Comprehensive Exercise Report

Team LAZREG-POILVE-LASNIER

240AEB015 LASNIER Aurélien

240AEB032 LAZREG Mehdi

240AEB011 POILVE Alexandre

[Requirements/Analysis 2](#_heading=)

[Journal 2](#_heading=)

[Software Requirements 3](#_heading=)

[Black-Box Testing 4](#_heading=)

[Journal 4](#_heading=)

[Black-box Test Cases 5](#_heading=)

[Design 6](#_heading=)

[Journal 6](#_heading=)

[Software Design 9](#_heading=)

[Implementation 11](#_heading=)

[Journal 11](#_heading=)

[Implementation Details 12](#_heading=)

[Testing 13](#_heading=)

[Journal 13](#_heading=)

[Testing Details 14](#_heading=)

[Presentation 15](#_heading=)

[Preparation 15](#_heading=)

# Requirements/Analysis

Week 2

## Journal

The following prompts are meant to aid your thought process as you complete the requirements/analysis portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

* After reading the client’s brief (possibly incomplete description), write one sentence that describes the project (expected software) and list the already known requirements.
  + <<The software will be a 2 player’s turn-based strategy video game where the first player who aligns 4 discs of his color in the grid wins.>>
    - <<Insert known requirements from client description, add more bullets as needed>>
    - 2 player’s game.
    - Create a grid.
    - Create 2 types of discs (red and yellow)
    - Each player gets 21 discs.
    - Player wins by aligning 4 discs of his color.
    - Selection of starting player
* After reading the client’s brief (possibly incomplete description), what questions do you have for the client? Are there any pieces that are unclear? After you have a list of questions, raise your hand and ask the client (your instructor) the questions; make sure to document his/her answers.
  + <<What size for the grid? Do players have a time limit on each round?>>
* Does the project cover topics you are unfamiliar with? If so, look up the topics and list your references.
  + <<No>>
* Describe the users of this software (e.g., small child, high school teacher who is taking attendance).
  + <<For kids ages 6 and up, so it must be easy to use.>>
* Describe how each user would interact with the software.
  + <<At each turn the user will have the option to choose a column to place his disc. >>
* What features must the software have? What should the users be able to do?
  + <<The software needs to detect the lowest empty location on the selected column.
  + It also needs to detect when 4 discs of the same color are aligned to stop the game and show the winner. Including when the disks are not on the same row or same columns.
  + It needs to ask both players if they want to start again.
  + Users should be able to quit if they want.>>
* Other notes:
  + <<Insert notes>>

## Software Requirements

<<Use your notes from above to complete this section of the formal documentation by writing a detailed description of the project, including a paragraph overview of the project followed by a list of requirements (see lecture for format of requirements). You may also choose to include user stories.>>

The main goal of our project is to develop a software for a client. Here, it will be a Connect 4 game. The concept is simple, it’s a 2 player’s turn based strategy game, where the goal is to connect 4 discs of the same color. To do that, each player throws their disc from the top of the same grid and then the disc goes down to the lowest position possible. This game is playable from 6 years old to 99. Since it’s originally a physical game, the objective here is to transform it into a digital game. Also, during the realization of the project, the client might add some options to it so it needs to be flexible.

To achieve this project, a list of requirement is necessary, first, the functional ones :

* First, it needs to be playable by the client on his own computer.
* Since it’s a 2 player game, the users need to be able to play their turn individually.
* Players should be able to choose who starts or be able to choose a random mode.
* The program must display a grid so the players know where they can play.
* 2 colors of discs must be created and must appear on the grid after being played.
* The software needs to know where the lowest position on each column is and also needs to know when 4 discs are aligned.
* The software should add an option to quit the game or restart a new one.

For the non-functional ones :

* It need to be accessible for young kids since it can be played at 6 years old
* The software must run fast, without lags or long loading time.
* It should be safe to play, meaning that the computer shouldn’t be damaged by the program.
* The game needs to be clear and well constructed.
* It also needs to respect the rules of the game described above.

# Black-Box Testing

Instructions: Week 4

## Journal

***Remember:*** Black box tests should only be based on your requirements and should work independent of design.

The following prompts are meant to aid your thought process as you complete the black box testing portion of this exercise. Please review your list of requirements and respond to each of the prompts below. Feel free to add additional notes.

