Comprehensive Exercise Report

Team 99 of Section 1

Mert Gulmus - 211ADB070

Pavlo Nikolaiev - 201ADB090

Rustam Rahimov - 211ADB058

[**Requirements/Analysis 2**](#_uwgqwd5ezv2w)

[Journal 2](#_lsityg2iq9m6)

[Software Requirements 3](#_2h0vru1u2mla)

[Functional Requirements: 3](#_k47yyzunvpih)

[Use Case Diagram 3](#_llgzmx75umwi)

[**Black-Box Testing 4**](#_prhaxdxmf8n8)

[Journal 4](#_18f11w613jft)

[Black-box Test Cases 5](#_2xn4jzot820y)

[**Design 7**](#_24fdizefyocn)

[Journal 7](#_esp2ocs9j6bk)

[Software Design 10](#_aifbl1x6rddt)

[Classes 10](#_jensawe68i41)

[GameSession Class 10](#_124qy7sgnem5)

[Properties 10](#_r22bx2lh79b0)

[Methods 10](#_bm6b81z7g08t)

[Player Class 10](#_hnq9umpdz4fy)

[Properties 10](#_lsi02xkdscrk)

[Methods 10](#_2bmj61ri9j5f)

[SessionHistory Class 10](#_czqh207y4vfy)

[Properties 10](#_ph05mey9crgl)

[Methods 10](#_3i9pfhqglu0c)

[**Implementation 12**](#_hya8f3jqkba6)

[Journal 12](#_acupzfhai7gz)

[Implementation Details 13](#_ojhtwkms2z3b)

[**Testing 15**](#_3qvya3vi836q)

[Journal 15](#_ckfs4xbl5pyr)

[Testing Details 16](#_bzt1547yxzxi)

[**Presentation 20**](#_hdjvrbf45b1p)

[Preparation 20](#_xbiquwtmf36n)

# 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.

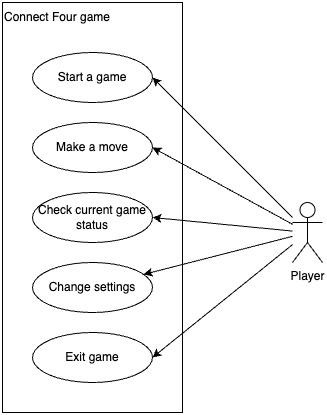
* After reading the client’s brief (possibly incomplete description), write one sentence that describes the project (expected software) and list the already known requirements.
  + A software that will allow users to play the "Connect Four" game.
    - The software must implement the known rules of the game.
* 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.
  + Should software save the results of game sessions?
* Does the project cover topics you are unfamiliar with? If so, look up the topics and list your references.
  + No, only the rules of the game are unfamiliar. However, it can be investigated using the given resources.
* Describe the users of this software (e.g., small child, high school teacher who is taking attendance).
  + Any user that wants to play the Connect Four game. No specific age group.
* Describe how each user would interact with the software
  + Using the graphical user interface, keyboard and mouse
* What features must the software have? What should the users be able to do?
  + Introduction page before starting a game session.
  + Game session page.
  + Game results page.
* Other notes:
  + None.

## Software Requirements

### Functional Requirements:

1. The system shall allow Player 1 to make the first move.
2. The system shall alternate turns between Player 1 and Player 2.
3. The system shall allow a player to drop a disc into one of the seven columns on their turn.
4. The system shall not allow a player to drop a disc into a full column.
5. The system shall check for a win condition after each move.
6. The system shall declare a player the winner if they connect four of their discs in a row vertically, horizontally, or diagonally.
7. The system shall declare the game a draw if the board is filled without a winner.

### Use Case Diagram



# Black-Box Testing

Instructions: Week 4

## Journal

***Remember:*** Black box tests should only be based on your requirements and should work independently 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)?
  + The input for the software would be the column number where the player wants to drop their disc. This would be an integer value between 1 and 7.
* What does output for the software look like (e.g., what type of data, how many pieces of data)?
  + The output of the software would be the updated game board after each move, which could be represented as a 6x7 matrix. The software would also output a message when a player wins or the game is a draw.
* What equivalence classes can the input be broken into?
  + The input can be broken into three equivalence classes:
    - Valid inputs: Numbers between 1 and 7 representing the columns of the game board.
    - Invalid inputs: Numbers less than 1 or greater than 7.
    - Non-numeric inputs: Any input that is not a number.
* What boundary values exist for the input?
  + The boundary values for the input are 1 and 7, which represent the first and last columns of the game board.
* Are there other cases that must be tested to test all requirements?
  + Yes, some additional cases that should be tested include:
    - A player attempts to drop a disc into a full column.
    - The game correctly identifies a win condition (vertically, horizontally, and diagonally).
    - The game correctly declares a draw when the board is filled without a winner.
* Other notes:
  + It’s important to remember that black box testing focuses on the functionality of the software and does not consider the internal structure. Therefore, the test cases should be designed to cover all possible scenarios that could occur during gameplay. This includes not only the normal operation of the game but also any potential error conditions.

