7/24/2024

Frederick Laroche

NMIT

Game Project

DATA-602

Table of Contents

[Project Mow-Tivated 3](#_Toc184177880)

[Overview of the game: 3](#_Toc184177881)

[In Depth - Description 3](#_Toc184177882)

[Gameplay mechanics: 3](#_Toc184177883)

[Scoring and Leaderboard 4](#_Toc184177884)

[Scoring System: 4](#_Toc184177885)

[Screen Design & Storyboarding 4](#_Toc184177886)

[Player Login, Player Registration, Locked-Out, Player Deletes Own Account 4](#_Toc184177887)

[Laying out tiles on a game board and player movement 6](#_Toc184177888)

[NPC (tile\_type : gnome) 8](#_Toc184177889)

[Player Game Scoring 10](#_Toc184177890)

[Player inventory, item location and placing item on a tile(tile\_type) 11](#_Toc184177891)

[Admin Interface 12](#_Toc184177892)

[Database Design 13](#_Toc184177893)

[Entity Relationship Diagram (ERD): 13](#_Toc184177894)

[Table Descriptions: 14](#_Toc184177895)

[CRUD Analysis 18](#_Toc184177896)

[SQL procedures and C# (milestone 2 & 3) 21](#_Toc184177897)

[Login procedure 21](#_Toc184177898)

[C# Code 22](#_Toc184177899)

[Registration procedure 24](#_Toc184177900)

[Laying out tiles on a game board 26](#_Toc184177901)

[Place items on tiles procedure 28](#_Toc184177902)

[Move player procedure 30](#_Toc184177903)

[NPC movement( I haven’t had time to set this one up) 32](#_Toc184177904)

[Game play scoring procedure 33](#_Toc184177905)

[Player inventory procedure 36](#_Toc184177906)

[Update player data 39](#_Toc184177907)

[Kill running games procedure 41](#_Toc184177908)

[Add new player procedure 43](#_Toc184177909)

[Delete player procedure 45](#_Toc184177910)

[Assignment 2 (stage three) PROTOTYPE V3 48](#_Toc184177911)

[Multiuser Concurrency (Milestone 3) 54](#_Toc184177912)

[References: 55](#_Toc184177913)

Project Mow-Tivated

## Overview of the game:

The game is about lawnmowing where players mow patterns on a tile-based map. The player needs to completely mow a specific pattern (rectangle, square, or diamond) as quickly as possible while avoiding obstacles like rocks and moving garden gnomes.

The players will earn points based on completion speed, remaining health point minus penalties for obstacles hit.

In single-player mode, the goal is to get the highest score possible. Same thing in multiplayer mode, whereas players compete against each other, and the one with the highest score is the winner.

There will be a global chat feature to allow players to communicate during the game or while waiting in the “lobby”.

There will also be a leaderboard to display the top player’s scores.

*(\*See Scoring system below)*

## In Depth - Description

### Gameplay mechanics:

Players move on a grid one tile at a time, either by clicking on an adjacent tile of their lawnmower or using keyboard arrows(I hope this is possible).

The tile has many types :

|  |  |
| --- | --- |
| Tile\_type | Description |
| pattern\_tiles | Mow these to gain points by finishing the pattern. |
| rock\_tiles | Hit these and the player lose 5 points and 1 health point. |
| gnome\_tile | Hit these and the player lose 5 points and 1 health point. |
| heart\_tiles: | Restores health. If health is full, the heart is stored in the inventory. Then if the player move on an obstacles or gnome , consume the heart instead of losing 1 health point.  If health is not full , recover 1 health point. |
| big\_blade\_Tiles | Allows the players to mow three tiles at once instead of one. It also gets stored as an item in the inventory under the name “Bigger Blade”. |
| home\_tile | Where both player starts the game and is the only tile that both player can be on at the same time. |
| Mowed\_tiles |  |

## Scoring and Leaderboard

### Scoring System:

Scores is based on player performance: time taken, health remaining minus penalties for hitting obstacles.

Scores are recorded in the score table, linked to specific players id and game id. This data is used to create the leaderboard.

#### Time :

|  |  |
| --- | --- |
| Time | Point |
| 10 sec | 100 points |
| 20 sec | 90 points |
| 30 sec | 80 points |
| 40 sec | 70 points |
| 50 sec | 50 points |
| 60 sec | 30 points |
| +60 sec | 1 points |

#### Obstacles :

|  |  |  |
| --- | --- | --- |
| Obstacle | Score | Health Point |
| Gnome | -5 | -1 |
| Rock | -5 | -1 |

#### Bonus Point for each health point left when finished :

+5 points for each health point

## Screen Design & Storyboarding

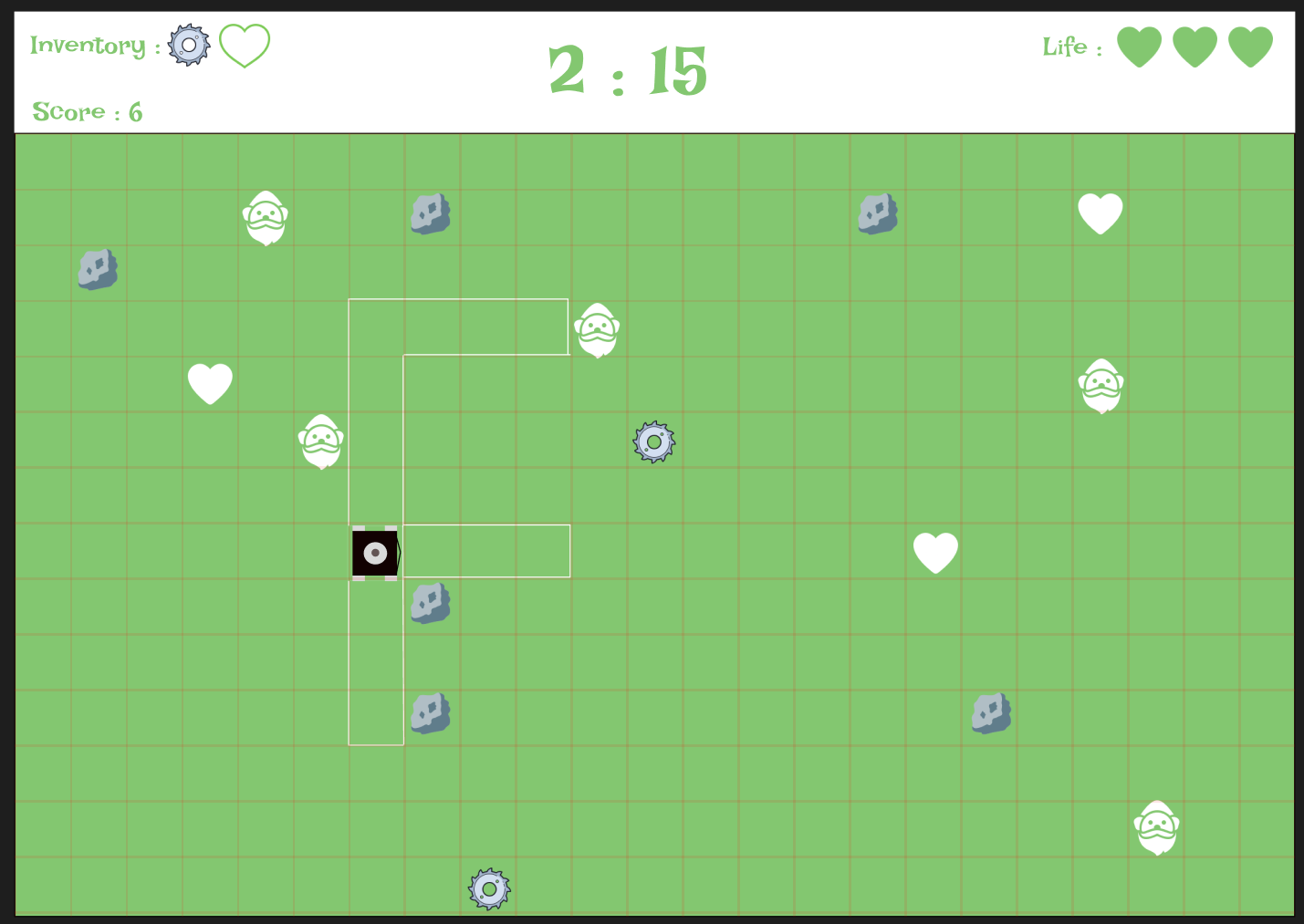
### Player Login, Player Registration, Locked-Out, Player Deletes Own Account