* What does input for the software look like (e.g., what type of data, how many pieces of data)?
  + <<Player’s move which is a column number (integer) where the player want to put their pieces>>
* What does output for the software look like (e.g., what type of data, how many pieces of data)?
  + <<Updated game board after the moves of the players, represented like a grid or a matrix showing the current state of the game.>>
* What equivalence classes can the input be broken into?
  + <<We can have : Valid Column = numbers of available columns from 1 to 7 on the game

Or Invalid Column = numbers of columns less than 1 or greater than the maximum, column already full, the player cannot add any more pieces.>>

* What boundary values exist for the input?
  + <<Minimum valid input = 1 / column number

Maximum valid input = 7 / number of columns equal to the maximum number of columns on the game

Invalid Input limit = 0 (less than 1 minimum) and +7 (more than the maximum).>>

* Are there other cases that must be tested to test all requirements?
  + <<1.Situation of initialization game : Where we start or start the game again (restart), and all the columns are empty and there is no message from the previous game, and players have all their coins available + first player selection

2. Situation of Victory conditions : When the first to succeed in aligning (horizontally, vertically, or diagonally) 4 pieces in a row.

3. Situation of Full board : When all columns are full and no more moves can be made.

4. Situation of a player that wants to quit the game.

>>

* Other notes:
  + <<Consider cases where the player enters wrong types of values, and the fluidity of the game response.>>

## Black-box Test Cases

Use your notes from above to complete the black-box test plan section of the formal documentation by writing black box test cases (other than actual results since no program currently exists). Remember to test each equivalence class, boundary value, and requirement.

| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
| --- | --- | --- | --- |
| Software start | Verify that the software launch well so the client can play directly | No error at launch | as expected |
| Player’s input | Player need to enter an integer to select a column to place his coin | Coin is placed in the selected column | as expected |
| Verify player’s input | The software must verify if the player input is acceptable (e g the integer must be between 1 and 7) | Between 1 and 7 :  Spawn the coin in the selected column | as expected |
| Software’s output | The software must spawn a coin in the column with the same integer has the player input | Show the coin placed in the correct column | as expected |
| Initialization of the game | Verify if the playground is empty at the start of the game | Empty playground | as expected |
| Player selection | Verify if the players can choose who start the game | Possibility for the player to choose who start | Since the two players play on the same computer, they just choose who clicks first. |
| Victory | The software need to detect when 4 coin of the same color are align | Show message : “(red or yellow) is the winner” | as expected |
| Full board | The software must detect when the board is full and must restart the game | A draw message is shown and game ask to restart | as expected |
| Quit option | The software must stop if one of the player ask to quit the game via a button | The game stop | as expected |
| Restart option | The game must restart if one of the players or both ask to replay at the end. | The game restart and initialize again | as expected |

# Design

Instructions: Week 6

## Journal

***Remember:*** You still will not be writing code at this point in the process.

The following prompts are meant to aid your thought process as you complete the design portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

* List the nouns from your requirements/analysis documentation.
  + <<We have many difference nouns, here are the main ones:
* Players
* Turns
* Game board
* Discs
* Client
* Goal
* Colors
* Position
* Column
* Options
* Grid
* User
* Audience
* Mode
* Game
* Winner
* >>
* Which nouns potentially may represent a class in your design?
  + <<Here are the potential class in our design :
* Players
* Discs
* Board or Grid
* Position
* Color maybe
* Option
* Game>>
* Which nouns potentially may represent attributes/fields in your design? Also list the class each attribute/field would be a part of.
  + <<I will directly write the class with their attributes (not all come from the requirement part) :
* Game : Player 1, Player 2, Current player, Winner, Game status
* Player : Name, Color
* Disc : Color, Position
* Color : List of color
* Grid/Board : Sizes, Layout
* Position : row, column
* Option : Various >>
* Now that you have a list of possible classes, consider different design options (***lists of classes and attributes***) along with the pros and cons of each. We often do not come up with the best design on our first attempt. Also consider whether any needed classes are missing. These two design options should not be GUI vs. non-GUI; instead you need to include the classes and attributes for each design. Reminder: Each design must include at least two classes that define object types.
  + <<First design which is more object oriented :
* Game :

Attributes : Player 1, Player 2, Current player, Winner, Game status

* Player :

Attributes : Name, Color

* Disc :

Attributes : Color, Position

* Grid/Board :

Attributes : Sizes, Layout

* Position :

Attributes : row, column

For the pros, this design is pretty clear. Each entity is well separated from the others. Each attribute is encapsulated and it’s very easy to add or modify some features.

