# Infernus

An Adventure Based MMORPG

*(Massively Multiplayer Online Rabbit Playing Game)*

Version 4.0

Cody Dillon, Enbai Kuang, Kyle Lotterer, Max Peaslee

## 1 Introduction

Team Cybermen intends to create a database that is used by an MMO game company in order to store and maintain multiple aspects of each of their customers. Data includes customer data, character data, along with analytical data that explains the patterns of the user within the game. Furthermore, the database will track most aspects of the ingame character such as their inventory, bank, skills and experience progression, quest completion, and friends. The game is free to play but allows players to purchase a monthly or yearly membership in order to unlock additional premium content.

## 2 Application

The application is used by game developers and would be used to facilitate communication of real time information between the game server and the user's game client in order to properly maintain and show the information of the character when a user is playing. It will also be used to store item information and character information on the server side which will allow administrators to better maintain, locate, and manipulate the data stored. This will be essential for a massive multiplayer online game because the database will have to keep track of many records from the customer and in-game player perspective in order to make sure that all data is properly stored and can be changed if issues arise. E.g. (player cheated?)

## 3 Entities

- ACCOUNT: Id, Username, Password, Email address, Status, Player_name
- BILLING INFO: First name, Middle name, Last name, Phone number, Membership, Expiration_date, account_id
- PLAYER: Player_health, Player_level, Display_name, Is_online, Time_played, account_id
- STORED_ITEM: Owner_name, Stored_in, Item_id, Item_amount, Item_name, Item_type, Item_picture, Item_upgrade
- SKILL: Skill_rank, Skill_exp_cap, Skill_name, Skill_id
- QUEST:Quest_id, Quest_name, Quest_stages
- STATUS_CONDITION: Condition_id, Condition_effect, Condition_duration

# 4 Use Cases

## 4.1 Introduction

Queries and their functionalities will be separated by user type and security privileges, with security privileges separated into different levels, each level will have different levels of control and ability to manipulate information stored in the database. Level 1 will be the admin level with ability to directly manipulate the entities stored with the database, Level 2 would be the user themselves who can only affect their own entities, while Level 3 would be the user only being able to view limited entity values.

## 4.2 Database Views

Admin (Level 1 security privilege)

- Admin should be able to retrieve and manipulate user game entities such as user inventory, skills, status, and quest log.
- Admin should be able to make changes to user account status such as banning an account.
- Admin should be able to see all accounts whose membership expires that month.
- Admin should be able to see membership totals grouped by status.
- Admin should be able to see which accounts have the most/least items or gold.

Player(Level 2 security privilege)

- User should be able to view and  delete items from their own inventory.
- User should be able to view and forget skills in their own skill list.
- User should be able to change their account password, username, membership, billing info.
- User should be able to view a list of completed and uncompleted quests as well as able to delete uncompleted quests.

# 5 Design

## 5.1 Introduction

This database models an online multiplayer adventure game where players venture about the world to take on and accept quests, train the character's skill levels, and gather items for use. As the player ventures around, they gather items they come across. Each player also has a state, which represents their progression along quests, skills, and their own condition statuses. The game is free but has a subscription which allows players to spend money to unlock additional content.

## 5.2 Walkthrough

A user creates an account and a playable character through an account creation page on an external website. In the future, the user can subscribe to a paid membership that unlocks additional content in the game. Their billing information will be stored in the database only when this occurs and can be removed on request. In the game, the player has access to quests, split up into multiple stages (checkpoints), that can be completed. The player also can level up their skills and rank them up by performing certain actions, such as killing enemies or stealing from NPCs. Players will encounter conditions in the form of buffs and debuffs, such as poison, along their journey which can empower or weaken their character.

