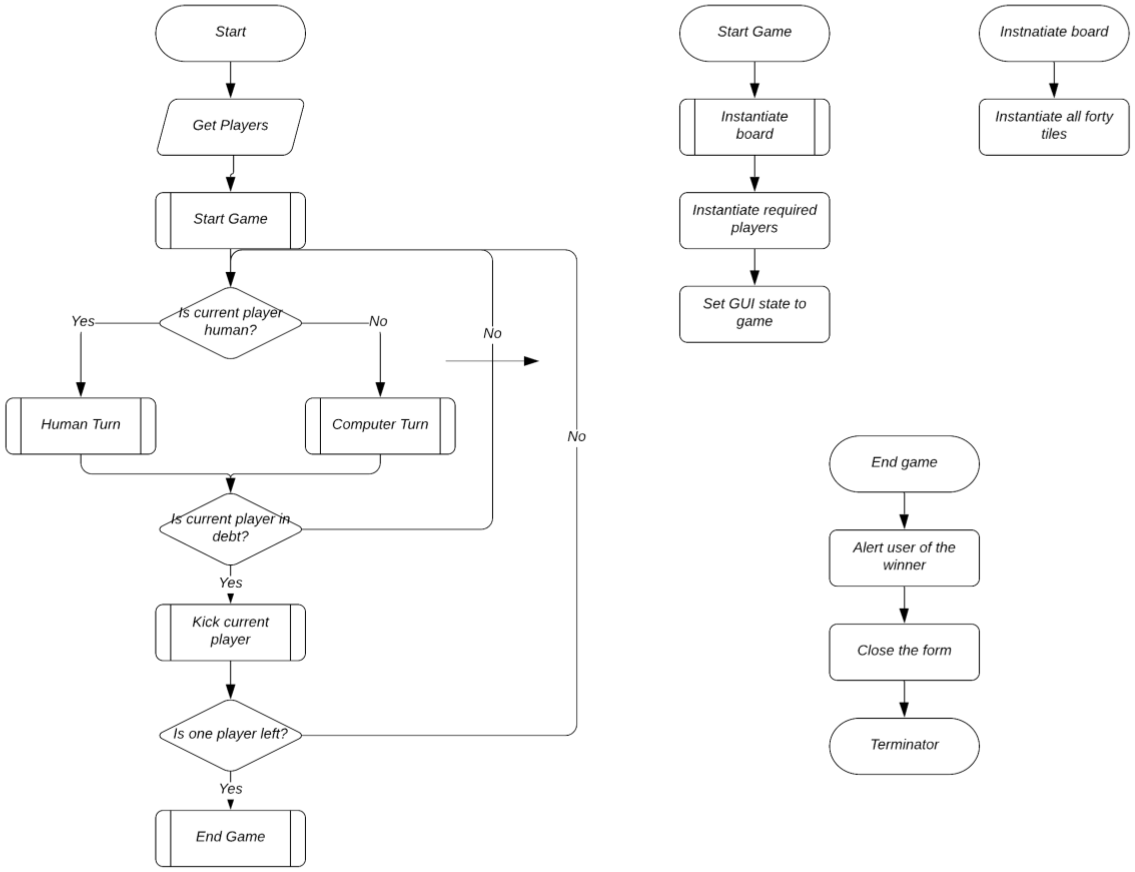


Limitations

Despite their being a lot of successful strategies in Monopoly – many of which rely upon exploiting the probabilities of landing on certain tiles – the overall game is also quite dependant on randomness as the dice rolls generally decide how the game is played. This means that also the hard difficulty setting on the computer-controlled player will not consistently be able to win against a player who makes random moves however they do have a much greater chance.

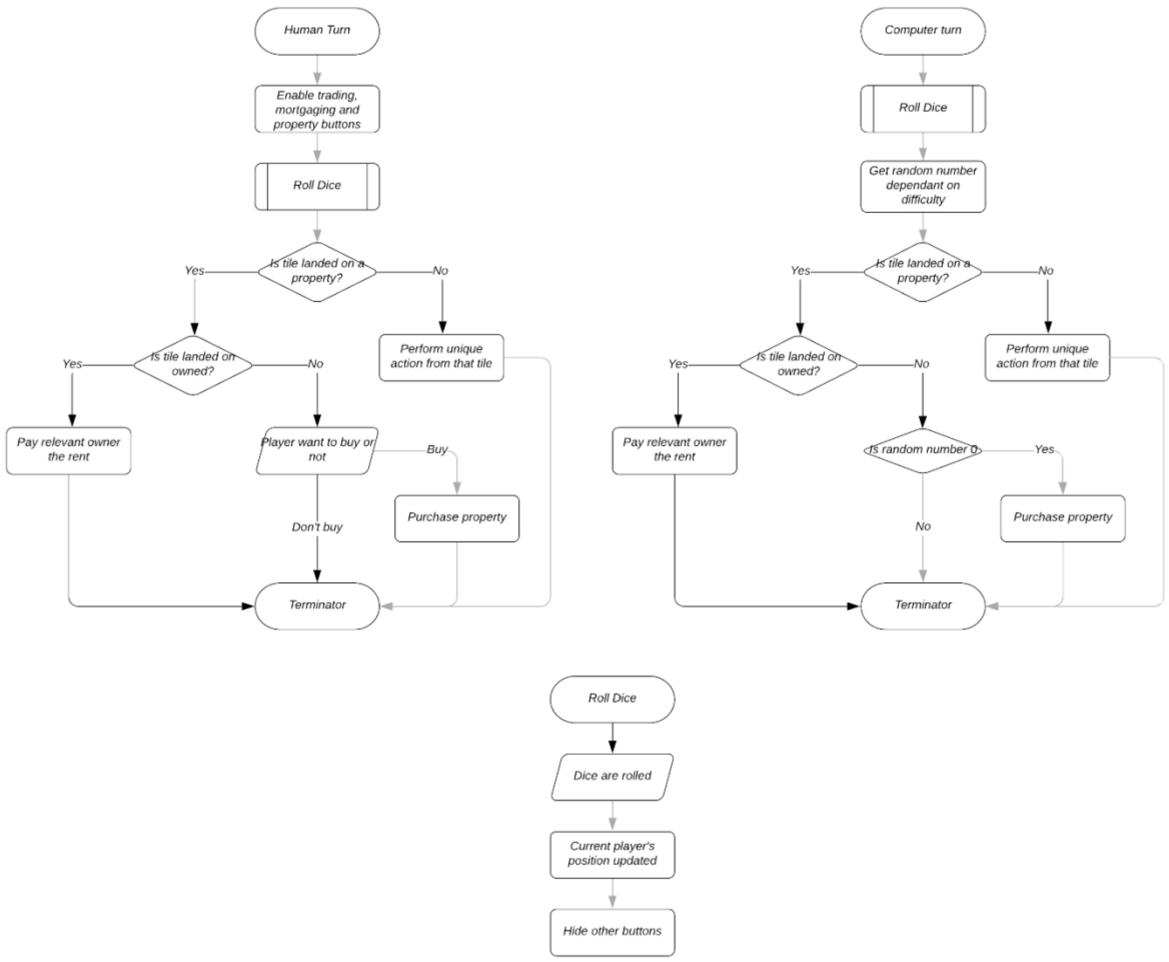
As a result of this, what could be done, once the whole game running as intended and many of the desired features of the end users are implemented, is enable the final game to be able to be played by only two computer controlled players which allows the game to be played very quickly and maybe even multiple times, then analyse what each computer did when they won and avoid doing things when they lose to not only improve their decisions but also possibly come up with new strategies which are based on actual evidence through simulations.

Documented Design



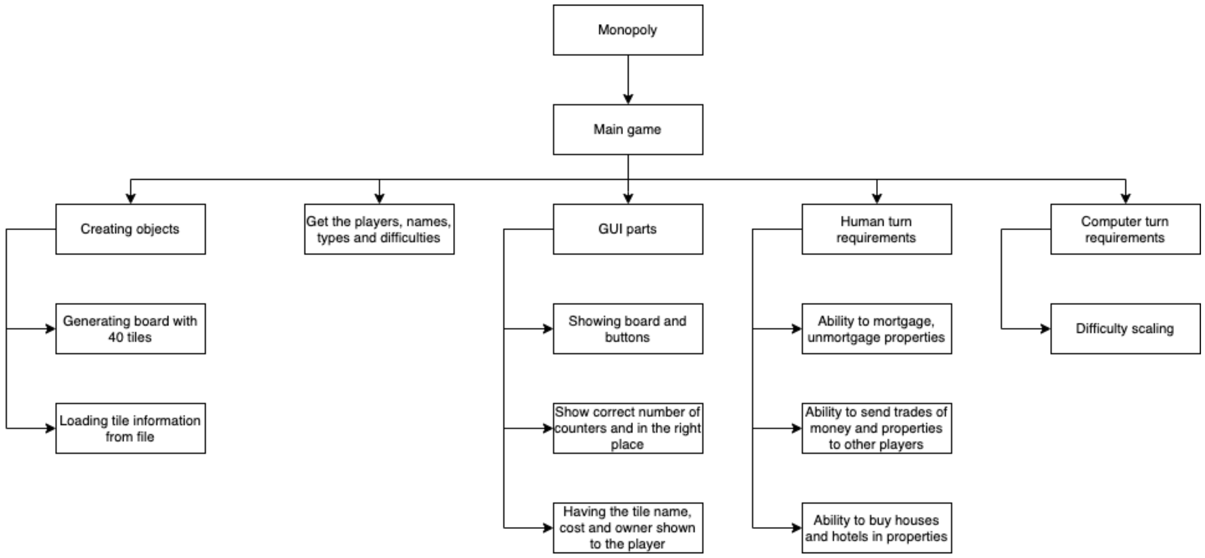
Above is a flow chart of the general procedure of the whole game. It demonstrates from the creation of the form to the end of form a general game with little extra; other functions included within the Monopoly game are absracted from the turn segment as are not frequently run such as trading, mortgaging and buying properties.

Initially, in the get players process, the user is required to input how many players will be playing, their names, and if they will be controlled by the computer or human. Upon the user then clicking start game, the board and output box are created and first player's turn is prepared. In creating the board, as the tiles and board follow a composition relationship, the board created all forty tiles involved in the Monopoly board and gets the names and values as information for that place. If that player is a human then the buttons are enabled and shown to can be clicked, otherwise if it is a computer then their turn is executed. At the end of that player's turn, if they have negative money then they are forced out of the game and can no longer win. The indicator in the circular queue of players is then incremented and loop repeated. If one player remains then that player is the winner and the game is over.


Human Turn
Enable trading, mortgaging and property buttons
Roll Dice
Is tile landed on a property?
Yes
No
Is tile landed on owned?
Yes
No
Perform unique action from that tile
Pay relevant owner the rent
Player want to buy or not
Buy
Purchase property
Don't buy
Terminator
Computer turn
Roll Dice
Get random number dependant on difficulty
Is tile landed on a property?
Yes
No
Is tile landed on owned?
Yes
No
Perform unique action from that tile
Pay relevant owner the rent
Is random number 0
Yes
Purchase property
No
Terminator
Roll Dice
Dice are rolled
Current player's position updated
Hide other buttons

Above is the general flow chart for the different types of players' turns. The human turn involves a lot more buttons to be pressed to control the flow of the program and nothing happens unless they are executed as in event driven programming. At the beginning of the turn, the buttons for mortgaging properties, buying houses and hotels, and creating a trade to another player is shown and the player can use them for its relevent action; otherwise the roll dice button can be clicked and position moved. If their new position is an unowned tile, they can buy it or leave it, otherwise they must pay the relevent owner the relevant rent expected. Most of this is done automatically for ease of use. On the other hand, a computer's turn is executed swiftly and procedurely meaning the game times are much shorter. The computer turn doesn't rely on any buttons to be pressed meaning the games are also much shorter times which appeals to end users.

Top-down diagram

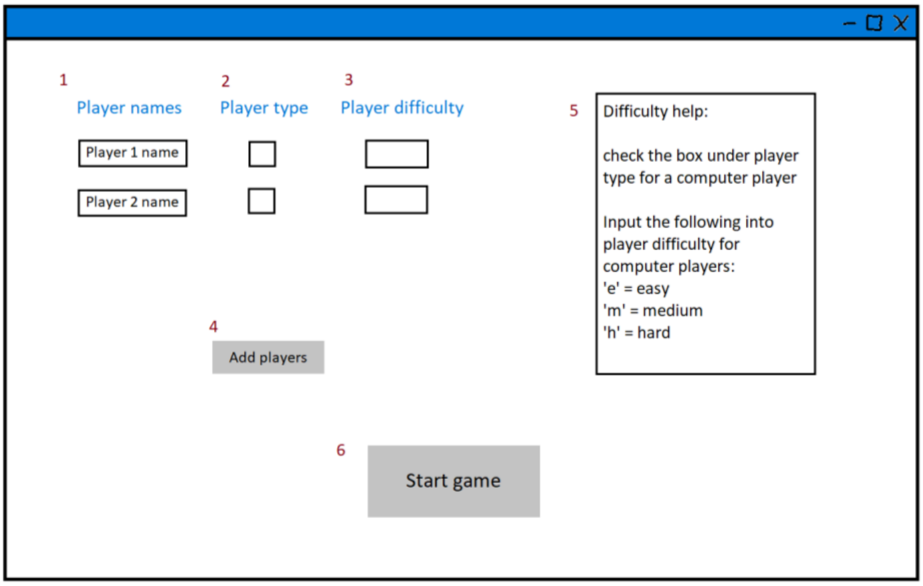

Monopoly
Main game
Creating objects
Generating board with 40 tiles
Loading tile information from file
Get the players, names, types and difficulties
GUI parts
Showing board and buttons
Show correct number of counters and in the right place
Having the tile name, cost and owner shown to the player
Human turn requirements
Ability to mortgage, unmortgage properties
Ability to send trades of money and properties to other players
Ability to buy houses and hotels in properties
Computer turn requirements
Difficulty scaling

IPSO table

Input	Getting player info	Initiates: P1Name, P1Type, P1Diff P2Name, P2Type, P2Diff P3Name, P3Type, P3Diff P4Name, P4Type, P4Diff
	Dice Roll	Updates Roll1 and Roll2 for the current player
	Buy/Don't buy properties	Determines what process should happen next
	Trade button	Creates a trade record and allows input into that
	Money in/out trade	Adds the data inputted here to the trade record and ensures they are integers using regular expression
	Right clicking on a property shape	Action performed on that specific tile upon clicking it depends on which button below is toggled on
	Mortgage button	Tiles shapes are clicked and if owned by the current player, are mortgaged; the user is notified
	Buy properties button	Tile shapes are clicked and if allowed, the current player purchases a house on it; the user is notified
	Use jail card, don't use jail card button	Only showed if the player is in jail and has the get out of jail free cards
Processing	Creating objects	All tiles in the board

		The players and putting them in the circular queue
	Positioning on board	Upon a dice roll, the current player's position is updated
	Normal turn	Action, depending on the current player's position, is executed
	Unowned property	The buy and don't buy buttons are shown
	Owned, un-mortgaged property	The relevant amount of rent is paid to the tile owner
	Owned, mortgaged property	No rent is paid as is mortgaged and current player is notified
	Chance card	A chance card is popped from the stack and action associated with it is executed
	Community chest	A community chest card is popped from the stack and action associated with it is executed
	Free parking	The sum of money collected on it is given to the player who landed on it
	Jail	The player is sent to jail and number of jail turns is started
	Passing go	The player who passes go receives a fixed amount from the bank
Storage	Storing trade	Store the trade inputted by user
Output	Board image	Showing the game board in the correct place
	Movement of counters	The relevant counter is moved to the relevant tile depending on current player's position
	Output memo	Any information such as counter movement or other essential stuff is written to the memo for the user
	Pending trade	User is notified if there is a pending trade to themselves
	A player has won	Create an alert window saying the name of that player has won
	Last remaining player	If there is only one player remaining, the final board state is outputted and a button to close the form is shown

Interface design

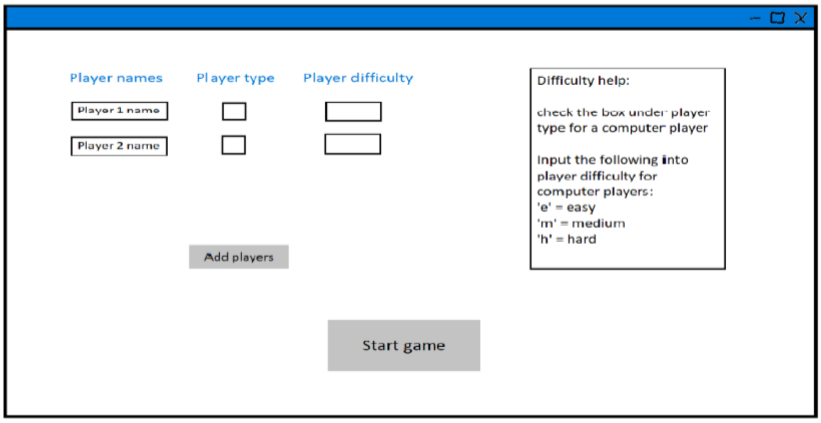


Difficulty help:

check the box under player type for a computer player

Input the following into player difficulty for computer players:
'e' = easy
'm' = medium
'h' = hard

1. The user inputs the names of the players into the edit boxes, if no information is inputted before the start game button is clicked then it defaults to e.g. Player 1
2. The user decides if that player will be controlled by the computer or not
3. If the checkbox in 2 is checked then the input into this box will determine what difficulty the computer will behave as, if no information is inputted as the player is a computer then it defaults to easy
4. This button allows the user to add more players. Initially there are only two rows for players but clicking it adds a row, at four rows the button can no longer be pressed
5. This un-editable box is only to help the user to know what to input into the difficulty boxes
6. Clicking this button confirms the number of players and their types to the game. It then clears the current GUI panel and sets up the main game

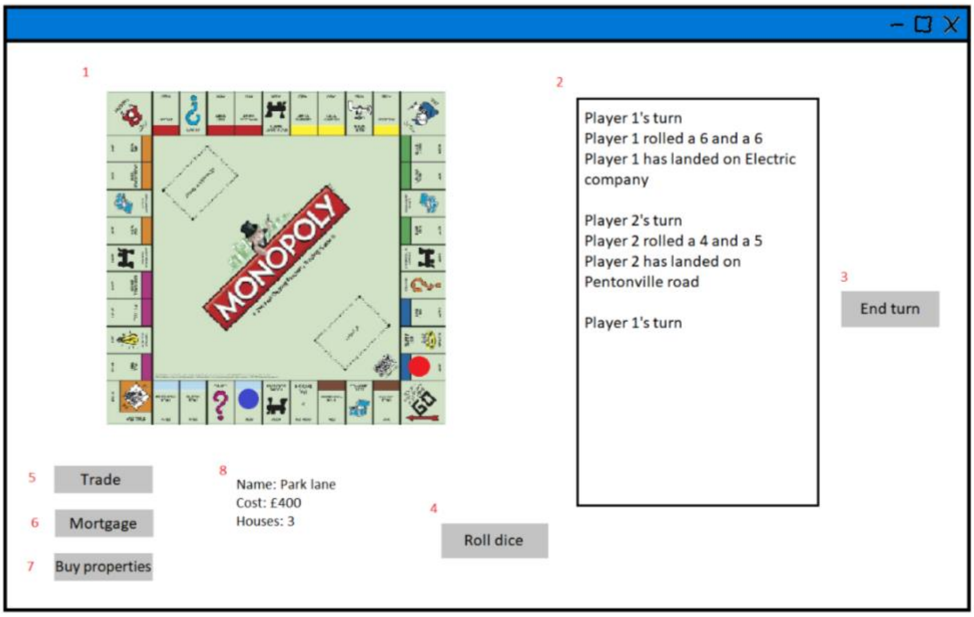


Difficulty help:

check the box under player type for a computer player

Input the following into player difficulty for computer players:
'e' = easy
'm' = medium
'h' = hard

Above is the main player picking page again, however this screenshot demonstrates how it will use a Grid Panel to scale the components in the form allowing for different sized or shaped screens meaning the program can appeal to more users as will dynamically size.



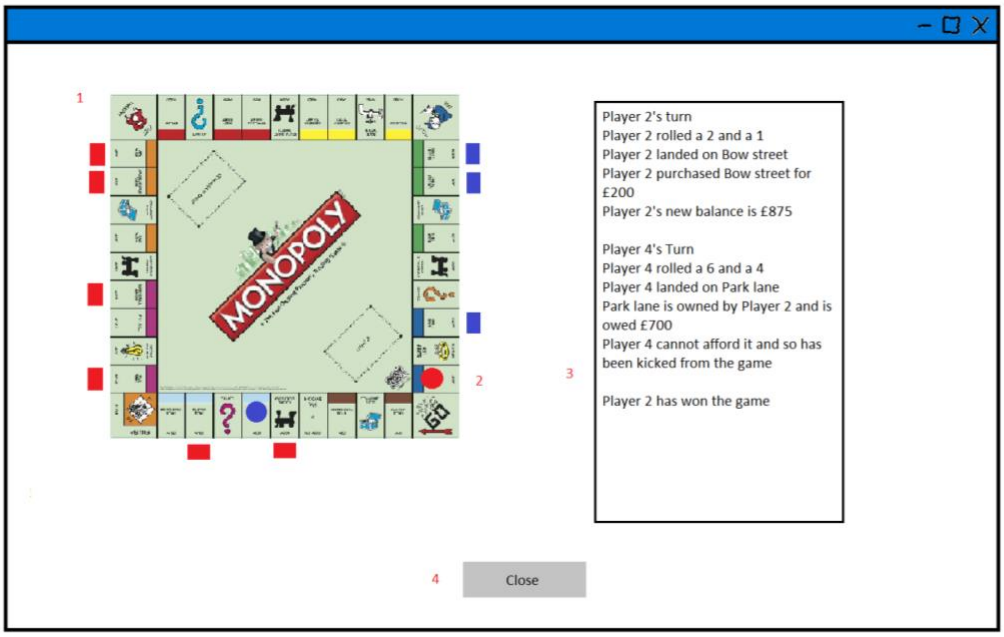
Player 1's turn
Player 1 rolled a 6 and a 6
Player 1 has landed on Electric company

Player 2's turn
Player 2 rolled a 4 and a 5
Player 2 has landed on Pentonville road

Player 1's turn

Name: Park lane
Cost: £400
Houses: 3

1. This is the board image and matches that of the original board game. The respective counters move around the board and match the positions of that player
2. This is the output memo which notifies the users of all the goings on in the game
3. Upon all requirements of a turn are completed such as rolling the dice and accepting trades (if any), this button is enabled and must be pressed before the next player can take their turn
4. Clicking the roll dice button generates two random numbers from 1-6 inclusive and move the player that number of spaces forward and then action on that tile performed – the counters are moved at this point
5. Clicking this initiates a trade and sets up the GUI to the later images
6. Clicking this allows players to then right click tiles that they own and mortgages/unmortgages them, the computer works out which one they wish to do by knowing its current state
7. Clicking this allows players to then right click tiles that they own and if able – by owning the whole range, having funds and not 5 properties already on that tile – purchase one
8. As the best image for a UK Monopoly board doesn't have good enough resolution, to make it easier for the user upon hovering over a tile with the mouse these 3 labels appear to show information about that tile and match the real time data about the board

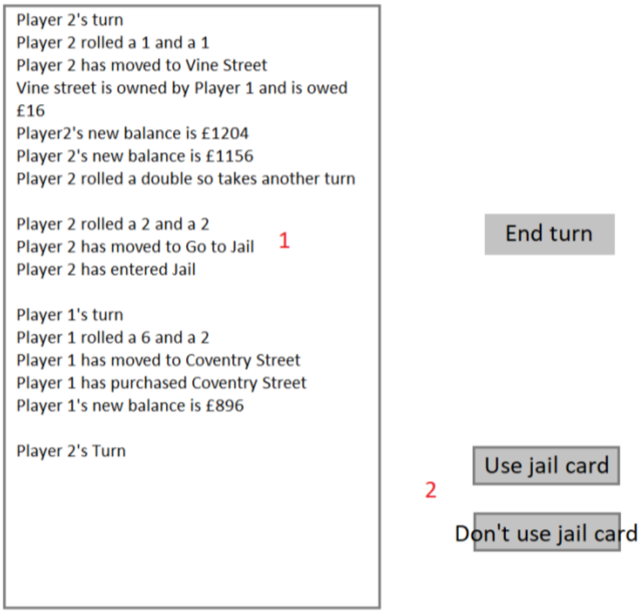


Player 2's turn
Player 2 rolled a 2 and a 1
Player 2 landed on Bow street
Player 2 purchased Bow street for £200
Player 2's new balance is £875

Player 4's Turn
Player 4 rolled a 6 and a 4
Player 4 landed on Park lane
Park lane is owned by Player 2 and is owed £700
Player 4 cannot afford it and so has been kicked from the game

Player 2 has won the game

1. This is an example of a board after a game. The game ran with four computer players against each other, and this would be an example result. Player 2 has won as you can see from the memo box
2. These are the counters that cycle around the board in the correct positions
3. The memo box shows that upon players being kicked out, they no longer play until the last player remains and another window as an alert pops out to notify the last player that they have won
4. This button only appears when the game is over, clicking it closes the form



Player 2's turn
Player 2 rolled a 1 and a 1
Player 2 has moved to Vine Street
Vine street is owned by Player 1 and is owed £16
Player2's new balance is £1204
Player 2's new balance is £1156
Player 2 rolled a double so takes another turn

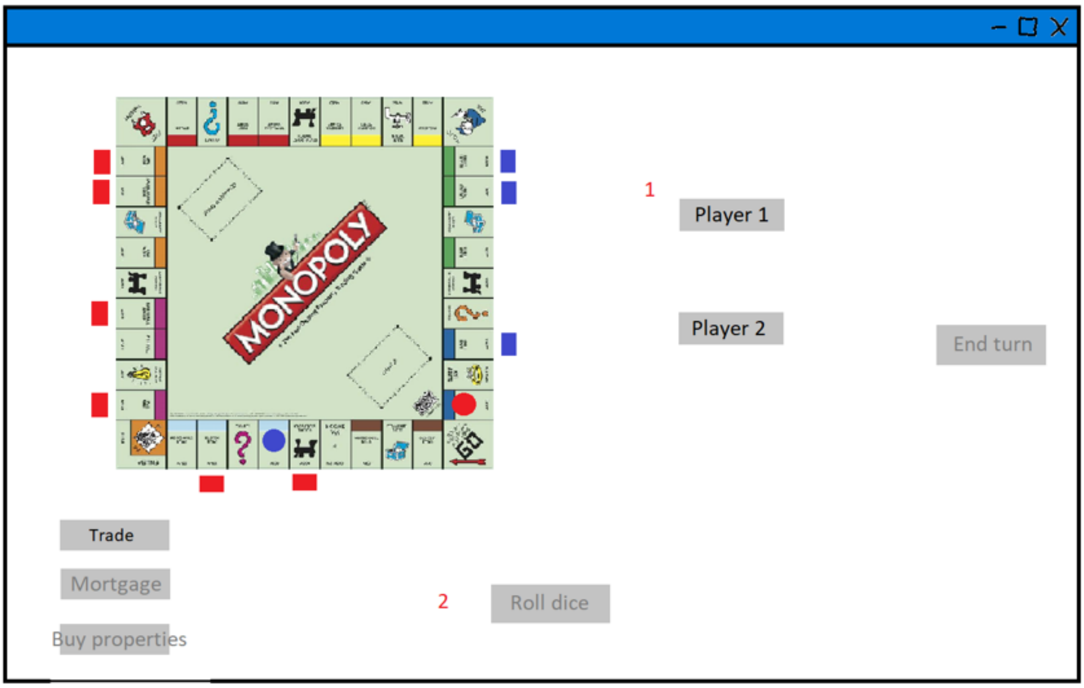
Player 2 rolled a 2 and a 2
Player 2 has moved to Go to Jail
Player 2 has entered Jail

Player 1's turn
Player 1 rolled a 6 and a 2
Player 1 has moved to Coventry Street
Player 1 has purchased Coventry Street
Player 1's new balance is £896

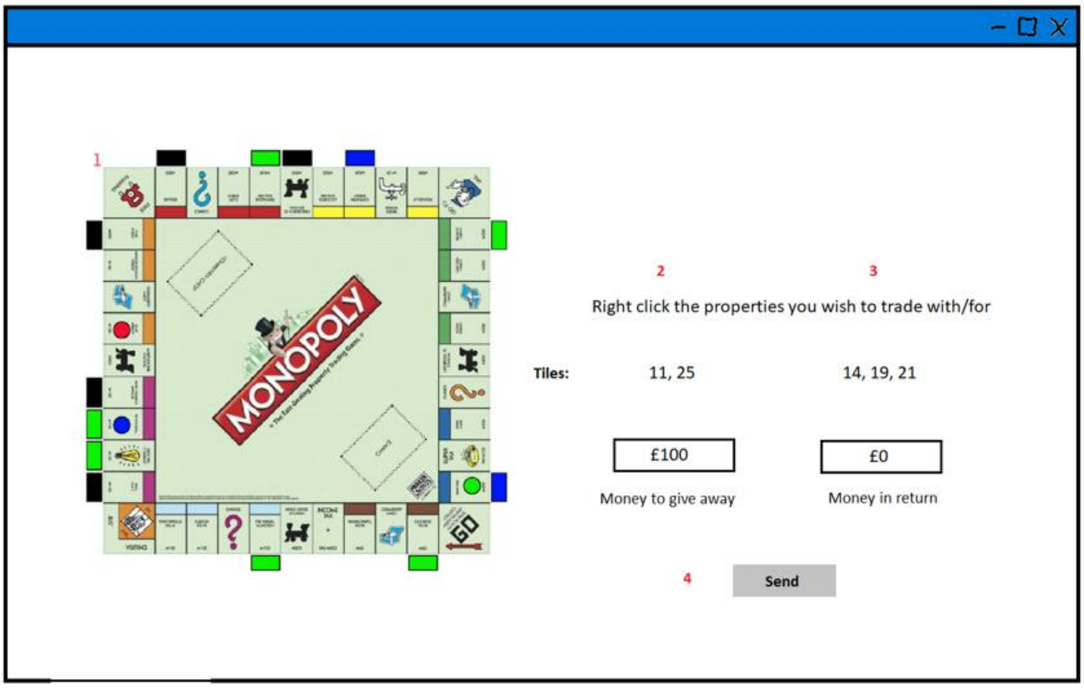
Player 2's Turn

1. This should appear in the memo box and indicates to the user that player 2 has gone to jail, also the counter on the board moves to the jail position