For the cons, it can be more complex to use because of the number of classes, we also need to be more careful on interactions between classes.

Second design :

* Game :

Attributes : Player 1, Player 2, Current player, Winner, Game status

Methods : Turn(), grid(), WinCheck()

* Grid :

Attributes : Sizes, Layout

* Colors :

Attributes : PlayersColors, DiscColors

For the pros, this design is more simplified, less complexity, it means that it is easier to understand and requires less implementations.

For the cons, there is a limited scalability and flexibility as well as a lack of separation which could cause some errors in the code.>>

* Which design do you plan to use? Explain why you have chosen this design.

We plan to use design 1 for many reasons :

* The ability to encapsulate every attributes in their classes, with the possibility to add more attributes or to do modification easily.
* The clear separation, while coding, we like to have our variables well separated to understand perfectly where each thing belongs to. It also makes the code more readable and understandable for someone outside the project.
* Also, the way the classes are done looks better to us in a more logical way.
* List the verbs from your requirements/analysis documentation.
  + <<Develop, Transform, Display, Choose, Play, Throws, Go down, Know, Add, Quit, Restart, Accessible, Run, Lag, Loading, Damage, Construct, Respect>>
* Which verbs potentially may represent a method in your design? Also list the class each method would be part of.
  + <<Here are the verbs that could be used as method in our design :
* Choose - Method: chooseStartingPlayer() Class : Game
* Display - Method: displayGrid() Class : Grid
* Add - Method: addPlayer() Class : Game or Player
* Restart - Method: restartGame() Class : Game
* Construct - Method: constructGame() Class : Game>>
* Other notes:
  + <<The Methods found upper might not be used in our game because the verbs we used in our requirements part weren’t really the most corresponding ones for methods design in our opinions.>>

## Software Design

<<Use your notes from above to complete this section of the formal documentation by planning the classes, methods, and fields that will be used in the software. Your design should include UML class diagrams along with method headers. ***Prior to starting the formal documentation, you should show your answers to the above prompts to your instructor.****>>*

For the software design, we decided to go with our first option, which is the object oriented one. The reasons are :

* The ability to encapsulate every attributes in their classes, with the possibility to add more attributes or to do modification easily.
* The clear separation, while coding we like to have our variables well separated to understand perfectly where each thing belongs to. It also makes the code more readable and understandable for someone outside the project.
* Also, the way the classes are done looks better to us in a more logical way.

We are going to use UML class diagrams but here is a short resume of our classes with attributes and methods :

* Game :

Attributes : Player 1, Player 2, Current player, Winner, Game status

Methodes: Game(),StartNewGame(),PlayerTurn(),WinCheck(),FullBoard(),Restart(),Display(),SwitchPlayer()

* Player :

Attributes : Name, Color

Method : Player()

* Disc :

Attributes : Color

Method : Disc()

* Grid/Board :

Attributes : Sizes, Layout

Method : Grid(), DisplayGrid()

* Position :

