Proposed Entity-Relation Diagram for MayAztec project

Profesor: Esteban Castillo Juárez

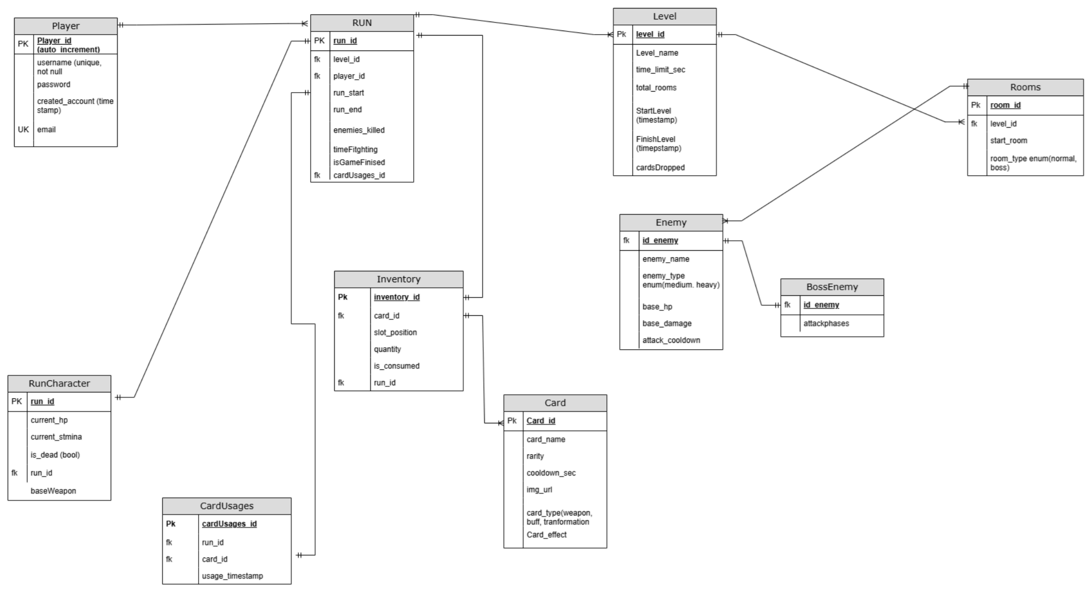
Materia: Construcción de Software para la toma de decisiones

Nombres:

- Mauricio Emilio Monroy González - A01029647
- Héctor Lugo Gabino - A01029811
- Nicolás Quintana Kluchnik- A01785655

Fecha: 11 de abril del 2025

<u>Justification</u>



Notes changes (version 2 changes):

Players - this table was deleted, because it made no sense to store players in another table, since they are already stored in the player table.

Chest - The chests table, we decided that it is not necessary to have it stored in the database, because we can manage it in a simpler way from the front-end. And having it in the database may be something not so necessary, because they are only likely to appear and have certain types of cards.

Added Run tables and card usages.

Remove enemy_card_drop.

- Justification: Drop probabilities are game rules, not critical data requiring persistence (e.g., do not affect player history).
- Simplifies the database schema and reduces the number of tables.

        - All this will be handled from the front end of the game, since there are probabilities that will remain fixed after local testing, we consider that it is better to handle it directly in the game attributes.

Simplify Enemy table.

Due to the decision to handle probabilities in the front end of the game, to simplify the testing and development process. These enemy related attributes should be removed.

Remove card_Effects.

Because we manage effects, usage times, and probabilities. We consider pertinent to eliminate the cardEffects table, since these will be implicit in the creation of the objects of each card in the front end of the game, simplifying its implementation without having to make a request to the database to instantiate them in the game.

Benefits of the Changes

- Performance: By reducing the number of database queries during the game, we improve the user experience.
- Simplicity: A simpler schema facilitates maintenance and reduces the probability of errors.
- Flexibility: Separation of game rules (frontend) and persistent data (backend) allows faster adjustments to the roll without changes to the database structure.
- Data Analysis: New RUN and CardUsages tables provide a solid foundation for statistical analysis, enabling improvements based on real game data.