2. If the player in jail also has a get out of jail free card received from the chance or community chest draws, then these buttons appear and give the player the choice to use them



1. This button only appears upon clicking the trade button. As player two clicked it, only the other players buttons who are in the game and alive should appear
2. The other buttons will be disabled for when in trading as will interfere with the trading and in normal Monopoly you cannot end a turn during a trade



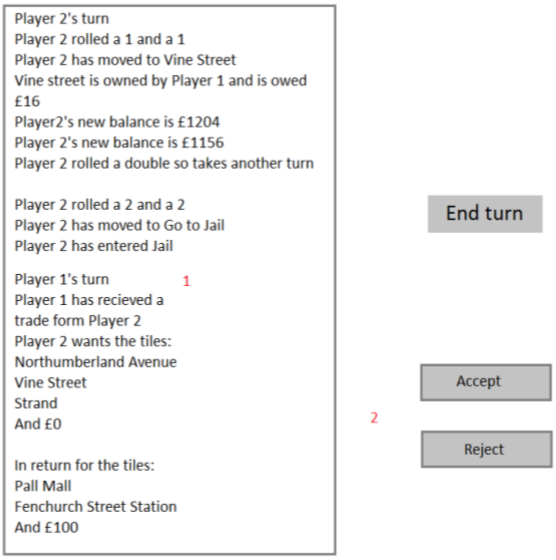
Right click the properties you wish to trade with/for

Tiles: 11, 25 14, 19, 21

£100 £0

Money to give away Money in return

1. Upon clicking one of the player buttons in a trade, in this case player 1, the next stage of trading appears.
2. This is the information about the current players turn (Who will have created the trade). Upon clicking any tile indicator owned by the players involved in the trade, they turn black and computer works out which side that tile should appear in e.g. if owned by the current player then the computer will know that the user is offering that to give away so is appended to that side. The money to give away box will default to zero but otherwise if player puts in a value then is assumed to be the money offered. Regular expression will be used to ensure that the data inputted into the boxes are integers only, if it doesn’t match the regex statement then they default to 0.
3. This is very similar to 2, but just is what the current player wants in return for the trade they are offering to give away
4. When the trade is decided, that player can send the trade to the other user and the record is submitted

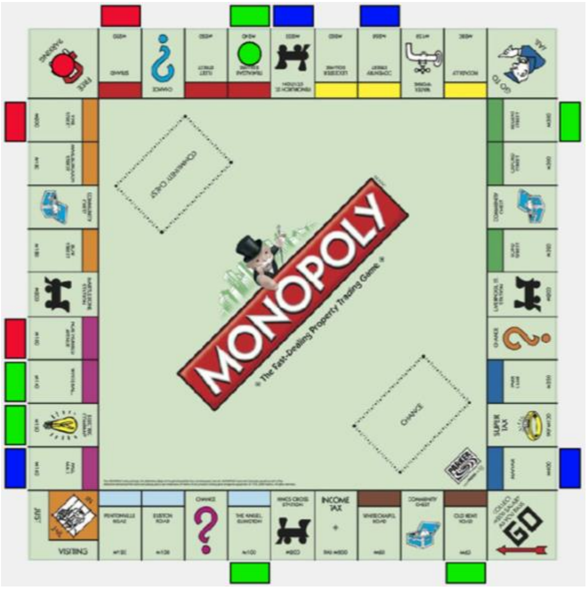


```
Player 2's turn
Player 2 rolled a 1 and a 1
Player 2 has moved to Vine Street
Vine street is owned by Player 1 and is owed
£16
Player2's new balance is £1204
Player 2's new balance is £1156
Player 2 rolled a double so takes another turn

Player 2 rolled a 2 and a 2
Player 2 has moved to Go to Jail
Player 2 has entered Jail

Player 1's turn
Player 1 has recieved a
trade form Player 2
Player 2 wants the tiles:
Northumberland Avenue
Vine Street
Strand
And £0

In return for the tiles:
Pall Mall
Fenchurch Street Station
And £100
```

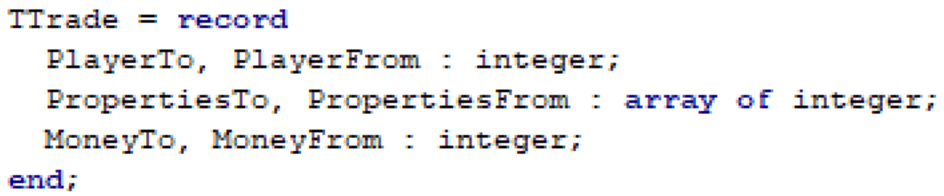
1. Once a trade is submitted, no other trades can be made until the player the trade was sent to has their turn. Once it is, the whole trade will be displayed in the memo box

2. The current player (who the trade has been sent to) must click either accept or reject to the trade before they can continue their turn. By clicking accept the trade occurs and program organises this, otherwise clicking reject clears the trade and future trades from other players can then occur again

This image is just to show an example board state after a trade has occurred (In particular, the one in the above images). It demonstrates how all the tiles offered in the trade are swapped and the ownership changes



Record definitions



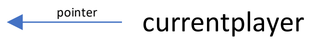
```
TTrade = record
  PlayerTo, PlayerFrom : integer;
  PropertiesTo, PropertiesFrom : array of integer;
  MoneyTo, MoneyFrom : integer;
end;
```

Record Element	Field Type	Use
PlayerTo	Integer	Stores the player number e.g. 0 = Player 1, of who the trade is directed to
PlayerFrom	Integer	Stores the player number e.g. 0 = Player 1, of who the trade record was created by
PropertiesTo	Dynamically sized array of integers	Stores an array of integers which refer to the tiles which the player who created the trade wants to give away e.g. 37, 39 = Park Lane and Mayfair
PropertiesFrom	Dynamically sized array of integers	Stores an array of integers which refer to the tiles which the player who created the trade wants in return e.g. 21, 29 = Strand and Piccadilly
MoneyTo	Integer	Stores the amount of money which the player who created the trade wants to give away e.g. 200 = £200
MoneyFrom	Integer	Stores the amount of money which the player who created the trade wants in return e.g. 0 = no money

Circular Queue of players

e.g. Players : array of TPlayer

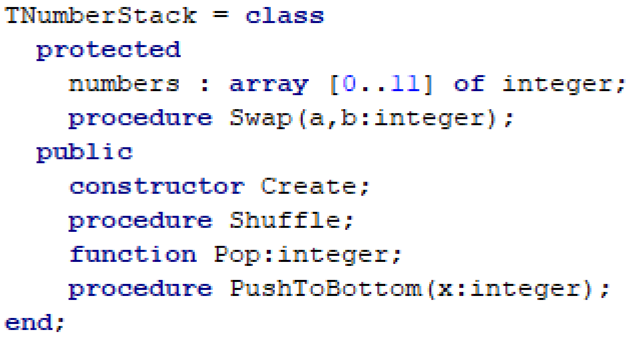
Player 1	Alive
Player 2	Alive
Player 3	Not Alive
Player 4	Alive



When the button ‘Start Game’ is clicked, the information inputted from the user at the GetPlayers stage is used to create all the players involved in that instance of the game, all of which start alive. In the above example we have initially started with four players. The currentplayer variable which acts as the circular pointer firstly points to player 1 by having the value 0 set to it. At the end of player 1’s turn in the procedure called EndTurnProcedure another procedure called nextplayer is called; this increments the currentplayer pointer by 1 and mods it with the number of players first declared when the game started. This allows for a circular nature as if at player 4, so currentplayer = 3, then incrementing it sets it to 4 and then modding it with 4 sets it back to 0 and repeats. If that player is not alive then the nextplayer procedure is called again as part of itself demonstrating recursion. In the above example the game has been going on for a bit and Player 3 has been forced out the game so is no longer alive; at the end of Player 2’s turn currentplayer is incremented from 1 to 2, however Player 3 is not alive so it is called again to go from 2 to 3 where player 4 is alive so the procedural calls are ended and the stack produced from the recursion is popped until empty resulting in player 4’s turn.

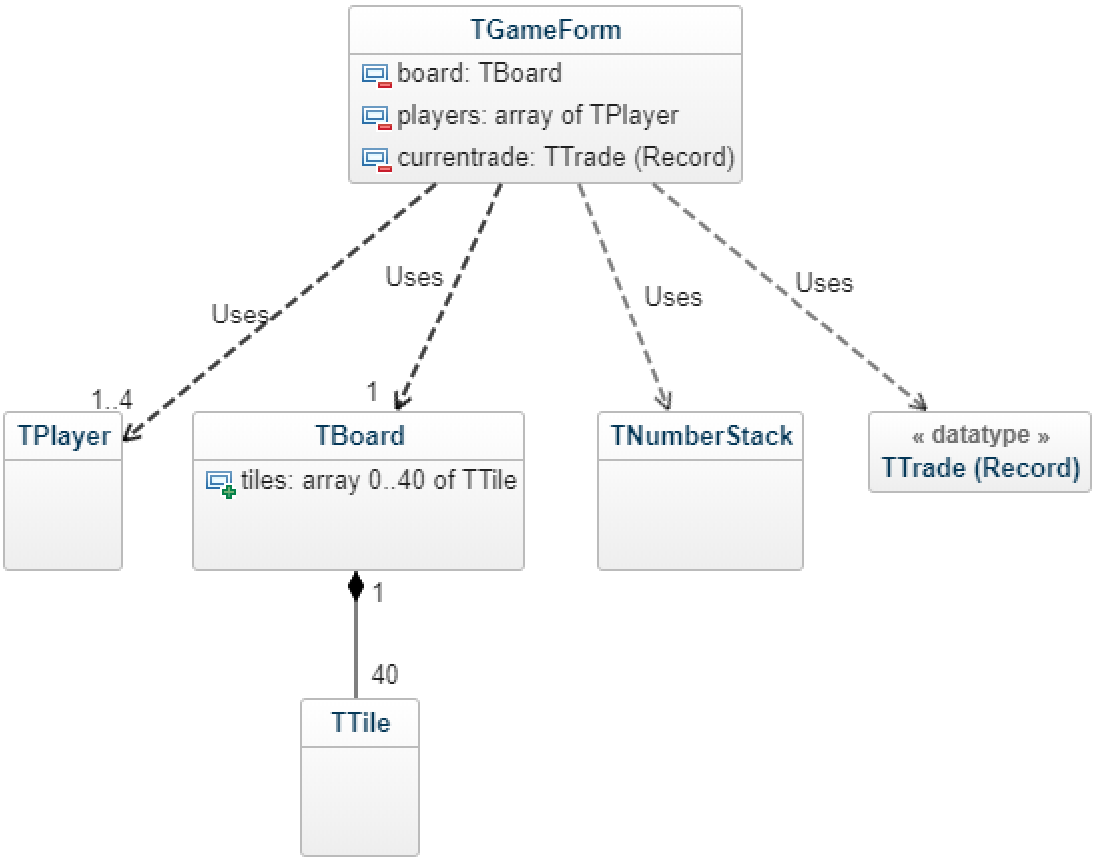
Stack of numbers

A stack of numbers which will behave slightly differently to a normal stack will be required. Firstly, when created, it should fill the array with the numbers 1-12 in order. Then it will use the Fisher-Yates shuffle algorithm to shuffle the numbers by repeatedly taking two random elements in the list and swapping their positions if the two elements are not matching. Once shuffled, it now behaves as a stack and the numbers are pushed and popped from it. It will be used for the stack of chance cards and stack of community chest cards as in the true board game they are a stack. They are only a stack of numbers as the actions that depend on the card drawn must be written in the main form due to encapsulation and data hiding as e.g. unable to write to the memo box from the uStack.pas unit as the memo box would be private in uGameForm.pas. Upon drawing the top number, a case statement determines what action is mapped to that number and it is executed. Then the number is effectively pushed onto another stack and shouldn’t be drawn until all other cards are drawn; however, could be optimally implemented instead by adding it to the bottom of the stack.

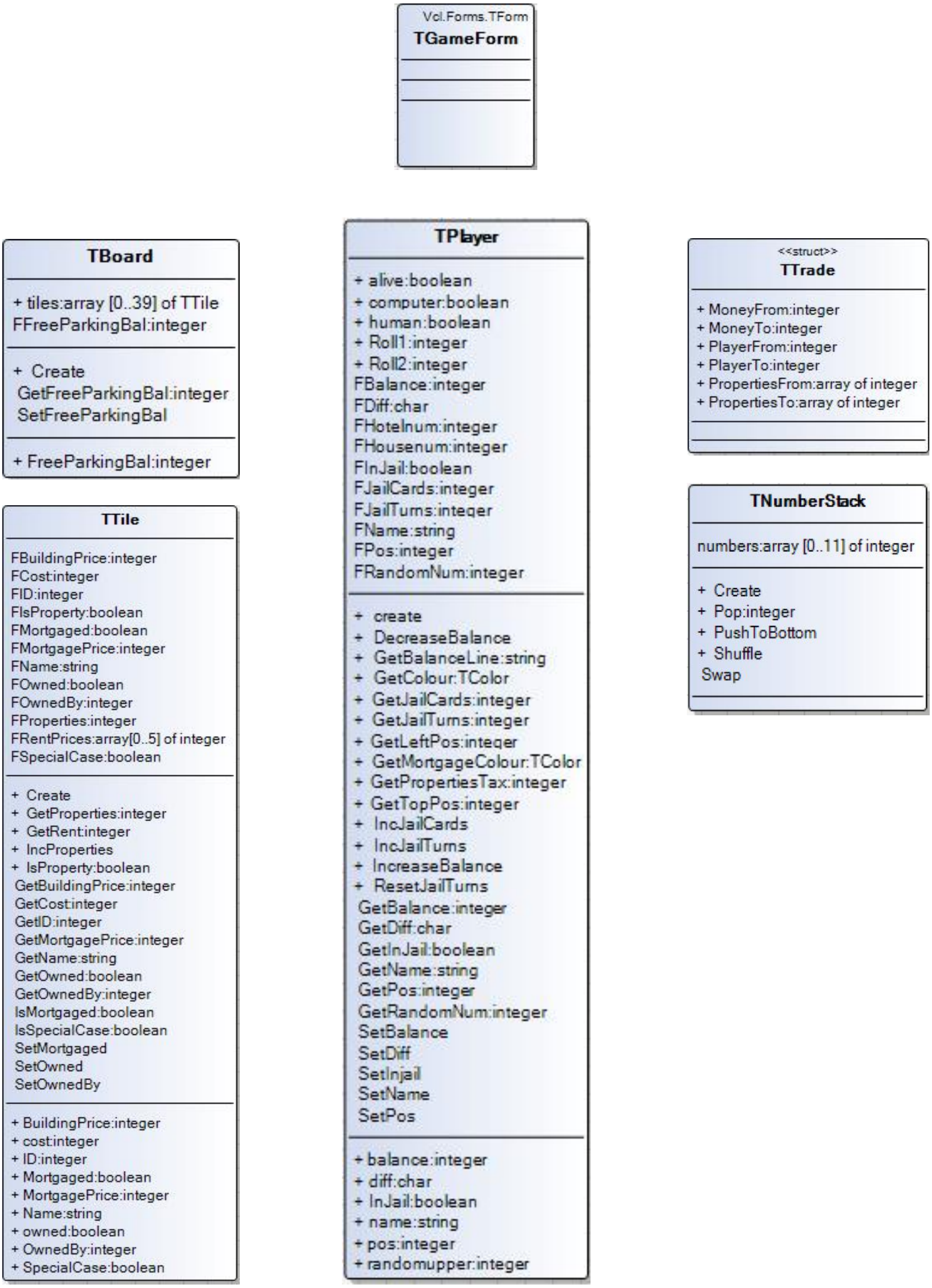


```
TNumberStack = class
  protected
    numbers : array [0..11] of integer;
    procedure Swap(a,b:integer);
  public
    constructor Create;
    procedure Shuffle;
    function Pop:integer;
    procedure PushToBottom(x:integer);
end;
```

OOP diagram

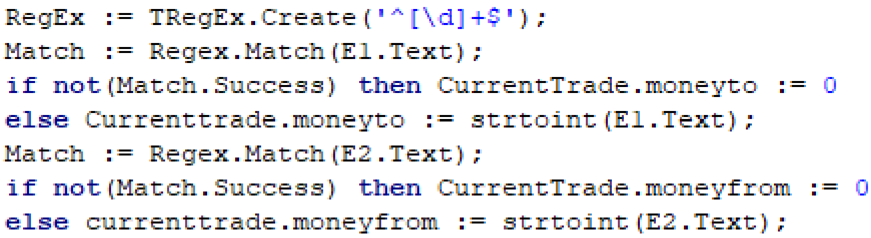


Upon further prototyping and research, It is not possible to use the architecture first described in the analysis OOP diagram as records would be used when creating a trade, as well as the fact that using TPlayer as an abstract class and then having two different classes, one for humans and one for computer players would mean that you would need to have an array of two different types. This a limitation of Delphi as cannot be done unless creating a record and have the data types in there but then would need to have every different possibility and it would no longer behave as a circular queue, therefore instead it would be best to go with one TPlayer object but they contain a specific procedure for the different types of players which are uniquely called at the start of each turn e.g. NormalTurn and ComputerTurn. This also allows for much easier implementation of the rolling a double feature as the turn procedures could just be recursively called.


Vcl.Forms.TForm
TGameForm
TBoard
+ tiles:array [0..39] of TTile
FFreeParkingBal:integer
+ Create
GetFreeParkingBal:integer
SetFreeParkingBal
+ FreeParkingBal:integer
TTile
FBuildingPrice:integer
FCost:integer
FID:integer
FIsProperty:boolean
FMortgaged:boolean
FMortgagePrice:integer
FName:string
FOwned:boolean
FOwnedBy:integer
FProperties:integer
FRentPrices:array[0..5] of integer
FSpecialCase:boolean
+ Create
+ GetProperties:integer
+ GetRent:integer
+ IncProperties
+ IsProperty:boolean
GetBuildingPrice:integer
GetCost:integer
GetID:integer
GetMortgagePrice:integer
GetName:string
GetOwned:boolean
GetOwnedBy:integer
IsMortgaged:boolean
IsSpecialCase:boolean
SetMortgaged
SetOwned
SetOwnedBy
+ BuildingPrice:integer
+ cost:integer
+ ID:integer
+ Mortgaged:boolean
+ MortgagePrice:integer
+ Name:string
+ owned:boolean
+ OwnedBy:integer
+ SpecialCase:boolean
TPlayer
+ alive:boolean
+ computer:boolean
+ human:boolean
+ Roll1:integer
+ Roll2:integer
FBalance:integer
FDiff:char
FHotelnum:integer
FHousenum:integer
FInJail:boolean
FJailCards:integer
FJailTurns:integer
FName:string
FPos:integer
FRandomNum:integer
+ create
+ DecreaseBalance
+ GetBalanceLine:string
+ GetColour:TColor
+ GetJailCards:integer
+ GetJailTurns:integer
+ GetLeftPos:integer
+ GetMortgageColour:TColor
+ GetPropertiesTax:integer
+ GetTopPos:integer
+ IncJailCards
+ IncJailTurns
+ IncreaseBalance
+ ResetJailTurns
GetBalance:integer
GetDiff:char
GetInJail:boolean
GetName:string
GetPos:integer
GetRandomNum:integer
SetBalance
SetDiff
SetInjail
SetName
SetPos
+ balance:integer
+ diff:char
+ InJail:boolean
+ name:string
+ pos:integer
+ randomupper:integer
<<struct>>
TTrade
+ MoneyFrom:integer
+ MoneyTo:integer
+ PlayerFrom:integer
+ PlayerTo:integer
+ PropertiesFrom:array of integer
+ PropertiesTo:array of integer
TNumberStack
numbers:array [0..11] of integer
+ Create
+ Pop:integer
+ PushToBottom
+ Shuffle
Swap

Input validation

Upon a user clicking the trade button, the trade GUI panel should be shown involving the correct form elements being shown. As well as clicking on properties to incorporate into the trade, the user will also enter amounts of money which the user is willing to give away/ wants in return. As the integer value of this data was going to be used in the TTrade record, whose MoneyTo and MoneyFrom fields are integer types, the program would need to ensure that the data was only of the integer type. The best way to do this was use regular expression. A regular expression statement which matches any digits only would need to be created, and then using the TMatch object it would need to compare the inputted data to the expression. If it accepts it then it added to the record, if not then it defaults to 0. This ought to be done for both edit boxes (for money to the directed player and money from the directed player). Overall the use of input validation would enable the program to be more robust for the user and means it is unable to crash and produce any errors making the software more professional.



```
RegEx := TRegEx.Create('^[\d]+$');
Match := Regex.Match(E1.Text);
if not(Match.Success) then CurrentTrade.moneyto := 0
else Currenttrade.moneyto := strtoint(E1.Text);
Match := Regex.Match(E2.Text);
if not(Match.Success) then CurrentTrade.moneyfrom := 0
else currenttrade.moneyfrom := strtoint(E2.Text);
```

Algorithms

```
Procedure ComputerTurn
   Randint ← random(players[currentplayer].RandomUpper)
      Double ← False

      If (PendingTrade = True) and (currentrade.playerto = currentplayer) then
         IF randint = 0 THEN accepttrade
         ELSE rejectrade

   Roll1 ← random(6) + 1
   Roll2 ← random(6) + 1
   Name ← players[currentplayer].name

   Output(Name + ''s has rolled a ' + Roll1 + ' and a ' + Roll2)

   IF Roll1 = Roll2 THEN double ← True

   IF (players[currentplayer].Pos + Roll1 + Roll2) > 39 THEN PassedGo

   IF players[currentplayer].InJail = False THEN
      Players[currentplayer].Pos  ← (Players[currentplayer].Pos+ Roll1 + Roll2) MOD 40
      NewPos← Players[currentplayer].Pos
      Output('Name has landed on ' + Board.Tiles[newpos].Name)
```

```
    IF Board.tiles[newpos].IsProperty = True THEN
        IF Board.tiles[newpos].owned = False THEN
            IF Randint = 0 THEN
                Board.tiles[newpos].owned ← True
                Board.tiles[newpos].OwnedBy ← currentplayer
                Output(Name + ‘ bought the tile ‘)
        ELSE
            IF Board.tiles[newpos].morgaged= True THEN
                Output(‘Property mortgaged’)
            ELSE
                IF Board.tiles[newpos].OwnedBy <> Currentplayer THEN
                    Rent ← Board.tiles[newpos].GetRent
                    Players[currentplayer].DecreaseBalanceBy(rent)
                    Owner ← Board.tiles[newpos].ownedby
                    Players[Owner].IncreaseBalanceBy(Rent)
                    Output(Players[Owner].name + ‘ owned ‘ Board.tiles[newpos].name)
                    Output(Name + ‘ paid £’ + Rent + ‘ to Players[Owner].Name)
    ELSE
        CASE NewPosOF
        4:
            Inc(Board.FreeParkingBalance,200)
            Players[currentplayer].DecreaseBalanceBy(200)
            Ouptut(Name + ‘ landed on income tax’)
        20:
            Players[currentplayer].IncreaseBalanceBy(Board.FreeParkingBalance)
            Output(Name + ‘ got £‘ + Board.FreeParkingBalance + ‘ from free parking’)
            Board.FreeParkingBalance ← 0
        30:
            Players[Currentplayer].InJail ← True
            Players[Currentplayer].IncJailTurns
            Players[Currentplayer].Pos← 10
            Output(Name + ‘ has been moved to jail’)
        38:
            Inc(Board.FreeParkingBalance,100)
            Players[currentplayer].DecreaseBalanceBy(100)
            Ouptut(Name + ‘ landed on super tax’)
        7, 22, 36: LandedOnChance
        2, 17, 33: LandedOnCommunityChest
ELSE
    IF players[currentplayer].JailCards > 0 THEN UseJailCard
    ELSE
        IF double = True THEN
            Players[currentplayer].InJail ← False
            Players[currentplayer].JailTurns ← 0
            Computerturn
        ELSE Ouptput(Name + ‘ is still in jail ‘)
    IF Randint = 2 THEN ComputerBuyProperties

    IF double = True THEN Computerturn
    ELSE EndOfTurnProcedure

END Procedure
```

The above pseudocode describes the computer turn procedure which is called at least once per turn then the player is controlled by the computer.