Attributes : row, column

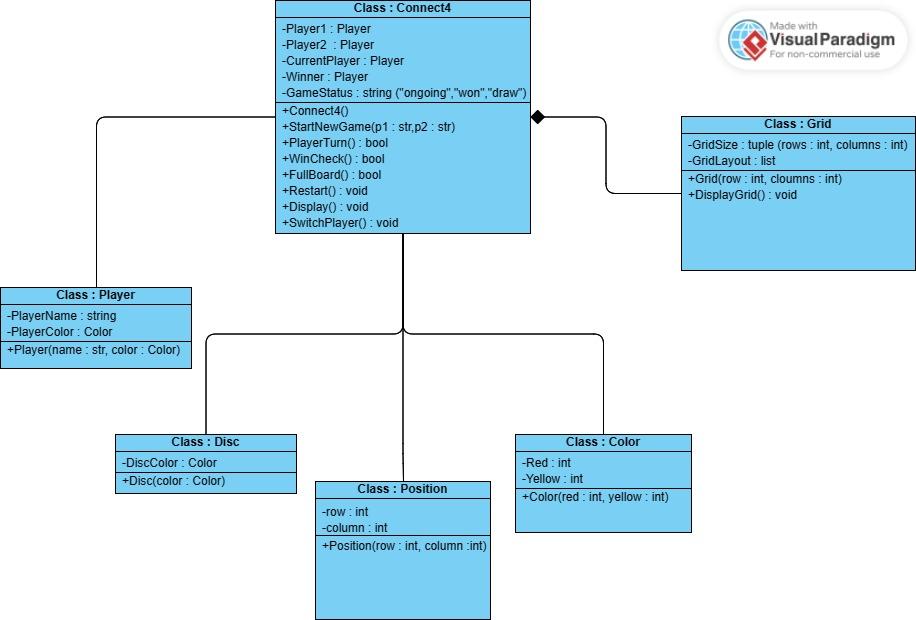
Metho : Position()

* Color :

Attributes : red,yellow

Method : Color()

Here is the UML design :



# Implementation

Instructions: Week 8

## Journal

The following prompts are meant to aid your thought process as you complete the implementation portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

* What programming concepts from the course will you need to implement your design? Briefly explain how each will be used during implementation.

<<To make this part, we used a kind of life cycle method, which means that we started by planning, then we started analyzing how we could write the code,and we finished with the implementation / testing. If we go deeper in the concept used in the code, here is an explanation :

First we use an **user input/output** to allow communication between the player and the script. The output is needed to let the player choose their moves. The output is used to display game’s information such as when one of the two players or when it’s a draw.

The **object oriented programming** (OOP) is used to implement classes and objects to organize our program, making it more readable.

We will use **conditional statements** to detect when 4 coins of the same color are aligned to declare the winner of the game (win situation). It will also be used to stop the game when the board is full but no player won (draw situation).

**Composition** will serve to represent relationship between our classes such as Connect4Game and Grid

**Event handling** will be used to spawn a coin when the player clicks on an empty case.

We will also use **Encapsulation** to protect the data by showing only what’s necessary outside the classes.

Finally we will also create a file named “readme” to **document** our code to help others understand our program..>>

* Other notes:
  + <<>>

## Implementation Details

<<Use your notes from above to write code and complete this section of the formal documentation with a README for the user that explains how he/she will interact with the system.>>

The code can be found on github, in the “src” folder. There is a first version “p4”, without graphical interface, to show how we started the reflexion on the code, and then there is the final version in the folder final\_code which is the version with a graphical interface using Tkinter. As explained above, we used a kind of lifecycle method as well as different programming methods like loops, conditions, objects and classes.. We also used Tkinter which is a package for GUI. The README is also on github.

README :

Connect 4 game

Welcome to Connect 4! This is a simple implementation of the classic Connect 4 game using Python with a graphical interface built using Tkinter.

Requirements

To run this game, you need to have Python installed on your system. The code has been tested with Python 3.

How to play

1. Clone or download this repository to your local machine.
2. Navigate to the directory containing the files (src -> final\_code).
3. Run the main.py file using Python. In the terminal type “python main.py”
4. The graphical interface of the Connect 4 game will open.
5. Player 1 and Player 2 take turns dropping colored discs into the grid.
6. The first player to connect four of their discs vertically, horizontally, or diagonally wins the game.
7. If the grid fills up with discs and no player has won, the game ends in a draw.

Interface control

* To make a move, click on the column where you want to drop your disc.
* When the game ends (either by a win or draw), a message box will prompt you to restart the game.

Enjoy the game !

Have fun playing Connect 4 with your friends.

