|  |
| --- |
| BHASVIC Computer Science |
| **Animalopoly** |
|  |

|  |
| --- |
| Charlie Jones  [Date] |

Contents

[Analysis 2](#_Toc200354882)

[What goes in this section? 2](#_Toc200354883)

[Design 7](#_Toc200354884)

[What goes in this section? 7](#_Toc200354885)

[Checklist 14](#_Toc200354886)

[Development 15](#_Toc200354887)

[What goes in this section? 15](#_Toc200354888)

[Checklist 17](#_Toc200354889)

[Evaluation 18](#_Toc200354890)

[What goes in this section? 18](#_Toc200354891)

[Checklist 20](#_Toc200354892)

[References 20](#_Toc200354893)

[References Explanation 20](#_Toc200354894)

[Reference List 23](#_Toc200354895)

# Analysis

## What goes in this section?

In your actual programming project, the analysis section is a significant piece of work in which you justify why the program/product you have decided to create should be created. You’ll look at existing solutions and competitors to examine what their strengths and weaknesses are and what opportunities and threats this creates for your project. You’ll also use questionnaires and interviews with your potential stakeholders/target audience to understand their needs.

Once you’ve gathered this information, you’ll create success criteria that will inform your development and design tests that will ensure that you can evidence your progress towards meeting these success criteria.

You’ll complete the Analysis Phase of your programming project before the Summer Break so that you can then focus on design in September.

For Animalopoly, we’ve given you the success criteria to save time! The only thing you need to do is to prioritize them based on their importance:

* Green = Most Important, do this first!
* Yellow
* Red = Least Important, do this last!

Success Criteria

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Success Criteria** | **Justification** | **Importance** |
| 1. Dice | | |  |
| 1.1 | Two Dice Implementation: Implement two dice that the player can roll, with each die generating a random number between 1 and 6. | Implementing two dice is fundamental to the game mechanics, as it determines how far players move on the board. This randomness adds excitement and variability to each turn, making the game more engaging. | Green |
| 1.2 | Dice Roll Outcome: Display the total number of spaces the player should move based on the sum of the dice roll. | Displaying the number of spaces the player should move based on the dice roll ensures players understand their movement clearly. This transparency is crucial for maintaining the flow of the game and preventing confusion. | Green |
| 1.3 | Double Roll Detection: Detect and confirm when two of the same dice have been rolled, triggering a special event. | Confirming when two of the same dice have been rolled is important because it triggers special events, such as drawing a card. This feature adds an extra layer of strategy and unpredictability, enhancing the overall gameplay experience. | Red |
| 2. Board | | |  |
| 2.1 | Board Display: Create and display a text-based board with 26 spaces to the user. | Creating a text-based board with 26 spaces and displaying it to the user provides a visual representation of the game state. This helps players keep track of their positions and the overall progress of the game, making it more immersive. | Green |
| 2.2 | Player Location Storage: Store and update the location of each player on the board after every move. | Storing the location of each player on the board is necessary for tracking player positions and game progress. It ensures that the game can accurately reflect each player's movements and interactions with the board spaces. | Green |
| 2.3 | Current Player Position: Display the current position of each player on the board at all times. | Showing on the board where each player currently is helps players understand their current status in the game. This visibility is essential for strategic planning and decision-making during their turns. | Yellow |
| 2.4 | Space Instructions: Provide specific instructions to the player based on the space they land on. | Informing the player what to do when they land on a specific space ensures players know the rules and actions required for each space. This clarity is vital for smooth gameplay and helps prevent misunderstandings. | green |
| 2.5 | Start Space Rewards: Automatically give the player £500 when they pass start and £1000 when they land on start. | Giving the player money when they pass (£500) or land on (£1000) start adds strategic elements and rewards for player movement. These incentives encourage players to keep moving and add excitement to reaching key points on the board. | yellow |
| 2.6 | Miss a Turn: Have the player miss their next turn when they land on the "miss a turn" space. | Having the player miss a turn when they land on the appropriate square introduces penalties and adds complexity to the game. This feature can change the dynamics of the game and requires players to adapt their strategies. | Red |
| 3. Player | | |  |
| 3.1 | Playing Piece Selection: Allow players to select a text-based playing piece (e.g., \*, @, ?). | Allowing players to pick a text-based playing piece personalizes the game experience for each player. This customization makes the game more enjoyable and helps players feel more connected to their in-game character. | Red |
| 3.2 | Player Name Storage: Store and display the player's name at the start of their turn. | Storing and announcing the player's name on their turn enhances player identification and engagement. It adds a personal touch to the game, making each turn feel more significant and interactive. | Yellow |
| 3.3 | Money Management: Track and update the amount of money each player has throughout the game. | Storing and updating the amount of money each player has is central to the game mechanics and determining the winner. This feature is crucial for tracking financial transactions and ensuring the game progresses correctly. | Green |
| 3.4 | Bankruptcy Notification: Notify players when they run out of money and are eliminated from the game. | Informing players when they run out of money indicates game status and player elimination. This notification is essential for maintaining the competitive aspect of the game and ensuring players are aware of their standing. | Green |
| 3.5 | Winning Condition: Announce the winner when they are the last player remaining with money. | Announcing the winner when they are the last player with money defines the end goal and victory condition of the game. This clear objective helps players understand what they are striving for and adds excitement to the competition. | Green |
| 4. Animals | | |  |
| 4.1 | Animal Assignment: Assign a unique animal to each space on the board. | Assigning an animal to each space on the board adds thematic elements and variety to the game. This feature enriches the gameplay experience by introducing unique attributes and interactions for each space. | Green |
| 4.2 | Animal Information Storage: Store detailed information for each animal, including name/species, level, cost to stop/visit, cost to buy, and owner. | Storing information for each animal (name/species, level, cost to stop/visit, cost to buy, owner) provides detailed attributes for game interactions. This data is essential for managing ownership, costs, and upgrades within the game. | Green |