- Initially it gets sets 'Randint' to a value dependent on the difficulty of the player to control what actions it does in the turn
- If there is a pending trade and randint happens to be 0 then the trade is accepted otherwise it is rejected
- If the player is in jail and has a get out of jail free card then it uses it by decreasing the number of cards they have, moving them out of jail and calling ComputerTurn again as a recursive function; all in the procedure UseJailCard
- Also, if the player rolls a double, they leave jail and the procedure is recursively called to act as another turn
- Then it rolls the dice and if they are a double it assigns true to that variable
- Following the dice roll, it moves the player around the circular board and checks if it goes past go, if so, it receives £200 from the bank in the procedure called 'PassedGo'
- It notifies the user what tile that player landed on
- If the player lands on an unowned property, then depending on the value of Randint it either buys it or leaves it
- If the player lands on a property, owned by another play and unmortgaged then the value of rent, determined by the number of houses on that tile, is paid from the current player to the owner
- If the player lands on a mortgaged property, then no money is exchanged
- If the player lands on position 4 then income tax is paid to free parking
- If the player lands on position 20 then the balance in free parking is paid to that player and the pot is reset
- If the player lands on position 30 then the player is sent to jail and variables involved in that are initialised
- If the player lands on position 38 then super tax is paid to free parking
- If the player lands on position 7, 22 or 36 then the LandedOnChance procedure is called which draws a chance card and executes the action described on it
- If the player lands on position 2, 17 or 33 then the LandedOnCommunityChest procedure is called which draws a community chest card and executes the action described on it
- If Randint allows it then the computer buys houses on its owned properties by going through as many as it can and buying at least one on each property. Also, if its

difficulty is hard then ComputerBuyProperties is similarly recursively defined so hard difficulty players buy as many houses as they can afford on the colour ranges they own

Data dictionary for global variables

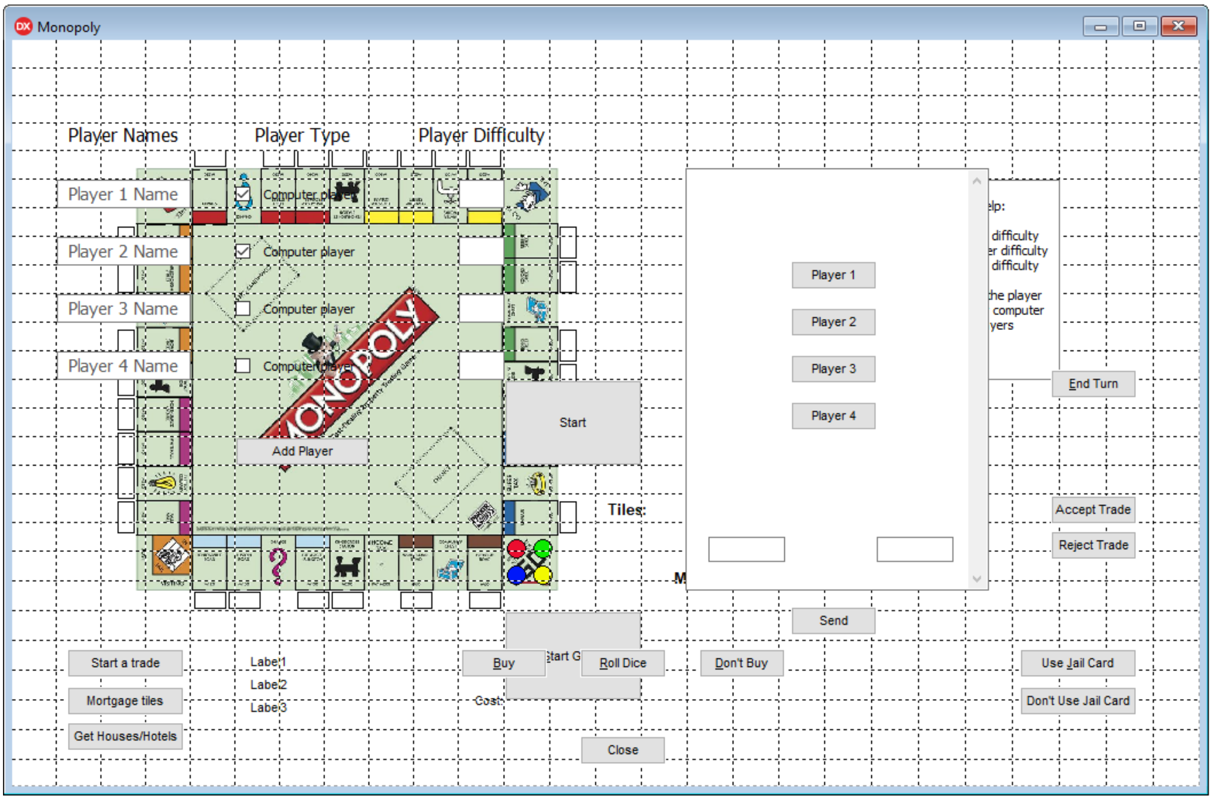
Variable name	Variable type	Sample data	Usage
Playernum	Integer	3	Holds the value of the number of alive players at the start of the game
Board	TBoard		Hold the board as an object including an array of TTiles which the game relies upon
Players	Array of TPlayer		Holds all the players who start the game and behaves as a circular queue
CurrentPlayer	Integer	0	Acts as a pointer for the circular queue to point to the player whose turn it is
TradingPlayer	Integer	1	Acts as a pointer to the player who created the CurrentTrade
ChanceStack	TNumberStack	[4,5,6…1]	Stores the number stack which determines what chance cards are drawn
CChestStack	TNumberStack	[11,1,3…5]	Stores the number stack which determines what community chest cards are drawn
Double	Boolean	False	Holds true if the most recent roll was a double
Intrade	Boolean	True	Holds true if the game is currently in trading mode, as determines what happens when a tile shape is clicked
Mortgaging	Boolean	False	Holds true if the current player is mortgaging properties, as determines what happens when a tile shape is clicked
BuyingHouses	Boolean	False	Holds true if the current player is purchasing houses, as determines what happens when a tile shape is clicked
Pendingtrade	Boolean	True	Holds true if there is a CurrentTrade active so no other trades can wipe the record

Currenttrade	TTrade		Holds all the information of a trade as a record if PendingTrade is true
P1alive	Boolean	True	Indicator just to show if that player is alive in the game
P2alive	Boolean	False	Indicator just to show if that player is alive in the game
P3alive	Boolean	True	Indicator just to show if that player is alive in the game
P4alive	Boolean	False	Indicator just to show if that player is alive in the game
Ingame	Boolean	True	Boolean to disable some features until the game has started e.g. hovering over tiles to display the labels
AllComputerPlayers	Boolean	False	Holds true if all the players in the game are computer controlled so doesn't update the GUI, as would be wasted, until the end of the game
ScaleFactor	Integer	1	Holds the value which the coordinates are scaled by to enable the counters to be in the correct position depending on the resolution of the screen
PropertyShapes	Array [0..39] of TShape		Holds the array of TShapes created on the form as were needed to by individually called by the position they are on the board
PropertyIDs	Array [0..27] of integer	*	Holds an array of integers as a constant which are the ID's of positions in the board which are properties

* = (1,3,5,6,8,9,11,12,13,14,15,16,18,19,21,23,24,25,26,27,28,29,31,32,34,35,37,39)

Technical Solution

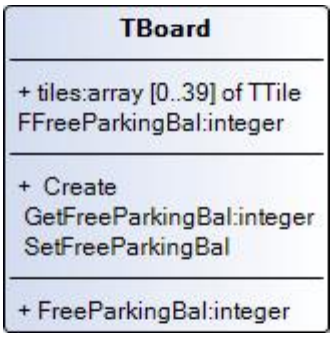
RAD Studio interface



Annotated object class definitions

TBoard

```
TBoard = class
  protected
    FFreeParkingBal:integer;

    function GetFreeParkingBal:integer;
    procedure SetFreeParkingBal(v:integer);

  public
    //This array holds the 40 tiles that compose the physical board
    tiles : array [0..39] of TTile;

    constructor Create;
    //This property enabled interaction with the Field FFreeParkingBal
    property FreeParkingBal:integer read GetFreeParkingBal write SetFreeParkingBal;
end;
```



TBoard

+ tiles:array [0..39] of TTile
FFreeParkingBal:integer

+ Create
GetFreeParkingBal:integer
SetFreeParkingBal

+ FreeParkingBal:integer

Fields

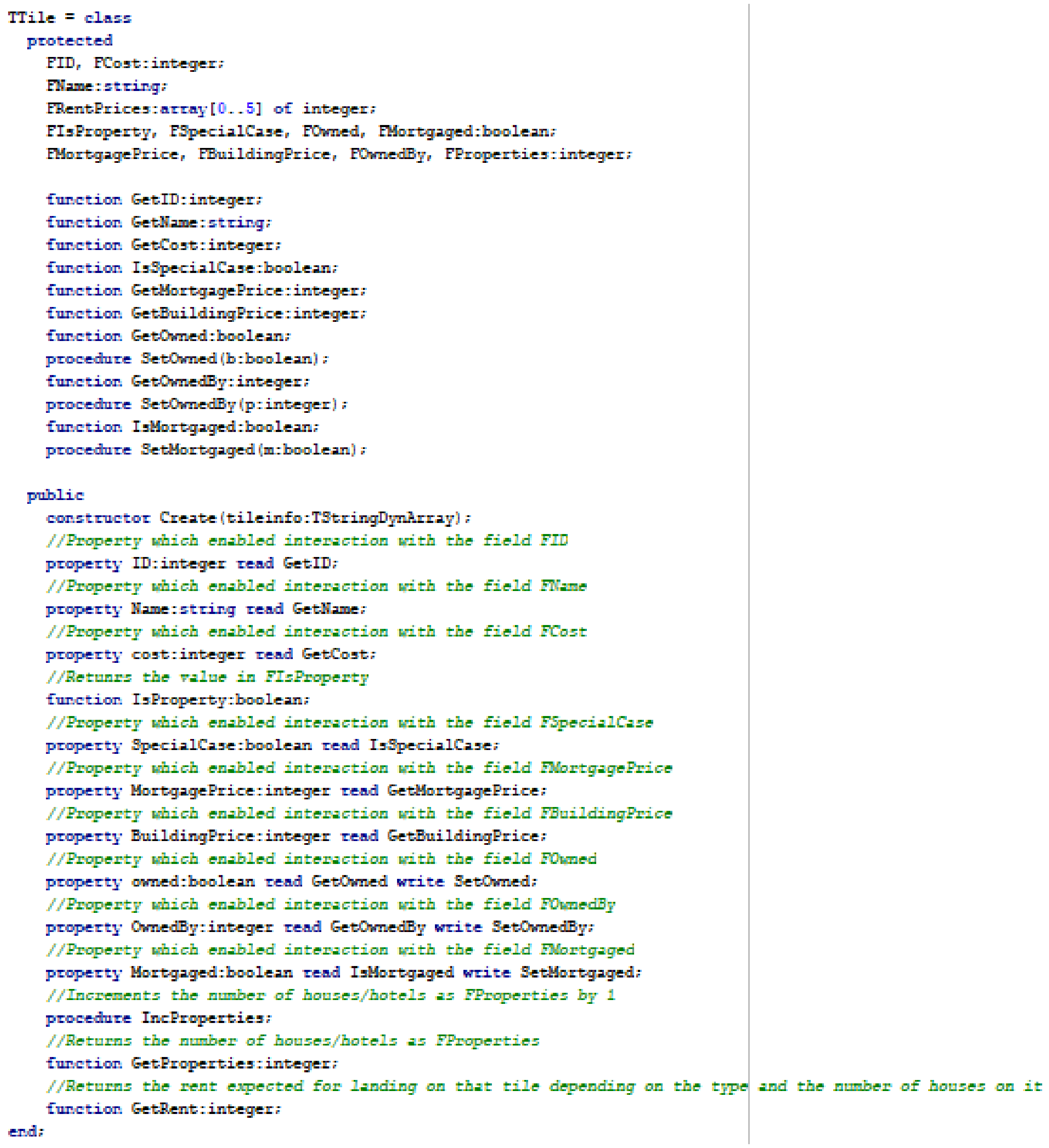
Name	Type	Description
Tiles	Array of 40 TTiles	Holds the tiles involved in the board so effectively the board state
FFreeParkingBal (Property)	Integer	Holds the amount of money stored in the free parking tile

Methods

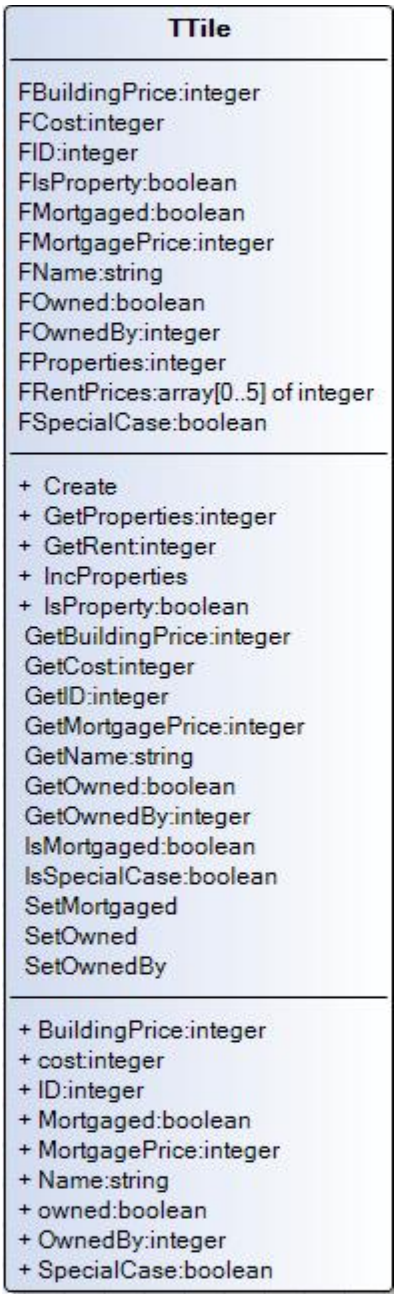
Name	Description
Create	Initialises the TTiles by creating them in the tiles array. It does this by reading from the text file associated with the program containing all the information used for the tiles, shown below. This also enables customisation of the game and the file is editable

```
0,False,Go
1,True,Old Kent Road,60,False,2,10,30,90,160,250,50
2,False,Community Chest
3,True,Whitechapel Road,60,False,4,20,60,180,320,450,50
4,False,Income Tax
5,True,Kings Cross Station,200,True,25,50,100,200
6,True,The Angel Islington,100,False,60,30,90,270,400,550,50
7,False,Chance
8,True,Euston Road,100,False,6,30,90,270,400,550,50
9,True,Pentonville Road,100,False,6,30,90,270,400,550,50
10,False,Just visiting
11,True,Pall Mall,140,False,10,50,150,450,625,750,100
12,True,Electric Company,150,True
13,True,Whitehall,140,False,10,50,150,450,625,750,100
14,True,Northumberland Avenue,160,False,12,60,180,500,700,900,100
15,True,Marylebone Station,200,True,25,50,100,200
16,True,Bow Street,180,False,14,70,200,550,750,950,100
17,False,CommunityChest
18,True,Marlborough Street,180,False,14,70,200,500,750,950,100
19,True,Vine Street,200,False,16,80,200,600,800,1000,100
20,False,Free Parking
21,True,Strand,220,False,18,90,250,700,875,1050,150
22,False,Chance
23,True,Fleet Street,220,False,18,90,250,700,875,1050,150
24,True,Trafalgar Square,240,False,20,100,300,750,925,1100,150
25,True,Fenchurch Street Station,200,True,25,50,100,200
26,True,Leicester Square,260,False,22,110,330,800,975,1150,150
27,True,Coventry Street,260,False,22,110,330,800,975,1150,150
28,True,Water Works,150,True
29,True,Picadilly,280,False,22,120,360,850,1025,1200,150
30,False,Go to Jail
31,True,Regent Street,300,False,26,130,390,900,1100,1275,200
32,True,Oxford Street,300,False,26,130,390,900,1100,1275,200
33,False,Community Chest
34,True,Bond Street,320,False,28,150,450,1000,1200,1400,200
35,True,Liverpool Street Station,200,True,25,50,100,200,200
36,False,Chance
37,True,Park Lane,350,False,35,175,500,1100,1300,1500,200
38,False,Super Tax
39,True,Mayfair,400,False,50,200,600,1400,1700,2000,200
```

TTile



```
TTile = class
  protected
    FID, FCost:integer;
    FName:string;
    FRentPrices:array[0..5] of integer;
    FIsProperty, FSpecialCase, FOwned, FMortgaged:boolean;
    FMortgagePrice, FBuildingPrice, FOwnedBy, FProperties:integer;

    function GetID:integer;
    function GetName:string;
    function GetCost:integer;
    function IsSpecialCase:boolean;
    function GetMortgagePrice:integer;
    function GetBuildingPrice:integer;
    function GetOwned:boolean;
    procedure SetOwned(b:boolean);
    function GetOwnedBy:integer;
    procedure SetOwnedBy(p:integer);
    function IsMortgaged:boolean;
    procedure SetMortgaged(m:boolean);

  public
    constructor Create(tileinfo:TStringDynArray);
    //Property which enabled interaction with the field FID
    property ID:integer read GetID;
    //Property which enabled interaction with the field FName
    property Name:string read GetName;
    //Property which enabled interaction with the field FCost
    property cost:integer read GetCost;
    //Returns the value in FIsProperty
    function IsProperty:boolean;
    //Property which enabled interaction with the field FSpecialCase
    property SpecialCase:boolean read IsSpecialCase;
    //Property which enabled interaction with the field FMortgagePrice
    property MortgagePrice:integer read GetMortgagePrice;
    //Property which enabled interaction with the field FBuildingPrice
    property BuildingPrice:integer read GetBuildingPrice;
    //Property which enabled interaction with the field FOwned
    property owned:boolean read GetOwned write SetOwned;
    //Property which enabled interaction with the field FOwnedBy
    property OwnedBy:integer read GetOwnedBy write SetOwnedBy;
    //Property which enabled interaction with the field FMortgaged
    property Mortgaged:boolean read IsMortgaged write SetMortgaged;
    //Increments the number of houses/hotels as FProperties by 1
    procedure IncProperties;
    //Returns the number of houses/hotels as FProperties
    function GetProperties:integer;
    //Returns the rent expected for landing on that tile depending on the type and the number of houses on it
    function GetRent:integer;
end;
```



TTile

FBuildingPrice:integer
FCost:integer
FID:integer
FIsProperty:boolean
FMortgaged:boolean
FMortgagePrice:integer
FName:string
FOwned:boolean
FOwnedBy:integer
FProperties:integer
FRentPrices:array[0..5] of integer
FSpecialCase:boolean

+ Create
+ GetProperties:integer
+ GetRent:integer
+ IncProperties
+ IsProperty:boolean
GetBuildingPrice:integer
GetCost:integer
GetID:integer
GetMortgagePrice:integer
GetName:string
GetOwned:boolean
GetOwnedBy:integer
IsMortgaged:boolean
IsSpecialCase:boolean
SetMortgaged
SetOwned
SetOwnedBy

+ BuildingPrice:integer
+ cost:integer
+ ID:integer
+ Mortgaged:boolean
+ MortgagePrice:integer
+ Name:string
+ owned:boolean
+ OwnedBy:integer
+ SpecialCase:boolean

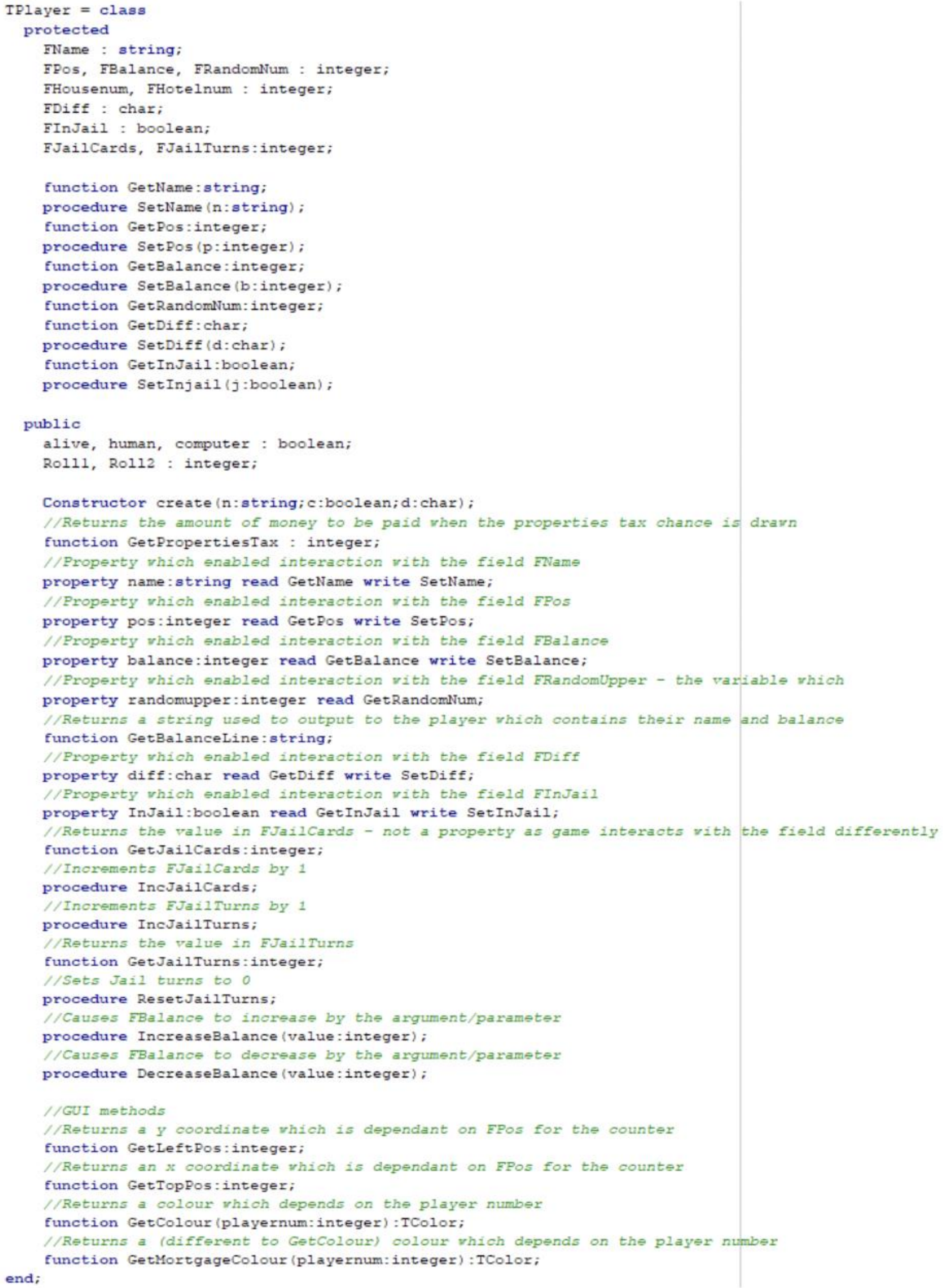
Fields

Name	Type	Description
FBuildingPrice (Property)	Integer	Holds the cost of building houses/hotels into that property
FCost (Property)	Integer	Holds the cost of purchasing that title deed
FID (Property)	Integer	Holds the ID which acts as an address for that property in the board's tiles
FIsProperty	Boolean	Boolean which is true for properties which can be purchased
FMortgaged (Property)	Boolean	Boolean which is true for properties which are mortgaged so no rent is paid
FMortgagePrice (Property)	Integer	Holds the amount of money a user gets for mortgaging this property and the cost to unmortgage it
FName (Property)	String	Holds the name of the tile
FOwned (Property)	Boolean	Boolean which is true when the title deed for that property is owned by a player
FOwnedBy (Property)	Integer	Holds the player number, in the current game, of who owns the property
FProperties	Integer	Holds the number of houses/hotels built on this tile
FRentPrices	Array of 5 integers	Holds the costs associated with relevant number properties which is paid upon a player landing on this tile
FSpecialCase (Property)	Boolean	Boolean which is true for trains stations or utilities as the rent action is unique for them

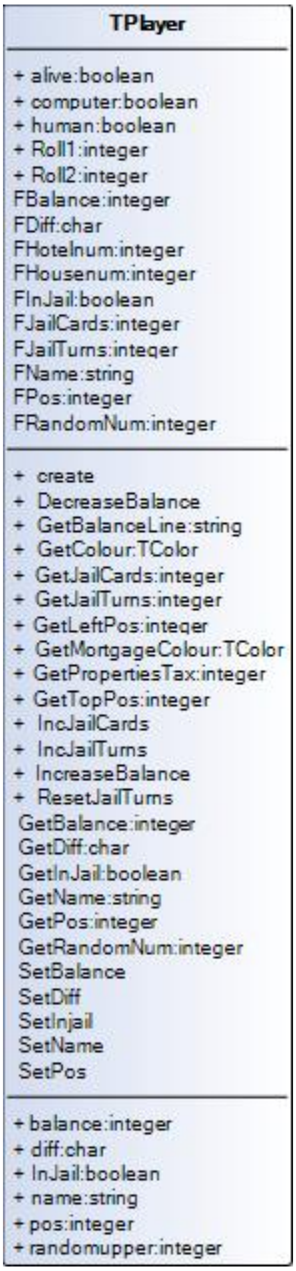
Methods

Name	Description
Create	Creates a tile with the input string containing the information needed to construct it e.g. the name, costs and other vital fields. This data is inputted into the right attribute as input string follows a strict format
F IsProperty	Returns a Boolean matching that in the field FProperty
F GetProperties	Returns an integer matching that in the field FProperties
P IncProperties	Increments the number in FProperties
F GetRent	Returns the amount of rent expected to be paid for landing on that property – in being FRentPrices[FProperties].

TPlayer



```
TPlayer = class
  protected
    FName : string;
    FPos, FBalance, FRandomNum : integer;
    FHousenum, FHotelnum : integer;
    FDiff : char;
    FInJail : boolean;
    FJailCards, FJailTurns:integer;

    function GetName:string;
    procedure SetName(n:string);
    function GetPos:integer;
    procedure SetPos(p:integer);
    function GetBalance:integer;
    procedure SetBalance(b:integer);
    function GetRandomNum:integer;
    function GetDiff:char;
    procedure SetDiff(d:char);
    function GetInJail:boolean;
    procedure SetInjail(j:boolean);

  public
    alive, human, computer : boolean;
    Roll1, Roll2 : integer;

    Constructor create(n:string;c:boolean;d:char);
    //Returns the amount of money to be paid when the properties tax chance is drawn
    function GetPropertiesTax : integer;
    //Property which enabled interaction with the field FName
    property name:string read GetName write SetName;
    //Property which enabled interaction with the field FPos
    property pos:integer read GetPos write SetPos;
    //Property which enabled interaction with the field FBalance
    property balance:integer read GetBalance write SetBalance;
    //Property which enabled interaction with the field FRandomUpper - the variable which
    property randomupper:integer read GetRandomNum;
    //Returns a string used to output to the player which contains their name and balance
    function GetBalanceLine:string;
    //Property which enabled interaction with the field FDiff
    property diff:char read GetDiff write SetDiff;
    //Property which enabled interaction with the field FInJail
    property InJail:boolean read GetInJail write SetInJail;
    //Returns the value in FJailCards - not a property as game interacts with the field differently
    function GetJailCards:integer;
    //Increments FJailCards by 1
    procedure IncJailCards;
    //Increments FJailTurns by 1
    procedure IncJailTurns;
    //Returns the value in FJailTurns
    function GetJailTurns:integer;
    //Sets Jail turns to 0
    procedure ResetJailTurns;
    //Causes FBalance to increase by the argument/parameter
    procedure IncreaseBalance(value:integer);
    //Causes FBalance to decrease by the argument/parameter
    procedure DecreaseBalance(value:integer);

    //GUI methods
    //Returns a y coordinate which is dependant on FPos for the counter
    function GetLeftPos:integer;
    //Returns an x coordinate which is dependant on FPos for the counter
    function GetTopPos:integer;
    //Returns a colour which depends on the player number
    function GetColour(playernum:integer):TColor;
    //Returns a (different to GetColour) colour which depends on the player number
    function GetMortgageColour(playernum:integer):TColor;
end;
```



TPlayer

+ alive:boolean
+ computer:boolean
+ human:boolean
+ Roll1:integer
+ Roll2:integer
FBalance:integer
FDiff:char
FHotelnum:integer
FHousenum:integer
FInJail:boolean
FJailCards:integer
FJailTurns:integer
FName:string
FPos:integer
FRandomNum:integer

+ create
+ DecreaseBalance
+ GetBalanceLine:string
+ GetColour:TColor
+ GetJailCards:integer
+ GetJailTurns:integer
+ GetLeftPos:integer
+ GetMortgageColour:TColor
+ GetPropertiesTax:integer
+ GetTopPos:integer
+ IncJailCards
+ IncJailTurns
+ IncreaseBalance
+ ResetJailTurns
GetBalance:integer
GetDiff:char
GetInJail:boolean
GetName:string
GetPos:integer
GetRandomNum:integer
SetBalance
SetDiff
SetInjail
SetName
SetPos

+ balance:integer
+ diff:char
+ InJail:boolean
+ name:string
+ pos:integer
+ randomupper:integer

Fields

Name	Type	Description
Alive	Boolean	A Boolean which holds true for players in the game who are currently alive and partaking
Computer	Boolean	A Boolean which holds true for players controlled by the computer
Human	Boolean	A Boolean which holds true for players controlled by a human
Roll1	Integer	Holds the value rolled by the first dice
Roll2	Integer	Holds the value rolled by the second dice
FBalance (Property)	Integer	Holds the balance of that player for the game. When negative, that player will be forced out the game
FDiff (Property)	Character	Holds the character of difficulty which is either ‘e’/’m’/’h’ and correspond to easy/medium/hard respectively
FHotelNum	Integer	Holds the number of hotels a player has, as is required for a chance card
FHouseNum	Integer	Holds the number of houses a player has, as is required for a chance card
FInJail (Property)	Boolean	A Boolean which holds true for when a player is in jail so should not receive money from rent and their turn is performed differently to normal e.g. only move if the roll is a double
FJailCards	Integer	Holds the number of get out of jail free cards which a player owns for leaving jail in one turn and not paying
FJailTurns	Integer	Holds the number of turns a player has been in jail for that session of being in jail – as resets after leaving jail. It is

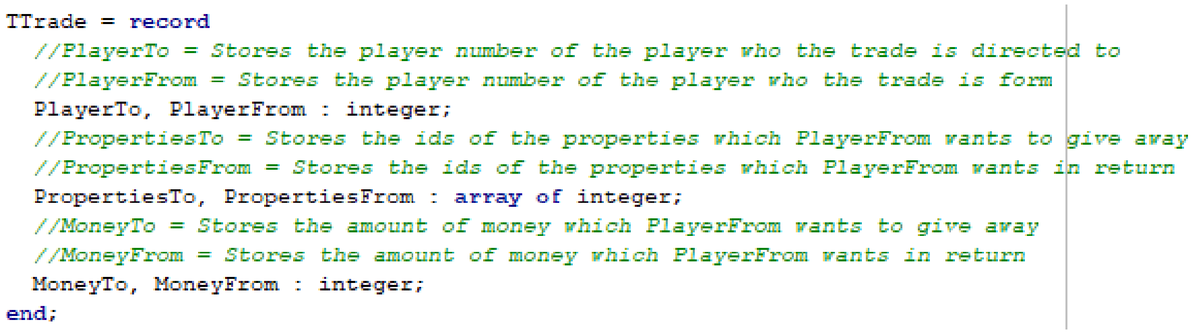
		used to enforce the 3 turns max in jail rule so players must pay to get out
FName (Property)	String	Holds the player name used to indicate in the memo box during the game
FPos (Property)	Integer	Holds the position on the board that the player's counter is currently in, and matches the ID of the tile is located on
FRandomNum (Property)	Integer	Holds the number mapped from the difficulty which determines what actions are completed by that player in that turn – a random number from 0 to FRandomNum is picked and if e.g. 0 means that it will buy the property that it lands on this turn if possible

Methods

Name	Description
Create	Creates the player by initialising the field variables required
P DecreaseBalance	Decreases the player's balance by the integer value given as a parameter
F GetBalanceLine	Returns a string containing the player's name and their balance as is frequently used and written to the memo
F GetColour	Returns the colour determined by the player's number
F GetJailCards	Returns the number of get out of jail free cards in FJailCards
F GetJailTurns	Returns the number of turns that player has spent in jail this session in FJailTurns
F GetLeftPos	Returns the left position on the form determined by the position of the player to move the counter to that place on the GUI
F GetMortgageColour	Returns the colour determined by the player's number which is slightly lighter than GetColour as represents a mortgaged property
F GetPropertiesTax	Returns the amount of money which the player should pay depending on the number of houses and hotels they have
F GetTopPos	Returns the top position on the form determined by the position of the player to move the counter to that place on the GUI
P IncJailCards	Increments the value in FJailCards
P IncJailTurns	Increments the value in FJailTurns

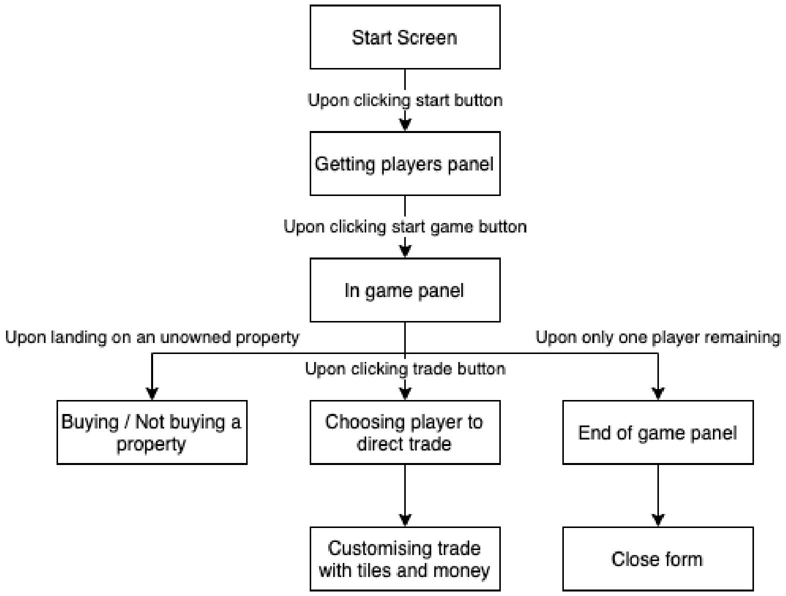
P IncreaseBalance	Increases the player’s balance by the integer value given as a parameter
P ResetJailTurns	Resets the value in FJailTurns back to zero

TTrade



```
TTrade = record
  //PlayerTo = Stores the player number of the player who the trade is directed to
  //PlayerFrom = Stores the player number of the player who the trade is form
  PlayerTo, PlayerFrom : integer;
  //PropertiesTo = Stores the ids of the properties which PlayerFrom wants to give away
  //PropertiesFrom = Stores the ids of the properties which PlayerFrom wants in return
  PropertiesTo, PropertiesFrom : array of integer;
  //MoneyTo = Stores the amount of money which PlayerFrom wants to give away
  //MoneyFrom = Stores the amount of money which PlayerFrom wants in return
  MoneyTo, MoneyFrom : integer;
end;
```

GUI overview


Start Screen
Upon clicking start button
Getting players panel
Upon clicking start game button
In game panel
Upon landing on an unowned property
Upon only one player remaining
Upon clicking trade button
Buying / Not buying a property
Choosing player to direct trade
End of game panel
Customising trade with tiles and money
Close form

Initially the game starts with a main screen and only one button clickable. Clicking this takes the form to the next panel which involves the user entering the information for the players and defining how many will be playing. There is a button labelled ‘Start game’ and when clicked takes the program to the next stage. This involves hiding all other elements and showing the panel of elements for being in the game causing the board to show up along with other necessary elements, for example the memo. There are other buttons for entering a trade which further changes the GUI and when a property is landed on, the two buy/don’t buy buttons appear. At the end of the game a button labelled ‘Close’ appears and when clicked frees the memory from the objects and exits out the form.