A screenshot of a login form

Description automatically generated

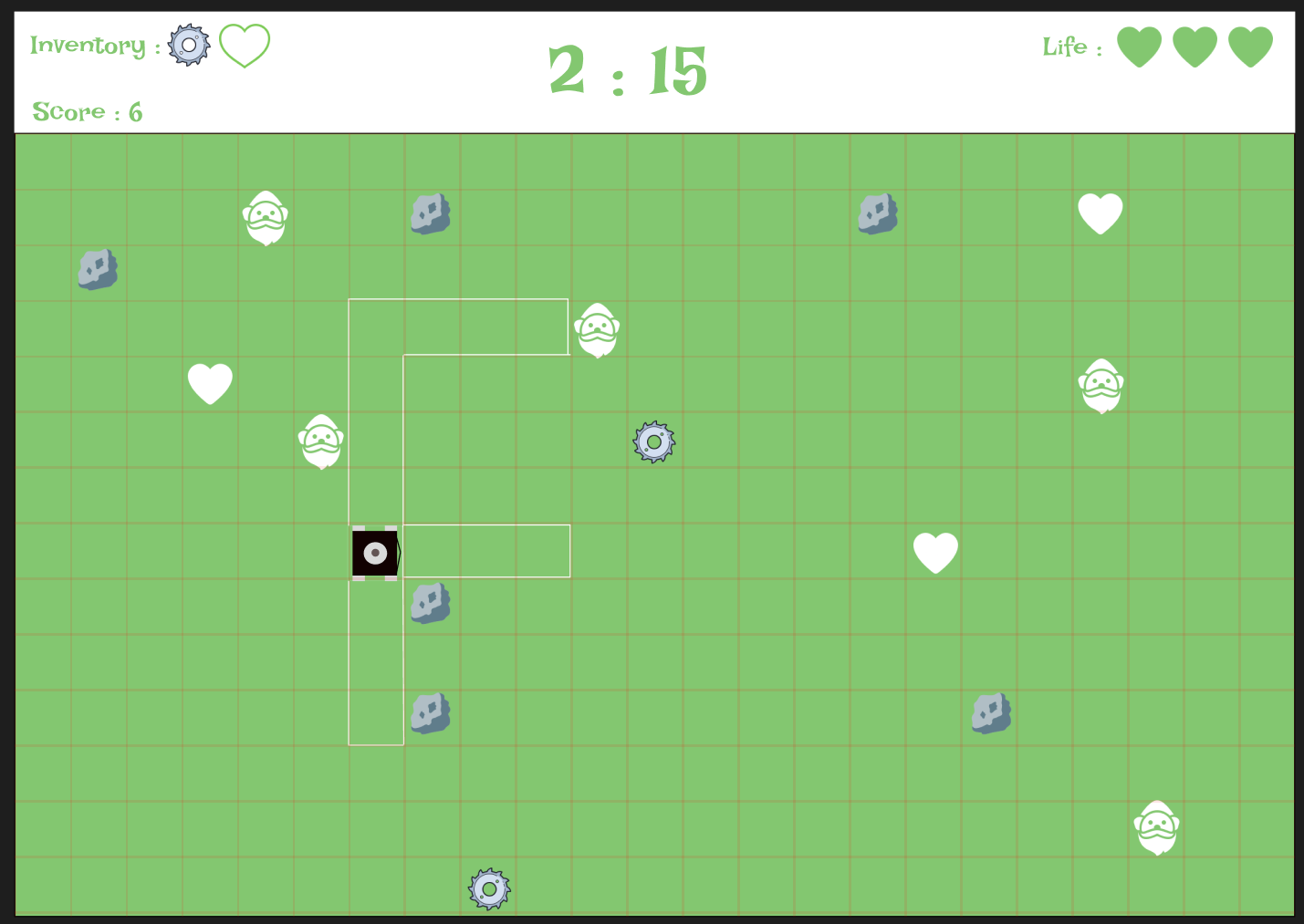
|  |  |  |
| --- | --- | --- |
| Player login | | |
| Description | **User stories** | **Operations** |
| Players need to enter his username and password to log into the game.  Steps:  The system checks if the username exists in the database.  If the username exists ,ask the user to enter their password. If the password is correct, they are logged in.  If the username does not exist already, give the player a "Creating a new account?" message with the options Yes or No.  If the player fails 5 consecutive login attempts, they are locked out.  The Yes or No options is to check if the player have made a mistake in their username as they could only have miss type. | *"As a player, I want to be able to log in so that I can access my player account and play the game."* | **SELECT:** Check the player table if the username exists.  **SELECT:** Check the player table for the player's status to make sure he isn’t ban or lock-out.  **SELECT:** If the username exists, check if the password matches the data recorded in the player table.  **UPDATE:** If login fails, add a login\_attempt for the player in the log table.  **UPDATE:** If the login is successful and before the player is locked out, reset the login\_attempt to 0.  **UPDATE:** Modify the player status to "online" in the player table on login.  **CREATE**: player\_session\_start timestamps on login in the log table.  **CREATE:** Player\_session\_end timestamp when player log out or quit the game.  **Lock-out**  **UPDATE:** When the player hits the maximum consecutive failed login attempts (5 times), update the status in the player table to "locked-out.".  **SELECT:** Get the administrator's email and display to the player on lockout. |
| Player registration | | |
| Description: | **User stories** | **Operations** |
| New players can register by entering their new username in the login form. The system will then check if the username already exists.  If the username doesn’t exist, the player will get a "Creating a new player?" message with Yes or No option.  After clicking Yes, the system will ask for a password to create the new player account. | *"As a new player, I want to register so that I can create an account to tie my data to and start playing the game."* | **CREATE:** Add the player's details into the player table (username, password).  **UPDATE:** Update the player's status to "online" in the player table.  **CREATE:** Add a login\_timestamp in the log table for the new player. |
| Player lock-out: | | |
| Description: | **User stories** | **Operations** |
| After 5 consecutive failed login attempts, the player's account will get locked-out.  The system will then send an email to the player on how to reset their account status.  Also, the player will have the option to click an "Ask for Help" button to contact an admin for assistance. | *"As a player, I want to understand why I can’t log in and how I can unlock my account"*  *“As an admin , I want to be able to change a player status to active.”* | **SELECT:** Get the administrator's email and display to the player on lockout.  **UPDATE:** In the admin email , there will be a link (I guess if possible)that can reset the player's status to "active" .  **OR**  **SELECT:** An admin can check the player's status in player table to verify if the player is either banned or locked-out.  **UPDATE:** When an admin reset the player's status to "active" in the player table . |
| Deletes own account | | |
| Description: | **User stories** | **Operations** |
| Players can delete their own account from the profile form. | *"As a player, I want to be able to delete my account and all my data if I no longer want to play this game."* | **DELETE:** Delete in cascade all the records associated with the player such as : inventory, score, tile, game, player table, log , tile location ,map and player\_chat. |

### Laying out tiles on a game board and player movement



|  |  |  |
| --- | --- | --- |
| Laying out tiles on a game board | | |
| Description | **User stories** | **Operations** |
| The game board is a tile-based grid where each tile represents different type such as grass, mowed tile, rocks, gnomes, hearts and saws.  The tile\_type of each tile will create the game environment and pattern to mow. | *"As a player, I want to play the game and see the different tiles to mow the pattern as fast as possible."* | **CREATE**: Insert data into the tile\_location table, to assign each tile a tile\_id, tile\_location and a tile\_type (ex: grass, rock, gnome, home).  **SELECT**: Get all the records above to initialize the game.  **UPDATE**: As the game advance, update the tile\_location and tile\_type changes (ex: grass is mowed, a rock is destroyed, or gnome is moving). |
| Player Movement | | |
| Description | **User Stories** | **Operations** |
| The player controls a lawnmower that moves one tile at a time on the game board.  The player can click on a tile adjacent to the lawnmower or use the arrow keys on the keyboard to move(*if it is possible*).  The game will then check the tile\_type to affect or not the player appropriately. | *“As a player, I want to move my lawnmower on the game board”*  *“As a player, I want to get affected by the type of tile I move on.”*  *“As a player, I want the game to stop iif my health drops to zero.”* | **SELECT:** When the player moves, the system checks the tile\_type of the tile and the tile\_location.  **If tile\_type is rock or gnome:**  **UPDATE:** Move the player to the tile and update the rock/gnome tile\_type to mowed.  **UPDATE:** Reduce the player's health by 1 in the player table.  **UPDATE:** Reduce 5 points from the player's score in the score table. If the score is negative, set it to 0.  **UPDATE:** Change the tile type to “mowed.”  **If tile\_type is heart:**  **UPDATE:** Increase the player's health by 1 (if not full).  **UPDATE:** Change the tile\_type “heart” to “mowed” from the tile\_type table.  **INSERT:** Add the heart to the player's inventory(if full health, add a full heart. If not full health , add an empty heart to the inventory.  **If tile\_type is grass\_pattern:**  **UPDATE:** Move the player to the tile and update the tile type to “mowed.”  **SELECT/UPDATE:** After each move, check the player's health points. If health is greater than 0, continue the game . If player’s health = 0, create the end\_time of this game.  **If tile\_type is bigger\_blade:**  **UPDATE:** Equip the player with a big blade, increasing mowing area(From 1 to 3 horizontally).  **UPDATE:** Change the big\_blade tile\_type to mowed.  **CREATE**: Add it to the player's inventory. |