## 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** |
| --- | --- | --- | --- |
| TC01 | Player 1 clicks on the 4th column to drop a disc into column 4. | The system updates the game board with Player 1’s disc in the bottom-most empty slot of column 4. |  |
| TC02 | Player 2 tries to click outside the game board, to place a disc to an invalid column. | The player can't place a disc outside the game board. | Player can’t. |
| TC03 | Player 2 tries to trigger an invalid behaviour by pressing random keys. | The system won't trigger any invalid event. | it won't trigger any invalid event. |
| TC04 | Player 1 tries to drop a disc into a full column. | The system won't let the player place a disc in a full column. |  |
| TC05 | Player 2 drops a disc which results in four of their discs in a row vertically. | The system declares Player 2 as the winner. |  |
| TC06 | Player 1 drops a disc which results in four of their discs in a row horizontally. | The system declares Player 1 as the winner. |  |
| TC07 | The game board is filled without a winner. | The system declares the game a draw. |  |

# 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.
  + Player 1
  + Player 2
  + Move
  + Disc
  + Turn
  + Win condition
  + Winner
  + Row
  + Column
  + Board
  + Draw
* Which nouns potentially may represent a class in your design?
  + GameSession, Move, Player, Settings, SessionHistory
* Which nouns potentially may represent attributes/fields in your design? Also list the class each attribute/field would be a part of.
  + GameSession
    - GameID
    - playerID
    - duration
    - moves
    - status
  + Move
    - moveID
    - moveType
  + Player
    - id
    - gameHistory
    - displayName
  + Settings
    - volume
    - resolution
    - language
  + SessionHistory
    - id
    - playerId
    - gameResult
    - date
* 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.
  + Design Option 1:
    - Classes: GameSession, Move, Player, Settings, SessionHistory
    - Pros: This design is straightforward and aligns well with the functional requirements. It clearly separates the concerns of game management (GameSession), individual moves (Move), player data (Player), game settings (Settings), and game history (SessionHistory).
    - Cons: This design might be a bit rigid and could become complex if we need to add more features in the future.
  + Design Option 2:
    - Classes: Game, Player, Move, GameHistory
    - Pros: This design is simpler and more flexible. The Game class handles the game management and settings, which could make it easier to add new features.
    - Cons: This design might not separate concerns as clearly as the first option. For example, the Game class could become too large and complex.
* Which design do you plan to use? Explain why you have chosen this design.
  + We plan to use Design Option 1 because it provides a clear separation of concerns, which can make the code easier to understand and maintain. Each class has a specific responsibility, which aligns with the Single Responsibility Principle of SOLID design principles.
* List the verbs from your requirements/analysis documentation.
  + Allow
  + Make
  + Drop
  + Check
  + Declare
  + Connect
  + Fill
  + Start
  + Set
  + Save
* Which verbs potentially may represent a method in your design? Also list the class each method would be part of.
  + GameSession class
    - startGame()
    - checkWinCondition()
    - declareWinner()
    - declareDraw()
  + Player class
    - makeMove()
  + Settings class
    - setVolume()
    - setResolution()
    - setLanguage()
  + SessionHistory class
    - saveGameSession()
* Other notes:
  + None.

## Software Design

## Classes

### GameSession Class

#### Properties

1. gameId (int): Unique identifier of the game session.
2. duration (int): Time lenght of the game session, in seconds.
3. moves (array): List of moves performed during the session.
4. status (string): Status of the game (still playing, draw, completed).

#### Methods

1. startGame(): Starts the game and allows player one to make the first move.
2. alternateTurns(): Changes the turn to the other player.
3. checkWinCondition(): Checks if the game session has a winner in the current state.
4. declareWinner(): Declares the game as completed and the corresponding player as the winner.
5. declareDraw(): Declares the game as completed and with no winner players.