|  |  |  |  |
| --- | --- | --- | --- |
| 4.3 | Animal Purchase: Allow players to purchase animals at their set cost when they land on an unowned animal space. | Allowing players to purchase animals at their set cost when they have no owner introduces ownership and investment mechanics. This feature adds strategic depth, as players must decide when to invest in animals to gain advantages. | Green |
| 4.4 | Animal Upgrade: Enable players to upgrade animals to increase their level at the same cost as the purchase price. | Allowing players to upgrade animals to increase their level at the same cost as purchase adds strategic depth and progression. This feature encourages players to enhance their assets, making the game more dynamic and competitive. | Green |
| 4.5 | Stop Cost Notification: Inform players of the cost to stop on an owned animal space. | Informing players of the cost to stop on an owned animal space ensures players are aware of financial penalties. This transparency is crucial for strategic planning and helps players make informed decisions during their turns. | Green |
| 5. Cards | | |  |
| 5.1 | Card Pack Creation: Create a pack of 20 cards with various scenarios where players gain or lose money. | Creating a pack of 20 cards with various scenarios where players gain/lose money adds random events and unpredictability to the game. This feature keeps the gameplay exciting and introduces new challenges and opportunities. | Yellow |
| 5.2 | Random Card Draw: Implement a feature to randomly draw a card from the pack when needed. | Implementing the ability for a random card to be drawn when needed enhances game dynamics and player decision-making. This randomness adds an element of surprise and requires players to adapt to changing circumstances. | Yellow |

# Design

## What goes in this section?

You need to include any whole project design/planning that your group did as well as the design/planning that you did for your objects – you do not need to include the design/planning that other members of your group did for their objects.

This should include:

* class diagrams
* flowcharts
* pseudocode
* data dictionaries

It should also include your test planning – remember that you need to have iterative testing, system testing and useability testing.

For each of these you need to plan **exactly** what test data should be used and **exactly** what output/result you would expect if that data were input.

**Charlie:**

Player:

• Allow them to pick a text-based playing piece (e.g. \*, |, ?)

• Store their name and announce it on their turn

• Store how much money they have

• Add to or decrease their money as they play

• Tell them that have lost if they run out of money

• Tell them they have won if they are the last player in the game

Cards:

• A pack of 20 cards with various scenarios where players gain / lose money

 • The ability for a random card to be drawn when needed

**Solomon:**

Board:

•Have a board with the above spaces which is displayed to the user (via text)

• Store the location of each user

• Show on the board where each player currently is

• Tell the user what to do when they land on that space

• Give the player money when they pass (£500) or land on (£1000) start

• Tell the player to miss a go when they land on the appropriate square

Main

**Taran:**

Animals:

• Have an animal assigned to each space on the board

• Store the above information for an animal

• Allow an animal to be purchased / owned by a particular player at it’s given              cost when it currently has no owner

•Allow an animal to be upgraded to increase in level at the same cost as it’s                                      purchased for

• Tell a player how much they need to pay when stopped on

Dice:

• Have two dice that the player can roll

• Tell the player how many spaces to move by

• Confirm when two of the same dice have been rolled

A diagram of a deck

AI-generated content may be incorrect.

This is my structure diagram for how the cards class will work.

I later decided that it would be easier to create a deck class that would contain a list for the 20 chards and logic for drawing a random card from the pack. This is because it will create a more modular approach that will enable more flexibility later.