### NPC (tile\_type : gnome)



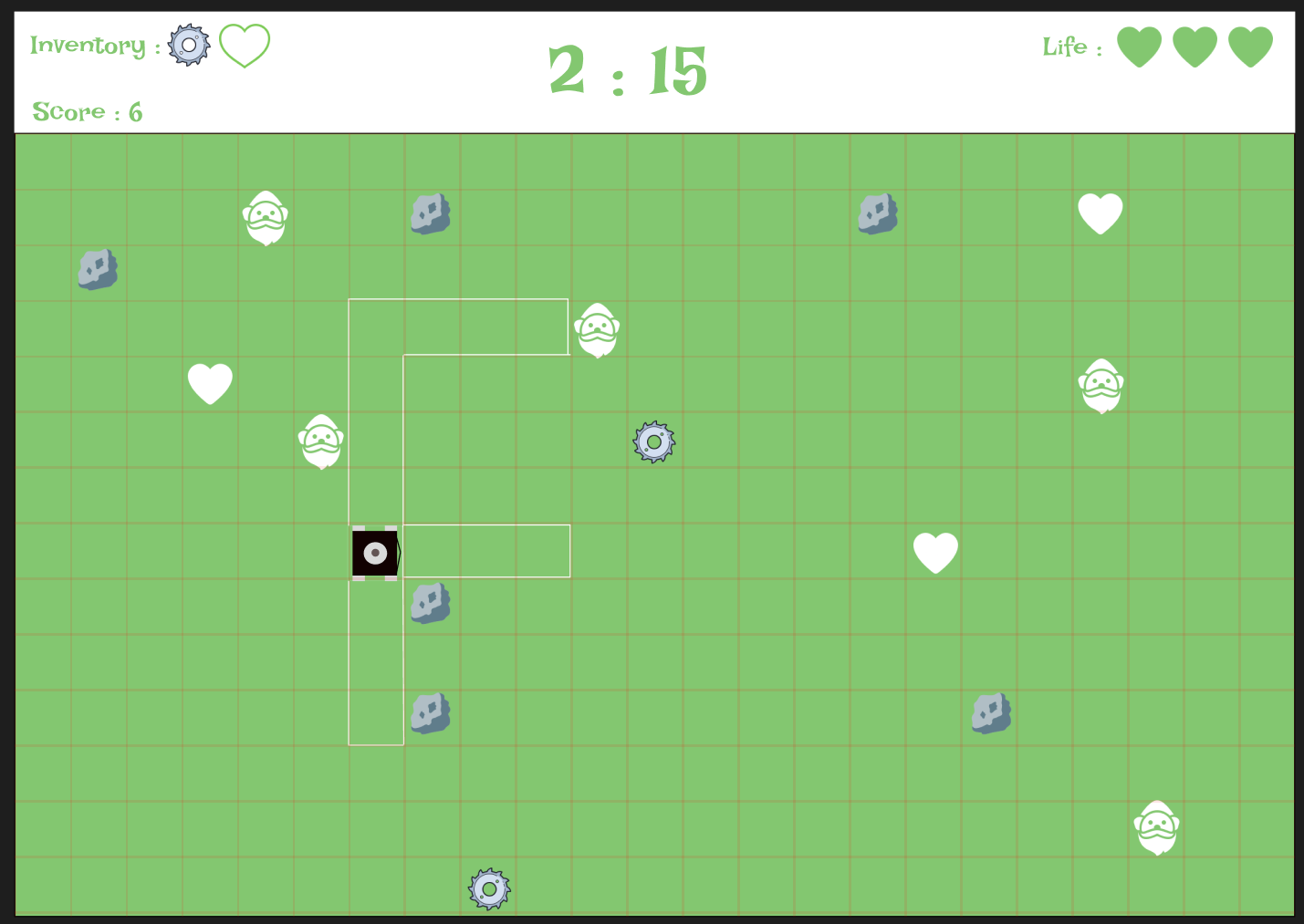
|  |  |  |
| --- | --- | --- |
| Gnome NPC | | |
| Description | **User stories** | **Operations** |
| Gnomes are a tile type. They are moving obstacles, to add a layer of difficulty.  If the player’s lawnmower moves on gnome tile, the player loses health and points.  Gnomes add a layer of difficulty. | *“As a player, I want to gnome to affect me when I come in contact with my lawnmower.”*  *“As a player, I want the gnome to move around the game board to feel some difficulty.”* | **SELECT:** Get the current position and tile\_id of each tile\_type gnome from the tile\_location table.  **UPDATE:** Move the gnomes by updating their location in the tile\_location table with their id.  **UPDATE:** As gnomes move, update the tile\_type they leave to its previous type(ex: tile\_type pattern or grass). |

### Player Game Scoring



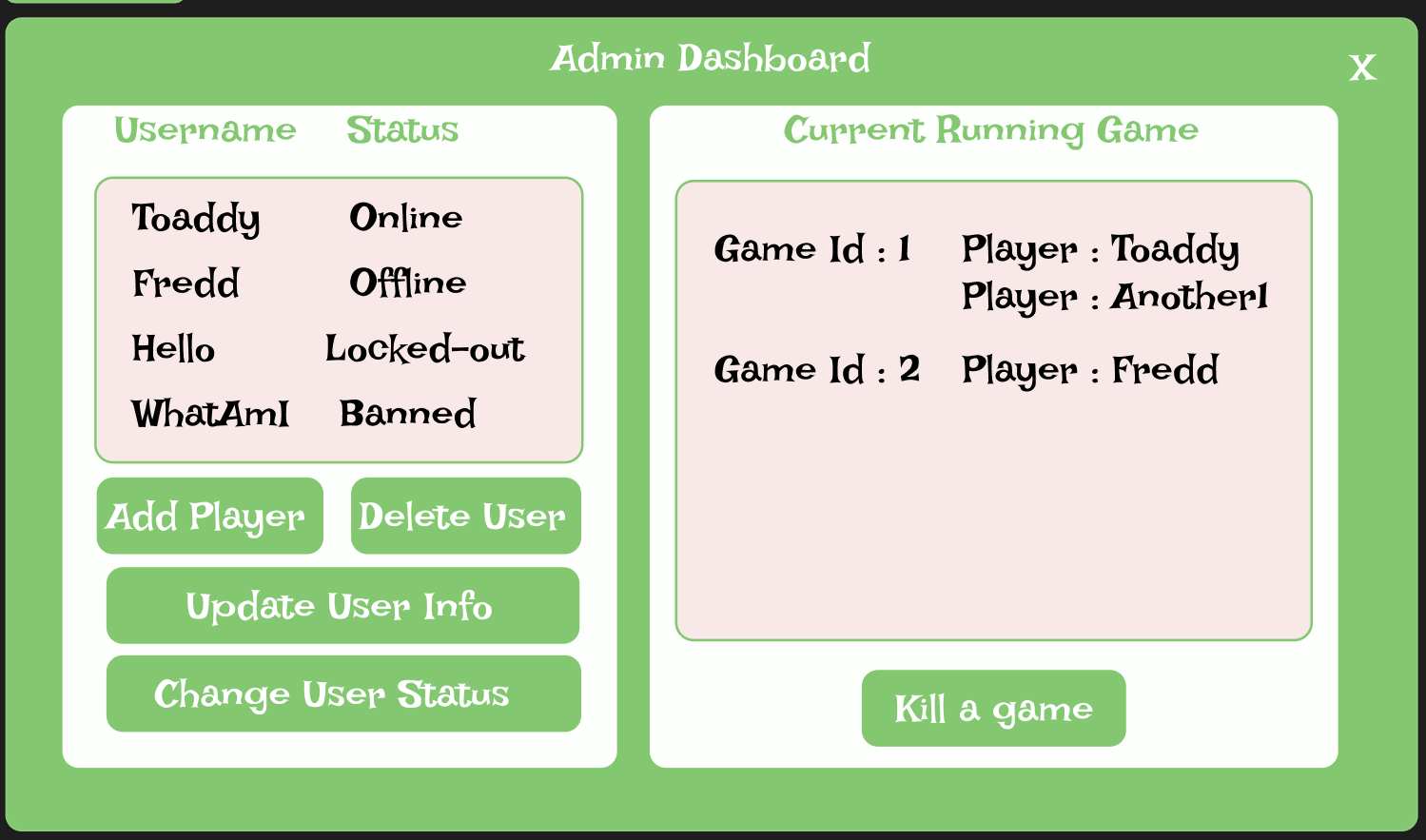
|  |  |  |
| --- | --- | --- |
| Leaderboard | | |
| Description | **User stories** | **Operations** |
| The leaderboard displays the top 5 players and their scores, sorted in desc. order.  Anyone(player and admin) can refresh the leaderboard to look at the latest highest scores. | “As a player, I want to see my score and compare it with others on the leaderboard.”  “As a user, I want to be able to refresh the leaderboard to get the latest top score.” | **SELECT:** Get the highest score, ordered by score\_value in desc. order from the score table to display the leaderboard.  **SELECT:** Get the score\_value and the player username from the player table to display player names and their scores. (Join)  **SELECT:** When the user clicks the "refresh" button, the system re-selects the info to updates the leaderboard. |
| Scoring mechanism | | |
| Description | **User stories** | **Operations** |
| Players will earn points based on their performance, such as time of completion , health remaining, and penalties for hitting obstacles like rocks or gnomes. | “As a player, I want to see my score to be updated in-game so that I know where I stand.” | **Update:** During gameplay, the player's score is updated constantly based on actions (ex: mowing grass in the pattern area, hitting obstacles) by adding or subtracting from the score\_value in the score table.  **Insert:** At the end of the game, the final score is stored in the score table linked to the player's player\_id and the game\_id. |

### Player inventory, item location and placing item on a tile(tile\_type)



|  |  |  |
| --- | --- | --- |
| Player inventory | | |
| Description | **User stories** | **Operations** |
| The inventory is showed at the top left. It shows the items collected during the game.  Only the full heart item can be used on delay (when health drop by one, it is automatically used in the inventory).  All the other items are use immediately after being picked up(Bigger saw and heart when the player has not all his health point already). | “As a player, I want to see the items I have collected in my current game and keep track of them in my inventory.” | **CREATE:** When a player step on an item(tile\_type: heart or bigger blade), it is added to the inventory table and linked to the player\_id.  **SELECT:** Get the inventory table and item table with the player\_id.  **UPDATE:** When a player uses an item, update the item table. |
| Item location | | |
| Description | **User stories** | **Operations** |
| Items like hearts, rocks, bigger blade, and gnomes are placed on specific tiles on the game board to add a layer of strategy and difficulty to the gameplay. | “As a player, I want to see where the items are located on the map so that I put a strategy in place.” | **SELECT:** Get the location of the tile\_type heart and bigger blade on the game board by searching the tile\_location table(row and column) for each tile\_id and tile\_type.  **UPDATE:** When a player moves on a tile\_type heart or bigger blade, change the tile\_type to mowed. |
| Item on a tile (tile\_type) | | |
| Description | **User stories** | **Operations** |
| Each item on the board is a specific tile\_type. They all affect the player differently . | “As a player, I want the items on the tiles to interact with my lawnmower based on the type of tile they are on, affecting my score and health.” | **SELECT:** Set the game board by getting the tile\_location ,tile id and tile\_type.  **UPDATE:** As the player moves on different tiles, the player can trigger some effects. Will need to update either health, inventory, tile\_type, mowing area or score . |