# Testing

Instructions: Week 10

## Journal

The following prompts are meant to aid your thought process as you complete the testing portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

* Have you changed any requirements since you completed the black box test plan? If so, list changes below and update your black-box test plan appropriately.
  + We updated our requirements to integrate the classes of our program.
  + We also put additional rows for the black block testing to test the classes.

| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
| --- | --- | --- | --- |
| Test color class | Test if the color attribute for one of the player is red and the other is yellow | Valid colors should be red and yellow, and invalid colors should raise an error. |  |
| Test disc Class | Test if the attribute of the object follow the appearance requirement | Valid disc colors should be red and yellow, and invalid disc creation should raise an error. |  |
| Test Player Class | Follow if the starting player attribute (number and color) are corrects | Valid player creation with correct number and color, and invalid player creation should raise an error. |  |
| Test Grid Class | Test if the size of the grid is correct | Valid grid creation with correct rows and columns, and invalid grid creation should raise an error. |  |

* List the classes of your implementation. For each class, list equivalence classes, boundary values, and paths through code that you should test.
  + Class : Color
    - equivalence classes =>
      * valid colors : red and yellow
      * Invalid colors : other
    - Boundary Values =>
      * choice 1 : red
      * choice 2 : yellow
    - Test =>
      * initialize players colors with valid color
      * initialize players colors with invalid color
  + Class : Disc
    - equivalence classes =>
      * Valid disc : disc with color
      * Invalid disc : disc without color
    - Boundary Values =>
      * Disc have color
    - Test =>
      * creation of a disc with color
      * creation of a disc without color
  + Class : Player
    - equivalence classes =>
      * player with number (1 or 2) and a color (red or yellow)
      * player without number (1 or 2) and/or a color (red or yellow)
    - Boundary Values =>
      * Correct number and color
    - Test =>
      * creating a player with a color and number
      * creating a player without a color and number
  + Class : Grid
    - equivalence classes =>
      * Grid with good size of rows and columns
      * Grid without good size of rows and columns
    - Boundary Values =>
      * Minimum rows/columns: 1
      * Maximum rows/columns: 7
    - Test =>
      * Creating a grid with a correct size.
      * Creating a grid with an incorrect size.
  + Class : Connect4Game
    - equivalence classes =>
      * Game with a correct number of player
      * Game with an incorrect number of player
      * Game with a draw situation
      * Game with a win situation
    - Boundary Values =>
      * Number of player = 2
      * Draw or win
    - Test =>
      * Start the game with 2 players
      * Start the game with another player count.
      * Fulfill the board the create a draw situation
      * Make one of the player win
* Other notes:

Some more test that are used :

* + Test different winning conditions such as 4 coins aligned vertically, horizontally and diagonally.
  + Test if each placement function correctly

## 

## 

## Testing Details

<<Use your notes from above to write your test programs and complete this section of the formal documentation by creating a list of your test programs along with descriptions of what they are testing. You will also complete the black-box test plan by running the program and filling in the Actual Results column.>>

We have divided our tests into two parts, firstly at the top in the black box part of the report there are the "visual" tests, this means that we check after launching the software that everything behaves as we expected. For this part, we directly filled in the boxes with the results obtained in the table on page 5.

For the second part of the tests, we used some lines of code to verify if everything is going well. We put the result in the second black box table on page 17. You will find below the different tests that we carried out.

The tests can be found in the test-files folder on github. To run them, they need to be added to the game folder and then just executed one by one. They are all individual python files. We use unittest to verify each class.

1. **test\_color.py**

Description of test 1 : This program tests the Color class to ensure that it correctly initializes player colors with valid colors and raises an error for invalid colors.

Results: Valid colors should be red and yellow, and invalid colors should raise an error.

2. **test\_disc.py**

Description of test 2 : This program tests the Disc class to ensure that it initializes discs with colors and raises an error for discs without colors.

Results: Valid disc creation with color, and invalid disc creation should raise an error.

3. **test\_player.py**

Description of test 3 : This program tests the Player class to ensure that it initializes players with correct numbers and colors and raises an error for incorrect inputs.

Results: Valid player creation with correct number and color, and invalid player creation should raise an error.

4. **test\_grids.py**

Description of test 4 : This program tests the Grid class to ensure that it initializes a grid with correct sizes and raises an error for incorrect sizes.

Results: Valid grid creation with correct rows and columns, and invalid grid creation should raise an error.