Procedure listings

Subroutine	Purpose	Local variables
P MoveCounter	Moves the current player's counter to the correct position on the board	
P PrepareTileShapeArray	Initialises the PropertyShapes array by getting the TShape components from the form	i : integer for iteration
P setGUIstart	Hides and shows the relevant form components which are required for the start	
P setGUIplayerChoices	Hides and shows the relevant form components which are required for the player choices stage	
P setGUIgame	Hides and shows the relevant form components which are required for the in game stage	
P Hoverover(t)	Shows the relavant labels to the user containing the name, cost and number of houses of the tile whose id is the argument t	
P StartPlayerGame	Creates the players from the information inputted from the user at the get players stage	Tempchar : char for the player's difficulty Tempname : string for the player's name
P PassedGo	Increases the balance of the current player by the value stored in the constant GoMoney	
P NormalTurn(r1,r2,b,g)	The main procedure which occurs for a human player's turn. R1 and R2 are passed into the procedure containing the values which the player rolled; B is a Boolean which determines if it needs to output the roll (if the player was in jail then doesn't need to) ; G is a Boolean which determines if the procedure should call PassedGo	Tempcost, Tempint : integer for working out the rents for utilities and train properties NewPos: integer for holding the tile id of the new position of the current player
P RollDiceProcedure	Called when the RollDice button is clicked to roll the dice and move the current player's position by that roll around the board and calls the correct next procedure depending on the	Randint1, Randint2, Randsum : integer for holding the rolled dice values and the sum

	player type and if they are in jail/have get out of jail free cards	Passgo : Boolean for determining if it needs to call PassedGo
P EndTurnProcedure	Called when the EndTurn button is clicked to kick out that player if they are in debt and call the procedure NextPlayer	i : integer for iteration
P NextPlayer	Increments the pointer ‘currentplayer’ to the next relevant player in the queue	
P CheckForDouble	Recalls the procedure NextTurn if the human player rolled a double	
P RollLine(x,y)	Adds a string which was frequently being written to the output memo containing the numbers x, y corresponding to the numbers rolled on the dice	
F GetAlivePlayers	Determines the number of remaining alive players in the game and returns it	i : integer for iteration x : integer for the number of alive players
F GetTrainOwned(P)	Determines the number of train tiles owned by the player P and returns it to work out how much money is owned to them	Count : integer for the number of train tiles owned by player P
P LandedOnChance	Gets a random chance card and executes the action on it to the current player	Tempint : integer to store the popped element from the stack and mapped to a chance card
P LandedOnCommunityChest	Gets a random community chest card and executes the action on it to the current player	Tempint : integer to store the popped element from the stack and mapped to a community chest card
P UseJailCard	Calls the relevant procedures required after a player uses their get out of jail free card	
P DontUseJailCard	Continues with a player’s turn if they don’t want to use their get out of jail free cards	
P ClearCurrentTrade	Clears the data in the CurrentTrade record and sets it to the default values	

P SetupTrade	Hides and shows the relevant form components which are required for the setting up a trade stage	
P AcceptTrade	Exchanges the tiles and money between the players involved in the trade and updates the GUI for the users	i : Integer for iteration
P RejectTrade	Calls ClearCurrentTrade to remove the current trade and alters the global variables which allow further trades to be made	i : integer for iteration
P TileShapeClick(P)	Performs the relevant actions - either adds the property with id P to the CurrentTrade record in the correct array; or mortgages the property with id P; or adds a house/hotel to the property with id P	
F PlayerOwnRange (player,clickedpos)	Determines if the player 'player' owns the colour range which contains the tile with id 'pos' and returns if that is true or false	
P ComputerTurn	Main computer turn, which carries out the sub-routines required in that computers turn and dynamically calls other parts depending on the player's difficulty	Randint1 : integer for determining what actions should be performed in that turn R1, R2, rollsum: integer for the rolls NewPos: integer for the new position of the current player Tempcost : integer for working out the rents for utilities and train properties
F FAllComputerPlayers	Determines if all the players in the game are controlled by a computer and returns the Boolean answer to know if it should update the GUI or not	
P ComputerBuyProperties	Buys houses/hotels for the current computer-controlled player on tiles that they can, if the player is on hard	i : integer for iteration

	difficulty then it buys as many properties as it can	
P EndGame	Called when the game has ended and finalises the GUI parts and enables the Close button to appear and be clicked	i : integer for iteration
F ArrayContains(arr, elem)	Determines if the value in ‘elem’ is present in the array ‘array’ and returns the Boolean answer	i : integer for iteration

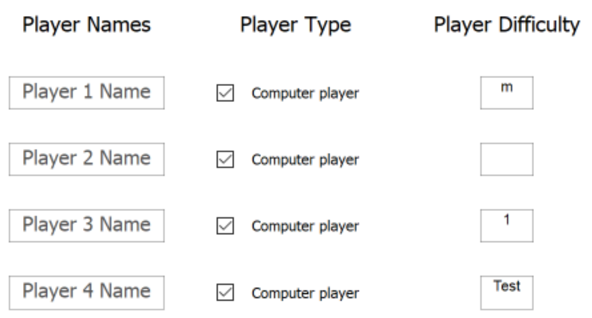
*The final code is located at the end of evaluation

Testing

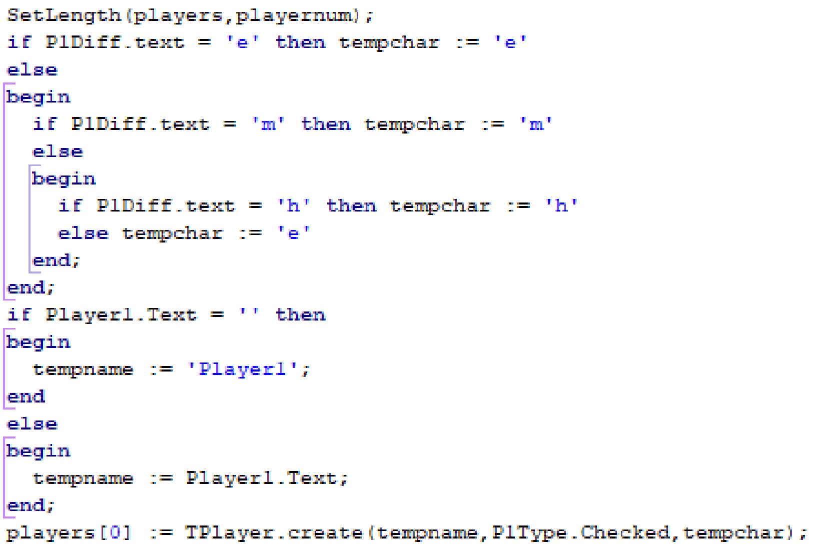
Testing strategy

The strategy I decided to use in the testing process for my program is a combination of multiple testing methods. Using black box testing I will be testing the interface options and interactions with the user to see if the program works how it is intended this includes avoiding any errors which result as a fact of unexpected inputs resolved by using input validation. I intend to combine the black box testing with end-user testing by using some end users to experiment with the program and test if they can break any part of it. The end-user testing will also indicate if the program needs help in explaining how to use the software if it isn't that easy to use as well as help me introduce some quality of life features which improve the game for example is the game easy enough to play with the information provided. Finally, I will complete some module testing on particular parts as I code the program and after I have completed the essential objectives then I will test the whole thing with all of its features and ensure that it works as intended.

Test 1

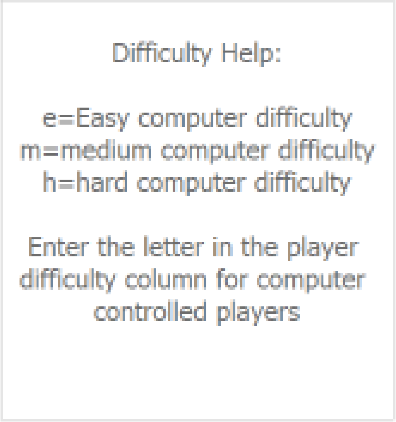
Test number	1
Purpose of test	To test if the input validation on the computer player difficulties prevents any errors occurring, while also giving that player a difficulty value
Test data	Player1: 'm' Player2: '' Player3: '1' Player4: 'Test' Player1 contains the typical data as the program expects an 'e', 'm' or 'h'. Player2 contains semi-typical data as the program assumes 'e' from an empty edit box. Player3 contains erroneous data as it expects at least a character and it is fed an integer. Player4 also contains erroneous data but this time it is in the form of a string.  Player Names | Player Type | Player Difficulty Player 1 Name | ☑ Computer player | m Player 2 Name | ☑ Computer player | Player 3 Name | ☑ Computer player | 1 Player 4 Name | ☑ Computer player | Test
Expected result	Overall there should be no errors and the program should continue. Player1 should accept the 'm' and cause the player to behave as a medium difficulty player. Player2 should assume the player to be an easy computer player from the empty box. Player3 and Player4 should

	ignore the information inputted from the user and default to easy difficulty. Also, if no information is inputted into the player name boxes, they should default to e.g. Player 1
Actual result	Upon clicking start game the program continues to the next stage and results in no alerting forms. As displayed by the memo box, the names default to the respective player e.g. Player 2's name is 'Player2'. Through playing the whole game through the players behave as they should and no errors are displayed so works as expected



```
SetLength(players,playernum);
if P1Diff.text = 'e' then tempchar := 'e'
else
begin
  if P1Diff.text = 'm' then tempchar := 'm'
  else
  begin
    if P1Diff.text = 'h' then tempchar := 'h'
    else tempchar := 'e'
  end;
end;
if Player1.Text = '' then
begin
  tempname := 'Player1';
end
else
begin
  tempname := Player1.Text;
end;
players[0] := TPlayer.create(tempname,P1Type.Checked,tempchar);
```

From some end-user testing, it was clear that the difficulty input values that were accepted were not clear or intuitive especially as the difficulty scaling option is a key factor for the use of my program. Therefore, I decided to implement an un-editable memo box which provides an explanation of what the program expects and how the user can change the computer player difficulties.

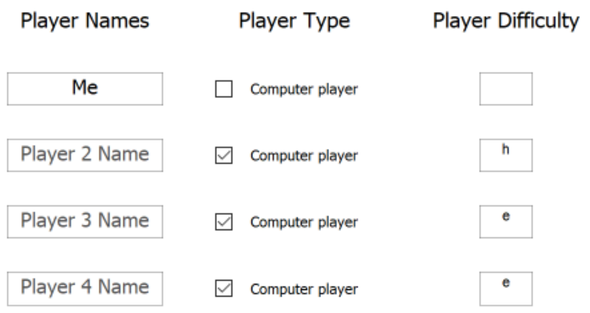


Difficulty Help:

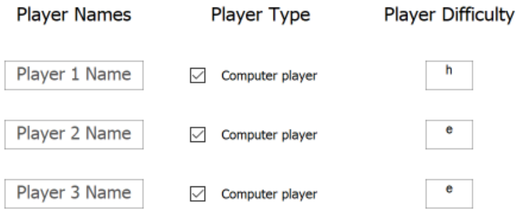
e=Easy computer difficulty
m=medium computer difficulty
h=hard computer difficulty

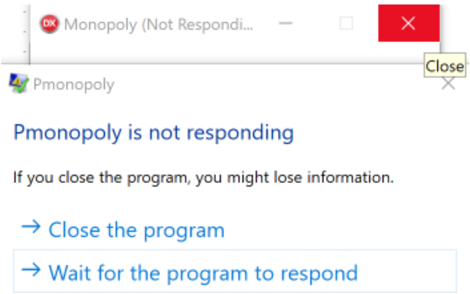
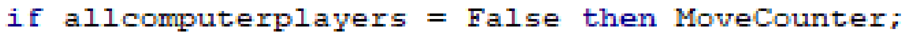
Enter the letter in the player difficulty column for computer controlled players

Test 2

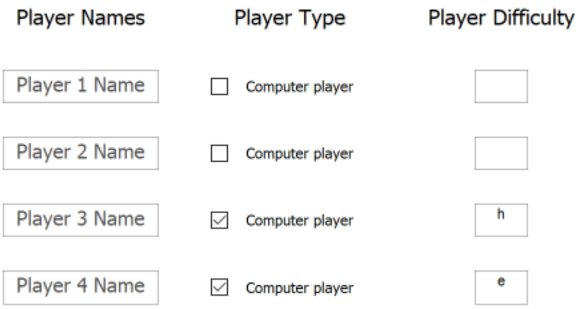
Test number	2
Purpose of test	Test if the new memo box provides sufficient information so a user knows how to modify the difficulty of computer-controlled players meaning that the program is user-friendly
Test data	Provide the end users with the program and ask them to play the game with computer players of difficulties they want
Expected result	The memo box displays for the user when they are at the getting player info stage, notifies them of what data to input to get what difficulties they want and then disappear when no longer needed
Actual result	 Player Names / Player Type / Player Difficulty Me / ☐ Computer player / Player 2 Name / ☑ Computer player / h Player 3 Name / ☑ Computer player / e Player 4 Name / ☑ Computer player / e Above is an example form page of when an end user was starting a game of Monopoly. It indicates that the memo box contained enough information for the users to know what to input demonstrating that the memo box is sufficient

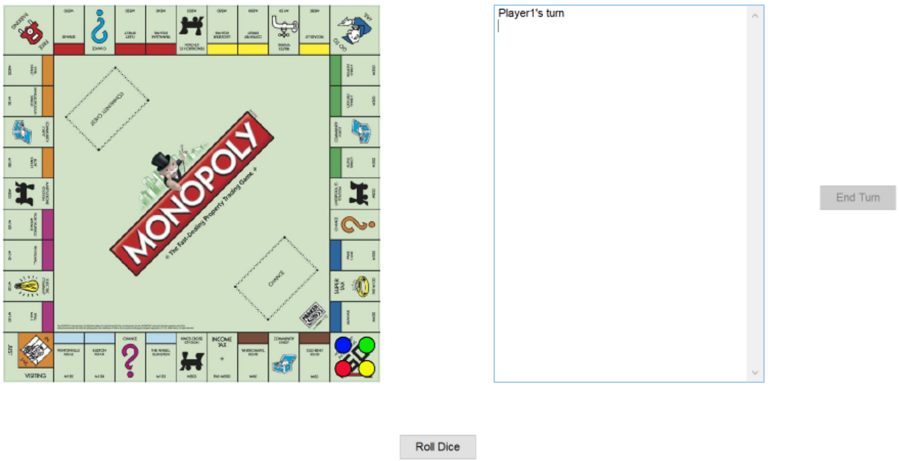
Test 3

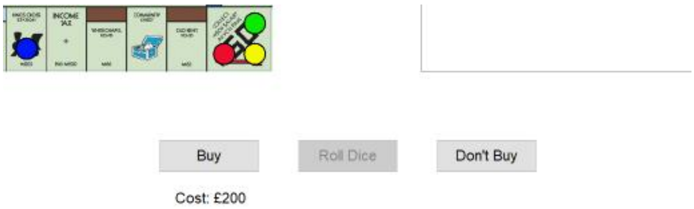
Test number	3
Purpose of test	To determine if the computer-controlled players act as they should and if the game can run with solely computer players, resulting in an end state and not throwing any errors
Test data	Three computer-controlled players, Player1 on hard difficulty with Player2 and Player3 on easy difficulty. This would be typical data  Player Names / Player Type / Player Difficulty Player 1 Name / ☑ Computer player / h Player 2 Name / ☑ Computer player / e Player 3 Name / ☑ Computer player / e
Expected result	The game runs smoothly with no errors and doesn't crash, also the players behave in accordance to their difficulty
Actual result	After programming the computer turn and testing it with the above test cases frequently if the game didn't end quickly, the program often resulted in not responding and being unable to close the form showed in the below screenshot. Therefore, the actual result did not match the expected result and some modifications were required

	 Monopoly (Not Respondi... Close Pmonopoly Pmonopoly is not responding If you close the program, you might lose information. → Close the program → Wait for the program to respond As I was unable to locate the source of the error, even with the in-built debugging help from RAD-studio with the breaks I began debugging it manually by commenting out most of the sub-routine and uncommenting out each section after running it until I determined which part was causing the problem. I came to the conclusion that during a computer turn, if the procedures which ran to help the GUI for example MoveCounter which moved the counter around the screen and BoardImg.Show which displayed the board were being used then this would result in the game crashing. I decided in the end, which also benefited the program, to only call these procedures and methods when the game involved at least one human player as not only would that mean that the human user still gets to see the game, but also it would mean that the purely computer players part of the game would run much faster as less processing would be required per call of the procedure. At the end of an all computer player game, I would call all the required sub-routines so the user still gets to see what happened in the game and the board state can be outputted. I also had to create a new function which determined if the game only contained computer players which would be called to see if the game needs to execute some of the GUI procedures e.g.  `if allcomputerplayers = False then MoveCounter;`

Test 4

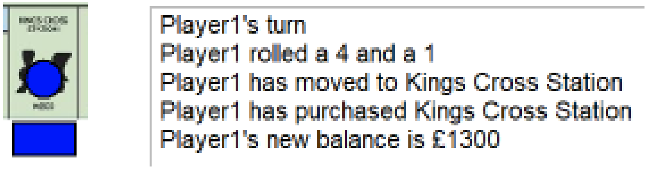
Test number	4
Purpose of test	Typical use of the game with all of its features
Test data	Have four players, two of which are controlled by a human and the other two which are computer controlled – one being hard and the other easy difficulty  Player Names | Player Type | Player Difficulty Player 1 Name | □ Computer player | Player 2 Name | □ Computer player | Player 3 Name | ☑ Computer player | h Player 4 Name | ☑ Computer player | e
Expected result	Part 1 is testing that upon clicking start game, the game initiates with the correct GUI components appearing and behaving as they should e.g. the counters in the correct place. Part 2 is testing that upon a player moving to an unowned property, the option to purchase it is outputted, the correct buttons for that action appear with the others disabled and the cost of that property is also displayed to the user. Part 3 is upon a player buying that property, the memo box is updated with that information, the board state is changed by that property being owned and the tile shape is shown and coloured the correct colour matching the player who bought it Part 4 combines two features as upon a player landing on community chest, they draw a card and the action is executed as well as if they roll a double then they take a turn again. Part 5 tests if the computer turns work by at the end of the two human player's turns, when the end turn button is clicked, the two computer players take their turns very quickly with what goes on in their turn displayed to the human users. After the two computer turns goes, the next human turn is player 1 so that is shown, and their turn begins. Furthermore, this test combines the chance card feature along with the computer turn feature, and upon a player landing on an owned property, the correct amount of money is exchanged and user notified via the memo box. Part 6 is a test that upon a player landing on go or passing it, they receive money from the bank matching the value in the true game.

	Part 7 is a test of when a player lands on free parking, they receive the money built upon the tile throughout the game from players paying the bank e.g. from income tax or chance cards causing people to pay. Part 8 is a test that players can trade with other players. Firstly, a human player would click on the start a trade button at the beginning of their go, then respective player buttons appear indicating which player the trade is directed to with only alive players in the game being shown. Then the player can click on properties that they own and properties that the player who they are trading own and the computer adds them to the display, so the users know, as well as the record behind the scenes. Also, the player can input money values into the trade boxes to incorporate money into the trade. Upon clicking the send button, the trade should be sent to the player it is directed to and at the start of the turn they are notified and must either accept it or reject it. In the meantime, the trade button should be greyed out as it should not be clickable as would interfere with the current trade record. Part 9 tests if a player can mortgage tiles which they own to increase their balance often so they can pay rent. Part 10 tests if a player can purchase houses/hotels on properties which they are able to so long as they can afford it.
Actual result	Part 1:  Player1's turn As seen from above, the correct number of counters appear along with only the roll dice button being enabled and the name of the first player and their turn is displayed in the memo output. Part 2:



As seen above, player 1 had moved to an unowned property and the correct buttons appear

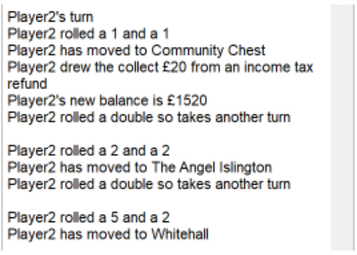
Part 3:



```
Player1's turn
Player1 rolled a 4 and a 1
Player1 has moved to Kings Cross Station
Player1 has purchased Kings Cross Station
Player1's new balance is £1300
```

Here it can be seen that player 1 clicked buy so the property shape was shown and updated to be the same colour as player 1 and the information is displayed to the user through the output box

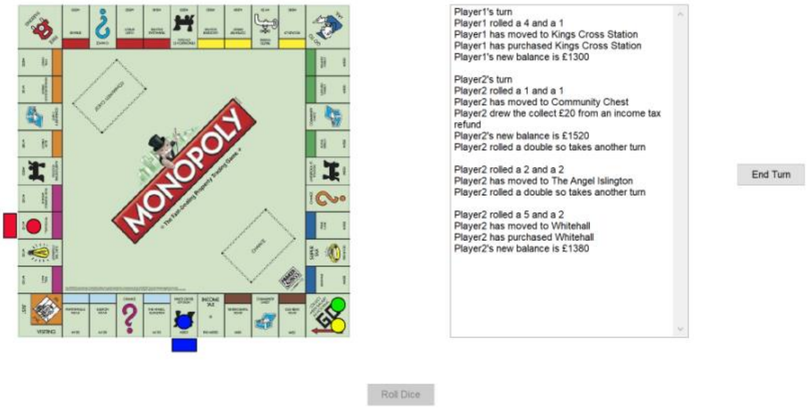
Part 4:



```
Player2's turn
Player2 rolled a 1 and a 1
Player2 has moved to Community Chest
Player2 drew the collect £20 from an income tax
refund
Player2's new balance is £1520
Player2 rolled a double so takes another turn

Player2 rolled a 2 and a 2
Player2 has moved to The Angel Islington
Player2 rolled a double so takes another turn

Player2 rolled a 5 and a 2
Player2 has moved to Whitehall
```

Here it can be seen that Player 2 rolled form go onto the community chest tile, they drew a card which was displayed to the user and completed the action which was to collect £20. As they rolled a double (1 and a 1) they take their turn again where they land on the Angel Islington, I clicked Don't buy so the property remains unowned and no money is exchanged. They again rolled a double, so they take a third turn where they roll to Whitehall

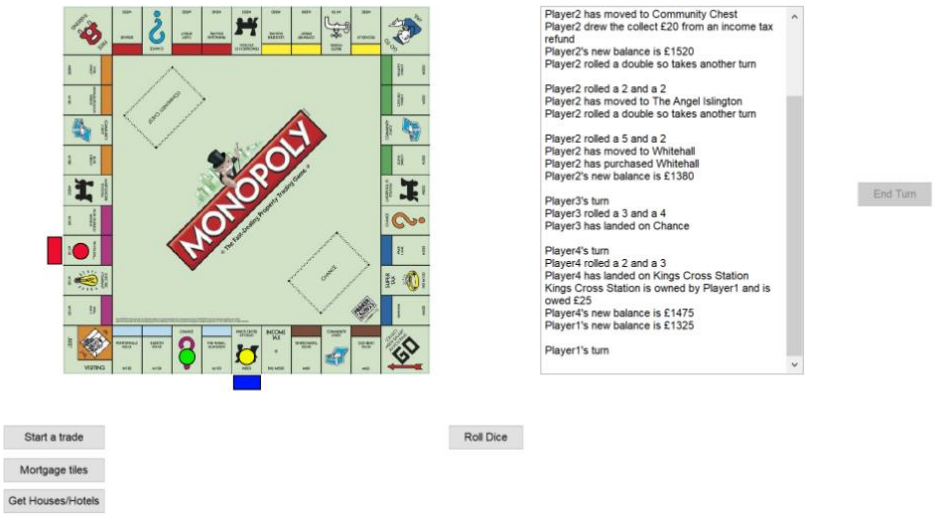
Part 5



```
Player1's turn
Player1 rolled a 4 and a 1
Player1 has moved to Kings Cross Station
Player1 has purchased Kings Cross Station
Player1's new balance is £1300

Player2's turn
Player2 rolled a 1 and a 1
Player2 has moved to Community Chest
Player2 drew the collect £20 from an income tax
refund
Player2's new balance is £1520
Player2 rolled a double so takes another turn

Player2 rolled a 2 and a 2
Player2 has moved to The Angel Islington
Player2 rolled a double so takes another turn

Player2 rolled a 5 and a 2
Player2 has moved to Whitehall
Player2 has purchased Whitehall
Player2's new balance is £1380
```

At the end of the Player 2's turn (above)

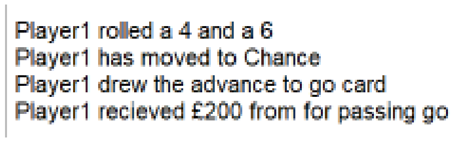
After clicking end turn on player 2's turn and the start of player 1's turn (below)



```
Player2 has moved to Community Chest
Player2 drew the collect £20 from an income tax
refund
Player2's new balance is £1520
Player2 rolled a double so takes another turn

Player2 rolled a 2 and a 2
Player2 has moved to The Angel Islington
Player2 rolled a double so takes another turn

Player2 rolled a 5 and a 2
Player2 has moved to Whitehall
Player2 has purchased Whitehall
Player2's new balance is £1380

Player3's turn
Player3 rolled a 3 and a 4
Player3 has landed on Chance

Player4's turn
Player4 rolled a 2 and a 3
Player4 has landed on Kings Cross Station
Kings Cross Station is owned by Player1 and is
owed £25
Player4's new balance is £1475
Player1's new balance is £1325

Player1's turn
```

As you can see from the above two images, after clicking the end turn button, in quick succession the two computer players took their turn. In Player 3's turn they moved to chance where they drew the card which causes a player to end their turn instantly and not permitted to complete any other actions in their turn e.g. start a trade or mortgage properties. Player 4 then took their turn where they rolled to Kings Cross station, this test also demonstrates that upon landing on an owned property, the correct amount of rent is paid from the landing player to the player who owns that tile

Part 6:

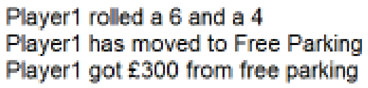


```
Player1 rolled a 4 and a 6
Player1 has moved to Chance
Player1 drew the advance to go card
Player1 recieved £200 from for passing go
```

Clearly in this picture it is shown that player 1 drew the chance to advance to go where they then collected money from the bank

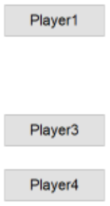
Part 7:

Clearly in this picture is it shown that player 1 moved to the free parking and they received £300. Then the free parking balance was set to 0 so it could be built up again for the players in the future.