### Player Class

#### Properties

1. playerId (int): Unique identifier of the player
2. gameHistory (array): List of game sessions completed by the player
3. displayName (string): Display name of the player

#### Methods

1. makeMove(): Drops a disc into one of the columns, if it's their turn.

### SessionHistory Class

#### Properties

1. sessionId (int)
2. playerId (int, foreign key)
3. gameResult (string)
4. date (datetime)

#### Methods

1. saveGameSession()

# 

# 

# 

# 

# 

# 

# 

# 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 prompt 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.
  + Object-oriented programming will be used to implement classes. These objects will encapsulate data and make the code modular and reusable.
* Other notes:
  + Use of ReactJS: ReactJS will be the primary library for building the user interface. Its component-based architecture aligns well with OOP principles, allowing for the creation of reusable, modular components. The virtual DOM in React helps efficiently update the UI in response to state changes.
  + We won't be using an actual backend. The whole process will occur in the frontend (the client). Local storage of the client will be used, instead of a SQL database. This is valid only for the prototype.
  + For version control, we will use Git to manage changes to the codebase and collaborate with team members.

## Implementation Details

# Connect4

## Implementation Details

This document provides detailed instructions on how users will interact with the system, which is implemented using object-oriented programming (OOP) principles and ReactJS.

### Implementation Approach

#### Frontend Only (Client-Side)

- \*\*Local Storage\*\*: Instead of using a backend database, the system will use the browser's local storage to persist data. This approach simplifies the prototype and allows for quick testing and iteration.

#### Version Control

- \*\*Git\*\*: Git will be used for version control to manage changes in the codebase. This allows team members to collaborate effectively, track changes, and revert to previous versions if necessary.

### User Interaction

1. \*\*Setting Up the Project\*\*

- \*\*Clone the Repository\*\*: Use Git to clone the repository to your local machine.

```sh

git clone git@github.com:mertgulmus/DIP392-1-99.git dip392

cd dip392

```

- \*\*Install Dependencies\*\*: Use npm or yarn to install the required dependencies.

```sh

npm install

```

- \*\*Start the Development Server\*\*: Start the React development server.

```sh

npm start

```

- \*\*Access the Application\*\*: Open your browser and navigate to `http://localhost:3000` to access the application.

2. \*\*Using the Application\*\*

- \*\*User Interface\*\*: The user interface is built with React components. Interact with different parts of the application by navigating through the available routes and components.

- \*\*Data Management\*\*: All user data and application state are managed using React state and stored in the browser's local storage.

### Additional Notes

- \*\*Prototype Limitations\*\*: Note that this prototype uses local storage for data persistence and does not include a backend server. For production, consider implementing a backend with a database for data management.

- \*\*React Dev Tools\*\*: Use React Developer Tools for debugging and inspecting React components and their state.

# 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.
  + No, there have been no changes to the requirements since the completion of the black box test plan.
* List the classes of your implementation. For each class, list equivalence classes, boundary values, and paths through code that you should test.
  + GameSession
    - Test initialization of a new game session.
    - Verify the correct initialization of session data.
    - Test making a series of valid game moves.
    - Ensure accurate recording of moves.
    - Test termination of a game session.
    - Verify the correct saving of session data.
    - Test handling of invalid game moves.
    - Ensure appropriate error handling.
  + Players
    - Initialization: Player instance initializes with ID and null display name.
    - Game history: Retrieves and returns game history correctly.
    - Change display name: Updates display name successfully.
    - Retrieve display name: Gets display name from local storage.
    - Save profile: Saves updated display name to local storage.
    - Check login: Determines player login status based on ID presence in local storage.
  + SessionHistory
    - Save Session History: Checks if session details are accurately saved in local storage by the SessionHistory class.
* Other notes:
  + No other notes.

## 

## 

## Testing Details

Game session testing

import { GameSession } from "../src/util/GameSession";

beforeEach(() => {

localStorage.clear();

});

const getSessionHistoryFromLocalStorage = () => {

return JSON.parse(localStorage.getItem('sessionHistory')) || [];

};

test('GameSession saves to SessionHistory correctly', () => {

localStorage.setItem('username', 'test');

const gameSession = new GameSession();

gameSession.startGame();

// Simulate some moves

gameSession.makeMove(0);

gameSession.makeMove(1);

gameSession.makeMove(0);

gameSession.makeMove(1);

gameSession.makeMove(0);

gameSession.makeMove(1);

gameSession.makeMove(0); // Red wins

const sessionHistory = getSessionHistoryFromLocalStorage();

expect(sessionHistory.length).toBe(1);

const savedSession = sessionHistory[0];

expect(savedSession.sessionID).toBe(gameSession.gameID);

expect(savedSession.playerID).toBe('test');

expect(savedSession.gameResult).toBe('red won');