Results of Tests

| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
| --- | --- | --- | --- |
| Test color class | Test if the color attribute for one of the player is red and the other is yellow | Valid colors should be red and yellow, and invalid colors should raise an error. | Results are as expected, all tests pass. |
| Test disc Class | Test if the attribute of the object follow the appearance requirement | Valid disc colors should be red and yellow, and invalid disc creation should raise an error. | Results are as expected, all tests pass. |
| Test Player Class | Follow if the starting player attribute (number and color) are corrects | Valid player creation with correct number and color, and invalid player creation should raise an error. | Results are as expected, all tests pass. |
| Test Grid Class | Test if the size of the grid is correct | Valid grid creation with correct rows and columns, and invalid grid creation should raise an error. | Results are as expected, all tests pass. |

# Presentation

Instructions:Week 12

## Preparation

The following prompts are meant to aid your thought process as you complete the presentation portion of this exercise. It is recommended that you examine the previous sections of the journal and your reflections as you work on the presentation as it is likely that you have already answered some of the following prompts elsewhere. Please respond to each of the prompts below and feel free to add additional notes.

* Give a brief description of your final project
  + <<Our final project is a Connect 4 video game coded in python with a simple interface, accompanied by instructions for using the computer program.>>(on peut développer un peu aussi)
* Describe your requirement assumptions/additions.
  + << We initially wanted a complete and functional game with relatively simple development, particularly in programming. >> (à développer un peu)
* Describe your design options and decisions. How did you weigh the pros and cons of the different designs to make your decision?
  + <<1. In our first design option, we opted for a more object-oriented approach with separate classes for Game, Player, Disc, Grid/Board, Position, and Color. Each class encapsulates relevant attributes and methods, promoting encapsulation and readability. This design offers clear separation of concerns but may introduce complexity due to the number of classes.

2. In the second design option, we simplified the structure by consolidating some functionality into the Game class and omitting separate classes for Player, Disc, and Position. While this reduces complexity, it may limit scalability and flexibility and potentially lead to errors due to a lack of separation.

3. We chose the first design for its clarity and flexibility, prioritizing encapsulation and clear separation of concerns. This design aligns better with object-oriented principles and facilitates easier maintenance and expansion of the codebase.

-------------------------------------------------------------------------------------------------

Pros and cons :

We evaluated the pros and cons of each design based on factors such as clarity, encapsulation, scalability, and flexibility. While the second design offered simplicity, it sacrificed separation of concerns.

In contrast, the first design provided a more robust foundation with clear separation of classes and attributes, facilitating easier maintenance and scalability despite the initial complexity. We prioritized long-term maintainability and flexibility over short-term simplicity, hence opting for the first design and also for the better understanding of the code for members.>>

* How did the extension affect your design?
  + <<We don’t understand this question.>>
* Describe your tests (e.g., what you tested, equivalence classes).
  + <<We have 4 test files :

1. **test\_color.py**

2. **test\_disc.py**

3. **test\_player.py**

4. **test\_grid.py**

- Each file targets a specific class, testing its initialization and handling of valid and invalid input. Testing ensures that colors, tokens, players, and grid are created correctly and throws errors for incorrect inputs. We also have conditions inside the code that verify if everything is going well during the game..>>

* What lessons did you learn from the comprehensive exercise (i.e., programming concepts, software process)?
  + <<We learned a lot about the structural development of an IT project. In particular the conception and design stages. One of the team members discovered UML with this project and the ease of understanding that it brings to a project regardless of the language used for development. We learned to work with the right stages of project development step by step which until then were still a little vague.>>
* What functionalities are you going to demo?
  + <<First we will show you the alpha, the first functional version of the code. This version shows a return in the terminal.
  + Then secondly we will show a demo of the production version with the colored interface without intervention in the terminal. We will show you a simple game of Connect 4, to demonstrate that our program works correctly without errors. >>
* Who is going to speak about each portion of your presentation? (Recall: Each group will have ten minutes to present their work; minimum length of group presentation is seven minutes. Each student must present for at least two minutes of the presentation.)

<< ???????????????

* + Mehdi Elias LAZREG :
  + Aurélien LASNIER :
  + Alexandre Noel J POILVE :

>>

* Other notes:
  + <<>>

<<Use your notes from above to complete create your slides and plan your presentation and demo.>>