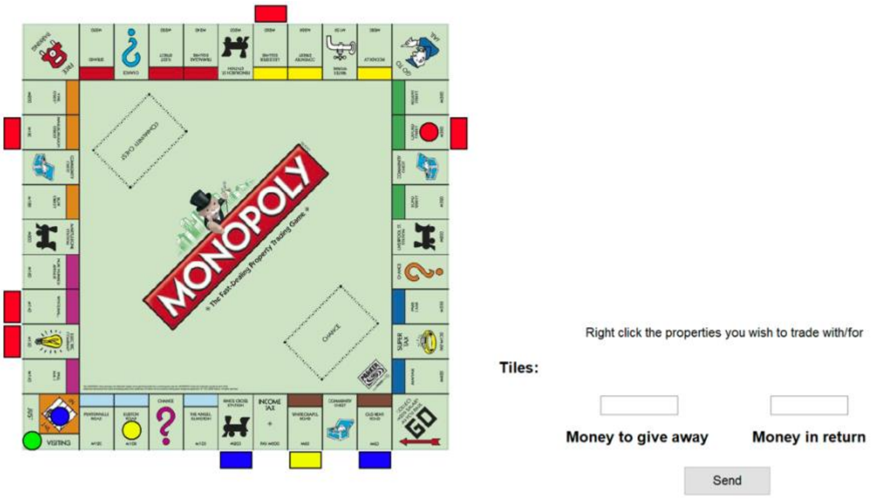
```
Player1 rolled a 6 and a 4
Player1 has moved to Free Parking
Player1 got £300 from free parking
```

Part 8:

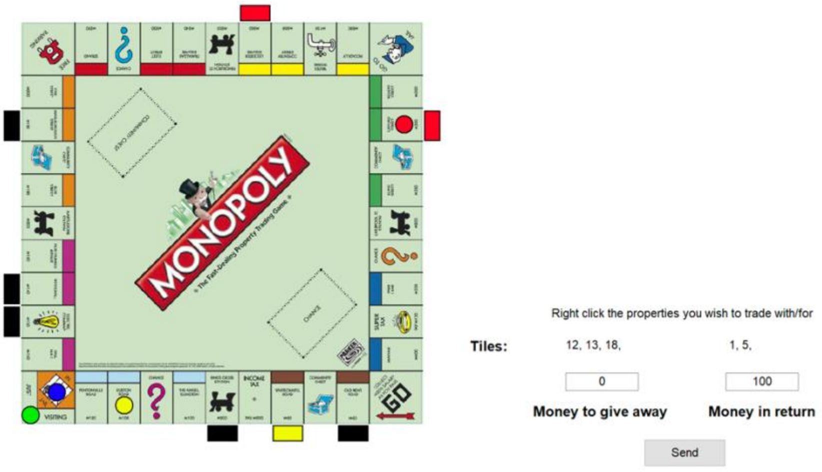


This picture shows that when Player 2 clicks the start a trade button, only players 1,3 and 4 can be clicked to direct a trade to as you cannot trade with yourself

Below is the board state initially before the trade and the trade when it starts off empty



Below is the board state during a trade, the player who created the trade has clicked on the blacked-out properties and the computer has worked out where the properties are intended to go and be used as, also the player has asked for £100 by putting it into the ‘Money in return’ edit

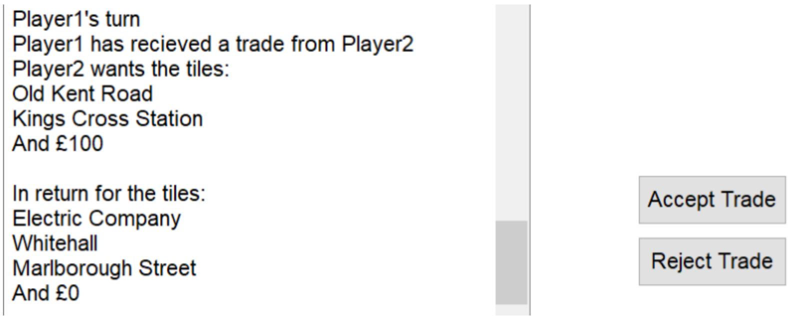


Right click the properties you wish to trade with/for

Tiles: 12, 13, 18, 1, 5,

0 Money to give away

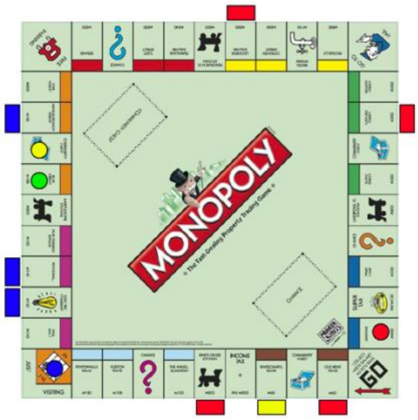
100 Money in return

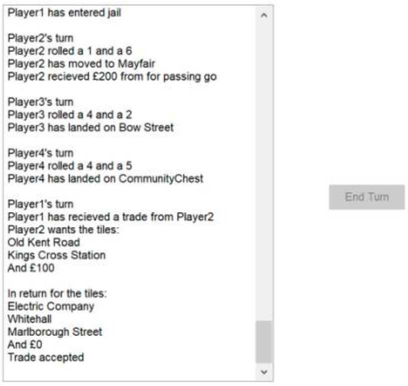


Player1's turn
Player1 has recieved a trade from Player2
Player2 wants the tiles:
Old Kent Road
Kings Cross Station
And £100

In return for the tiles:
Electric Company
Whitehall
Marlborough Street
And £0

Upon a player receiving a trade, this is displayed to that user to tell them what the trade involves, they then must click the accept trade or reject trade to decide what to do.

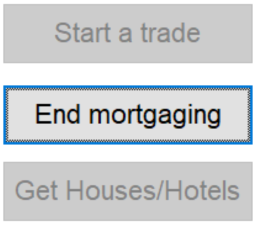




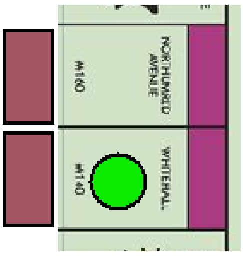
```
Player1 has entered jail

Player2's turn
Player2 rolled a 1 and a 6
Player2 has moved to Mayfair
Player2 recieved £200 from for passing go

Player3's turn
Player3 rolled a 4 and a 2
Player3 has landed on Bow Street

Player4's turn
Player4 rolled a 4 and a 5
Player4 has landed on CommunityChest

Player1's turn
Player1 has recieved a trade from Player2
Player2 wants the tiles:
Old Kent Road
Kings Cross Station
And £100

In return for the tiles:
Electric Company
Whitehall
Marlborough Street
And £0
Trade accepted
```

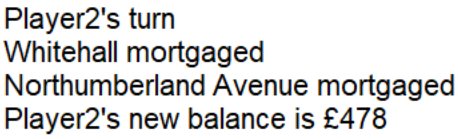
Above it can be seen that after player 1 clicked accept trade, the trade was followed through and the exchange of relevant properties and amounts of money occurred resulting in a different board state e.g. red (player 2) now owning the first property – Old Kent Road.

Part 9:



Upon a player clicking on the mortgage tiles button, the other two are disabled and the caption is updated to say end mortgaging

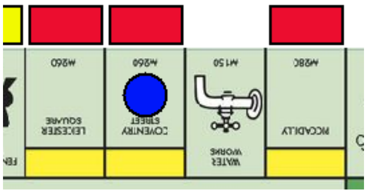


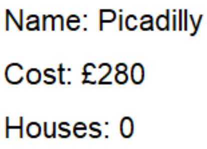


```
Player2's turn
Whitehall mortgaged
Northumberland Avenue mortgaged
Player2's new balance is £478
```

The two above images demonstrate that Player 2 mortgaged two properties which it previously owned. They did this by right clicking on the properties or the tile shapes, if a property is either not owned by the player doing the mortgaging or already mortgaged then the action of clicking on the property just does nothing instead. The colour of the tile shape is updated such that it is slightly greyed out to indicate to the users that the properties are mortgaged, output is displayed in the memo box. Finally, when the mortgaging is complete and the end mortgaging button is clicked, that player's new balance is displayed in the box, so they know their new balance following the mortgaging.

Part 10:



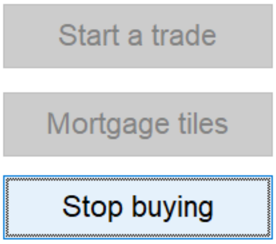


Name: Picadilly

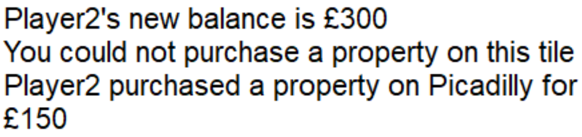
Cost: £280

Houses: 0

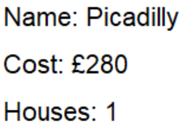
Above the pictures show the state of the yellow properties before player 2 buys any houses on them



Similarly, to when mortgaging tiles, upon clicking ‘Get houses/hotels’ the button updates to now say stop buying and the other buttons are disabled so only one event is performed at once by the user



Player2's new balance is £300
You could not purchase a property on this tile
Player2 purchased a property on Picadilly for £150



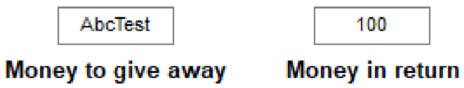
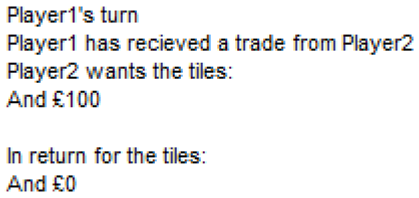
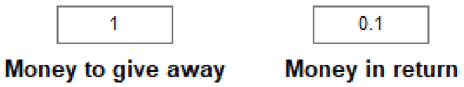
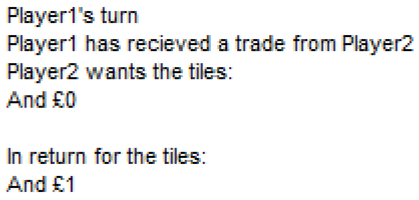
Name: Picadilly

Cost: £280

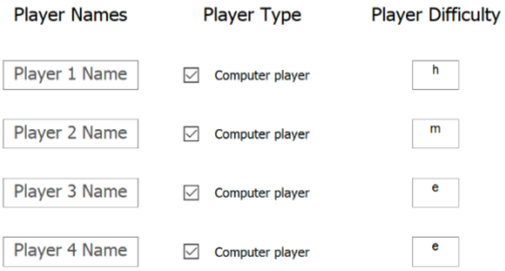
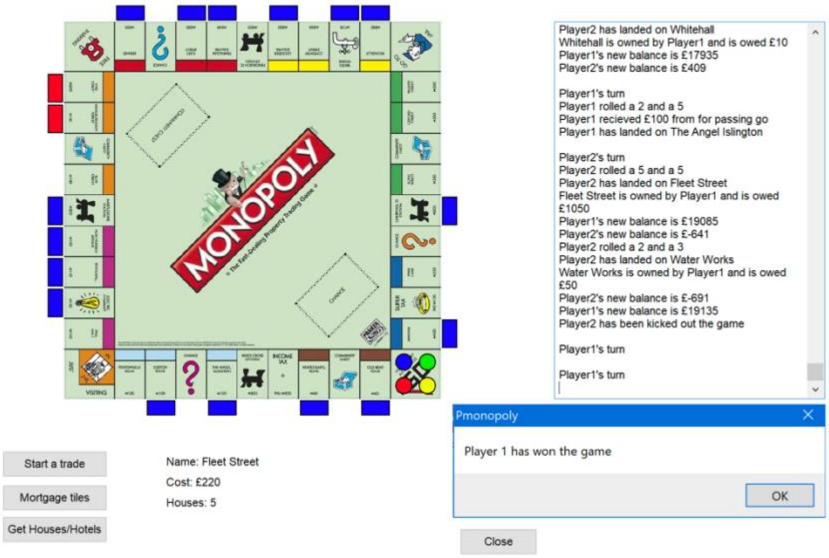
Houses: 1

The left screenshot shows the first line when player 2 tried clicking on a property which they did own, but just didn’t own the range so they were unable to buy a house on that property and this was outputted to the user. The second line explains how they clicked on Piccadilly and as they could afford it and were allowed to, they bought a house on that property. The right screenshot shows how now the labels are updated - indicating the board state has changed and Piccadilly now has an extra house therefore the rent on it has increased.

Test 5

Test number	5
Purpose of test	To test the use of regular expression for input validation when sending a trade
Test data	100 – Typical data One – Erroneous data Abc123 – Erroneous data 0 – Typical data 0.1 – Invalid extreme data
Expected result	With the typical data inputs (0,100) the program should accept these as they match the regular expression of being integers only. The data which are not of the right type (One, Abc123) so therefore are erroneous should be ignored and therefore cause the field in the record to default to 0. The Invalid extreme data (0.1) also should not be accepted as I do not intend to have the program deal with money in smaller increments than £1, therefore should again default to 0
Actual result	 AbcTest Money to give away 100 Money in return  Player1's turn Player1 has recieved a trade from Player2 Player2 wants the tiles: And £100 In return for the tiles: And £0  1 Money to give away 0.1 Money in return  Player1's turn Player1 has recieved a trade from Player2 Player2 wants the tiles: And £0 In return for the tiles: And £1 As you can see from above, when erroneous data was inputted in AbcTest, it didn't accept it and instead set it to 0. When typical data in 100 and 1 were entered the program accepted it. When extreme invalid data in 0.1 was inputted it did not accept it meaning it matches the expected result and works as intended.

Test 6

Test number	6
Purpose of test	To test whether a full game can be completed where all the players are computer players
Test data	All 4 players will be computer controlled, player 1 will be on hard difficulty, player 2 will be on medium difficulty and the others will be easy  Player Names — Player Type — Player Difficulty Player 1 Name — ☑ Computer player — h Player 2 Name — ☑ Computer player — m Player 3 Name — ☑ Computer player — e Player 4 Name — ☑ Computer player — e
Expected result	Player1 hopefully should win, however if they do not it doesn't mean that the algorithm is wrong, just they got unlucky. However, I repeat the test case with the same data and consistently Player1 doesn't win then either there is an error with the ComputerTurn algorithm, or I need to improve my hard difficulty setting such that it is actually hard to beat.
Actual result	The below screenshot it shows that player 1 won the game as expected by the difficulties. A new alert form was displayed with the name of the winner of the game. The labels show that Fleet Street had 5 houses on it therefore the rent on it was very high and so resulted in player 2 going into debt initially and then they landed on another tile owned by player 1 so at the end of player 1's turn, player 2 was kicked out resulting in only one player left, the winner. Overall this shows that the actual result matched the expected result and the computer players seem to behave as expected and the program didn't stop due to any errors or crashing.  Player2 has landed on Whitehall Whitehall is owned by Player1 and is owed £10 Player1's new balance is £17935 Player2's new balance is £409 Player1's turn Player1 rolled a 2 and a 5 Player1 recieved £100 from for passing go Player1 has landed on The Angel Islington Player2's turn Player2 rolled a 5 and a 5 Player2 has landed on Fleet Street Fleet Street is owned by Player1 and is owed £1050 Player1's new balance is £19085 Player2's new balance is £-641 Player2 rolled a 2 and a 3 Player2 has landed on Water Works Water Works is owned by Player1 and is owed £50 Player2's new balance is £-691 Player1's new balance is £19135 Player2 has been kicked out the game Player1's turn Player1's turn Pmonopoly Player 1 has won the game Name: Fleet Street Cost: £220 Houses: 5 Start a trade Mortgage tiles Get Houses/Hotels OK Close MONOPOLY

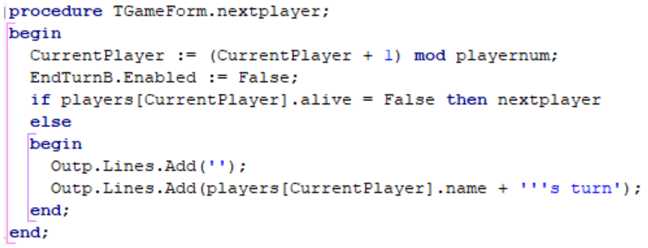
Test 7

Test number	7
Purpose of test	To test and demonstrate the working program in two situations. The first one with all 4 computer players and demonstrating the quick gameplay and opportunity to experiment with it. The second test is a general game
Test data	With 4 computer players I ensure one is played at hard difficulty and the rest at easy. With the normal game, I make the first player a human controlled player and the second player a medium computer player
Expected result	Both results of playing the game should just result in normal gameplay without resulting in any errors or crashes, then at the end the form should close, and memory is freed.
Actual result	www.bit.ly/3cH9L0J The above links takes you to a DropBox folder containing both video files which provide evidence for the above test. The one named 4Computerplayers.mov is a video of my setting up a game with four computer players playing against each other, also player 1 is put on hard difficulty. The game runs for a bit as the players are competing against each other, then at the end the last remaining player is player 1 (named the hard player) so winner is alerted. Then the game state is outputted by showing all the properties and who owns what. Then the close button is shown and clickable to free the memory and close the form. The second file named NormalGame.mov is an example video of a normal game containing a human player played by me and a medium difficulty computer player. Firstly, I take my go and upon clicking end turn, the computer takes its go very swiftly ready for me to take my next go. I continue to play the game by clicking the relevant buttons and decide to buy or don't buy some properties, then as it continues I use the key binds I implemented to increase the game speed as e.g. 'r' corresponds to roll and 'e' to end turn. I continue to buy properties which results in me running out of money and as a result the computer wins the game.

Evaluation

Comparison against core objectives

Objective number	Objective	Evaluation
1.1.1	The whole game matches that of the board game - being similar to an economic simulation with money flow and trading of assets	How effectively I achieved this objective was subject to interpretation and would vary between persons, so instead I got several end users to play the game and then asked them how they felt I mimicked the board game. All of the users said that my game was a clear replica of the true game and contained all the features and more of which the game exhibits, one even said that they preferred the speed of my game relative to the board game which is very key as matches what one of my end users wanted from the program
1.1.2.1	The board object will be composed of 40 tile objects demonstrating the strong composition relationship as matches the relationship of the actual board game	The TBoard object contains a field named 'tiles' which is a static array of TTile objects indicating the composition relationship is hard coded into the game. Throughout the running of the game, the manipulation of the tiles is through the global object board and field board.tiles enabling

1.1.3.1	The player objects will be stored in a circular queue to ensure they all get a go and are in order	The game's form object contains a field called 'players' which is a dynamic array containing the TPlayer objects. I also was required to use a pointer under the variable name 'currentplayer' which directed the list to a specific player. In order to increment currentplayer whilst still pointing to an alive player, I was able to use recursion as the procedure would repeat until it found a player whose field 'alive' was True
		
		`procedure TGameForm.nextplayer;`
		`begin`
		`  CurrentPlayer := (CurrentPlayer + 1) mod playernum;`
		`  EndTurnB.Enabled := False;`
		`  if players[CurrentPlayer].alive = False then nextplayer`
		`  else`
		`  begin`
		`    Outp.Lines.Add('');`
		`    Outp.Lines.Add(players[CurrentPlayer].name + '''s turn');`
		`  end;`
		`end;`

1.1.3.2	The human and computer player objects will be inherited from an abstract TPlayer class and will have many	*Not completed:* *I planned that I would create an abstract class called TPlayer and have two other classes as human and computer players which would inherit the abstract class and be instantiated in my game. However this objective was not completed as*

	similar attributes and methods	*during my prototyping I realised that to have a circular queue of players then it would be a dynamic array of different object types which you cannot do so either I would complete the above objective and not this or vice versa so I opted for the above one 1.1.3.1 which involved having the circular array*
1.1.4	Property ownership – the players will be able to buy properties that they land on and own them, the program will also allow the properties to be mortgaged and traded between players	In the TTile object I have a Boolean field called FOwned and an integer field called FOwnedBy. When a tile is bought, the owned Boolean is set to true and OwnedBy is set to the number of the player buying it. By default, these fields are set to false and -1. This means that each player can own several tiles and all the other functions which rely on an owner such as paying rent are directed to the player in OwnedBy
1.1.4.1	To organise trading between the players the program will create a record which contains the information involved in the trade	I created a new unit containing the definition for the TTrade record. When a trade is made the fields are filled with data and the variable PendingTrade is set to true. Then at the start of the turn of player who the trade is directed, they are given the option to accept or reject the trade. If accepting, the computer completes the trade and the GUI is updates. If rejecting the record is cleared and set to default for future trades, as well as PendingTrade being set to false
1.1.4.1.1	There should be input validation in the money involved in the trade using regular expression to ensure it is only an integer	I have used a regular expression to validate whether or not the inputted data, when in a trade, can be an integer; this ensures that the program doesn't crash or output an error making it a better experience for the end users as well as removing bugs they can exploit
1.1.4.1.2	Upon a player receiving a trade, they should have the option to accept or reject the trade. If accept the computer organises the exchange, if reject then the record is cleared and prepared for a future trade	If there is a pending trade when it is the start of a turn and trade's field called PlayerTo is the same as the CurrentPlayer then two buttons appear which force the user to either accept or reject the trade. Upon accepting the computer organises the exchange and the GUI is updated
1.1.4.2	To organise the mortgaging of properties, the players will be able to choose properties which they	Each TTile object contains a FMortgaged Boolean field, in order to make this field true the user must click on the mortgage properties button and then right click the tile shape causing the computer to determine that tile is mortgage-able, if it is then

	own and mortgage them, however they will no longer receive rent from them	the colour is updated and money given to the user. Upon a player landing on a mortgaged tile, no rent is paid matching that in the true board game
1.1.5	Money management: each player will have a fixed initial balance as defined by the Monopoly rules and will leave the game and lose if they run out	All the players start with an initial balance of £1500 and can spend it on what they want. Upon a player passing go they also receive money; this value is stored as a constant amount at the top of my program as when a smaller value allows for shorter computer-only player games. Upon a player going into negative money they are notified and must mortgage or trade their way up to a positive balance otherwise they will get kicked from the game when clicking the end turn button
1.1.6.1	The two jail tiles will be built in – go to jail will send the player to jail where the player must either roll a double or at the end of three turns pay to get out; the just visiting state will also be on this tile	Upon a player landing on just visiting (tile.id = 10) nothing occurs however if a player lands on the go to jail tile (tile.id = 30) then they are sent to jail. If a player is in jail then their turns consists of only rolling the dice, if it is a double then they leave jail and can continue on normally. If not, then they must stay in jail. If a player has any get out of jail free cards, then they are also given the option to use it when in jail enabling them to leave early. If they remain in jail for three turns, they must pay £50 to free parking and may leave jail
1.1.6.2	Free parking will collect all the tax money and other many required from the chance/community chest cards; anyone who lands on it will be rewarded with the totalled money and its value will be reset	There is a public field as part of the TBoard object which acts as a running total for the free parking balance. It builds up from tax tiles and other methods and resets when a player lands on it, giving the total to that player
1.1.7	Chances and community chests – all the cards and functions implemented with random chances of drawing them for every tile	All the board game's chance and community chest card labels and functions are implemented into the program. Upon a player landing on the tile, a random card is drawn, and action described on that card is executed
1.2.1	Original state for user to choose how many people are playing, their name and colour – these will loop round the	At the start, the game defaults to two players and the user can add more. They can customise each player's name and method of control. If the player is to be controlled by the computer, then they can also choose a difficulty. Upon clicking start game,

	board and will be located on the correct tile during their turn, indicating to the user where they are	the panel for the GetPlayers stage is hidden and replaced by the panel for InGame. The board is shown, and relevant number of player counters are shown. As the users or computer rolls the dice, the counters move around the board the rolled number of places
1.2.1.1	If that player is computer controlled and if so, what difficulty they are behaving as	There is a TEdit which the user can input a character which corresponds to the difficulty. This character is only read if that player's checkbox is checked to show they are a computer player, then there is some input validation to ensure that only accepted characters are inputted. The key which the computer uses is displayed to the user in the difficulty help memo box
1.2.2.1	The Monopoly board - the program will have the normal board as an image with the pieces located on it, also it will indicate what properties are owned and by which colour with an indicator (maybe with a colour)	The Monopoly board is located on the left with up to 4 counters which represent each player. The counter's colour corresponds to the player numbers. When a player purchases a tile on the board, an indicator appears whose colour matches the colour of the tile and so player. This enables the other users to know who owns that property. Also, when the mouse hovers over a tile, 3 labels appear with the name, cost and number of houses on that tile so it can be easily known. The counters also iterate around the board and their position on the board corresponds to their position in the game
1.2.2.2	A read-only output box for the user to understand the flow and play of the game- the box will update with any events occurring at that moment in the game and will be easy to read – e.g. 'Player 1 has moved to go' or 'Player 1 has paid £100 to Player 2'	There is a large memo box on the right of the board which contains all the information a user could want about all the previous goes and their current go. At the start of a player's turn it notifies them then everything they do in that turn has an output which is written to the box e.g. buying properties, drawing chance cards and moving to jail
1.2.2.3	All the relevant buttons to perform each action for example when clicking roll dice, the current player's turn will roll, and their piece will move around the board. Then if they land on an	On a normal turn there are five buttons which a player could click. At the start of the turn – before they have rolled the dice – the trade, mortgage properties and buy houses buttons are enabled and can be clicked performing their functions. The end turn button can only be clicked if the roll dice button has been pressed as a player needs to complete their turn. Upon rolling, the counter is

	unowned property buttons will appear to allow them to buy it or not	moved, and action performed on the tile landed on is executed e.g. if able to buy then two new buttons representing the buy property choice are shown, also the end turn button is still disabled until whole turn is completed
2.1.1	The difficulty variable of the computer player will determine what decision it makes	The difficulty variable corresponds to a number in a range stored as FRandomNum. This value helps determine what actions are taken in the computer's turn – for example if a random number from 0 to FRandomNum for that player is 1, then if a computer lands on an unowned property that turn, it will buy it. To implement difficulty scaling, by having the hard difficulty be harder than the medium difficulty, which is harder than the easy difficulty, I have FRandomNum vary per difficulty. Also, through research I have done I found that the oranges were generally the best, therefore if a hard player lands on an unowned orange, it will always buy it. Also, if a player is seven tiles behind the tile that it is deciding whether to buy it or not, it increases their chances of buying it as 7 is the most likely roll with two dice. Furthermore, most games involve a player buying houses and hotels on their property arrays. The hard and medium difficulties therefore have access to buying rent increasing properties on tiles they can afford. The procedure which enables them to buy these also is recursive as for the hard difficulty and if they have enough funds the procedure calls itself and repeats until it is in the base case where it no longer is able to buy any properties or its money drops below £500
2.1.2	Complete research into the best strategies and decisions to make enabling the hard difficulty to actually be hard to beat for the human player	This objective was mainly completed in the research phase of the project in the analysis. I ran a program which only has the function in Monopoly which move the player implemented such that as to simulate the movement of counters in the game. This provided me with results including the most common tiles and best properties to buy being the oranges. I then used this information to increase the difficulty of my hard computer player to always buy the oranges if given the opportunity. Evidence to suggest this really did help is that through many repeated tests from end users the hard difficulty did almost always beat other difficulties and medium often

		always beat the easy difficulty setting without making them too hard or too easy

Comparison against extension objectives

Objective number	Objective	Evaluation
3	Being able to save current game states and loading them from save files in the future	*Not completed:* *I was unable to complete this objective of my program due to an underestimate of the time needed to complete the above objectives. The implementation of this functionality would mean creating my own format for storing information to a form containing all the information which a board would need to resume the game as well as being able to save a game state such as all the tiles, their ownership status, the houses on top of all the players, their name, balance and many other things. As a result, the file may also have been reasonably large relative to the actual program so wouldn't have been that practical. If I had more time I would at least attempt to try and introduce this to the program as would improve the game as well as user experience and most likely will try even after the project is due*
4	Once achieved the initial game and some kind of computer-controlled user, have the game play out without human players and analyse the results to see what strategies are the most effective at winning.	*Partially completed:* *I was able to achieve a computer-controlled user which contained difficulty scaling, and behaved as a normal human player and had access to the same resources and actions as a human player meaning there was no straight-up unfair advantage given; however I was unable to produce some sort of output from the program which contained the game information and player statistics which some analysis could be applied to the numbers. If implemented, I would not only be able to increase and modify the computer player's actions and introduce some machine learning elements, but also generate strategies which could be executed by human players in an all human game meaning it would also benefit the users. I also plan to complete this functionality after the project deadline as despite it being quite tedious and difficult, it will definitely increase the programs playability as well as increase user experience as the hard difficulty players will be challenging*

Independent end-user feedback

Objective	Evaluation
Visual icons representing piece movement	The game has up to four icons which moved around the board on the positions matching that player's current position. They also move onto the right positions irrelevant of the computers resolution as the resolution does affect the pixel positions however with the small hidden edit box, it enables users to change it and play with the right positions. Also, the increase efficiency and usability of the program, when only computer-controlled players are involved, the counters are not shown and moved until the end
Indicators to show what tiles are owned by who	For every tile there is a tile shape as part of an array which is initialised at run-time and contains the form elements which are shown when a property is bought. The tile element is also filled with a colour matching the players colour and counter colour, so it is clear who owns that property
Match the normal board game in rules and game flow	From the start and even in the design process I have always tried to implement all the rules in the true game into my program. Almost all of the core objectives in my analysis are matching the aspects of the real game, therefore after completing these I have demonstrated the completeness of this objective
Piece icons could be similar to the true board game e.g. an iron or plane. Also, option to upload your own image to represent you	*Not completed:* *Due to the implementation of my current counters being TShapes in the Delphi form, I was unable to modify the representation of these unless they are a form of shape meaning an iron for example would be impossible. In order to add this to my program I would have to change the object to be e.g. a TImage which may produce errors and problems as a result of some TShape methods which a TImage doesn't have*

After speaking to Lucas Hockley, my end user, he gave me the following feedback:

The majority of his initial objectives were completed to at least the standard he expected and even the indicators of the properties he thought were better, as they are clickable and are incorporated into other parts of the project – for example the trading or mortgaging. He would still like to see the modification of the piece icons to be at least the Monopoly default ones as believes that they are quite unique to Monopoly and despite not being that useful, is a defining part of the game. Also he thought that the saving a game feature despite not being part of the true board game would be really useful as would mean you could come back to a game in the future and would probably mean he would rather play my version over the true board game version. Overall, he believes that every other factor of Monopoly has been effectively implemented and really enjoys playing the games especially watching and experimenting with all computer players in seeing who wins and how they do.