### Admin Interface



|  |  |  |
| --- | --- | --- |
| Kill running games | | |
| Description | **User Stories** | **Operations** |
| Function to kill running games when an admin user selects a game and click the "Kill Game" button in the admin interface. | “As an admin, I want to have the possibility to terminate running games to manage the system.” | **SELECT:** Get the game table where end\_time is null to get a list of current running games.  **UPDATE:** When the admin selects a game and clicks "Kill Game," update the end\_time in the game table to now. |
| Add new players | | |
| Description | **User stories** | **Operations** |
| Admin can add new players by filling out a form(username and password). | “As an admin, I want to add new players to the game easily in my interface.” | **CREATE:** Insert a new record in the player table (username, password).  **SELECT:** Check if the username already exists in the player table to prevent duplicates. |
| Update players data | | |
| Description | **User stories** | **Operations** |
| Function to update existing players' data by the admin. | “As an admin, I want to be able to update player information to manage user accounts in my interface.” | **SELECT:** Get the player details from the player table when the admin selects a player in the admin interface.  **UPDATE:** Modify the player’s details (username, password, status) and update the record in the player table. |
| Remove existing players | | |
| Description | **User stories** | **Operations** |
| Function for admin to remove existing players by deleting their account. | “As an admin, I want to be able to change a player status to manage user’s access to the game.” | **SELECT:** Get the player's status and details from the player table when the admin selects a player in the admin interface.  **DELETE:** Completely remove the player’s record from the player table and delete in cascade in the other related tables. |

# Database Design

### Entity Relationship Diagram (ERD):

### Table Descriptions:

|  |  |  |
| --- | --- | --- |
| Player | | |
| Purpose | **Attributes** | **Relationships** |
| Stores all relevant information about the players.  A player can only have one character as the player is the character and their player username is the character name. | * **player\_id:**   Primary key to identify each unique player.   * **game\_id:**   Foreign key linking the game table to identify the current game to the associated player.   * **tile\_id:**   Foreign key linking the tile table to identify the current player position on the game board.   * **username:**   The player’s username used for login and in the game.   * **player\_password:**   The player’s password for login and in the game.   * **is\_admin:**   Boolean(true or false) if the player has administrative privileges or not.   * **status:**   Current status of the player (ex: active, locked-out, banned).   * **healthpoint:**   Current player’s health points during the game. | **Relationship with game**:  A player is associated with a game via game\_id. A player can only have one game running at a time .  A game can have many player as it is needed to play multiplayer.  **Relationship with tile**:  A player’s position on the game board is tracked by their tile\_id.  A tile can have many player(Only the home tile on multiplayer mode) but a player can only have 1 tile. |
| Game | | |
| Purpose | **Attributes** | **Relationships** |
| Tracks the details of each game session. | * **game\_id:**   Primary key to identify each unique game session.   * **game\_type:**   Type of game (single-player or multiplayer).   * **start\_time:**   Timestamp of when the game started.   * **end\_time:**   Timestamp of when the game ended (null for ongoing games). | **Relationship with player**:  Multiple players can participate in a game, linked through game\_id.  A player can only one game at a time.  **Relationship with map**:  Each game has one map.  Each map can only have one game. |

|  |  |  |
| --- | --- | --- |
| Score | | |
| Purpose | **Attributes** | **Relationships** |
| Records the scores for each player in each game session. | * **score\_id:**   Primary key to identify each unique score.   * **player\_id:**   Foreign key linking the player table to indicate which player got this specific score.   * **game\_id:**   Foreign key linking the game table to indicate in which game the score was made.   * **score\_timestamp:**   Timestamp of when the score was made.   * **score\_value:**   The actual score value made by the player. | **Relationship with player**:  A score is linked to a player by the player\_id.  **Relationship with game**:  A score is linked with a specific game by the game\_id. |
| Log | | |
| Purpose | **Attributes** | **Relationships** |
| To keep track of player login attempts and log time. | * **log\_id:**   Primary key to identify each unique log entry.   * **player\_id:**   Foreign key linking the player table to indicate which player the log is from.   * **login\_attempt:**   To count the number of login attempt made by the player.   * **login\_timestamp:**   Timestamp of each login attempt. | **Relationship with player**:  Log data is linked with a specific player by a player\_id. |
| Player chat | | |
| Purpose | **Attributes** | **Relationships** |
| To store every chat messages sent by players in the global chat. | * **player\_id:**   Foreign key linking the player table to indicate which player sent the message.   * **chat\_id:**   Foreign key linking the chat\_session table, indicating the chat session the message belongs to.   * **timestamp:**   Timestamp of when the message was sent.   * **message:**   The message content. | **Relationship with player**:  A chat message is linked to a player by a player\_id.  **Relationship with chat\_session**:  A message is linked with a specific chat session. |

|  |  |  |
| --- | --- | --- |
| Chat session | | |
| Purpose | **Attributes** | **Relationships** |
| To track chat sessions that happen during game and lobby sessions. | * **chat\_id:**   Primary key to identify each unique chat session.   * **session\_start:**   Timestamp of when the chat session started.   * **session\_end:**   Timestamp of when the chat session ended (null for ongoing chat sessions). | **Relationship with player\_chat**:  Many messages can be associated with one chat session by chat\_id. |
| Map | | |
| Purpose | **Attributes** | **Relationships** |
| To define the map and the tiles linked with it for each game . | * **map\_id:**   Primary key to identify each unique map.   * **game\_id:**   Foreign key to link the game table to indicate which game this map has. | **Relationship with game**:  Each map is linked with a game session by a game\_id.  **Relationship with tile**:  The map has multiple tiles linked by the tile\_id. |
| Tile | | |
| Purpose | **Attributes** | **Relationships** |
| To store each tile and details on the game board. | * **tile\_id:**   Primary key to identify each unique tile.   * **map\_id:**   Foreign key to link the map table to which map has the specific tile . | **Relationship with map**:  Each tile is part of a map\_id.  **Relationship with tile\_location**:  The placement of a tile on the game board is made by the tile\_location. |
| Tile location | | |
| Purpose | **Attributes** | **Relationships** |
| To set the position of each tile on the board. | * **tile\_id:**   Foreign key to link the tile table to identify the specific tile.   * **tile\_type\_id:**   Foreign key to link the tile\_type table to a type of tile (ex: pattern, rock, gnome).   * **row:**   The row position of the tile on the game board.   * **column:**   The column position of the tile on the game board. | **Relationship with tile**:  The tile location is tracked by a tile\_id.  **Relationship with tile\_type**:  The type of each tile is tracked by a tile\_type\_id. |

|  |  |  |
| --- | --- | --- |
| Tile type | | |
| Purpose | **Attributes** | **Relationships** |
| To define the types of tiles that are on the game board. | * **tile\_type\_id:**   Primary key to identify each unique tile type.   * **name:**   The name of the tile type (ex: pattern, rock, gnome).     * **effect:**   The effect that the tile type has on the player (ex: +/- health, +/- score).   * **score\_value:**   The score value of the tile type if there is any (ex: pattern= 0 point). | **Relationship with tile\_location**:  Each tile type is linked to a tile on the board by the tile\_type\_id.  **Relationship with item**: Items are linked with the tile types by the tile\_type\_id. |
| Inventory | | |
| Purpose | **Attributes** | **Relationships** |
| To track the items that players have collected during the current game. | * **inventory\_id:**   Primary key to identify each unique inventory.   * **player\_id:**   Foreign key to link the player table to which player has this inventory.   * **game\_id:**   Foreign key to link the game table to the game the item is collected.   * **quantity:**   The quantity of the item held by the player. | **Relationship with player**:  The inventory is linked to a specific player by the player\_id.  **Relationship with game**:  The items in the inventory are linked to a specific game session by the game\_id. |
| Item | | |
| Purpose | **Attributes** | **Relationships** |
| To store all the item’s details that can be collected by a players. | * **item\_id:**   Primary key to identify each unique item.   * **inventory\_id:**   Foreign key to link the inventory table to the inventory the item belongs to.   * **tile\_type\_id:**   Foreign key to link the tile\_type table to a type of tile the item is associated with.   * **item\_name:**   The name of the item.   * **item\_description:**   Description of the item and his effects. | **Relationship with inventory**:  Each item is linked to a player's inventory by the inventory\_id.  **Relationship with tile\_type**:  The items are linked to a specific tile type by a tile\_type\_id. |