In our proposed ER model for our game MayAztec, we proposed 11 Tables with no intermediate tables.

Main entities and their attributes:

Player

- This table manages user account information.
- Attributes:
    - player_id (int auto_increment) PK

  - username (varchar)
  - password (varchar)
  - date_create (time_stamp)
  - email (varchar) UK
- Relations:
  - One-to-one with RUN:
    - This relationship allows multiple games to be registered for the same player, implementing the “runs” system, characteristic of roguelites.
    - The player is connected to the Run table that contains all the records he made during a game, and having access to each run, he can also indirectly know the statistics of his playable character and the world of each run.
- Normalization Justification:
  - 1FN Each column stores a single atomic value. For example, the email field contains a single address and username is a single string.
  - 2FN Since the primary key is player_id (unique field), all other attributes are completely dependent on that identifier. There are no composite keys, so there are no partial dependencies.
  - 3FN Each non-key attribute depends only on player_id and there is no attribute that depends on another non-key attribute.

RUN

- This table will serve as a support, to save the complete information of the level, defeated enemies, combat time, if I finished the game or not, how long it took to complete it.
- Attributes:
  - run_id (PK) auto_increment
  - level_id(fk)
  - player_id(fk)
  - run_star (time_stamp)
  - run_finised (time_stamp)
  - timefighting (int)
  - isGameFinished (bool)
- Relations:

- One to many with player
    - It will allow the player to generate statistics through the run game. This will allow us to create specific views of the runs.
- One to many with level
    - A room can have many levels. In our case 2 levels. This allows us to track the entire game of player u and not just one level.

Normalization Justification:

Making everything in the game depend on the run table allows to avoid redundancies, and circular dependencies.

- Complies 2NF: All attributes depend on the complete PK (run_id)
- A player can have multiple independent games (runs)
- Records the full life cycle of each run

RunCharacter

- Purpose**:** Store in-game character attributes (Hp, stamina, base weapon) associated with a character
- isDeath will be used to know whether or not I passed a level or a room. Thanks to combining the time stamp with this attribute we will be able to make summary screens, because it is a bool that indicates the state of the character.
- The structure of how the relationships are connected was changed.
    - First the character depends completely on the run, even if you restart the game and start with “the same stats of your character”, it changes from run to run how it is managed. And it does not remain constant as it was proposed before.
    - Secondly, it is now connected to your inventory through the run_id since both depend on this specific run to differentiate between run and run.

- Attributes:
    - run_id (PK) auto_increment
    - current_hp (tiny int)
    - current_stamina (tiny int)
    - isDeath
- Relations:

    - One-to-one with Run
        - The character you control depends on the run. Since you will not always get the same attributes throughout the run. This is why it depends on the run and not on the player, since the player only plays the runs, but the character does not depend on the player but rather on the game (run).

Normalization Justification:

- 2NF Each game is independent; therefore the attributes of the character may vary and depend entirely on the run (complying 2FN: all data depend on the key, which can be a unique identifier or the run_id).
- 3NF - has no partial dependencies on another table

Inventory:

- Purpose: Store the card that the character has in his "deck" or available slots.
- The limited quantity allows the gameplay to be strategic for the player, without abusing the cards. And have a centralized space for this field

- Attributes:
    - inventory_id (int) PK
    - card_id (int) FK to Card
    - is_consumed (boolean)
    - slot_position
- Relations:
    - One to one with Run:
        - Each inventory depends on the run, because the cards you use will be how the inventory of that run is managed.
    - One Mandatory-to-Many Mandatory with Card:
        - An inventory can contain different cards and this inventory is unique for each run.