Final Delphi code

uTile.pas

```
unit uTile;

interface

uses
  SysUtils, Dialogs, Classes, types;

type

  TTile = class
    protected
      FID, FCost:integer;
      FName:string;
      FRentPrices:array[0..5] of integer;
      FIsProperty, FSpecialCase, FOwned, FMortgaged:boolean;
      FMortgagePrice, FBuildingPrice, FOwnedBy, FProperties:integer;

      function GetID:integer;
      function GetName:string;
      function GetCost:integer;
      function IsSpecialCase:boolean;
      function GetMortgagePrice:integer;
      function GetBuildingPrice:integer;
      function GetOwned:boolean;
      procedure SetOwned(b:boolean);
      function GetOwnedBy:integer;
      procedure SetOwnedBy(p:integer);
      function IsMortgaged:boolean;
      procedure SetMortgaged(m:boolean);

    public
      constructor Create(tileinfo:TStringDynArray);
      //Property which enabled interaction with the field FID
      property ID:integer read GetID;
      //Property which enabled interaction with the field FName
      property Name:string read GetName;
      //Property which enabled interaction with the field FCost
      property cost:integer read GetCost;
      //Retunrs the value in FIsProperty
      function IsProperty:boolean;
      //Property which enabled interaction with the field FSpecialCase
      property SpecialCase:boolean read IsSpecialCase;
      //Property which enabled interaction with the field FMortgagePrice
      property MortgagePrice:integer read GetMortgagePrice;
      //Property which enabled interaction with the field FBuildingPrice
      property BuildingPrice:integer read GetBuildingPrice;
      //Property which enabled interaction with the field FOwned
      property owned:boolean read GetOwned write SetOwned;
      //Property which enabled interaction with the field FOwnedBy
      property OwnedBy:integer read GetOwnedBy write SetOwnedBy;
      //Property which enabled interaction with the field FMortgaged
      property Mortgaged:boolean read IsMortgaged write SetMortgaged;
      //Increments the number of houses/hotels as FProperties by 1
      procedure IncProperties;
      //Returns the number of houses/hotels as FProperties
      function GetProperties:integer;
      //Returns the rent expected for landing on that tile depending on
the type and the
      //number of houses on it
      function GetRent:integer;
  end;

implementation

{ TTile }

constructor TTile.Create(tileinfo:TStringDynArray);
var
  i : integer;
begin
  Fid := strtoint(tileinfo[0]);
  Fname := tileinfo[2];

  if tileinfo[1] = 'True' then
  begin
    FisProperty := True;
    FCost := strtoint(tileinfo[3]);

    if tileinfo[4] = 'True' then
    begin
      FSpecialcase := True;
    end
    else
    begin
      for i := 0 to 5 do
      begin
        FRentPrices[i] := strtoint(tileinfo[i+5]);
      end;
      Fbuildingprice := strtoint(tileinfo[11]);
    end;
    Fmortgageprice := cost div 2;
  end
  else
  begin
    FisProperty := False;
  end;

  owned := False;
  ownedby := -1;
  FProperties := 0;
  mortgaged := False;

  if Fid = 0 then
  begin
    Fisproperty := False;
  end;
end;

function TTile.GetBuildingPrice: integer;
begin
  result := FBuildingPrice;
end;

function TTile.GetCost: integer;
begin
  result := FCost;
end;
```

```
function TTile.GetID: integer;
begin
  result := FID;
end;

function TTile.GetMortgagePrice: integer;
begin
  result := FMortgagePrice;
end;

function TTile.GetName: string;
begin
  result := Fname;
end;

function TTile.GetOwned: boolean;
begin
  result := FOwned;
end;

function TTile.GetOwnedBy: integer;
begin
  result := FOwnedBy;
end;

function TTile.GetProperties:integer;
begin
  result := FProperties;
end;

function TTile.GetRent: integer;
begin
  result := FRentPrices[FProperties];
end;

procedure TTile.IncProperties;
begin
  inc(FProperties);
  try
    FMortgageprice := FMortgageprice + round(0.5 * FBuildingprice);
  except
    On E: EInvalidOp do
    begin
      FMortgagePrice := FMortgagePrice + (FBuildingPrice div 2);
    end;
  end;
end;

function TTile.IsMortgaged: boolean;
begin
  result := FMortgaged;
end;

function TTile.IsProperty: boolean;
begin
  result := FIsProperty;
end;

function TTile.IsSpecialCase: boolean;
begin
```

```
  result := FSpecialCase;
end;

procedure TTile.SetMortgaged(m: boolean);
begin
  FMortgaged := m;
  if m = True then FProperties := 0
end;

procedure TTile.SetOwned(b: boolean);
begin
  FOwned := b;
end;

procedure TTile.SetOwnedBy(p: integer);
begin
  FOwnedBy := p;
end;

end.
```

uBoard.pas

```
unit uBoard;

interface

uses
  StrUtils, SysUtils, Dialogs, Classes, uTile, types;

type
  TBoard = class
    protected
      FFreeParkingBal:integer;

      function GetFreeParkingBal:integer;
      procedure SetFreeParkingBal(v:integer);

    public
      //This array holds the 40 tiles that compose the physical board
      tiles : array [0..39] of TTile;

      constructor Create;
      //This property enabled interaction with the Field FFreeParkingBal
      property FreeParkingBal:integer read GetFreeParkingBal write
SetFreeParkingBal;
  end;

implementation

{ TBoard }

constructor TBoard.Create;
var
  tilefile : textfile;
  line : string;
  tileinfo : TStringDynArray;
  tilenum : integer;
begin
  AssignFile(tilefile,'tiles.txt');
  Reset(tilefile);

  tilenum := 0;
  while not(EOF(tilefile)) do
  begin
    readln(tilefile,line);
    tileinfo := splitstring(line,',');

    tiles[tilenum] := TTile.Create(tileinfo);
    inc(tilenum);
  end;
  closefile(tilefile);
  FFreeParkingBal := 0;
end;

function TBoard.GetFreeParkingBal: integer;
begin
  result := FFreeParkingBal
end;

procedure TBoard.SetFreeParkingBal(v: integer);
begin
  FFreeParkingBal := v;
end;

end.
```

uPlayers.pas

```
unit uPlayers;

interface

uses
  uTile, SysUtils, Vcl.Graphics;

type
  TPlayer = class
    protected
      FName : string;
      FPos, FBalance, FRandomNum : integer;
      FHousenum, FHotelnum : integer;
      FDiff : char;
      FInJail : boolean;
      FJailCards, FJailTurns:integer;

      function GetName:string;
      procedure SetName(n:string);
      function GetPos:integer;
      procedure SetPos(p:integer);
      function GetBalance:integer;
      procedure SetBalance(b:integer);
      function GetRandomNum:integer;
      function GetDiff:char;
      procedure SetDiff(d:char);
      function GetInJail:boolean;
      procedure SetInjail(j:boolean);

    public
```

```
      alive, human, computer : boolean;
      Roll1, Roll2 : integer;

      Constructor create(n:string;c:boolean;d:char);
      //Returns the amount of money to be paid when the properties tax
chance is drawn
      function GetPropertiesTax : integer;
      //Property which enabled interaction with the field FName
      property name:string read GetName write SetName;
      //Property which enabled interaction with the field FPos
      property pos:integer read GetPos write SetPos;
      //Property which enabled interaction with the field FBalance
      property balance:integer read GetBalance write SetBalance;
      //Property which enabled interaction with the field FRandomUpper -
the variable which
      property randomupper:integer read GetRandomNum;
      //Returns a string used to output to the player which contains
their name and balance
      function GetBalanceLine:string;
      //Property which enabled interaction with the field FDiff
      property diff:char read GetDiff write SetDiff;
      //Property which enabled interaction with the field FInJail
      property InJail:boolean read GetInJail write SetInJail;
      //Returns the value in FJailCards - not a property as game
interacts with the field
      //differently
      function GetJailCards:integer;
      //Increments FJailCards by 1
      procedure IncJailCards;
      //Increments FJailTurns by 1
      procedure IncJailTurns;
      //Returns the value in FJailTurns
      function GetJailTurns:integer;
      //Sets Jail turns to 0
      procedure ResetJailTurns;
      //Causes FBalance to increase by the argument/parameter
      procedure IncreaseBalance(value:integer);
      //Causes FBalance to decrease by the argument/parameter
      procedure DecreaseBalance(value:integer);

      //GUI methods
      //Returns a y coordinate which is dependant on FPos for the counter
      function GetLeftPos:integer;
      //Returns an x coordinate which is dependant on FPos for the
counter
      function GetTopPos:integer;
      //Returns a colour which depends on the player number
      function GetColour(playernum:integer):TColor;
      //Returns a (different to GetColour) colour which depends on the
player number
      function GetMortgageColour(playernum:integer):TColor;
  end;

implementation

{ TPlayer }

constructor TPlayer.create(n:string;c:boolean;d:char);
begin
  Fname := n;
  if c = False then
  begin
    human := True;
  end
  else
  begin
    computer := True;
    Fdiff := d;
    if Fdiff = 'e' then FRandomNum := 5;
    if Fdiff = 'm' then FRandomNum := 3;
    if Fdiff = 'h' then FRandomNum := 2;
  end;
  Fbalance := 1500;
  Fjailturns := 0;
  Fpos := 0;
  alive := True;
end;

function TPlayer.GetPos: integer;
begin
  result := FPos;
end;

function TPlayer.GetPropertiesTax: integer;
var
  runningcost : integer;
begin
  runningcost := 25 * Fhousenum;
  runningcost := runningcost + (120 * Fhotelnum);
  result := runningcost;
end;

function TPlayer.GetRandomNum: integer;
begin
  result := FRandomNum;
end;

procedure TPlayer.DecreaseBalance(value: integer);
begin
  FBalance := FBalance - value;
end;

function TPlayer.GetBalance: integer;
begin
  result := FBalance;
end;

function TPlayer.GetBalanceLine: string;
begin
  result := FName + '''s new balance is £' + inttostr(FBalance);
end;

function TPlayer.GetColour(playernum: integer): TColor;
begin
  case playernum of
    0 : result := $00F91800;
    1 : result := $002F0DEC;
    2 : result := $0000EA0C;
    3 : result := $0000F9F2;
  end;
end;
```

```
function TPlayer.GetDiff: char;
begin
  result := FDiff;
end;

function TPlayer.GetInJail: boolean;
begin
  result := FInJail;
end;

function TPlayer.GetJailCards: integer;
begin
  result := FJailCards;
end;

function TPlayer.GetJailTurns: integer;
begin
  result := FJailTurns;
end;

function TPlayer.GetLeftPos: integer;
var
  relativepos : integer;
begin
  if FInJail = True then
  begin
    result := 132;
  end
  else
  begin
    case pos of
      0: begin
        result := 443
      end;
      1..9: begin
        relativepos := pos mod 10;
        result := (434 - (relativepos*30));
      end;
      10: begin
        result := 108;
      end;
      11..19: begin
        result := 117;
      end;
      20: begin
        result := 124;
      end;
      21..29: begin
        relativepos := pos mod 10;
        result := (134 + (relativepos * 30));
      end;
      30: begin
        result := 132;
      end;
      31..39: begin
        result := 451;
      end;
    end;
  end;
end;
```

```
function TPlayer.GetMortgageColour(playernum: integer): TColor;
begin
  case playernum of
    0 : result := $00AE7D68;
    1 : result := $006456A3;
    2 : result := $005C8D63;
    3 : result := $0044AFB5;
  end;
end;

function TPlayer.GetName: string;
begin
  result := FName;
end;

function TPlayer.GetTopPos: integer;
var
  relativepos : integer;
begin
  if Finjail = True then
  begin
    result := 443;
  end
  else
  begin
    case pos of
      0: begin
        result := 448
      end;
      1..9: begin
        result := 455;
      end;
      10: begin
        if Finjail = true then
        begin
          result := 443
        end
        else
        begin
          result := 464
        end;
      end;
      11..19: begin
        relativepos := pos mod 10;
        result := (439 - (relativepos * 30));
      end;
      20: begin
        result := 129;
      end;
      21..29: begin
        result := 121;
      end;
      30: begin
        result := 443;
      end;
      31..39: begin
        relativepos := pos mod 10;
        result := ((137 + relativepos * 30));
      end;
    end;
  end;
```

```
end;

procedure TPlayer.IncJailCards;
begin
  inc(FJailCards);
end;

procedure TPlayer.IncJailTurns;
begin
  inc(FJailTurns);
end;

procedure TPlayer.IncreaseBalance(value: integer);
begin
  FBalance := FBalance + value;
end;

procedure TPlayer.ResetJailTurns;
begin
  FJailTurns := 0;
end;

procedure TPlayer.SetBalance(b: integer);
begin
  FBalance := b;
end;

procedure TPlayer.SetDiff(d: char);
begin
  FDiff := d;
end;

procedure TPlayer.SetInjail(j: boolean);
begin
  FInJail := j;
end;

procedure TPlayer.SetName(n: string);
begin
  FName := n;
end;

procedure TPlayer.SetPos(p: integer);
begin
  FPos := p;
end;

end.
```

uTrade.pas

```
unit uTrade;

interface

type
  TTrade = record
    //PlayerTo = Stores the player number of the player who the trade is
directed to
```

```
    //PlayerFrom = Stores the player number of the player who the trade
is form
    PlayerTo, PlayerFrom : integer;
    //PropertiesTo = Stores the ids of the properties which PlayerFrom
wants to give away
    //PropertiesFrom = Stores the ids of the properties which PlayerFrom
wants in return
    PropertiesTo, PropertiesFrom : array of integer;
    //MoneyTo = Stores the amount of money which PlayerFrom wants to give
away
    //MoneyFrom = Stores the amount of money which PlayerFrom wants in
return
    MoneyTo, MoneyFrom : integer;
  end;

implementation

end.
```

uStack.pas

```
unit uStack;

interface

type

  TNumberStack = class
    protected
      numbers : array [0..11] of integer;
      //Swaps the numbers in the positions a and b
      procedure Swap(a,b:integer);
    public
      constructor Create;
      //Shuffle which uses the Fisher-Yates shuffle algorithm
      procedure Shuffle;
      //Pops the topmost number of the array and returns it
      function Pop:integer;
      //Pushes the argument x to the bottom of the stack
      procedure PushToBottom(x:integer);
  end;

implementation

{ TNumberStack }

constructor TNumberStack.Create;
var
  i: Integer;
begin
  for i := 0 to 11 do
  begin
    numbers[i] := i+1;
  end;
end;

function TNumberStack.Pop: integer;
var
  tempint:integer;
  i: Integer;
begin
  tempint := numbers[0];
  for i := 0 to 10 do numbers[i] := numbers[i+1];
  result := tempint;
end;

procedure TNumberStack.PushToBottom(x: integer);
begin
  numbers[11] := x;
end;

procedure TNumberStack.Shuffle;
var
  i, Random1, Random2: Integer;
begin
  for i := 0 to 200 do
  begin
    Random1 := Random(12);
    Random2 := Random(12);
    if Random1 <> Random2 then Swap(Random1,Random2);
  end;
end;

procedure TNumberStack.Swap(a,b:integer);
var
  tempint:integer;
begin
  tempint := numbers[a];
  numbers[a] := numbers[b];
  numbers[b] := tempint;
end;

end.
```