# CRUD Analysis

**C= Insert R= SELECT U= UPDATE D= DELETE**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Process | Table | Attributes affected | CRUD | Description |
| Player login/log out  Including lockout | Player  log | Username  player\_password  status  login\_attempt  player\_id  player\_session\_start  player\_session\_end | **R** | Check if the username exists |
| **R** | Check the player’s status |
| **R** | Check the password if the username exists |
| **U** | Add a login attempt in the log table |
| **U** | Reset to 0 the login attempts of player on login |
| **U** | Update the player status to : "online" ,“offline” or ”locked-out” |
| **C** | Add a player\_session\_start time |
| **C** | Add a player\_session\_end time on logout |
| **U** | Update the player status to "locked-out" after 5 consecutive failed attempts |
| **C** | Add a login time in the log table |
| **R** | Get and display the admin email to the player on lockout |
| Process | **Table** | **Attributes affected** | **CRUD** | **Description** |
| Player registration | Player  log | Username  player\_password  status    player\_id  login\_timestamp | **C** | Insert the new player info in the player table |
| **R** | Check if the username exists |
| **U** | Update the player status to "online" when the account is created |
| **C** | Add a login\_timestamp in the log table |
| **C** | Add a player\_session\_start time |
| Process | **Table** | **Attributes affected** | **CRUD** | **Description** |
| Laying out tiles on a game board | tile  tile\_location  tile\_type | tile\_id    tile\_type\_id    row    column | **C** | Insert data to define each tile\_type and tile\_location on the board |
| **R** | Get the tile data to initialize the game board |
| **U** | Update the tile\_type as the game progresses (ex: tile\_type pattern to mowed, rock to mowed etc) |
| Process | **Table** | **Attributes affected** | **CRUD** | **Description** |
| Placing an item on a tile | tile\_location  item | tile\_id  tile\_type\_id  item\_id  row  column | **C** | Insert an item on a tile in the tile\_location table |
| **R** | Get the item and tile details(type and location) |
| **U** | Update the item state and location |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Process | Table | Attributes affected | CRUD | Description |
| Player movement | Player  tile\_location  tile\_type  score  inventory  Game | tile\_id  tile\_type\_id  player\_id  healthpoint  score\_value  inventory\_id  end\_time | **R** | Check the tile\_type and tile\_location before moving to determine what will interaction is with the player. |
| **U** | Move the player to the new tile, update healthpoint if necessary, and change the tile\_type to "mowed." |
| **U** | Update the score\_value according to the tile\_type and effect. |
| **C** | Insert an end\_time for the game if the player mows the last tile\_type pattern. |
| **U** | Update inventory if necessary (ex: add heart to inventory). |
| Process | **Table** | **Attributes affected** | **CRUD** | **Description** |
| Gameplay scoring | score  player | score\_value  player\_id  game\_id  score\_timestamp | **U** | Update the score\_value continuously in the score table during the game based on the player/players interactions. |
| **R** | Get the score\_value and the score\_timestamp from the score table to create the leaderboard. |
| Process | **Table** | **Attributes affected** | **CRUD** | **Description** |
| Player Gameplay Acquiring Inventory  (ex: picking up items off a tile and putting them in an inventory) | Inventory  item | inventory\_id  player\_id  item\_id  quantity | **C** | Add an item to the player’s inventory table. |
| **U** | Update the quantity in the inventory table for the specific player. |
| **R** | Get the player's id and inventory by selecting the inventory table to show the inventory on screen. |
| Process | **Table** | **Attributes affected** | **CRUD** | **Description** |
| Move an item (NPC effect) | tile\_location  item  tile\_type | tile\_id  tile\_type\_id  item\_id | **R/U** | Get and update the row and column of the tile\_type gnome to move the NPC to a new tile. |
| **R/U** | Get the previous tile\_type of the tile and update the type by the old value when the gnome moves. |
| Process | **Table** | **Attributes affected** | **CRUD** | **Description** |
| Kill running games | game  player | game\_id  end\_time  status | **R** | Get the ongoing games from the game table where end\_time is null. |
| **U** | Update the status in the player table to set player/players as offline. |
| **U** | Update the end\_time in the game table to set the game as ended. |
| Process | **Table** | **Attributes affected** | **CRUD** | **Description** |
| Add new players | player | username  player\_password  status  player\_id | **C** | Insert new player record in the player table(username, password). |
| **R** | To make sure the username does not exist before creating the account. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Process | Table | Attributes affected | CRUD | Description |
| Update data of a player | player  score  log | username  player\_password  status  player\_id | **R** | Get the player info from the player table to do the update. |
| **U** | Modify and update the player info from the player table. |
| Process | **Table** | **Attributes affected** | **CRUD** | **Description** |
| Delete a player | player  score  inventory  log  player\_chat    game | player\_id  username  status  score\_value  score\_timestamp  login attempt  quantity  message  start\_time  end\_time | **R** | Retrieve player data before deletion. |
| **Delete in cascade** | |
| **D** | Delete the player's score\_value and score\_timestamp in the score table where the player\_id matches the selected player. |
| **D** | Delete all inventory items associated with the player from the inventory table where the player\_id matches. |
| **D** | Remove all login attempt records related to the player from the log table where the player\_id matches the selected player. |
| **D** | Delete all chat messages associated with the player from the player\_chat table where the player\_id matches the selected player. |
| **D** | Remove all the game session data(start\_time and end\_time) of the player from the game table where the player\_id matches the selected player. |

List to do :

1. Player login, including lock out. [4]

2. Player registration,[4]

3. Laying out tiles on a game board. [4]

4. Placing an item on a tile. [4]

5. Player game play movement, i.e., moving a player to a “legal” tile. [4]

6. Game play scoring. How do players gain and lose points? [4]

7. Player game play acquiring inventory, e.g., picking up items off a tile and putting them in an

inventory (bag?) [4]

8. Move an Item (NPC effect). [4]

9. Kill running games. [4]

10. Add new players. [4]

11. Update data of a player. [4]

12. Delete a player. [4]

# SQL procedures and C# (milestone 2 & 3)

## Login procedure

**MySQL file (line 202 to 290)**

#### MySQL test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Results | Screenshot | Test data | Pass/fail |
| Login with valid username and password. | User logs in successfully. |  | **CALL Login('123', '123');** | **pass** |
| Login attempt with incorrect password. | Login fails. Login attempt count is incremented by 1 .  an error message is returned with the number of failed attempts out of 5. |  | **CALL Login('123', 'badpass');** | **pass** |
| Login attempt with non-existent username. | Login fails. An error message is returned .It then ask to the user if they want to register. |  | **CALL Login('1234', '123');** | **pass** |
| Login for a locked-out user. | Login fails.  error message returned : the account is locked out. |  | **CALL Login('Hello', 'password12345');** | **pass** |
| Login attempts with locked-out account . | Login fails.  Return a lockout message. |  | **CALL Login('Hello', 'password12345');** | **pass** |
| Login attempt for a banned user. | Login fails. An error message is returned : the account is banned. |  | **CALL Login('WhatAmI', 'password123456');** | **pass** |

## C# Code

#### C# Test

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Test data | Expected: | Screenshot |
| Valid login: | Username: 123  Password: 123 | Login successful |  |
| Invalid password: | Username: 123  Password: 1234 | Invalid credentials message |
| Admin status testing | | | |
| Test case | Test data | Expected: | Screenshot |
| Valid admin check: | PlayerID: 2 | true |  |
| Non-admin check | PlayerID: 1 | false |
|  |

#### C# Interaction and GUI

|  |  |
| --- | --- |
|  |  |
| Login screen: | Users enter their username and password in the login form. |
| Login button : | Get the username and password |
| Call the UserDao.Login method with the credentials. |
| Success: | Redirect to the lobby |
| Fail: | Displays an error message(depend on the error) |
| Screenshot: (example of wrong password error) | *A screenshot of a computer  Description automatically generated* |

|  |  |
| --- | --- |
| Exception handling | |
| DAO | UserDao.cs |
| Connection check |  |
| Data validation |
| Exception block |
| Test code |  |

## Registration procedure

**MySQL file (line 292 to 303)**

#### MySQL Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Results | Screenshot | Test data | Pass/fail |
| Register a new user (valid credentials) | Registration is successful.    Return the new player ID. |  | **CALL Register('12345', 'newpass123');** | **pass** |
| Register with an existing username. | Registration fails.  Error message is returned. |  | **CALL Register('123', '123');** | **pass** |

C# Test

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Test data | Expected: | Screenshot |
| Valid registration | Username: 123456  Password: pass123 | Registration successful |  |
| Invalid registration | Username: 123  Password: 123 | Username already exist |

#### C# Interaction and GUI

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Registration screen: | If a register attempt fails due to an existent username, the user is prompted to choose another one. | |
| Registration button : | Get the username and password.  Call UserDao.Register method. | |
| Success: | Redirect to the login screen. | |
| Failure: | Displays an error message | |
| Screenshot: (example of existing username error) |  | |

|  |  |
| --- | --- |
| Exception handling | |
| DAO | **UserDao.cs** |
| Connection check |  |
| Data validation |
| Exception block |
| Test code |  |

## Laying out tiles on a game board

**MySQL file line 657 to 876**

#### MySQL Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Results | Screenshot | Test data | Pass/fail |
| Generate single player map | Board created with 10x10 tiles  All tile type placed correctly. |  | **CALL StartGame(2, 'single-player');**  **CALL GenerateSinglePlayerMap(1);** | **pass** |
| Get game board layout | Shows all tiles and positions.  Shows tile types and player location. |  | **CALL GetGameBoard(2);** | **pass** |

#### C# Test (GameDao)

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Test data | Expected: | Screenshot |
| Get game board | Game ID: 2 | Returns:  Tile positions  Tile types  Player positions | (see GameDao for testing code. Code is run in the program file) |
| Board generation | Game ID: 1 | Creates new 10x10 board Places all required features |

#### C# Interaction and GUI

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Game screen: | 10x10 grid displayed  Different tiles shown with different colors/symbols:  NonMowed: Basic grass (green)  Mowed: Cut grass (light green)  Pattern: Special areas to mow  Rock: Obstacles  Items: Heart, BiggerBlade  Home: Starting position  Gnome: Moving enemy | |
| Board layout: | U-shaped pattern to mow  Rocks placed as obstacles  Power-ups (Heart, BiggerBlade) placed  Gnome enemy placed  Home tile at bottom middle  Rest filled with basic grass | |
| Success: | Board generated and displayed  All tile placed correctly  Player positioned at home tile (H tile Yellow) | |
| Screenshot: | Initial board (player in blue on H tile(Home tile)    After player move, the board update the tile color(tile type)  Non-mowed to mowed(green N to red M)  Home tile is yellow | |

|  |  |
| --- | --- |
| Exception handling | |
| DAO | GameDao.cs |
| Connection check |  |
| Data validation |
| Exception block |
| Test code |  |

## Place items on tiles procedure

**MySQL file line 657 to 876**

(same as map generation since items are placed during map creation)

#### MySQL Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Results | Screenshot | Test data | Pass/fail |
| For testing purposes, I can do it this way  Place heart item | Item placed on board at specified position. Tile type updated. |  | **SELECT \* FROM tile\_location tl JOIN tile t ON tl.tile\_id = t.tile\_id WHERE tl.tile\_type\_id = 7;**  (checks heart placement at row 3, column 1) | **pass** |
| For testing purposes, I can do it this way  Place bigger blade item | Item placed on board at specified position. Tile type updated. |  | **SELECT \* FROM tile\_location tl JOIN tile t ON tl.tile\_id = t.tile\_id WHERE tl.tile\_type\_id = 8;**  (checks bigger blade placement at row 7, column 4) | **pass** |