Normalization Justification:

- 1NF: Each row represents a single card in the inventory of a run

- two other attributes (such as is_consumed) are completely dependent on inventory_id (fulfilling 2FN and 3FN) because that field is the primary key of the table.
- Eliminates redundancies by separating the temporary (run) from the permanent (player) status
- Having your own primary key of inventory_id allows for no duplicate data, since even if there are two cards of the same type omas, integrity rules are not violated.

Card

- Purpose: To model each lottery card. The games revolve around the cards, these are the power ups of the adventure.
- Attributes:
    - Card_id (int) PK
    - card_name (varchar)
    - description (varchar)
    - rarity (enum("low_tier, high_tier))
    - cooldown_sec (smallint)
    - img_url
    - cardEffect
- Relations:
    - Many Mandatory-to-One Mandatory with Inventory
        - Allows many cards (5 plus base card weapon) to be attributed to only one inventory

Annotations:

- img_url: This field stores the path to the image that represents the chart graphically in the game.

- Normalization Justification:
    - 1FN Each attribute stores a single value. For example, rarity is a single value defined by the ENUM.
    - 2NF All attributes depend entirely on card_id, since each card is an independent entity.
    - 3FN There are no derived attributes (there is no field whose information is redundant or calculated from another non-key attribute).

CardUsages

- **Purpose:** This card gives us the possibility to generate more accurate statistics, which otherwise could not be generated. Since the inventory only serves as a connection between the player and the cards, but does not count specifically which card was used. The purpose of this table is that when we want to know which are the most used cards by the players, it would give us an idea of how they use them, also if a query is made with the drop rate we can generate inferences. This table is a record to generate statistics.
- Attributes:
    - cardUsages_id
    - run_id
    - card_id
- Relations:
    - Run id one to one:
        - In card usages the redudnacia with player and inventory was eliminated because they all share a unique run_id for that moment, since having the character_id and run_id provided the same information, the same with the inventory, that's why the redundant relations were limited.
- Normalization Justification:
    - CardUsages uses a unique identifier for each usage (usage_id) and stores FK to Character, Inventory and Card.
    - In this way, the individual usage is captured without relying on a complex composite key, staying in 1FN, 2FN and 3FN.

Enemy

- To store the data of each enemy. Centralizes the common information of all enemies.
- The important attributes of the enemy as its attackcooldown is to not attack you repeatedly, detection range is to switch between pursuit and dispersion modes a very simple implementation of a finite state machine.
- Its attributes like base_hp, damage, are fundamental to instantiate the enemies with these base values for each one.

- Attributes:
  - id_enemy (int) PK
  - enemy_name (varchar)
  - enemy_type (enum("medium", "heavy", "boss"))
  - base_hp (tinyint)
  - base_damage (tinyint)
  - detection_range (smallint)
  - attack_cooldown (float)
- Relations:
  - One-to-one with enemy_card_drop:
    - This relationship implements the probabilistic drop system. Each enemy can drop several cards with different probabilities.
    - **Constraints:** Foreign key ensure referential integrity.
  - One-to-one with Boss_enemy:
    - This is a specialization relationship where bosses are a special type of enemy with additional attributes. It allows reusing common attributes while adding specific characteristics.
    - **Constraints**: the **primary key** of Boss_enemy is also a **foreign key** that references Enemy, ensuring that only some enemies are bosses.
  - Many Mandatory-to-One Mandatory with Rooms

Annotations:

  - detection_range: Defines the distance at which the enemy can detect the player, crucial for AI systems and enemy behavior.
  - attack_cooldown: Represents the time the enemy must wait between attacks, balancing the difficulty of the game.
- Normalization Justification:
  - 1FN Each attribute is atomic: enemy_name is a unique string, enemy_type is a single value, etc.
  - 2FN Since the primary key is unique (enemy_id), all attributes depend entirely on this value, recording the enemy's base information.

- 3FN There are no transitive dependencies; the classification and characteristics of an enemy are defined independently of other attributes.