uGameForm.pas

```
unit uGameForm;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
System.Classes,
  Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls,
uBoard,
  Vcl.ExtCtrls, Vcl.Imaging.jpeg, Vcl.Imaging.pngimage, Vcl.Buttons,
  uPlayers, uTile, uStack, uTrade;

  //Smaller = shorter game; larger = longer game
  const GoMoney:integer = 100;

type
  TGameForm = class(TForm)
    GridPanel: TGridPanel;
    Start: TButton;
    PlayerNames: TLabel;
    Player1, Player2, Player3, Player4: TEdit;
    P1Type, P2Type, P3Type, P4Type: TCheckBox;
    AddPlayer, StartGame: TButton;
    PlayerType, PlayerDiff: TLabel;
    BoardIMG: TImage;
    P1Diff, P2Diff, P3Diff, P4Diff: TEdit;
    DiffHelp, Outp: TMemo;
    RollDiceB, EndTurnB: TButton;
    Buy, DBuy: TButton;
    CostLbl: TLabel;
    UseJC, DUseJc: TButton;
    ScaleFactorEdit: TEdit;
    p1c, p2c, p3c, p4c: TShape;
    tile1, tile3, tile5, tile6, tile8, tile9, tile11, tile12, tile13,
tile14, tile15,
    tile16, tile18, tile19, tile21, tile23, tile24, tile25, tile26,
tile27, tile28,
    tile29, tile31, tile32, tile34, tile35, tile37, tile39: TShape;
    p1b, p2b, p3b, p4b: TButton;
    OutL: TLabel;
    E1, E2: TEdit;
    L1, L2: TLabel;
    SendB: TButton;
    Lfrom: TLabel;
    TileL: TLabel;
    Lto: TLabel;
    AcceptB, RejectB: TButton;
    MortgageB, PropertiesB, TradeB: TButton;
    tile1img, tile3img, tile5img, tile6img, tile8img, tile9img,
tile11img, tile12img,
    tile13img, tile14img, tile15img, tile16img, tile18img, tile19img,
tile21img,
    tile23img, tile24img, tile25img, tile26img, tile27img, tile28img,
tile29img,
    tile31img, tile32img, tile34img, tile35img, tile37img, tile39img:
TImage;
    NameL, CostL, PropertiesL: TLabel;
    CloseB: TButton;

    procedure StartClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure AddPlayerClick(Sender: TObject);
    procedure StartGameClick(Sender: TObject);
    procedure RollDiceBClick(Sender: TObject);
    procedure EndTurnBClick(Sender: TObject);
    procedure BuyClick(Sender: TObject);
    procedure DBuyClick(Sender: TObject);
    procedure DUseJcClick(Sender: TObject);
    procedure UseJCClick(Sender: TObject);
    procedure GetScaleFactor;
    procedure tile1ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure TradeBClick(Sender: TObject);
    procedure p1bClick(Sender: TObject);
    procedure p2bClick(Sender: TObject);
    procedure p3bClick(Sender: TObject);
    procedure p4bClick(Sender: TObject);
    procedure tile3ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile5ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure RejectBClick(Sender: TObject);
    procedure SendBClick(Sender: TObject);
    procedure AcceptBClick(Sender: TObject);
    procedure tile6ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile8ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile9ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile11ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile12ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile13ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile14ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile15ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile16ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile18ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile19ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile21ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile23ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile24ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile25ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile26ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile27ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile28ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile29ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile31ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile32ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile34ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile35ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile37ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure tile39ContextPopup(Sender: TObject; MousePos: TPoint; var
Handled: Boolean);
    procedure MortgageBClick(Sender: TObject);
    procedure PropertiesBClick(Sender: TObject);
    procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y:
Integer);
    procedure tile3imgMouseEnter(Sender: TObject);
    procedure tile1imgMouseEnter(Sender: TObject);
    procedure tile5imgMouseEnter(Sender: TObject);
    procedure tile6imgMouseEnter(Sender: TObject);
    procedure tile8imgMouseEnter(Sender: TObject);
    procedure tile9imgMouseEnter(Sender: TObject);
    procedure tile11imgMouseEnter(Sender: TObject);
    procedure tile12imgMouseEnter(Sender: TObject);
    procedure tile13imgMouseEnter(Sender: TObject);
    procedure tile14imgMouseEnter(Sender: TObject);
    procedure tile15imgMouseEnter(Sender: TObject);
    procedure tile16imgMouseEnter(Sender: TObject);
    procedure tile18imgMouseEnter(Sender: TObject);
    procedure tile19imgMouseEnter(Sender: TObject);
    procedure tile21imgMouseEnter(Sender: TObject);
    procedure tile23imgMouseEnter(Sender: TObject);
    procedure tile25imgMouseEnter(Sender: TObject);
    procedure tile24imgMouseEnter(Sender: TObject);
    procedure tile26imgMouseEnter(Sender: TObject);
    procedure tile27imgMouseEnter(Sender: TObject);
    procedure tile28imgMouseEnter(Sender: TObject);
    procedure tile29imgMouseEnter(Sender: TObject);
    procedure tile31imgMouseEnter(Sender: TObject);
    procedure tile32imgMouseEnter(Sender: TObject);
    procedure tile34imgMouseEnter(Sender: TObject);
    procedure tile35imgMouseEnter(Sender: TObject);
    procedure tile37imgMouseEnter(Sender: TObject);
    procedure tile39imgMouseEnter(Sender: TObject);
    procedure CloseBClick(Sender: TObject);

  private
  //Setup variables
    playernum : integer;

  //Game variables
    board:TBoard;
    players:array of TPlayer;
    ChanceStack, CChestStack : TNumberStack;
    CurrentPlayer, tradingplayer:integer;
    double, intrade, pendingtrade, mortgaging, buyinghouses:boolean;
    currenttrade:TTrade;
    p1alive,p2alive,p3alive,p4alive:boolean;
    ingame, allcomputerplayers:boolean;

  //Gui parts
    scalefctr : integer;
    propertyshapes : array [0..39] of TShape;
    procedure MoveCounter;
    procedure PrepareTileShapeArray;
    procedure setGUIstart;
    procedure setGUIplayerChoices;
    procedure setGUIgame;
    procedure HoverOver(t:integer);

    procedure StartPlayerGame;
    procedure PassedGo;
    procedure NormalTurn(r1,r2:integer;b,g:boolean);
    procedure RollDiceProcedure;
    procedure EndTurnProcedure;
    procedure nextplayer;
    procedure CheckForDouble;
    procedure RollLine(x,y:integer);
    procedure removePlayer(player:integer);
    function GetAlivePlayers:integer;
    function GetTrainOwned(p:integer):integer;
    procedure LandedOnChance;
    procedure LandedOnCommunityChest;
    procedure UseJailCard;
    procedure DontUseJailCard;
    procedure ClearCurrentTrade;
    procedure SetupTrade;
    procedure AcceptTrade;
    procedure RejectTrade;
    procedure TileShapeClick(p:integer);
    function PlayerOwnRange(player,clickedpos:integer):boolean;
    procedure ComputerTurn;
    function FAllComputerPlayers:boolean;
    procedure ComputerBuyProperties;
    procedure EndGame;
    function ArrayContains(arr:array of integer;elem:integer):boolean;
  end;

  Const PropertyIDs  : array  [0..27] of integer =
(1,3,5,6,8,9,11,12,13,14,15,16,18,19,21,
  23,24,25,26,27,28,29,31,32,34,35,37,39);

var
  GameForm: TGameForm;

implementation

{$R *.dfm}

//When the accept trade button is clicked
procedure TGameForm.AcceptBClick(Sender: TObject);
begin
  accepttrade;
end;

//This uses the record created when making the trade and engages the
exchange between the
//players, then it modifies the property images to match the correct
owner
procedure TGameForm.accepttrade;
var
  i : integer;
begin
  players[CurrentPlayer].IncreaseBalance(currenttrade.moneyto);
  players[currenttrade.playerfrom].DecreaseBalance(currenttrade.moneyto);
  players[CurrentPlayer].DecreaseBalance(currenttrade.moneyfrom);

players[currenttrade.playerfrom].IncreaseBalance(currenttrade.moneyfrom);
  for i := 0 to Length(currenttrade.propertiesto)-1 do
  begin
    board.tiles[currenttrade.propertiesto[i]].ownedby := CurrentPlayer;
    propertyshapes[currenttrade.propertiesto[i]].Brush.Color :=
    players[0].GetColour(CurrentPlayer);
  end;
  for i := 0 to Length(currenttrade.propertiesfrom)-1 do
  begin
    board.tiles[currenttrade.propertiesfrom[i]].ownedby :=
currenttrade.playerfrom;
    propertyshapes[currenttrade.propertiesfrom[i]].Brush.Color :=
    players[0].GetColour(currenttrade.playerfrom);
  end;
  ClearCurrentTrade;
  AcceptB.Hide;
  RejectB.Hide;
  RollDiceB.Enabled := True;
  pendingTrade := False;
  outp.Lines.Add('Trade accepted');
  intrade := false;
end;

//Enabled more players to be added at the getting players stage
procedure TGameForm.AddPlayerClick(Sender: TObject);
begin
  if playernum = 2 then
  begin
    Player3.Show;
    P3Type.Show;
    p3diff.show;
    inc(playernum)
  end
  else
  begin
    Player4.Show;
    P4Type.Show;
    p4diff.Show;
    inc(playernum);
    AddPlayer.Hide;
  end;
end;

//A generic function required for determining if a property had already
been clicked in the
//trading section
function TGameForm.arraycontains(arr: array of integer; elem: integer):
boolean;
var
  i: Integer;
begin
  for i := 0 to length(arr)-1 do
  begin
    if arr[i] = elem then result := True;
  end
end;

//Subtracts the cost of the propety from the player and changes the
colour of the tile
//image
procedure TGameForm.BuyClick(Sender: TObject);
begin
  Buy.Hide;
  DBuy.Hide;
  CostLbl.Hide;

players[CurrentPlayer].DecreaseBalance(board.tiles[players[CurrentPlayer]
.pos].cost);
  board.tiles[players[CurrentPlayer].pos].owned := True;
  board.tiles[players[CurrentPlayer].pos].ownedby := CurrentPlayer;
  Outp.Lines.Add(players[CurrentPlayer].name + ' has purchased ' +
  board.tiles[players[CurrentPlayer].pos].name);
  Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
  propertyshapes[players[CurrentPlayer].pos].Show;
  propertyshapes[players[CurrentPlayer].pos].Brush.Color :=
  players[0].GetColour(CurrentPlayer);
  if players[CurrentPlayer].balance < 0 then
  begin
```

```
    TradeB.Hide;
    outp.Lines.Add(players[CurrentPlayer].name + ' is currently in
debt');
    outp.Lines.Add(players[CurrentPlayer].name +
    ' needs to mortgage properties to a positive balance or will be
kicked out the game');
    MortgageB.Show;
  end;
  CheckForDouble;
end;

//Recursive call for the main turn function of a human player
procedure TGameForm.CheckForDouble;
begin
  if double = True then
  begin
    outp.Lines.Add(players[CurrentPlayer].name + ' rolled a double so
takes another turn');
    outp.Lines.Add('');
    double := False;
    RollDiceProcedure;
  end
  else EndTurnB.Enabled := True;
end;

//Upon accepting or rejectying a trade, the record must be cleared
procedure TGameForm.ClearCurrentTrade;
begin
  Currenttrade.playerto := -1;
  Currenttrade.playerfrom := -1;
  setLength(Currenttrade.propertiesto, 1);
  setLength(Currenttrade.propertiesfrom, 1);
  Currenttrade.moneyto := 0;
  Currenttrade.moneyfrom := 0;
  tradingplayer := -1;
  E1.Text := '';
  E2.Text := '';
  Lfrom.Hide;
  lto.Hide;
  Currenttrade := Default(TTrade);
end;

//At the end of the game the close button closes the form
procedure TGameForm.CloseBClick(Sender: TObject);
var
  i: Integer;
begin
  for i := 0 to 39 do board.tiles[i].Free;
  board.Free;
  for i := 0 to length(players)-1 do players[i].Free;
  GameForm.Close;
end;

//Recursive call for the computer to buy properties on any tiles it can
procedure TGameForm.computerbuyproperties;
var
  i : integer;
begin
  for i := 0 to 27 do
  begin
    if (playerownrange(CurrentPlayer,propertyids[i]) = true) and
```

```
       (players[CurrentPlayer].balance >
board.tiles[propertyids[i]].buildingprice) and
       (board.tiles[propertyids[i]].Getproperties < 5) then
    begin
      board.tiles[propertyids[i]].IncProperties;

players[CurrentPlayer].DecreaseBalance(board.tiles[propertyids[i]].buildi
ngprice);
      outp.Lines.Add(players[CurrentPlayer].name + ' purchased a property
on ' +
      board.tiles[propertyids[i]].name + ' for ' +
      inttostr(board.tiles[propertyids[i]].buildingprice));
      if (players[CurrentPlayer].balance > 5 *
board.tiles[propertyids[i]].buildingprice)
         and (players[CurrentPlayer].diff = 'h') then
computerbuyproperties;
    end;
  end;
end;

//main function for the computer turn
procedure TGameForm.computerturn;
var
  randint1, r1, r2, newpos, rollsum, tempcost: integer;
begin
  randomize;
  if (pendingtrade = True) and (CurrentPlayer = currenttrade.playerto)
then
  begin
    //Randomly decides to accept or reject a trade offer - more likely to
accept if easy
    //difficulty
    if players[CurrentPlayer].diff = 'e' then randint1 := random(3);
    if players[CurrentPlayer].diff = 'm' then randint1 := random(5);
    if players[CurrentPlayer].diff = 'h' then randint1 := random(6);
    if randint1 = 0 then accepttrade
    else rejecttrade;
  end;
  r1 := random(6) + 1;
  r2 := random(6) + 1;
  //randint1 is determined by difficulty of the computer player and its
value decides what
  //actions are taken on its turn e.g. buying properties or not
  randint1 := random(players[CurrentPlayer].randomupper);
  double := False;
  RollLine(r1,r2);
  if r1 = r2 then double := true;
  rollsum := r1 + r2;
  if players[CurrentPlayer].pos + rollsum > 39 then passedgo;
  if players[CurrentPlayer].injail = False then
  begin
    players[CurrentPlayer].pos := ((players[CurrentPlayer].pos + r1 + r2)
mod 40);
    newpos := players[CurrentPlayer].pos;
    outp.Lines.Add(players[CurrentPlayer].name + ' has landed on ' +
    board.tiles[players[CurrentPlayer].pos].name);
    if (board.tiles[newpos].owned = False) and
(board.tiles[newpos].isproperty) then
    begin
      if randint1 = 0 then
      begin
```

```
        //If possible and ranint1 is 0 then the property is bought
        if players[CurrentPlayer].diff = 'h' then
        begin
          if players[CurrentPlayer].balance > (2 *
board.tiles[newpos].cost) then
          begin
            if (newpos = (players[0].pos - 7)) or
               (newpos = (players[1].Pos - 7)) or
               (newpos = 16) or
               (newpos = 18) or
               (newpos = 19) then
            begin
              players[CurrentPlayer].DecreaseBalance
              (board.tiles[players[CurrentPlayer].pos].cost);
              board.tiles[players[CurrentPlayer].pos].owned := True;
              board.tiles[players[CurrentPlayer].pos].ownedby :=
CurrentPlayer;
              Outp.Lines.Add(players[CurrentPlayer].name + ' has
purchased ' +
              board.tiles[players[CurrentPlayer].pos].name);
              Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
              if allcomputerplayers = False then
              propertyshapes[players[CurrentPlayer].pos].Show;
              propertyshapes[players[CurrentPlayer].pos].Brush.Color :=
              players[CurrentPlayer].GetColour(CurrentPlayer);
            end;
          end;
        end
        else
        begin
          players[CurrentPlayer].DecreaseBalance(
          board.tiles[players[CurrentPlayer].pos].cost);
          board.tiles[players[CurrentPlayer].pos].owned := True;
          board.tiles[players[CurrentPlayer].pos].ownedby :=
CurrentPlayer;
          Outp.Lines.Add(players[CurrentPlayer].name + ' has purchased '
+
          board.tiles[players[CurrentPlayer].pos].name);
          Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
          if allcomputerplayers = False then
propertyshapes[players[CurrentPlayer].pos]
          .Show;
          propertyshapes[players[CurrentPlayer].pos].Brush.Color :=
          players[CurrentPlayer].GetColour(CurrentPlayer);
        end;
      end
    end
    else
    begin
      if (board.tiles[players[CurrentPlayer].pos].owned = True) and
         (board.tiles[players[CurrentPlayer].pos].ownedby <>
CurrentPlayer) and
         (board.tiles[players[CurrentPlayer].pos].mortgaged = False) and
         (board.tiles[players[CurrentPlayer].pos].specialcase = False)
then
      begin

players[CurrentPlayer].DecreaseBalance(board.tiles[players[CurrentPlayer]
.pos]
        .GetRent);
```

```
players[board.tiles[players[CurrentPlayer].pos].ownedby].IncreaseBalance
        (board.tiles[players[CurrentPlayer].pos].GetRent);
        Outp.Lines.Add(board.tiles[players[CurrentPlayer].pos].name + '
is owned by ' +
        players[board.tiles[players[CurrentPlayer].pos].ownedby].name +
        ' and is owed ' +
inttostr(board.tiles[players[CurrentPlayer].pos].getrent));
        Outp.Lines.Add(

players[board.tiles[players[CurrentPlayer].pos].OwnedBy].GetBalanceLine);
        Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
      end
      else
      begin
        if (board.tiles[players[CurrentPlayer].pos].owned = True) and
           (board.tiles[players[CurrentPlayer].pos].ownedby <>
CurrentPlayer) and
           (board.tiles[players[CurrentPlayer].pos].specialcase = False)
then
        begin
          Outp.Lines.Add(board.tiles[players[CurrentPlayer].pos].name +
          ' is mortgaged so no rent is paid');
        end
        else
        begin
          if (board.tiles[players[CurrentPlayer].pos].owned = True) and
             (board.tiles[players[CurrentPlayer].pos].ownedby <>
CurrentPlayer) and
             (board.tiles[players[CurrentPlayer].pos].specialcase = True)
then
          begin
            if (board.tiles[players[CurrentPlayer].pos].id = 12) or
               (board.tiles[players[CurrentPlayer].pos].id = 28) then
            begin
              //Exchanges the rent in the unique utilities method
              if board.tiles[28].ownedby = board.tiles[12].ownedby then
              tempcost := 10 * (r1+r2)
              else tempcost := 4 * (r1+r2);
              players[CurrentPlayer].balance :=
players[CurrentPlayer].balance -
              tempcost;

players[board.tiles[players[CurrentPlayer].pos].ownedby].balance :=

players[board.tiles[players[CurrentPlayer].pos].ownedby].balance +
              tempcost;
              Outp.Lines.Add(board.tiles[players[CurrentPlayer].pos].name
+
              ' is owned by ' +

players[board.tiles[players[CurrentPlayer].pos].ownedby].name +
              ' and is owed ' + inttostr(tempcost));
              Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);

Outp.Lines.Add(players[board.tiles[players[CurrentPlayer].pos]
              .ownedby].GetBalanceLine);
            end
            else
            begin
              if (board.tiles[players[CurrentPlayer].pos].id = 5) or
```

```
                 (board.tiles[players[CurrentPlayer].pos].id = 15) or
                 (board.tiles[players[CurrentPlayer].pos].id = 25) or
                 (board.tiles[players[CurrentPlayer].pos].id = 35) then
              begin
                //Exchanges the rent in the unique station method
                tempcost :=
GetTrainOwned(board.tiles[players[CurrentPlayer].pos].OwnedBy);
                players[CurrentPlayer].balance :=
players[CurrentPlayer].balance - tempcost;

players[board.tiles[players[CurrentPlayer].pos].ownedby].balance :=

players[board.tiles[players[CurrentPlayer].pos].ownedby].balance +
tempcost;

Outp.Lines.Add(board.tiles[players[CurrentPlayer].pos].name +
                ' is owned by ' +
players[board.tiles[players[CurrentPlayer].pos]
                .ownedby].name + ' and is owed ' + inttostr(tempcost));
                Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);

Outp.Lines.Add(players[board.tiles[players[CurrentPlayer].pos].ownedby]
                .GetBalanceLine);
              end
              else
              begin
                //The case statements for the unique tile locations
                case board.tiles[players[CurrentPlayer].pos].id of
                  20 : begin
                    //Upon landing on free parking, that player recieved
all the
                    //amount collected from e.g. tax tiles

players[CurrentPlayer].IncreaseBalance(board.freeparkingbal);
                    if board.freeparkingbal = 0 then
                    begin
                      Outp.Lines.Add('There was no money on the free
parking tile');
                    end
                    else
                    begin
                      Outp.Lines.Add(players[CurrentPlayer].name + ' got
' +
                      inttostr(board.freeparkingbal) + ' from free
parking');
                    end;
                    board.freeparkingbal := 0;
                  end;

                  4 : begin
                    //Upon landing on income tax, the player pays 200 to
the free
                    //parking balance
                    Outp.Lines.Add(players[CurrentPlayer].name +
                    ' paid $200 to free parking');
                    board.freeparkingbal := board.freeparkingbal + 200;
                    players[CurrentPlayer].DecreaseBalance(200);
                  end;

                  38 : begin
```

```
          //Upon landing on super tax, the player pays 100 to
the free parking
          //balance
          Outp.Lines.Add(players[CurrentPlayer].name +
          ' paid $100 to free parking');
          board.freeparkingbal := board.freeparkingbal + 100;
          players[CurrentPlayer].DecreaseBalance(100);
         end;

         30 :begin
          //Upon landing on go to jail, that player is sent
tojail and
          //GUI updated
          players[CurrentPlayer].injail := True;
          players[CurrentPlayer].IncJailTurns;
          Outp.Lines.Add(players[CurrentPlayer].name + ' has
entered jail');
          players[CurrentPlayer].pos := 10;
          MoveCounter;
         end;

         //7, 22 or 36 are all chance tiles
         7 : landedonchance;
         22 : landedonchance;
         36 : landedonchance;

         //2, 17, 33 are all community chest tiles
         2 : landedoncommunitychest;
         17 : landedoncommunitychest;
         33 : landedoncommunitychest;
        end;
       end;
      end;
     end;
    end;
   end;
  end;
 end
 else
 begin
  //If in jail and the player has get out of jail free cards then the
computer uses them
  if players[CurrentPlayer].GetJailCards > 0 then usejailcard
  else
  begin
   if double = True then
   begin
    players[CurrentPlayer].InJail := False;
    players[CurrentPlayer].ResetJailTurns;
    computerturn;
   end
   else
   begin
    outp.Lines.Add('computer player is in jail');
    players[CurrentPlayer].IncJailTurns;
   end;
  end;
 end;
 //Only update the positions of the counter if there is a human player

  if allcomputerplayers = False then MoveCounter;
  if (randint1 = 2) then
  begin
    //The computer recursively buys properties if it is able to
    computerbuyproperties;
  end;

  if double = True then computerturn
  else EndTurnProcedure;
end;

//If the player doesn't want to buy the property then the buttons are
hidden
procedure TGameForm.DBuyClick(Sender: TObject);
begin
  DBuy.Hide;
  Buy.Hide;
  CostLbl.Hide;
end;

//If the player doesn't want to user their get out of jail free cards
then a normal turn
//is initiated
procedure TGameForm.DontUseJailCard;
begin
  if double = True then
  begin
    RollLine(players[CurrentPlayer].roll1,players[CurrentPlayer].roll2);
    players[CurrentPlayer].injail := False;
    players[CurrentPlayer].ResetJailTurns;

NormalTurn(players[CurrentPlayer].roll1,players[CurrentPlayer].roll2,true
,false);
  end
  else
  begin
    RollLine(players[CurrentPlayer].roll1,players[CurrentPlayer].roll2);
    players[CurrentPlayer].IncJailTurns;
    if players[CurrentPlayer].Getjailturns > 3 then
    begin
      //As in the real game, if the player has been in jail for 3 turns,
then on the 4th
      //they are charged
      Outp.Lines.Add(players[CurrentPlayer].name +
      ' has been in jail for 3 turns and will be charged 100');
      players[CurrentPlayer].balance := players[CurrentPlayer].balance -
100;
      players[CurrentPlayer].injail := False;
      players[CurrentPlayer].Resetjailturns;

NormalTurn(players[CurrentPlayer].roll1,players[CurrentPlayer].roll2,true
,false);
      MoveCounter;
    end
  end;
end;

//If the player doesn't wish to use their get out of jail free cards then
the buttons
//are hidden
```

```
procedure TGameForm.DUseJcClick(Sender: TObject);
begin
  DontUseJailCard;
  DUseJc.Hide;
  UseJc.Hide;
end;

//At the end of the computer turn, the final game state is shown to the
user
procedure TGameForm.EndGame;
var
  i: Integer;
begin
  players[CurrentPlayer].computer := False;
  TradeB.hide;
  MortgageB.Hide;
  PropertiesB.Hide;
  CloseB.show;
  for i := 0 to 27 do
  begin
    if board.tiles[propertyids[i]].owned = True then
    begin
      propertyshapes[propertyids[i]].Show;
      propertyshapes[propertyids[i]].Brush.Color :=

players[CurrentPlayer].GetColour(board.tiles[propertyids[i]].ownedby);
    end;
  end;
end;

//For when the end turn button is clicked
procedure TGameForm.EndTurnBClick(Sender: TObject);
begin
  endturnprocedure;
end;

//At the end of a turn, it prepares the game for the next players turn
procedure TGameForm.endturnprocedure;
var
  i :integer;
begin
  double := False;
  randomize;
  buy.Hide;
  dbuy.Hide;
  if players[CurrentPlayer].balance < 0 then
  begin
    //If the players balance is below zero then they are kicked out
    outp.Lines.Add(players[CurrentPlayer].name + ' has been kicked out
the game');
    removePlayer(CurrentPlayer);

    //If that player is involved in the pending trade then it is cleared
as would block
    //other trades in the future
    if (pendingtrade = True) and ((currenttrade.playerto = CurrentPlayer)
or
        (currenttrade.playerfrom = CurrentPlayer)) then
    begin
      outp.Lines.Add('The current trade involved ' +
players[CurrentPlayer].name +
   ' so has been cleared');
   for i := 0 to Length(currenttrade.propertiesto)-1 do
   begin
    propertyshapes[currenttrade.propertiesto[i]].Brush.Color :=

players[CurrentPlayer].GetColour(board.tiles[currenttrade.propertiesto[i]
].ownedby)
   end;
   for i := 0 to Length(currenttrade.propertiesfrom)-1 do
   begin
    propertyshapes[currenttrade.propertiesfrom[i]].Brush.Color :=
    players[CurrentPlayer].GetColour(
    board.tiles[currenttrade.propertiesfrom[i]].ownedby)
   end;
   clearcurrenttrade;
  end;
  //If there is 1 player remaining then the game is over
  if GetAlivePlayers = 1 then
  begin
   EndTurnB.Hide;
   RollDiceB.Hide;
   nextplayer;
   showmessage(players[CurrentPlayer].name + ' has won the game');
   EndGame;
  end
 end;
 //Organises the next alive player to become 'CurrentPlayer'
 nextPlayer;
 if players[CurrentPlayer].computer = True then
 begin
  computerturn;
 end
 else
 begin
  if (pendingtrade = True) and (CurrentPlayer = currenttrade.playerto)
then
  begin
   RollDiceB.Enabled := False;
   outp.Lines.Add(players[CurrentPlayer].name + ' has recieved a trade
from ' +
   players[currenttrade.playerfrom].name);
   outp.lines.Add(players[currenttrade.playerfrom].name + ' wants the
tiles:');
   for i := 0 to Length(currenttrade.propertiesfrom)-1 do
   begin
    outp.Lines.Add(board.tiles[currenttrade.propertiesfrom[i]].name);
   end;
   outp.lines.Add('And ' + inttostr(currenttrade.moneyfrom));
   outp.Lines.Add('');
   outp.Lines.Add('In return for the tiles:');
   for i := 0 to Length(currenttrade.propertiesto)-1 do
   begin
    outp.Lines.Add(board.tiles[currenttrade.propertiesto[i]].name);
   end;
   outp.lines.Add('And ' + inttostr(currenttrade.moneyto));
   AcceptB.Show;
   RejectB.Show;
  end
  else
  begin
   RollDiceB.Enabled := True;
```

```
      if pendingtrade = False then TradeB.Show;
      MortgageB.Show;
      propertiesb.Show;
    end;
  end;
  CostLbl.Hide;
end;

//Function to determine if all the players involved in the game are
computer controlled
function TGameForm.fallcomputerplayers: boolean;
begin
  if playernum = 3 then
  begin
    if (p1type.Checked = True) and (p2type.Checked = True) and
(p3type.Checked = True)
    then result := True
    else result := False;
  end
  else
  begin
    if playernum = 4 then
    begin
      if (p1type.Checked = True) and (p2type.Checked = True) and
(p3type.Checked = True)
      and (p4type.Checked = True) then result := True
      else result := False;
    end
    else
    begin
      if (p1type.Checked = True) and (p2type.Checked = True) then result
:= True
      else result := False;
    end
  end;
end;

//Prepares the form and ses up the gui
procedure TGameForm.FormCreate(Sender: TObject);
begin
  setGUIstart;
  GridPanel.Show;
  BoardIMG.Hide;
  scaled := true;
  ingame := False;
  preparetileshapearray;
end;

//If the mouse hovers over the form then the labels are hidden
procedure TGameForm.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  NameL.Hide;
  CostL.Hide;
  PropertiesL.Hide;
end;

//Determines how many players are alive
function TGameForm.GetAlivePlayers: integer;
var
  i, x : integer;
```

```
begin
  x := 0;
  for i := 0 to playernum-1 do
  begin
    if players[i].alive = True then inc(x)
  end;
  result := x;
end;

//Allows for the counters to be in the correct place as varies with
screen reslution
procedure TGameForm.GetScaleFactor;
begin
  if strtoint(scaleFactoredit.Text) in [0..9] then
  begin
    scalefctr := strtoint(ScaleFactorEdit.Text);
  end
  else
  begin
    scalefctr := 1;
  end;
end;

//Determines the number of trains stations a particular player owns
function TGameForm.GetTrainOwned(p:integer): integer;
var
 count : integer;
begin
  count := 0;
  if board.tiles[5].ownedby = p then
  begin
    inc(count);
  end;
  if board.tiles[15].ownedby = p then
  begin
    inc(count);
  end;
  if board.tiles[25].ownedby = p then
  begin
    inc(count);
  end;
  if board.tiles[35].ownedby = p then
  begin
    inc(count);
  end;

  case count of
    1 : result := 25;
    2 : result := 50;
    3 : result := 100;
    4 : result := 200;
  end;
end;

//If the mouse hovers over a tile then the infomation about that is
outputted in
//the labels
procedure TGameForm.hoverover(t: integer);
begin
  if ingame = True then
  begin
```

```
    NameL.Show;
    CostL.Show;
    PropertiesL.Show;
    NameL.Caption := 'Name: ' + board.tiles[t].name;
    CostL.Caption := 'Cost: ' + inttostr(board.tiles[t].cost);
    PropertiesL.Caption := 'Houses: ' +
inttostr(board.tiles[t].Getproperties);
  end;
end;

procedure TGameForm.tile14imgMouseEnter(Sender: TObject);
begin
  hoverover(14);
end;

procedure TGameForm.tile3imgMouseEnter(Sender: TObject);
begin
  hoverover(3);
end;

//Completes a random and appropriate action for landed on a chance tile
procedure TGameForm.landedonchance;
var
  tempint : integer;
begin
  tempint := ChanceStack.Pop;
  case tempint of
    1 : begin
      Outp.Lines.Add(players[CurrentPlayer].name + ' drew the advance to
go card');
      passedgo;
      players[CurrentPlayer].pos := 0;
    end;
    2 : begin
      players[CurrentPlayer].injail := True;
      players[CurrentPlayer].IncJailTurns;
      Outp.Lines.Add(players[CurrentPlayer].name + ' drew the advance to
jail card');
      Outp.Lines.Add(players[CurrentPlayer].name + ' has entered jail');
      players[CurrentPlayer].pos := 10;
    end;
    3 : begin
      Outp.Lines.Add(players[CurrentPlayer].name +
      ' drew the advance to Pall Mall card');
      if players[CurrentPlayer].pos > 13 then passedgo;
      players[CurrentPlayer].pos := 13;
      NormalTurn(0,0,false,false);
    end;
    4 : begin
      Outp.Lines.Add(players[CurrentPlayer].name + ' drew the advance to
Kings Cross card');
      if players[CurrentPlayer].pos > 5 then passedgo;
      players[CurrentPlayer].pos := 5;
      NormalTurn(0,0,false,false);
    end;
    5 : begin
      Outp.Lines.Add(players[CurrentPlayer].name +
      ' drew the advance to Trafalgar Square card');
      if players[CurrentPlayer].pos > 24 then passedgo;
      players[CurrentPlayer].pos := 24;
      NormalTurn(0,0,false,false);
  end;
  6 : begin
   Outp.Lines.Add(players[CurrentPlayer].name +
   ' drew the collect 150 from the bank card');
   players[CurrentPlayer].IncreaseBalance(150);
   Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
  end;
  7 : begin
   Outp.Lines.Add(players[CurrentPlayer].name + ' drew the advance to
Mayfair card');
   players[CurrentPlayer].pos := 39;
   NormalTurn(0,0,false,false);
  end;
  8 : begin
   Outp.Lines.Add(players[CurrentPlayer].name + ' drew the go back 3
spaces card');
   players[CurrentPlayer].pos := players[CurrentPlayer].pos - 3;
   Outp.Lines.Add(players[CurrentPlayer].name + ' is now on ' +
   board.tiles[players[CurrentPlayer].pos].name);
   NormalTurn(0,0,false,false);
  end;
  9 : begin
   outp.Lines.Add(players[CurrentPlayer].name +
   ' drew the pay 40 per house and 120 per hotel card');
   tempint := players[CurrentPlayer].GetPropertiesTax;
   outp.Lines.Add(players[CurrentPlayer].name + ' has to pay ' +
inttostr(tempint));
   players[CurrentPlayer].DecreaseBalance(tempint);
   Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
  end;
  10 : begin
   Outp.Lines.Add(players[CurrentPlayer].name +
   ' drew the collect 100 from the bank card');
   players[CurrentPlayer].IncreaseBalance(100);
   Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
  end;
  11 : begin
   Outp.Lines.Add(players[CurrentPlayer].name +
   ' drew the pay 150 for school fees card');
   players[CurrentPlayer].DecreaseBalance(150);
   Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
  end;
  12 : begin
   outp.Lines.Add(players[CurrentPlayer].name + ' drew the get out of
jail free card');
   players[CurrentPlayer].IncJailCards;
  end;
 end;
 ChanceStack.PushToBottom(tempint);
 MoveCounter;
end;

//Completes a random and appropriate action for landed on a community
chest tile
procedure TGameForm.landedoncommunitychest;
var
 tempint:integer;
begin
 tempint := CChestStack.Pop;
 case tempint of
  1 : begin
```

```
      Outp.Lines.Add(players[CurrentPlayer].name + ' drew the advance to 
go card');
      passedgo;
      players[CurrentPlayer].pos := 0;
    end;
    2 : begin