#### C# Test (GameDao)

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Test data | Expected: | Screenshot |
| Check heart placement | Game ID: 2 Row: 3 Column: 1 | Heart item found at correct position | (see GameDao for testing code. Code is run in the program file) |
| Check bigger blade placement | Game ID: 2 Row: 7 Column: 4 | Bigger blade item found at correct position |

#### C# Interaction and GUI

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Game screen: | Items appear on board as letter and color:  Heart: h  Bigger blade: B  Rock: R  Gnome:G  Home: H | |
| Item placement: | I only have one hardcoded map at the moment for singleplayer  Heart at (3,1)  Bigger blade at (7,4)  Rocks at (2,2), (2,7), (6,3), (6,6)  Gnome at (4,8)  Home at (9,5) | |
| Success: | Items visible on the game board  Items interact correctly with the player | |
| Screenshot: | Items are placed correctly on the board(as expected) | |

## Move player procedure

**MySQL file line 877 to 1011**

#### MySQL Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Results | Screenshot | Test data | Pass/fail |
| Valid move (adjacent tile) | Player moved successfully  Position updated in database. |  | CALL MovePlayer(1, 2, 8, 5);  (moving up from home position) | **pass** |
| Invalid move (too far) | Move rejected. Position unchanged. |  | CALL MovePlayer(1, 2, 1, 1);  (trying to move across board) | **pass** |

#### C# Test (GameDao)

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Test data | Expected: | Screenshot |
| Valid move | Player ID: 1  Game ID: 2  New Row: 8  New Column: 5 | Move successful  Position updated | (see GameDao for testing code. Code is run in the program file) |
| Invalid move | Player ID: 1  Game ID: 2  New Row: 1  New Column: 1 | Move rejected Position unchanged |

#### C# Interaction and GUI

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Game screen: | Player position shown on board(blue) | |
| Valid moves: | One tile up/down  One tile left/right  One tile in diagonal | |
| Movement rules: | Can only move to adjacent tiles(see valid move)  Must have health above 0  Can't move if the game is over  Movement triggers the tile effects | |
| Success: | Player moves to new position  Tile effects applied to game ,player,inventory(pattern,health,gnome,rock,biggerblade)  Score/health updated | |
| Failure: | Invalid move: Error message | |
| Screenshot: | See the other screenshot from the other procedure as this one could be too extensive | |

|  |  |
| --- | --- |
| Exception handling | |
| DAO | **GameDao.cs** |
| Connection check |  |
| Exception block |
| Test code |  |

## NPC movement( I haven’t had time to set this one up)

I have added a gnome tile type, it is on my single player map but I haven’t added a movement logic. I tried but I was getting too many error to fix for the time I had remaining to submit the assessment.

## Game play scoring procedure

**MySQL file line 1120 to 1156**

#### MySQL Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Results | Screenshot | Test data | Pass/fail |
| Add score for collecting pattern tiles | Score added (10 points) Score updated in database |  | **CALL UpdateGameScore(1, 1, 10);** | **pass** |
| Add score for normal grass tiles | Score added (1 point) Score updated in database |  | **CALL UpdateGameScore(1, 1, 1);** | **pass** |
| Add penalty for moving on a rock/gnome | Penalty applied (-5 points) Score updated in database |  | **CALL UpdateGameScore(1, 1, -5);** | **pass** |

#### C# Test (GameDao)

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Test data | Expected: | Screenshot |
| Add positive score  (normal grass tile) | Player ID: 1  Game ID: 1  Score: 1 | Score +1  successfully added | (see GameDao for testing code. Code is run in the program file) |
| Add negative score  (gnome or rock) | Player ID: 1  Game ID: 1  Score: -5 | -5 Score  successfully added |
| Add positive score  (pattern tile) | Player ID: 1  Game ID: 1  Score: +10 | Score +10  successfully added |

#### C# Interaction and GUI

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Game screen: | Points are awarded automatically during gameplay when a player move on a tile(score depend on the tile type):  Pattern tiles: +10 points  Normal grass: +1 point  Rock/Gnome: -5 points | |
| Scoring mechanism: | Points are gained when moving on any tiles (some are positif or negative points)  Score updates in real-time  Score history for the game is kept to update the leaderboard (if top 10 score) | |
| Success: | Update the score in the GUI | |
| Screenshot: (example of score update) | Game start at 0 points :  Score updated after 1 move(regular grass +1 point) :  Score updated after move(rock -5 points) :    Score updated after move(pattern +10 points) :    Same with gnome: | |

|  |  |
| --- | --- |
| Exception handling | |
| DAO | **GameDao.cs** |
| Connection check |  |
| Exception block |
| Test code |  |

## Player inventory procedure

**MySQL file line 1012 to 1052**

#### MySQL Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Results | Screenshot | Test data | Pass/fail |
| Add an item to empty inventory | Item added successfully  Quantity set to 1 |  | **CALL AddItemToInventory(4, 1, 7);**  **CALL GetPlayerInventory(4, 1);**  **(player 1, game 1, heart item)** | **pass** |
| Add item to existing inventory | Quantity increased by 1  Inventory updated |  | **CALL AddItemToInventory(4, 1, 8);**  **CALL GetPlayerInventory(4, 1);**  **(player 1, game 1, bigger blade item)** | **pass** |

#### C# Test (GameDao)

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Test data | Expected: | Screenshot |
| Get player inventory | Player ID: 1 Game ID: 1 | Return list of items | (see GameDao for testing code. Code is run in the program file) |
| Add item to inventory | Player ID: 1  Game ID: 1  Item type: 7 (heart) | Item added successfully |
| Add second item to inventory | Player ID: 1  Game ID: 1  Item type: 8 (bigger blade) | Item added successfully |

#### C# Interaction and GUI

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Game screen: | Shows current inventory items Items picked up automatically when moving onto them Heart items can be used to heal Bigger blade items improve mowing range | |
| Inventory system: | Items collected by walking over them Heart: Stored if health is full, used if health low Bigger blade: Increases mowing area Inventory persists during game Items counted individually | |
| Success: | Item added to inventory  display item | |
| Screenshot: (example of score update) | Pick up heart when full health(store it in inventory)    Then hit a rock( lose the heart item instead of losing 1 hp)    Pick up heart when not full health(recover 1 hp)    Pick up bigger blade: | |

|  |  |
| --- | --- |
| Exception handling | |
| DAO | **GameDao.cs** |
| Connection check |  |
| Exception block |
| Test code |  |

## Update player data

**MySQL file line 342 to 373**

#### MySQL Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Results | Screenshot | Test data | Pass/fail |
| Update player data with a non-existent username. | Player data is updated. |  | **CALL UpdatePlayerData(1, '12344', '12344');** | **pass** |
| Update player data with an existing username. | Operation fails.  An error message is returned :  the username already exists. |  | **CALL UpdatePlayerData(1, '123', '12344');** | **pass** |

C# Test (UserDao)

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Test data | Expected: | Screenshot |
| Valid update | Player ID: 1  New username: 1234  New password: 1234 | Player data updated | A black screen with white text  Description automatically generated |
| Invalid update | Player ID: 999  New username: BadUser  New password: pass | Player not found |

#### C# Code

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Profile screen: | Players can update their username and password. | |
| Update button : | Calls PlayerDao.UpdatePlayerData method with the new data.  Displays messages based on the outcome. | |
| Success: | Success message | |
| Failure: | Displays an error message | |
| Screenshot: (example of existing username error) |  | |

|  |  |
| --- | --- |
| Exception handling | |
| DAO | UserDao.cs |
| Connection check |  |
| Data validation |
| Exception block |
| Test code |  |

## Kill running games procedure

**MySQL file line 374 to 401**

#### MySQL Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Results | Screenshot | Test data | Pass/fail |
| Stop an existing game | Game stopped successfully.  End time updated in database. |  | **CALL StopGame(1);** | **pass** |
| Stop a non-existent game | Operation fails. Error message: Game not found. |  | **CALL StopGame(999);** | **pass** |

#### C# Test (AdminDao)

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Test data | Expected: | Screenshot |
| Valid game termination | Game ID: 1 | Game stopped successfully | (see AdminDao for testing code. Code is run in the program file) |
| Invalid game termination | Game ID: 999 | Game not found Error message returned |

C# Interaction and GUI

#### C# Code

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Admin screen: | Show a list of active games  Admin can select game to terminate  ‘Kill Game’ button stops selected game | |
| Kill game process: | Select game from the active games list  Click Kill Game button  Game is stopped | |
| Success: | Game stopped message shown  Game removed from active games list | |
| Failure: | No game selected: Error message  Invalid game ID: Error message | |
| Screenshot: |  | |

|  |  |
| --- | --- |
| Exception handling | |
| DAO | AdminDao.cs |
| Connection check |  |
| Exception block |
| Test code |  |

## Add new player procedure

(I used the same register procedure to add a player, it made sense to me)

**MySQL file line 292 to 303**

**MySQL Test**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Results | Screenshot | Test data | Pass/fail |
| Register with a non-existent username | Registration is successful |  | CALL Register('TestUser1', 'pass123'); | **pass** |
| Register with existing username | Registration fails  Error message: Username already exists. |  | CALL Register('123', '123'); | **pass** |

#### C# Test (UserDao)

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Test data | Expected: | Screenshot |
| Valid Registration | Username: user123 Password: pass123 | Player added to database | (see UserDao for testing code. Code is run in the program file) |
| Invalid Registration | Username: 123 Password: pass123 | Registration fails username already exists |

#### C# Interaction and GUI

|  |  |
| --- | --- |
|  |  |
| Admin screen: | Admin can add player from admin screen. |
| Add player/Register process: | Enter username and password  Click Add/Register button  System checks if username exists  If username available, creates account |
| Success: | Success message displayed |
| Fail: | Username already exists. Error message displayed |
| Screenshot: | example of existing username error and success |

|  |  |
| --- | --- |
| Exception handling | |
| DAO | UserDao.cs |
| Connection check |  |
| Exception block |
| Test code |  |