**Deck Pseudocode**

Class deck

private cards

private templates [,[],[],[],[],[],[],[],[],[],[],[],[]]

public procedure new()

for 0 to 19

cards[i] = Card.new(templates[i,0],templates[i,1])

endfor

endprocedure

public procedure drawCard(player)

number = random(1,20) <- (SaveMyExams, 2024)

cards[number].useCard(player)

endprocedure

**Cards Pseudocode**

class Card

private description // description of card

private value // number of spaces or amount of money

public procedure new(descriptionInput, actionInput, valueInput)

description = descriptionInput

action = actionInput

value = valueInput

endprocedure

public procedure useCard(player)

player.updateMoney(value)

endprocedure

A screenshot of a computer program

AI-generated content may be incorrect.

This was my class diagram for player and cards.

I later added more attributes like isBankrupt and position. This is because they where required for other parts of the program to run properly.

A diagram of a computer game

AI-generated content may be incorrect.

This is my structure diagram for how the player class will work. The main features are that you can setup the player, add or decrease money and manage their turn.

When it came to writing the program, I took most of the logic out like input name because we decided we would have most of the logic within the game class.

**Player Pseudocode**

class Player

private name

private money

private symbol

private isBankrupt

private position

public procedure new(nameInput, symbolInput)

name = nameInput

symbol = symbolInput

money = 1000

Bankrupt = false

endprocedure

public procedure setPosition(pos)

position = pos

endprocedure

public procedure getPostition()

return position

endprocedure

public procedure updateMoney(value)

money = money + value

if money <= 0 then

Bankrupt = true

endif

endprocedure

public function getName()

return name

endfunction

public function getMoney()

return money

endfunction

public function getSymbol()

return symbol

endfunction

public function getIsBankrupt()

return Bankrupt

endfunction

endclass

A black and white table with white text

AI-generated content may be incorrect.

This is the data dictionary that stores any variables that will be used in the program for player and card.

A diagram of a card

AI-generated content may be incorrect.

This is my flowchart for card. The main logic is to randomly generate a number.

## Checklist

This is the top band marking criteria for the design section of your programming project – you should aim to tick off every statement on this list. You should also be clear about what evidence you have included for each statement and what page of your document this evidence is on.

|  |  |  |  |
| --- | --- | --- | --- |
| **I have…** | **y/n** | **As evidence, I have included…** | **On page…** |
| Broken the problem down systematically into a series of smaller problems suitable for computational solutions, explaining and justifying the process |  |  |  |
| Defined in detail the structure of the solution to be developed. |  |  |  |
| Described the solution fully using appropriate and accurate algorithms justifying how these algorithms form a complete solution to the problem. |  |  |  |
| Described, justifying choices made, the usability features to be included in the solution. |  |  |  |
| Identified and justified the key variables / data structures / classes (as appropriate to the proposed solution) justifying and explaining any necessary validation. |  |  |  |
| Identified and justified the test data to be used during the iterative development of the solution. |  |  |  |
| Identified and justified any further data to be used in the post development phase. |  |  |  |

# Development

## What goes in this section?

This section is about explaining the process you went through to develop the program – you should be taking regular screenshots of your **fully annotated** code and then adding this to your development section with information about what you did and why you did it.

You also need to show evidence that you were regularly running and testing your programming – you should have an iterative test plan to follow but this will/should change and evolve as you are programming, and you start to encounter issues you didn’t predict.

It is **super** important that you include evidence of any failed tests, what you changed to make your code work and **why** these changes were important/the best solution for that situation. If you try to run your code and it doesn’t run then take a screenshot, add it to your development section, make the change, explain how you solved the problem, then add a screenshot of the fixed code.

A computer screen with text

AI-generated content may be incorrect.

This was the 2d array of card descriptions and money values. I had to put both the description and value as a string because java wouldn’t let me put an integer and a string into the same array. There is likely a more effective way to do this where you can store both int and string in a list however I wasn’t sure, so I stuck with what I knew would work and then later converted to string value back into an integer.



A screenshot of a computer

AI-generated content may be incorrect.

Here I had an error where I incorrectly typed the code for indexing a 2d array. To fix this I changed the code to: cards.add(new Cards(deck[i][0],Integer.parseInt(deck[i][1]))). This was the best change for the situation because it’s the only way to do it without using 2 1d arrays and writing it like deck1[i], deck2[0] but this would require unnecessary code.



A screenshot of a computer

AI-generated content may be incorrect.A black screen with white text

AI-generated content may be incorrect.

Here I had a problem with the random number generation where it was creating a number out of the size of the list. I found it was because the range of the random number was 0-20 while the list was only 0-19. To fix this I changed it to be 20. This is best fit for the solution because I could have a variable that can change depending on the number of cards I would like in a deck but because I’m not intending on changing the deck size there is no need.