}

);

describe('GameSession', () => {

let game;

beforeEach(() => {

game = new GameSession();

});

test('should start a game', () => {

game.startGame();

expect(game.status).toBe('started');

expect(game.startTime).not.toBeNull();

});

test('should alternate turns', () => {

const initialPlayer = game.currentPlayer;

game.alternateTurn();

expect(game.currentPlayer).not.toBe(initialPlayer);

});

});

Players tests

import { Player } from "../src/util/Player";

describe('Player class', () => {

beforeEach(() => {

localStorage.clear();

});

test('Player initialization', () => {

localStorage.setItem('username', 'test');

const player = new Player();

expect(player.playerID).toBe('test');

expect(player.displayName).toBe(null); // Since no display name set initially

});

test('Getting game history', () => {

localStorage.setItem('username', 'test');

localStorage.setItem('sessionHistory', JSON.stringify([

{ sessionID: 1, playerID: 'test', gameResult: 'win' },

{ sessionID: 2, playerID: 'other', gameResult: 'lose' }

]));

const player = new Player();

expect(player.gameHistory).toEqual([{ sessionID: 1, playerID: 'test', gameResult: 'win' }]);

});

test('Changing display name', () => {

localStorage.setItem('username', 'test');

const player = new Player();

player.changeDisplayName('NewName');

expect(player.displayName).toBe('NewName');

});

test('Retrieving display name', () => {

localStorage.setItem('username', 'test');

localStorage.setItem('test', 'DisplayName');

const player = new Player();

expect(player.retrieveDisplayName()).toBe('DisplayName');

});

test('Saving player profile', () => {

localStorage.setItem('username', 'test');

const player = new Player();

player.changeDisplayName('NewName');

player.savePlayerProfile();

expect(localStorage.getItem('test')).toBe('NewName');

});

test('Checking if player is logged in', () => {

localStorage.setItem('username', 'test');

const player = new Player();

expect(player.isLogged()).toBe(true);

});

});

Session History tests

import { SessionHistory } from "../src/util/SessionHistory";

describe('SessionHistory class', () => {

beforeEach(() => {

localStorage.clear();

});

test('Creating and saving session history', () => {

const sessionID = 1;

const playerID = 'test';

const gameResult = 'win';

const sessionHistory = new SessionHistory(sessionID, playerID, gameResult);

const savedSession = JSON.parse(localStorage.getItem('sessionHistory'))[0];

expect(savedSession.sessionID).toBe(sessionID);

expect(savedSession.playerID).toBe(playerID);

expect(savedSession.gameResult).toBe(gameResult);

expect(new Date(savedSession.date)).toEqual(sessionHistory.date);

});

});

# 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
  + The app we created is a Connect 4 game. It was mainly written using such programming languages as React.js, HTML, CSS, SCCS and JavaScript. It is a two-player game where one wins if it stacks 4 circles of one color either horizontally, vertically or diagonally.
* Describe your requirement assumptions/additions.
  + The game is playable by two players on the same device.
  + The user interface should be intuitive and easy to navigate.
  + The game board should dynamically update to show the current state of play.
  + The game should detect a win condition and display a message to the players.
  + The game should offer a way to reset and start a new game.
* Describe your design options and decision. How did you weigh the pros and cons of the different designs to make your decision?
  + Several design options were taken into consideration. The main idea of the game's design is to make it minimalistic and with sufficient resolution.
* How did the extension affect your design?
  + It was necessary to make the design responsive, making it work on all devices without any visual bugs appearing during gameplay.
* Describe your tests (e.g., what you tested, equivalence classes).
  + The main tests consider the following:
    - User interface tests: testing the responsiveness of the game UI design
    - Game decision tests: testing the decision the game is going to make based on the state of the game
    - Game history tests: testing what is going to be saved in history after each game
* What lessons did you learn from the comprehensive exercise (i.e., programming concepts, software process)?
  + The main lessons that we learned are:
    - Working with React.Js
    - Designing the game and creating UI
    - Teamwork and separation of the workflow
* What functionalities are you going to demo?
  + The main functions of the game (Placing circles, Defining the state of the game)
  + Saving and retrieving final decisions of the previous games.
* 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.)
  + The presentation will be divided into three parts:
    - Coding part
    - Implementation part
    - Challenges and lessons learned part