## Delete player procedure

**MySQL file line 414 to 452**

#### MySQL Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Results | Screenshot | Test data | Pass/fail |
| Delete an existing player. | Player is deleted |  | **CALL DeletePlayer(5);** | **pass** |
| Delete a non-existent player. | Operation fails.  An error message is returned :  the player does not exist or not found. |  | **CALL DeletePlayer(999);** | **pass** |

C# Test

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Test data | Expected: | Screenshot |
| Valid Delete | Player ID: 1 | Player data deleted |  |
| Invalid Delete | Player ID: 999 | Player not found |
| \*See UserDao for the test code(run in the program file) | | | |

#### C# Code

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Profile & Admin screen: | Players can delete their account in the profile screen  Admin can delete an account from the admin screen. | |
| Delete account button : | Calls UserDao.DeletePlayer(playerId); method.  (on admin screen, select a player from the player list)  Calls the DeletePlayer stored procedure.  Displays messages based on the result. | |
| Success: | Profile screen :Are you sure message and then success message and redirected to login screen.  Admin screen : Same without the redirection to login screen | |
| Failure: | Are you sure message and then displays an error message. | |
| Screenshot: | Profile screen:    Admin screen:  No player selected | |

|  |  |
| --- | --- |
| Exception handling | |
| DAO | UserDao.cs |
| Connection check |  |
| Exception block |
| Test code |  |

# Assignment 2 (stage three) PROTOTYPE V3

Refactor each TSQL procedure and function enabling correct concurrency using TSQL transaction management statements and error and exception handling facilities. For each include a set of relevant test data. LO1, LO3

#### 1. Player login (MySQL file line 202 to 290)

|  |  |
| --- | --- |
| An TSQL procedure that encapsulates the code has the correct SQL for handling concurrency exceptions: | |
| Transaction handling: | Track player status and login attempts in the database |
| The SQL statements encapsulated in the procedure produce the correct COMMIT and ROLLBACK: | |
| COMMIT case: | When login works, changes status to online and resets attempts |
| ROLLBACK case: | Not needed as updates happen one at a time |
| Calls that test the COMMIT and ROLLBACK in the SQL procedure: | |
| Test scenario 1: | CALL Login('123', '123') - Works and updates status |
| Test scenario 2: | CALL Login('Hello', 'password12345') - Shows locked account message |
| The TSQL procedure produces a result that the application can use: | |
| Success result: | Sends "Login successful" and player ID to the game |
| Failure result: | Tells the game why login failed (wrong password, locked, banned) |

#### 2. Player registration (MySQL file line 292 to 303)

|  |  |
| --- | --- |
| An TSQL procedure that encapsulates the code has the correct SQL for handling concurrency exceptions: | |
| Transaction handling: | Check username availability and create new account |
| The SQL statements encapsulated in the procedure produce the correct COMMIT and ROLLBACK: | |
| COMMIT case: | Creates new player account and logs first connection |
| ROLLBACK case: | Not needed as it checks for existing username first |
| Calls that test the COMMIT and ROLLBACK in the SQL procedure: | |
| Test scenario 1: | CALL Register('12345', 'newpass123') - Creates new account |
| Test scenario 2: | CALL Register('123', '123') - Fails as username exists |
| The TSQL procedure produces a result that the application can use: | |
| Success result: | Tells game registration worked and gives new ID |
| Failure result: | Tells game username already exists |

#### 3. Laying out game board (MySQL file line 657 to 876)

|  |  |
| --- | --- |
| An TSQL procedure that encapsulates the code has the correct SQL for handling concurrency exceptions: | |
| Transaction handling: | Creates complete game board in one transaction |
| The SQL statements encapsulated in the procedure produce the correct COMMIT and ROLLBACK: | |
| COMMIT case: | Board created with all tiles and objects placed |
| ROLLBACK case: | If any part fails, whole board creation cancelled |
| Calls that test the COMMIT and ROLLBACK in the SQL procedure: | |
| Test scenario 1: | CALL StartGame(2, 'single-player') - Makes new board |
| Test scenario 2: | CALL GenerateSinglePlayerMap(2) - Places all tiles |
| The TSQL procedure produces a result that the application can use: | |
| Success result: | Returns complete board layout to game |
| Failure result: | Tells game if board creation failed |

#### 4. Placing items on tiles (MySQL file line 657 to 876, part of map generation)

|  |  |
| --- | --- |
| An TSQL procedure that encapsulates the code has the correct SQL for handling concurrency exceptions: | |
| Transaction handling: | Places items as part of map generation transaction |
| The SQL statements encapsulated in the procedure produce the correct COMMIT and ROLLBACK: | |
| COMMIT case: | All items placed in correct positions on map |
| ROLLBACK case: | If item placement fails, map generation rolls back |
| Calls that test the COMMIT and ROLLBACK in the SQL procedure: | |
| Test scenario 1: |  |
| Test scenario 2: |  |
| The TSQL procedure produces a result that the application can use: | |
| Success result: | Map includes all items in the correct positions |
| Failure result: | Error with the item placements , no error allowed |

#### 5. Player movement (MySQL file line 877 to 1011)

|  |  |
| --- | --- |
| An TSQL procedure that encapsulates the code has the correct SQL for handling concurrency exceptions: | |
| Transaction handling: | Updates player position and handles tile effects |
| The SQL statements encapsulated in the procedure produce the correct COMMIT and ROLLBACK: | |
| COMMIT case: | Valid move completed, position updated, effects applied |
| ROLLBACK case: | Invalid move or error reverts all changes |
| Calls that test the COMMIT and ROLLBACK in the SQL procedure: | |
| Test scenario 1: | CALL MovePlayer(1, 1, 8, 5) -- Good move succeeds |
| Test scenario 2: | CALL MovePlayer(1, 1, 1, 1) --Bad move fails |
| The TSQL procedure produces a result that the application can use: | |
| Success result: | New position, health, and score returned |
| Failure result: | Keeps old position, explains why move failed |

#### 6. Game play scoring (MySQL file line 1120 to 1156)

|  |  |
| --- | --- |
| An TSQL procedure that encapsulates the code has the correct SQL for handling concurrency exceptions: | |
| Transaction Handling: | Updates score history and totals in one transaction |
| The SQL statements encapsulated in the procedure produce the correct COMMIT and ROLLBACK: | |
| COMMIT Case: | Score added and total updated correctly |
| ROLLBACK Case: | If score update fails, no points added |
| Calls that test the COMMIT and ROLLBACK in the SQL procedure: | |
| Test Scenario 1: | CALL UpdateGameScore(1, 1, 10) -- Adds pattern points |
| Test Scenario 2: | CALL UpdateGameScore(1, 1, -5) -- Adds penalty points |
| The TSQL procedure produces a result that the application can use: | |
| Success Result: | Returns new total score to game |
| Failure Result: | Keeps old score if update fails |

7. Player inventory (MySQL file line 1012 to 1052)

|  |  |
| --- | --- |
| An TSQL procedure that encapsulates the code has the correct SQL for handling concurrency exceptions: | |
| Transaction handling: | Manages inventory updates in single transaction |
| The SQL statements encapsulated in the procedure produce the correct COMMIT and ROLLBACK: | |
| COMMIT case: | Item added or quantity updated successfully |
| ROLLBACK case: | Failed updates don't affect inventory |
| Calls that test the COMMIT and ROLLBACK in the SQL procedure: | |
| Test scenario 1: | CALL AddItemToInventory(1, 1, 7) -- Adds heart |
| Test scenario 2: | CALL AddItemToInventory(1, 1, 8) -- Adds bigger blade |
| The TSQL procedure produces a result that the application can use: | |
| Success result: | Updated inventory shown to player |
| Failure result: | Keeps old inventory if update fails |

#### 8. NPC movement (Not implemented)

|  |  |
| --- | --- |
| An TSQL procedure that encapsulates the code has the correct SQL for handling concurrency exceptions: | |
| Transaction handling: | N/A |
| The SQL statements encapsulated in the procedure produce the correct COMMIT and ROLLBACK: | |
| COMMIT case: | N/A |
| ROLLBACK case: | N/A |
| Calls that test the COMMIT and ROLLBACK in the SQL procedure: | |
| Test scenario 1: | N/A |
| Test scenario 2: | N/A |
| The TSQL procedure produces a result that the application can use: | |
| Success result: | Gnome exists just as a static obstacle currently |
| Failure result: | N/A |

#### 9. Kill running games (MySQL file line 374 to 401)

|  |  |
| --- | --- |
| An TSQL procedure that encapsulates the code has the correct SQL for handling concurrency exceptions: | |
| Transaction handling: | Stops games safely with transaction |
| The SQL statements encapsulated in the procedure produce the correct COMMIT and ROLLBACK: | |
| COMMIT case: | Game ended with proper end time recorded |
| ROLLBACK case: | If game not found, no changes made |
| Calls that test the COMMIT and ROLLBACK in the SQL procedure: | |
| Test scenario 1: | CALL StopGame(1) -- Ends existing game |
| Test scenario 2: | CALL StopGame(999) – Fails, missing game |
| The TSQL procedure produces a result that the application can use: | |
| Success result: | Confirms game stopped successfully |
| Failure result: | Reports if game wasn't found |

#### 10. Add new players (MySQL file line 292 to 303, I used the Register procedure)

|  |  |
| --- | --- |
| An TSQL procedure that encapsulates the code has the correct SQL for handling concurrency exceptions: | |
| Transaction handling: | Creates new player accounts safely |
| The SQL statements encapsulated in the procedure produce the correct COMMIT and ROLLBACK: | |
| COMMIT case: | New player account created successfully |
| ROLLBACK case: | Username exists, no account created |
| Calls that test the COMMIT and ROLLBACK in the SQL procedure: | |
| Test scenario 1: | CALL Register('TestUser1', 'pass123') -- Creates account |
| Test scenario 2: | CALL Register('123', '123') -- Fails as name exists |
| The TSQL procedure produces a result that the application can use: | |
| Success result: | Returns new player ID and success message |
| Failure result: | Returns clear error about username conflict |