Level

- Purpose: define game levels. Is a way to persist the configuration of each level required.
  - Timestamps are useful to know how much time was spent in each level.
  - The count of how many cards you drop can help us to know the difference of cards used vs. cards left unused or uncollected.
- Attributes
  - level_id (int) PK
  - name (varchar)
  - time_limit_sec (float)
  - total_rooms (tinyint)
  - startLevel (timeStamp)
  - finishLevel (timestamp)
- Relations:
  - One-to-one with Player_Runstats:
    - In a level a set of statistics is generated to the player, and that is unique for that run.
  - One Mandatory-to-Many Mandatory with Rooms:
    - One level contains many rooms

Annotations:

- time_limit_sec: This field sets a time limit to complete the level, adding pressure to the player and an additional element of challenge.
- total_rooms: Defines the number of rooms in a level, allowing to generate levels of different complexity.
- room_type: Classifies the rooms (combat, boss), allowing to create varied and progressive experiences.
- card_drops was removed as it can generate confusion as this is handled with chests and enemies through the front end.

Rooms

- Purpose: store rooms whitin the level, each room its own room-type. A level can have multiple rooms. This facilitates the generation of dungeons and the assignment of enemies to each room.
- The rooms, for example, are fundamental to know which cards to use in the boss room. It helps us to know specific things about the rooms of the level.

- Attributes
  - room_id (int) PK
  - level_id (int) FK to Level
  - room_type (enum("normal", "boss"))
- Relations:
  - Many Mandatory-to-One Mandatory with Level
  - One Mandatory-to-Many Mandatory with Enemy
  - One-to-one with Chest

Boss Enemy

- Purpose: Separates to empathize with specialization from a normal enemy to one with boss attributes.
- Attributes
  - id_enemy (int) FK
  - attack_phases (tinyint)
  - phase_threshold (int)
  - bendition_reward (int) FK to Card_effec
- Relations
  - One-to-one with Enemy
    - It is a derivative of a normal enemy, containing unique attributes, phases of attacks.
- Normalization Justification:
  - The separation avoids storing null or unnecessary attributes in the Enemy table for normal enemies.
  - Normalization is met by having all BossEnemy attributes depend on the key (id_enemy, which is PK and FK), avoiding redundancies.

1. First Normal Form (1NF): Atomic Attributes

Compliance:

- All attributes store indivisible values. For example:
    - card_type in Card uses an ENUM(“weapon”, “transformation”, “buff”), which defines unique and specific values.
    - enemy_type in Enemy classifies enemies as “medium”, “heavy”, or “boss”, without mixing multiple values in one field.

2. Second Normal Form (2NF): No Partial Dependencies

Compliance:

- All tables have simple primary keys (e.g. player_id, run_id), and non-key attributes are fully dependent on them.
- Example in Card_effect:
    - effect_id is the PK, and attributes like effect_value or duration depend only on it, not on card_id (which is a FK to link effects to cards).

Points where IA helped:

Normalization of Tables:

- Proposal to remove current_size from Inventory (violated 3NF as derived data)
- Suggestion to use composite PK (run_id + card_id) in Inventory to avoid artificial IDs.
- Correction of relationships in CardUsages (remove redundant character_id and inventory_id).

Inventory table:

- I proposed inventory_id for unique tracking.
- The AI suggested restriction UNIQUE(run_id, card_id) to avoid unwanted duplicates.
-

RUN-RunCharacter relationship:

- I defined that the character state depends 100% on the run.
- The AI validated that this meets 2NF/3NF by eliminating Player dependencies.

The importance of normalization and how generative tools helped improve the model:
Normalization is important since it helps with data organization, its efficiency, reduces redundancy, and ensures data integrity within the database. Since we broke down larger tables into smaller ones, this makes them related and allows the database to be more scalable and easier to maintain
Generative language tools helped us improve the database as it gave us multiple suggestions for structuring, identifying missing relations and/or attributes, helping us clarify the ambiguity of the design. Said tools can also assist in explaining complex database concepts, making it easier for developers to understand, refine, and optimize the models.