      Outp.Lines.Add(players[CurrentPlayer].name +
      ' drew the advance to Old Kent Road card');
      if players[CurrentPlayer].pos > 1 then passedgo;
      players[CurrentPlayer].pos := 1;
      NormalTurn(0,0,false,false);
    end;
    3 : begin
      outp.Lines.Add(players[CurrentPlayer].name + ' drew the advance to 
jail card');
      players[CurrentPlayer].injail := True;
      players[CurrentPlayer].IncJailTurns;
      Outp.Lines.Add(players[CurrentPlayer].name + ' has enterd jail');
      players[CurrentPlayer].pos := 10;
    end;
    4 : begin
      Outp.Lines.Add(players[CurrentPlayer].name +
      ' drew the pay 100 for hospital fees card');
      players[CurrentPlayer].DecreaseBalance(100);
      Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
    end;
    5 : begin
      Outp.Lines.Add(players[CurrentPlayer].name + ' drew the pay 50 for 
a doctor card');
      players[CurrentPlayer].DecreaseBalance(50);
      Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
    end;
    6 : begin
      Outp.Lines.Add(players[CurrentPlayer].name +
      ' drew the collect 200 for a bank error card');
      players[CurrentPlayer].IncreaseBalance(200);
      Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
    end;
    7 : begin
      Outp.Lines.Add(players[CurrentPlayer].name +
      ' drew the collect 100 from the bank');
      players[CurrentPlayer].IncreaseBalance(100);
      Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
    end;
    8 :  begin
       Outp.Lines.Add(players[CurrentPlayer].name +
       ' drew the collect 100 as inheritance');
      players[CurrentPlayer].IncreaseBalance(100);
      Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
    end;
    9 : begin
      Outp.Lines.Add(players[CurrentPlayer].name +
      ' drew the collect 20 from an income tax refund');
      players[CurrentPlayer].IncreaseBalance(20);
      Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
    end;
    10 : begin
       Outp.Lines.Add(players[CurrentPlayer].name + ' drew the take a 
chance card');
       landedonchance;
    end;
    11 : begin
      outp.Lines.Add(players[CurrentPlayer].name +
      ' drew the get out of jail free card');
      players[CurrentPlayer].IncJailCards;
    end;
    12 : begin
      Outp.Lines.Add(players[CurrentPlayer].name +
      ' drew the collect 50 from your stocks');
      players[CurrentPlayer].IncreaseBalance(50);
      Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
    end;
  end;
  CChestStack.PushToBottom(tempint);
  MoveCounter
end;

//Following 4 procedures are for when trading is being completed and
enabled the choice
//of who the trade is sent to
procedure TGameForm.p1bClick(Sender: TObject);
begin
  tradingplayer := 0;
  SetupTrade;
end;

procedure TGameForm.p2bClick(Sender: TObject);
begin
  tradingplayer := 1;
  SetupTrade;
end;

procedure TGameForm.p3bClick(Sender: TObject);
begin
  tradingplayer := 2;
  SetupTrade;
end;

procedure TGameForm.p4bClick(Sender: TObject);
begin
  tradingplayer := 3;
  SetupTrade;
end;

//Upon passing go, the correct player recieves a fixed amount of money
procedure TGameForm.passedgo;
begin
  Outp.Lines.Add(players[CurrentPlayer].name + ' recieved ' +
inttostr(gomoney) +
  ' from for passing go');
  players[CurrentPlayer].IncreaseBalance(gomoney);
end;

//Determines if a player owns a colour range which the clicked position
is part of
function TGameForm.playerownrange(player, clickedpos:integer): boolean;
begin
  if board.tiles[clickedpos].ownedby = CurrentPlayer then
  begin
    case clickedpos of
      1 : if board.tiles[clickedpos].ownedby = board.tiles[3].ownedby
then result := True;
```

```
      3 : if board.tiles[clickedpos].ownedby = board.tiles[1].ownedby
then result := True;
      6 : if (board.tiles[clickedpos].ownedby = board.tiles[8].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[9].ownedby) then
result := True;
      8 : if (board.tiles[clickedpos].ownedby = board.tiles[9].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[6].ownedby) then
result := True;
      9 : if (board.tiles[clickedpos].ownedby = board.tiles[6].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[8].ownedby) then
result := True;
      11 : if (board.tiles[clickedpos].ownedby = board.tiles[13].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[14].ownedby) then
result := True;
      13 : if (board.tiles[clickedpos].ownedby = board.tiles[14].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[11].ownedby) then
result := True;
      14 : if (board.tiles[clickedpos].ownedby = board.tiles[11].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[13].ownedby) then
result := True;
      16 : if (board.tiles[clickedpos].ownedby = board.tiles[18].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[19].ownedby) then
result := True;
      18 : if (board.tiles[clickedpos].ownedby = board.tiles[19].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[16].ownedby) then
result := True;
      19 : if (board.tiles[clickedpos].ownedby = board.tiles[16].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[18].ownedby) then
result := True;
      21 : if (board.tiles[clickedpos].ownedby = board.tiles[23].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[24].ownedby) then
result := True;
      23 : if (board.tiles[clickedpos].ownedby = board.tiles[24].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[21].ownedby) then
result := True;
      24 : if (board.tiles[clickedpos].ownedby = board.tiles[21].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[23].ownedby) then
result := True;
      26 : if (board.tiles[clickedpos].ownedby = board.tiles[27].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[29].ownedby) then
result := True;
      27 : if (board.tiles[clickedpos].ownedby = board.tiles[29].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[26].ownedby) then
result := True;
      29 : if (board.tiles[clickedpos].ownedby = board.tiles[26].ownedby)
and
```

```
        (board.tiles[clickedpos].ownedby = board.tiles[27].ownedby) then
result := True;
      31 : if (board.tiles[clickedpos].ownedby = board.tiles[32].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[34].ownedby) then
result := True;
      32 : if (board.tiles[clickedpos].ownedby = board.tiles[34].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[31].ownedby) then
result := True;
      34 : if (board.tiles[clickedpos].ownedby = board.tiles[31].ownedby)
and
        (board.tiles[clickedpos].ownedby = board.tiles[32].ownedby) then
result := True;
      37 : if board.tiles[clickedpos].ownedby = board.tiles[39].ownedby
then result := True;
      39 : if board.tiles[clickedpos].ownedby = board.tiles[37].ownedby
then result := True;
    else result := False;
    end;
  end
  else result := False;
end;

//Creates an array of TShapes at runtime from the elements on the form
procedure TGameForm.PrepareTileShapeArray;
var
  i: Integer;
begin
  for i := 0 to 27 do
  begin
    propertyshapes[propertyids[i]] := FindComponent('tile' +
inttostr(propertyids[i]))
    as TShape;
    propertyshapes[propertyids[i]].Hide;
  end;
end;

//This allows the button to behave as a toggle for buying the properties
procedure TGameForm.PropertiesBClick(Sender: TObject);
begin
  if buyinghouses = False then
  begin
    buyinghouses := True;
    propertiesb.Caption := 'Stop buying';
    RollDiceB.enabled := False;
    TradeB.Enabled := False;
    EndTurnB.Enabled := False;
    MortgageB.Enabled := False;
    Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
  end
  else
  begin
    buyinghouses := False;
    propertiesb.Caption := 'Get Houses/Hotels';
    RollDiceB.enabled := True;
    TradeB.Enabled := True;
    MortgageB.Enabled := True;
  end;
end;
```

```
//This allows the button to behave as a toggle for mortgaging properties
procedure TGameForm.MortgageBClick(Sender: TObject);
begin
  if mortgaging = False then
  begin
    mortgageb.Caption := 'End mortgaging';
    mortgaging := True;
    RollDiceB.enabled := False;
    TradeB.Enabled := False;
    EndTurnB.Enabled := False;
    propertiesb.Enabled := False;
  end
  else
  begin
    mortgaging := False;
    Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);
    mortgageb.caption := 'Mortgage tiles';
    RollDiceB.enabled := True;
    TradeB.Enabled := True;
    outp.Lines.Add('');
    propertiesb.Enabled := True;
  end;
end;

//This procedure is called to set the counter in the correct place for
the user
procedure TGameForm.MoveCounter;
begin
  case CurrentPlayer of
    0: begin
      p1c.Left := scalefctr*players[CurrentPlayer].GetLeftPos;
      p1c.Top := scalefctr*players[CurrentPlayer].GetTopPos;
    end;
    1 : begin
      p2c.Left := scalefctr*players[CurrentPlayer].GetLeftPos;
      p2c.Top := scalefctr*players[CurrentPlayer].GetTopPos;
    end;
    2 : begin
      p3c.Left := scalefctr*players[CurrentPlayer].GetLeftPos;
      p3c.Top := scalefctr*players[CurrentPlayer].GetTopPos;
    end;
    3 : begin
      p4c.Left := scalefctr*players[CurrentPlayer].GetLeftPos;
      p4c.Top := scalefctr*players[CurrentPlayer].GetTopPos;
    end;
  end;
end;

//Called at the end of a complete turn to get the next player in the
circular queue
procedure TGameForm.nextplayer;
begin
  CurrentPlayer := (CurrentPlayer + 1) mod playernum;
  EndTurnB.Enabled := False;
  if players[CurrentPlayer].alive = False then nextplayer
  else
  begin
    Outp.Lines.Add('');
    Outp.Lines.Add(players[CurrentPlayer].name + '''s turn');
  end;
end;
```

```
//Called for human players only and is similar to computerturn but just
gives the user
//choices to make using buttons in the GUI
procedure TGameForm.NormalTurn(r1,r2:integer;b,g:boolean);
var
  tempcost, tempint, newpos :integer;
begin
  newpos := (players[CurrentPlayer].pos + r1 + r2) mod 40;
  players[CurrentPlayer].pos := newpos;
  if not((r1 = 0) and (r2 = 0)) then
  begin
    if b = false then
    begin
      RollLine(r1,r2);
      MoveCounter;
    end;
    Outp.Lines.Add(players[CurrentPlayer].name + ' has moved to ' +
    board.tiles[players[CurrentPlayer].pos].Name);
    if board.tiles[players[CurrentPlayer].pos].isproperty = False then
    begin
      Buy.Hide;
      DBuy.Hide;
    end;
    if g = true then passedgo;
  end;
  newpos := players[CurrentPlayer].pos;
  if board.tiles[newpos].IsProperty then
  begin
    if board.tiles[newpos].Owned = False then
    begin
      Buy.Show;
      DBuy.Show;
      CostLbl.Show;
      CostLbl.Caption := 'Cost: ' +
inttostr(board.tiles[players[CurrentPlayer].pos].cost);
    end
    else
    begin
      if board.tiles[newpos].OwnedBy = CurrentPlayer then
      begin
        outp.Lines.Add(players[CurrentPlayer].name + ' owns ' +
        board.tiles[players[CurrentPlayer].pos].name);
      end
      else
      begin
        if board.tiles[newpos].Mortgaged = True then
        begin
          Outp.Lines.Add(board.tiles[players[CurrentPlayer].pos].name +
          ' is mortgaged so no rent is paid');
        end
        else
        begin
          if (board.tiles[newpos].SpecialCase = True) then
          begin
            if (board.tiles[newpos].ID = 12) or (board.tiles[newpos].ID =
28) then
            begin
              if board.tiles[28].ownedby = board.tiles[12].ownedby then
tempcost :=
              10 * (r1+r2)
              else tempcost := 4 * (r1+r2);
              players[CurrentPlayer].DecreaseBalance(tempcost);
              players[board.tiles[players[CurrentPlayer].pos].ownedby]
              .IncreaseBalance(tempcost);
              Outp.Lines.Add(board.tiles[players[CurrentPlayer].pos].name
+' is owned by '
              +
players[board.tiles[players[CurrentPlayer].pos].ownedby].name +
              ' and is owed ' + inttostr(tempcost));
              Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);

Outp.Lines.Add(players[board.tiles[players[CurrentPlayer].pos]
              .ownedby].GetBalanceLine);
            end
            else
            begin
              tempcost := 0;
              tempint := GetTrainOwned(board.tiles[newpos].OwnedBy);
              if tempint = 1 then tempcost := 25;
              if tempint = 2 then tempcost := 50;
              if tempint = 3 then tempcost := 100;
              if tempint = 4 then tempcost := 200;
              players[CurrentPlayer].DecreaseBalance(tempcost);
              players[board.tiles[players[CurrentPlayer].pos].ownedby].
              IncreaseBalance(tempcost);
              Outp.Lines.Add(board.tiles[players[CurrentPlayer].pos].name
+
              ' is owned by ' +
players[board.tiles[players[CurrentPlayer].pos]
              .ownedby].name + ' and is owed ' + inttostr(tempcost));
              Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);

Outp.Lines.Add(players[board.tiles[players[CurrentPlayer].pos].ownedby]
              .GetBalanceLine);
            end;
          end
          else
          begin

players[CurrentPlayer].DecreaseBalance(board.tiles[newpos].GetRent);
            players[board.tiles[newpos].OwnedBy].IncreaseBalance(
            board.tiles[newpos].GetRent);
            Outp.Lines.Add(board.tiles[players[CurrentPlayer].pos].name +
' is owned by '
            +
players[board.tiles[players[CurrentPlayer].pos].ownedby].name +
            ' and is owed ' +
inttostr(board.tiles[players[CurrentPlayer].pos].getrent));
            Outp.Lines.Add(players[CurrentPlayer].GetBalanceLine);

Outp.Lines.Add(players[board.tiles[newpos].OwnedBy].GetBalanceLine);
          end;
        end;
      end;
    end;
  end
  else
  begin
    case board.tiles[players[CurrentPlayer].pos].id of
      20 : begin
        players[CurrentPlayer].IncreaseBalance(board.freeparkingbal);
        if board.freeparkingbal = 0 then
        begin
          Outp.Lines.Add('There was no money on the free parking tile');
        end
        else
        begin
          Outp.Lines.Add(players[CurrentPlayer].name + ' got ' +
          inttostr(board.freeparkingbal) + ' from free parking');
        end;
        board.freeparkingbal := 0;
      end;

      4 : begin
        Outp.Lines.Add(players[CurrentPlayer].name + ' paid $200 to free
parking');
        board.freeparkingbal := board.freeparkingbal + 200;
        players[CurrentPlayer].DecreaseBalance(200);
      end;

      38 : begin
        Outp.Lines.Add(players[CurrentPlayer].name + ' paid $100 to free
parking');
        board.freeparkingbal := board.freeparkingbal + 100;
        players[CurrentPlayer].DecreaseBalance(100);
      end;

      30 :begin
        players[CurrentPlayer].injail := True;
        players[CurrentPlayer].IncJailTurns;
        Outp.Lines.Add(players[CurrentPlayer].name + ' has entered
jail');
        players[CurrentPlayer].pos := 10;
        MoveCounter;
      end
    end;
    if (board.tiles[players[CurrentPlayer].pos].id = 7) or
       (board.tiles[players[CurrentPlayer].pos].id = 22) or
       (board.tiles[players[CurrentPlayer].pos].id = 36) then
    begin
      landedonchance;
    end;
    if (board.tiles[players[CurrentPlayer].pos].id = 2) or
       (board.tiles[players[CurrentPlayer].pos].id = 17) or
       (board.tiles[players[CurrentPlayer].pos].id = 33) then
    begin
      landedoncommunitychest;
    end
  end;
  CheckForDouble
end;

//Required to reject trade but as is called by computer turn, I turned it
into
//a procedure
procedure TGameForm.RejectBClick(Sender: TObject);
begin
  rejecttrade;
end;

//Just clears the trade as doesn't need to exchange anything
procedure TGameForm.rejecttrade;
```

```
var
  i : integer;
begin
  for i := 0 to Length(currenttrade.propertiesto)-1 do
  begin
    propertyshapes[currenttrade.propertiesto[i]].Brush.Color :=

players[0].GetColour(board.tiles[currenttrade.propertiesto[i]].ownedby)
  end;
  for i := 0 to Length(currenttrade.propertiesfrom)-1 do
  begin
    propertyshapes[currenttrade.propertiesfrom[i]].Brush.Color :=

players[0].GetColour(board.tiles[currenttrade.propertiesfrom[i]].ownedby)
  end;
  RollDiceB.Enabled := True;
  Outp.Show;
  AcceptB.Hide;
  REjectB.Hide;
  OutL.Hide;
  PendingTrade := False;
  outp.Lines.Add('Trade rejected');
  intrade := False;
  RollDiceB.Enabled := True;
  ClearCurrentTrade;
end;

//Sets the alive variable of that player to false as is required by other
procedures
procedure TGameForm.removePlayer(player: integer);
var
  i : integer;
begin
  players[player].alive := False;
  case player of
    0 : p1alive := False;
    1 : p2alive := False;
    2 : p3alive := False;
    3 : p4alive := False;
  end;
  if GetAlivePlayers > 1 then
  begin
    for i := 0 to 27 do
    begin
      if board.tiles[propertyids[i]].OwnedBy = player then
      begin
        board.tiles[propertyids[i]].owned := False;
        propertyshapes[propertyids[i]].Brush.Color := clWhite;
        propertyshapes[propertyids[i]].Hide;
      end;
    end;
  end;
end;

//Required to clicking the button but as a procedure for the procedure
computerturn
procedure TGameForm.RollDiceBClick(Sender: TObject);
begin
  RollDiceProcedure;
end;
```

```
//Rolls the dice and moves the player appropriately, also notifies the
user if they are
//in debt to deal with it
procedure TGameForm.RollDiceProcedure;
var
  randint1, randint2, randsum : integer;
  passgo : boolean;
begin
  GetScaleFactor;
  Buy.Hide;
  DBuy.Hide;
  CostLbl.Hide;
  RollDiceB.Enabled := False;
  randomize;
  double := false;
  randint1 := random(6) + 1;
  randint2 := random(6) + 1;
  if randint1 = randint2 then double := true;
  randsum := randint2 + randint2;
  players[CurrentPlayer].roll1 := randint1;
  players[CurrentPlayer].roll2 := randint2;
  passgo := False;
  if players[CurrentPlayer].pos + randsum > 39 then passgo := True;
  if players[CurrentPlayer].injail = False then
  begin
    NormalTurn(randint1,randint2,false,passgo);
  end
  else
  begin
    if players[CurrentPlayer].GetJailCards > 0 then
    begin
      UseJc.Show;
      DUseJc.Show;
    end
    else
    begin
      DontUseJailCard;
    end;
  end;
  propertiesb.Hide;
  if players[CurrentPlayer].balance >= 0 then
  begin
    TradeB.hide;
    MortgageB.hide;
  end
  else
  begin
    TradeB.Hide;
    outp.Lines.Add(players[CurrentPlayer].name + ' is currently in
debt');
    outp.Lines.Add(players[CurrentPlayer].name +
    ' needs to mortgage properties to a positive balance or will be
kicked out the game');
    MortgageB.Show;
  end;
end;

//As often required, I turned it into a procedure
procedure TGameForm.RollLine(x, y: integer);
begin
```

```
  Outp.Lines.Add(players[CurrentPlayer].name + ' rolled a ' + inttostr(x)
+ ' and a '
  + inttostr(y));
end;

//Creates the trade record when send button clicked
procedure TGameForm.SendBClick(Sender: TObject);
var
  i : integer;
begin
  Outp.Show;
  E1.Hide;
  E2.Hide;
  L1.Hide;
  L2.Hide;
  TileL.Hide;
  SendB.hide;
  lfrom.Hide;
  Outl.Hide;
  lto.Hide;
  lfrom.Caption := '';
  lto.Caption := '';
  if E1.Text = '' then CurrentTrade.moneyto := 0
  else Currenttrade.moneyto := strtoint(E1.Text);
  if E2.Text = '' then CurrentTrade.moneyfrom := 0
  else currenttrade.moneyfrom := strtoint(E2.Text);
  MortgageB.Enabled := True;
  RollDiceB.Enabled := True;
  propertiesb.Enabled := True;
  for i := 0 to Length(currenttrade.propertiesto)-1 do
  begin
    propertyshapes[currenttrade.propertiesto[i]].Brush.Color :=

players[0].GetColour(board.tiles[currenttrade.propertiesto[i]].ownedby)
  end;
  for i := 0 to Length(currenttrade.propertiesfrom)-1 do
  begin
    propertyshapes[currenttrade.propertiesfrom[i]].Brush.Color :=

players[0].GetColour(board.tiles[currenttrade.propertiesfrom[i]].ownedby)
  end;
end;

//Acts as a panel by hiding/showing components
procedure TGameForm.setGUIgame;
begin
  GridPanel.Hide;
  BoardIMG.Show;
  Outp.Show;
  RollDiceB.Show;
  EndTurnB.Enabled := False;
  EndTurnB.Show;
end;

//Acts as a panel by hiding/showing components
procedure TGameForm.setGUIPlayerChoices;
begin
  Start.Hide;
  PlayerNames.Show;
  PlayerType.Show;
  Player1.Show;
  Player2.Show;
  Player3.Hide;
  Player4.Hide;
  P1Type.Show;
  P2Type.Show;
  P3Type.Hide;
  P4Type.Hide;
  AddPlayer.Show;
  StartGame.Show;
  PlayerDiff.Show;
  P1Diff.Show;
  p2diff.show;
  DiffHelp.Show;
end;

//Acts as a panel by hiding/showing components
procedure TGameForm.setGUIstart;
begin
  PlayerNames.Hide;
  PlayerType.Hide;
  Player1.Hide;
  Player2.Hide;
  Player3.Hide;
  Player4.Hide;
  P1Type.Hide;
  P2Type.Hide;
  P3Type.Hide;
  P4Type.Hide;
  AddPlayer.Hide;
  StartGame.Hide;
  playerdiff.hide;
  p1diff.hide;
  p2diff.Hide;
  p3diff.Hide;
  p4diff.Hide;
  DiffHelp.Hide;
  Outp.Hide;
  EndTurnB.Hide;
  RollDiceB.Hide;
  Buy.Hide;
  DBuy.Hide;
  CostLbl.Hide;
  UseJc.Hide;
  DUseJC.Hide;
  p1c.Hide;
  p2c.Hide;
  p3c.Hide;
  p4c.Hide;
  p1b.Hide;
  TileL.Hide;
  E1.Hide;
  E2.Hide;
  p2b.Hide;
  p3b.Hide;
  p4b.Hide;
  TradeB.Hide;
  AcceptB.Hide;
  RejectB.Hide;
  OutL.Hide;
  L1.Hide;
  L2.Hide;
  SendB.Hide;
  lfrom.Show;
  lto.Show;
  Start.Show;
  MortgageB.Hide;
  propertiesb.Hide;
  NameL.Hide;
  CostL.Hide;
  PropertiesL.Hide;
  CloseB.Hide;
end;

//Acts as a panel by hiding/showing components
procedure TGameForm.SetupTrade;
begin
  L1.Show;
  L2.Show;
  OutL.Show;
  E1.Show;
  E2.Show;
  SendB.Show;
  TileL.Show;
  currenttrade.playerto := tradingplayer;
  p1b.Hide;
  p2b.Hide;
  p3b.Hide;
  p4b.Hide;
  intrade := True
end;

//Creates the objects
procedure TGameForm.StartClick(Sender: TObject);
begin
  board := TBoard.Create;
  ChanceStack := TNumberStack.Create;
  CChestStack := TNumberStack.Create;
  ChanceStack.Shuffle;
  CChestStack.Shuffle;
  playernum := 2;
  setGUIPlayerChoices;
end;

//Moves the counters to the first place and initiates the first turn
procedure TGameForm.StartGameClick(Sender: TObject);
begin
  GetSCaleFactor;
  StartPlayerGame;
  setGUIgame;
  Outp.Lines.Add(players[CurrentPlayer].name + '''s turn');
  p1c.Left := scalefctr*432;
  p1c.top := scalefctr*436;
  p2c.Left := scalefctr*432;
  p2c.Top := scalefctr*459;
  p3c.Left := scalefctr*455;
  p3c.top := scalefctr*436;
  p4c.Left := scalefctr*455;
  p4c.Top := scalefctr*459;
  ingame := True;
  allcomputerplayers := fallcomputerplayers;
  if P1Type.Checked = True then computerturn;
end;
```

```
//From the input into the get players process, creates the required
amount of players
// and of types determined by user
procedure TGameForm.StartPlayerGame;
var
  tempchar : char;
  tempname : string;
begin
  p1alive := True;
  p2alive := True;
  CurrentPlayer := 0;
  SetLength(players,playernum);
  if P1Diff.text = 'e' then tempchar := 'e'
  else
  begin
    if P1Diff.text = 'm' then tempchar := 'm'
    else
    begin
      if P1Diff.text = 'h' then tempchar := 'h'
      else tempchar := 'e'
    end;
  end;
  if Player1.Text = '' then
  begin
    tempname := 'Player1';
  end
  else
  begin
    tempname := Player1.Text;
  end;
  players[0] := TPlayer.create(tempname,P1Type.Checked,tempchar);

  if P2Diff.text = 'e' then tempchar := 'e'
  else
  begin
    if P2Diff.text = 'm' then tempchar := 'm'
    else
    begin
      if P2Diff.text = 'h' then tempchar := 'h'
      else tempchar := 'e'
    end;
  end;
  if Player2.Text = '' then
  begin
    tempname := 'Player2';
  end
  else
  begin
    tempname := Player2.Text;
  end;
  players[1] := TPlayer.create(tempname,P2Type.Checked,tempchar);

  p1c.Show;
  p2c.Show;
  p1b.caption := players[0].name;
  p2b.caption := players[1].name;
  if playernum = 3 then
  begin
    p3alive := True;
    if P3Diff.text = 'e' then tempchar := 'e'
```

```
      else
      begin
        if P3Diff.text = 'm' then tempchar := 'm'
        else
        begin
          if P3Diff.text = 'h' then tempchar := 'h'
          else tempchar := 'e'
        end;
      end;
      if Player3.Text = '' then
      begin
        tempname := 'Player3';
      end
      else
      begin
        tempname := Player3.Text;
      end;
      players[2] := TPlayer.create(tempname,P3Type.Checked,tempchar);
      p3c.Show;
      p3b.caption := players[2].name;
    end;
    if playernum = 4 then
    begin
      p3alive := True;
      p4alive := True;
      p3c.Show;
      p4c.Show;
      if P3Diff.text = 'e' then tempchar := 'e'
      else
      begin
        if P3Diff.text = 'm' then tempchar := 'm'
        else
        begin
          if P3Diff.text = 'h' then tempchar := 'h'
          else tempchar := 'e'
        end;
      end;
      if Player3.Text = '' then
      begin
        tempname := 'Player3';
      end
      else
      begin
        tempname := Player3.Text;
      end;
      players[2] := TPlayer.create(tempname,P3Type.Checked,tempchar);

      if P4Diff.text = 'e' then tempchar := 'e'
      else
      begin
        if P4Diff.text = 'm' then tempchar := 'm'
        else
        begin
          if P4Diff.text = 'h' then tempchar := 'h'
          else tempchar := 'e'
        end;
      end;
      if Player4.Text = '' then
      begin
        tempname := 'Player4';
      end
      else
      begin
        tempname := Player4.Text;
      end;
      players[3] := TPlayer.create(tempname,P4Type.Checked,tempchar);
      p3b.caption := players[2].name;
      p4b.caption := players[3].name;
    end;
  end;

  //The below short procedures are from clicking the images and tiles
  procedure TGameForm.tile11ContextPopup(Sender: TObject; MousePos: TPoint;
    var Handled: Boolean);
  begin
    tileshapeclick(11);
  end;

  procedure TGameForm.tile11imgMouseEnter(Sender: TObject);
  begin
    hoverover(11);
  end;

  procedure TGameForm.tile12ContextPopup(Sender: TObject; MousePos: TPoint;
    var Handled: Boolean);
  begin
    tileshapeclick(12);
  end;

  procedure TGameForm.tile12imgMouseEnter(Sender: TObject);
  begin
    hoverover(12);
  end;

  procedure TGameForm.tile13ContextPopup(Sender: TObject; MousePos: TPoint;
    var Handled: Boolean);
  begin
    tileshapeclick(13);
  end;

  procedure TGameForm.tile13imgMouseEnter(Sender: TObject);
  begin
    hoverover(13);
  end;

  procedure TGameForm.tile14ContextPopup(Sender: TObject; MousePos: TPoint;
    var Handled: Boolean);
  begin
    tileshapeclick(14);
  end;

  procedure TGameForm.tile15ContextPopup(Sender: TObject; MousePos: TPoint;
    var Handled: Boolean);
  begin
    tileshapeclick(15);
  end;

  procedure TGameForm.tile15imgMouseEnter(Sender: TObject);
  begin
    hoverover(15);
  end;
```

```
procedure TGameForm.tile16ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(16);
end;

procedure TGameForm.tile16imgMouseEnter(Sender: TObject);
begin
  hoverover(16);
end;

procedure TGameForm.tile18ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(18);
end;

procedure TGameForm.tile18imgMouseEnter(Sender: TObject);
begin
  hoverover(18);
end;

procedure TGameForm.tile19ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(19);
end;

procedure TGameForm.tile19imgMouseEnter(Sender: TObject);
begin
  hoverover(19);
end;

procedure TGameForm.tile1ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(1);
end;

procedure TGameForm.tile1imgMouseEnter(Sender: TObject);
begin
  hoverover(1);
end;

procedure TGameForm.tile21ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(21);
end;

procedure TGameForm.tile21imgMouseEnter(Sender: TObject);
begin
  hoverover(21);
end;

procedure TGameForm.tile23ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(23);
end;
```

```
procedure TGameForm.tile23imgMouseEnter(Sender: TObject);
begin
  hoverover(23);
end;

procedure TGameForm.tile24ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(24);
end;

procedure TGameForm.tile24imgMouseEnter(Sender: TObject);
begin
  hoverover(24);
end;

procedure TGameForm.tile25ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(25);
end;

procedure TGameForm.tile25imgMouseEnter(Sender: TObject);
begin
  hoverover(25);
end;

procedure TGameForm.tile26ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(26);
end;

procedure TGameForm.tile26imgMouseEnter(Sender: TObject);
begin
  hoverover(26);
end;

procedure TGameForm.tile27ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(27);
end;

procedure TGameForm.tile27imgMouseEnter(Sender: TObject);
begin
  hoverover(27);
end;

procedure TGameForm.tile28ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(28);
end;

procedure TGameForm.tile28imgMouseEnter(Sender: TObject);
begin
  hoverover(28);
end;

procedure TGameForm.tile29ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(29);
end;

procedure TGameForm.tile29imgMouseEnter(Sender: TObject);
begin
  hoverover(29);
end;

procedure TGameForm.tile31ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(31);
end;

procedure TGameForm.tile31imgMouseEnter(Sender: TObject);
begin
  hoverover(31);
end;

procedure TGameForm.tile32ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(32);
end;

procedure TGameForm.tile32imgMouseEnter(Sender: TObject);
begin
  hoverover(32);
end;

procedure TGameForm.tile34ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(34);
end;

procedure TGameForm.tile34imgMouseEnter(Sender: TObject);
begin
  hoverover(34);
end;

procedure TGameForm.tile35ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(35);
end;

procedure TGameForm.tile35imgMouseEnter(Sender: TObject);
begin
  hoverover(35);
end;

procedure TGameForm.tile37ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(37);
end;

procedure TGameForm.tile37imgMouseEnter(Sender: TObject);
```

```
begin
  hoverover(37);
end;

procedure TGameForm.tile39ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(39);
end;

procedure TGameForm.tile39imgMouseEnter(Sender: TObject);
begin
  hoverover(39);
end;

procedure TGameForm.tile3ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(3);
end;

procedure TGameForm.tile5ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(5);
end;

procedure TGameForm.tile5imgMouseEnter(Sender: TObject);
begin
  hoverover(5);
end;

procedure TGameForm.tile6ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(6);
end;

procedure TGameForm.tile6imgMouseEnter(Sender: TObject);
begin
  hoverover(6);
end;

procedure TGameForm.tile8ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(8);
end;

procedure TGameForm.tile8imgMouseEnter(Sender: TObject);
begin
  hoverover(8);
end;

procedure TGameForm.tile9ContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
begin
  tileshapeclick(9);
end;

procedure TGameForm.tile9imgMouseEnter(Sender: TObject);
```

```
begin
  hoverover(9);
end;

//Determines what action to perform from the global variables when the
TShapes are clicked
procedure TGameForm.tileshapeclick(p: integer);
begin
  if intrade = True then
  begin
    if (board.tiles[p].ownedby = CurrentPlayer) and
       (arraycontains(currenttrade.propertiesto,p) = False) then
    begin
      SetLength(currenttrade.propertiesto,
Length(currenttrade.propertiesto)+1);
      currenttrade.propertiesto[high(currenttrade.propertiesto)] := p;
      propertyshapes[p].Brush.Color := clBlack;
      lfrom.Show;
      lto.Caption := lto.Caption + inttostr(p) + ', ';
    end
    else
    begin
      if (board.tiles[p].ownedby = tradingplayer) and
         (arraycontains(currenttrade.propertiesfrom,p) = False) then
      begin
        SetLength(currenttrade.propertiesfrom,
Length(currenttrade.propertiesfrom)+1);
        currenttrade.propertiesfrom[high(currenttrade.propertiesfrom)] :=
p;
        propertyshapes[p].Brush.Color := clBlack;
        lto.Show;
        lfrom.Caption := lfrom.Caption + inttostr(p) + ', ';
      end;
    end;
  end
  else
  begin
    if mortgaging = True then
    begin
      if (board.tiles[p].ownedby = CurrentPlayer) and
         (board.tiles[p].mortgaged = False) then
      begin
        propertyshapes[p].Brush.Color :=
        players[CurrentPlayer].GetMortgageColour(CurrentPlayer);
        players[CurrentPlayer].balance :=
        players[CurrentPlayer].balance + board.tiles[p].mortgageprice;
        board.tiles[p].mortgaged := True;
        outp.Lines.Add(board.tiles[p].name + ' mortgaged');
      end
      else
      begin
        if board.tiles[p].ownedby = CurrentPlayer then
        begin
          players[CurrentPlayer].balance :=
players[CurrentPlayer].balance -
          board.tiles[p].mortgageprice;
          board.tiles[p].mortgaged := False;
          outp.Lines.Add(board.tiles[p].name + ' unmortgaged');
        end;
      end;
    end
    else
    begin
      if buyinghouses = True then
      begin
        if (playerownrange(CurrentPlayer,p) = true) and
           (players[CurrentPlayer].balance >
board.tiles[p].buildingprice) and
           (board.tiles[p].Getproperties < 5) then
        begin
          board.tiles[p].IncProperties;
          players[CurrentPlayer].balance :=
players[CurrentPlayer].balance -
          board.tiles[p].buildingprice;
          outp.Lines.Add(players[CurrentPlayer].name + ' purchased a
property on ' +
          board.tiles[p].name + ' for ' +
inttostr(board.tiles[p].buildingprice));
        end
        else
        begin
          outp.Lines.Add('You could not purchase a property on this
tile');
        end;
      end;
    end;
  end;
end;

//Sets up the buttons for the correct number of players and hides the
current player's
//button
procedure TGameForm.TradeBClick(Sender: TObject);
begin
  pendingtrade := True;
  currenttrade.playerfrom := CurrentPlayer;
  outp.Hide;
  if p1alive = True then p1b.Show;
  if p2alive = True then p2b.Show;
  if p3alive = True then p3b.Show;
  if p4alive = True then p4b.Show;
  case CurrentPlayer of
    0 : p1b.Hide;
    1 : p2b.Hide;
    2 : p3b.Hide;
    3 : p4b.Hide;
  end;

  EndTurnB.Enabled := False;
  TradeB.Hide;
  MortgageB.Enabled := False;
  propertiesb.Enabled := False;
  RollDiceB.Enabled := False;
end;

//Moves the player out of jail and calls the procedure to start their
turn
procedure TGameForm.UseJailCard;
begin
  Outp.Lines.Add(players[CurrentPlayer].name + ' has used their get out
of jail free card');
  Outp.Lines.Add(players[CurrentPlayer].name + ' rolled a ' +
  inttostr(players[CurrentPlayer].roll1) + ' and a ' +
inttostr(players[CurrentPlayer].roll2));
  players[CurrentPlayer].injail := False;
  players[CurrentPlayer].Resetjailturns;

NormalTurn(players[CurrentPlayer].roll1,players[CurrentPlayer].roll2,true
,false);
  MoveCounter;
end;


//Calls the correct procedure to remove the player from jail and hide the
buttons no
//longer required
procedure TGameForm.UseJCClick(Sender: TObject);
begin
  UseJailCard;
  UseJC.Hide;
  DUseJC.Hide;
end;

end.
```