#### 11. Update player data (MySQL file line 342 to 373)

|  |  |
| --- | --- |
| An TSQL procedure that encapsulates the code has the correct SQL for handling concurrency exceptions: | |
| Transaction handling: | Updates player details safely in transaction |
| The SQL statements encapsulated in the procedure produce the correct COMMIT and ROLLBACK: | |
| COMMIT case: | Player data updated if username available |
| ROLLBACK case: | Rolls back if username taken or player not found |
| Calls that test the COMMIT and ROLLBACK in the SQL procedure: | |
| Test scenario 1: | CALL UpdatePlayerData(1, '12344', '12344') -- Updates data |
| Test scenario 2: | CALL UpdatePlayerData(1, '123', '12344') -- Fails |
| The TSQL procedure produces a result that the application can use: | |
| Success result: | Confirms update successful |
| Failure result: | Explains why update failed |

#### 12. Delete player (MySQL file line 414 to 452)

|  |  |
| --- | --- |
| An TSQL procedure that encapsulates the code has the correct SQL for handling concurrency exceptions: | |
| Transaction handling: | Removes all player data in single transaction |
| The SQL statements encapsulated in the procedure produce the correct COMMIT and ROLLBACK: | |
| COMMIT case: | Player and all related data removed completely |
| ROLLBACK case: | If any deletion fails, all changes reversed |
| Calls that test the COMMIT and ROLLBACK in the SQL procedure: | |
| Test scenario 1: | CALL DeletePlayer(5) -- Deletes player successfully |
| Test scenario 2: | CALL DeletePlayer(999) - Fails ,invalid ID |
| The TSQL procedure produces a result that the application can use: | |
| Success result: | Confirms player deleted |
| Failure result: | Report that the player wasn't found |

In your C# .Net application refactor each GUI and DAO method to include error and exception handling facilities, and to test error handling in the GUI through to connections to the database. For each include a set of relevant test data. LO2

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*See section : SQL procedures and C# (milestone 2 & 3) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Multiuser Concurrency (Milestone 3)

#### Concurrency management in MySQL and .NET C#

When working on my game project, I learned about different ways MySQL and .NET C# handle multiple users trying to access or change data at the same time. This is important because without proper management, players could interfere with each other's actions or corrupt the game data.

MySQL manages concurrent access through what they call transaction isolation levels. When I implemented my stored procedures, I used transactions to make sure operations like player movement or inventory updates happened correctly even with multiple players. For instance, when one player picks up an item, MySQL's transaction system ensures another player can't grab the same item at the same time (Microsoft, 2023). This helped me avoid problems in my game where two players might try to do conflicting actions.

The way MySQL handles this is different from how .NET C# approaches the same problem. In .NET, developers can choose between two main strategies: optimistic and pessimistic concurrency (Microsoft, 2023). Optimistic concurrency assumes conflicts won't happen often, like expecting players won't usually try to move to the exact same spot at the same time. Pessimistic concurrency is more cautious and locks data while it's being used, like temporarily preventing other players from interacting with a tile while one player is moving onto it.

In my project, I decided to use MySQL's transaction management for handling concurrent actions. I found this worked better because it kept all the rules about who can do what directly in the database. This made it easier to make sure everything stayed consistent, especially when multiple players are moving around the game board or collecting items at the same time. The database would automatically prevent any conflicts that could mess up the game state (Oracle, 2023).

The main difference I noticed is that MySQL's approach handles everything at the database level, while .NET gives programmers more options to handle concurrency in their code. MySQL's way was simpler for me to implement in my game, but .NET's approach might work better for other types of applications where you need more control over how concurrent actions are managed.

Through working on this project, I learned that both platforms have good ways to handle concurrent access, but they're designed for different situations. MySQL's transaction system worked well for my game because it automatically handled most of the complicated parts of making sure multiple players could play together without problems.

# References:

Microsoft. (2023). Transaction and concurrency in SQL Server. SQL Server Documentation.

<https://learn.microsoft.com/en-us/sql/relational-databases/>

Microsoft. (2023). Handling concurrency in .NET applications. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/framework/>

Oracle. (2023). MySQL transaction isolation levels. MySQL Documentation.

<https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html>

Vecteezy. (2024) Lawn mowing vectors. Vecteezy.

<https://www.vecteezy.com/free-vector/lawn-mowing>

PNG All. (2024). Lawn mower PNG.

<https://www.pngall.com/lawn-mower-png/download/134075/>

**C# code:**  
Branx. (2023). Graphics user interface to create a simple Windows Forms countdown timer in C#.

<https://steemit.com/utopian-io/@branx/c-graphics-user-interface-to-create-a-simple-windows-forms-countdown-timer-in-c>

ByteHide. (2023). Timer in C#.

<https://www.bytehide.com/blog/timer-csharp>

Stack Overflow. (2023). How to convert seconds in min:sec format.

<https://stackoverflow.com/questions/3665012/how-to-convert-seconds-in-minsec-format>

Microsoft. (2023). Walkthrough: Creating and running unit tests for managed code. Visual Studio Documentation.

<https://learn.microsoft.com/en-us/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code?view=vs-2022>

W3Schools. (2023). C# inheritance.

<https://www.w3schools.com/cs/cs_inheritance.php>

Microsoft. (2023). C# constructors. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/constructors>

W3Schools. (2023). C# constructors.

<https://www.w3schools.com/cs/cs_constructors.php>

Microsoft. (2023). Create a C# WinForm in Visual Studio. Visual Studio Documentation.

<https://learn.microsoft.com/en-us/visualstudio/ide/create-csharp-winform-visual-studio?view=vs-2022>

Microsoft. (2023). Get started with Windows Forms. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/desktop/winforms/get-started/create-app-visual-studio?view=netdesktop-8.0>

YouTube. (2022). Windows Forms tutorial

<https://www.youtube.com/watch?v=5Dm8kXii7zU>

YouTube. (2022). Windows Forms design basics

<https://www.youtube.com/watch?v=JSJ1JI2alJg>

CodeProject. (2023). Designing the layout of Windows Forms using a TableLayoutPanel.

<https://www.codeproject.com/Tips/842418/Designing-the-Layout-of-Windows-Forms-using-a-Tabl>

Microsoft. (2023). Colors in Windows Forms. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/api/system.windows.media.colors?view=windowsdesktop-9.0>

Microsoft. (2023). ColorDialog in Windows Forms. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/api/system.windows.forms.colordialog?view=windowsdesktop-9.0>

Microsoft. (2023). Generic Dictionary in C#. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-9.0>

Microsoft. (2023). Timer in Windows Forms. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/api/system.windows.forms.timer?view=windowsdesktop-9.0>

Microsoft. (2023). Walkthrough: Arranging controls on Windows Forms using a TableLayoutPanel. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/desktop/winforms/controls/walkthrough-arranging-controls-on-windows-forms-using-a-tablelayoutpanel>

Microsoft. (2023). Convert.ToInt32 method in C#. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/api/system.convert.toint32?view=net-9.0>

Microsoft. (2023). How to convert a string to a number in C#. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/types/how-to-convert-a-string-to-a-number>

FreeCodeCamp. (2023). How to convert a string to an integer in C#.

<https://www.freecodecamp.org/news/how-to-convert-a-string-to-an-integer-in-c-sharp/>

GeeksforGeeks. (2023). C# Convert.ToInt32(string, IFormatProvider) method.

<https://www.geeksforgeeks.org/c-sharp-convert-toint32string-iformatprovider-method/>

GeeksforGeeks. (2023). Virtual functions in C++.

<https://www.geeksforgeeks.org/virtual-function-cpp/>

Microsoft. (2023). Virtual functions in C++. .NET Documentation.

<https://learn.microsoft.com/en-us/cpp/cpp/virtual-functions?view=msvc-170>

Stack Overflow. (2023). Why do we need virtual functions in C++?.

<https://stackoverflow.com/questions/2391679/why-do-we-need-virtual-functions-in-c>

Microsoft. (2023). Partial classes and methods in C#. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods>

GeeksforGeeks. (2023). Partial classes in C#.

<https://www.geeksforgeeks.org/partial-classes-in-c-sharp/>

Microsoft. (2023). Override keyword in C#. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override>

Stack Overflow. (2023). How to override a void method in C#?.

<https://stackoverflow.com/questions/73986921/how-override-void-method>

Microsoft. (2023). Protected keyword in C#. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/protected>

Microsoft. (2023). Internal keyword in C#. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/internal>

Microsoft. (2023). DockStyle enumeration in Windows Forms. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/api/system.windows.forms.dockstyle?view=windowsdesktop-9.0>

**MySQL:**  
  
W3Schools. (2023). SQL JOIN.

<https://www.w3schools.com/sql/sql_join.asp>

W3Schools. (2023). SQL Alias.

<https://www.w3schools.com/sql/sql_alias.asp>

W3Schools. (2023). SQL Tutorial.

<https://www.w3schools.com/sql/default.asp>

W3Schools. (2023). SQL WHERE.

<https://www.w3schools.com/sql/sql_where.asp>

W3Schools. (2023). SQL ORDER BY.

<https://www.w3schools.com/sql/sql_orderby.asp>

W3Schools. (2023). SQL EXISTS.

<https://www.w3schools.com/sql/sql_exists.asp>

W3Schools. (2023). SQL CASE.

<https://www.w3schools.com/sql/sql_case.asp>

W3Schools. (2023). SQL Stored Procedures.

<https://www.w3schools.com/sql/sql_stored_procedures.asp>

W3Schools. (2023). SQL COUNT.

<https://www.w3schools.com/sql/sql_count.asp>