**Final Code**

A screen shot of a computer program

AI-generated content may be incorrect.This is the final code for the Card class

This is the final code for the deck class A computer screen shot of a program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

## This is the code for Player class

## Checklist

This is the top band marking criteria for the development section of your programming project – you should aim to tick off every statement on this list. You should also be clear about what evidence you have included for each statement and what page of your document this evidence is on.

|  |  |  |  |
| --- | --- | --- | --- |
| **I have…** | **y/n** | **As evidence, I have included…** | **On page…** |
| Provided evidence of each stage of the iterative development process for a coded solution, relating this to the breakdown of the problem from the analysis stage and explaining what I did and justifying why |  |  |  |
| Provided evidence of prototype versions of my solution for each stage of the process. |  |  |  |
| Ensured the solution is well structured and modular in nature. |  |  |  |
| Annotated the code to aid future maintenance of the system. |  |  |  |
| Ensured all variables and structures are appropriately named. |  |  |  |
| Included evidence of validation for all key elements of the solution. |  |  |  |
| Shown my review at all key stages in the process. |  |  |  |
| Provided evidence of testing at each stage of the iterative development process. |  |  |  |
| Provided evidence of any failed tests and the remedial actions taken with full justification for any actions taken. |  |  |  |

# Evaluation

## What goes in this section?

This section is about showcasing your system and usability testing results – it’s ok if your program doesn’t fully work (it probably won’t!).

Wherever possible the evidence in your testing table should be a screen recording of the program in action. The videos should be stored in the same folder as this document.

If it doesn’t fully work, then you actually have more to write about and show your evaluation skills – make sure that you comment on why you were unable to get the program fully working and how you could meet the missing criteria through further development.

Make sure to also think about whether the code you have created would be easy, difficult or somewhere in the middle for somebody else to maintain – have you coded in a weird way? Could you have been more modular? Used more inheritance?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test description | Expected result | Actual result | Evidence | Pass or fail |
| Does the program correctly convert a sting to and integer | The player’s money is correctly changed in relation to the amount of money in the original string | The money is correctly adjusted. | A black background with white text  AI-generated content may be incorrect. | Pass |
| The card description is correctly outputted | The description is printed on screen | The description was printed out correctly |  | Pass |
| The player class cam be correctly created with a symbol and a name | There won’t be an error when a new player class is made | There was no error |  | Pass |

A screenshot of a computer

AI-generated content may be incorrect.

This is what the program looks like when its run. I’m very happy with how it turned out. It looks very good visually and functions very well.

One of the issues we found with the program was that the players were gaining money faster than they could spend it. This can be fixed by just reducing the amount from doing a lap around the board and also increase the amount you pay for landing on another players animal.

I think that the code we have written would be easy for someone to maintain because we made sure to write comments throughout the program, the make it more understandable for someone who is new to the program.

## Checklist

This is the top band marking criteria for the evaluation section of your programming project – you should aim to tick off every statement on this list. You should also be clear about what evidence you have included for each statement and what page of your document this evidence is on.

|  |  |  |  |
| --- | --- | --- | --- |
| **I have…** | **y/n** | **As evidence, I have included…** | **On page…** |
| Provided annotated evidence of post development testing for function and robustness |  |  |  |
| Provided annotated evidence for usability testing. |  |  |  |
| Used the test evidence to cross reference with the success criteria to evaluate the solution explain how the evidence shows that the criteria has been fully, partially or not met in each case. |  |  |  |
| Provided comments on how any partially or unmet criteria could be addressed in further development. |  |  |  |
| Provided evidence of the usability features justifying their success, partial success or failure as effective usability features. |  |  |  |
| Provided comments on how any issues with partially or unmet usability features could be addressed in further development. |  |  |  |
| Considered maintenance issues and limitations of the solution. |  |  |  |
| Described how the program could be developed to deal with limitations and potential improvements / changes. |  |  |  |
| There is a well-developed line of reasoning which is clear and logically structured. The information presented is relevant and substantiated. |  |  |  |

# References

# Reference List

Anon., 2024. [Online]   
Available at: https://www.w3schools.com/java/java\_arraylist.asp  
[Accessed 04 June 2025].

geeks For Geeks, 2024. *Geeks For Geeks - How to convert a string to an int in java.* [Online]   
Available at: https://www.geeksforgeeks.org/how-to-convert-a-string-to-an-int-in-java/  
[Accessed 04 June 2025].

SaveMyExams, 2024. *SaveMyExams - Random Number Generation.* [Online]   
Available at: https://www.savemyexams.com/gcse/computer-science/aqa/20/revision-notes/2-programming/random-number-generation/random-number-generation/  
[Accessed 27 May 2025].

W3Schools, 2024. *W3Schools - Java how to random number.* [Online]   
Available at: https://www.w3schools.com/java/java\_howto\_random\_number.asp  
[Accessed 04 June 2025].

W3Schools, 2025. *W3Schools - Java Strings.* [Online]   
Available at: https://www.w3schools.com/java/java\_strings.asp  
[Accessed 3 June 2025].