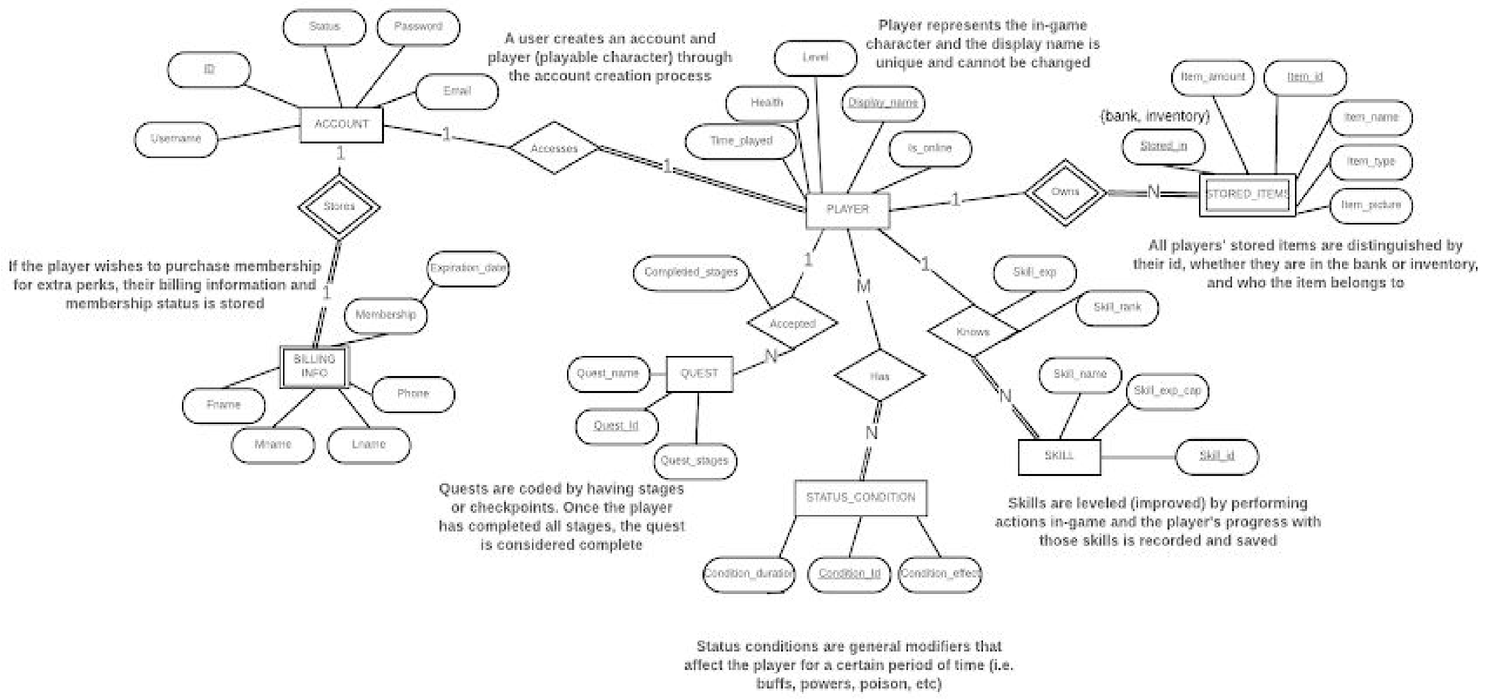
The players will have access to items that will help them as they progress through the game. They can store these items in their inventory, which has a limited size, or their bank which can be accessed inside towns in-game. These stored items will be tracked through three variables. These variables consist of who the item belongs to, what item it is (its type), and whether it is inside the inventory or the bank.

## 5.3 Design Discussion

The design has changed frequently throughout the project diagramming. We initially had a STORAGE entity that broke into weak entities such as INVENTORY_ITEMS and BANK_ITEMS, but we concluded it would be more efficient to just have everything in one table. For example, when a player picks up an item that they are not currently holding, the in-game code makes a query to the database which brings up a selected subset of items that the player owns, and creates a new tuple dedicated to that item if the player does not own it. The item_id is stored alongside the player_name as the owner, the amount that the player received of that item (i.e 453 gold pieces means item_amount is 453), and it is designated to be of type inventory.
The other attributes are filled in as assumed. We are concerned that the relation will be extremely large with a big player-base, but this diagram is assuming a relatively small player-base.

We also collectively decided that the SHIPPING entity does not make sense for our database project as they are not purchasing anything physical. We decided that BILLING entity would be just fine by itself as they are purchasing virtual membership for the account. The QUEST entity is not weak because a quest can exist without any players that have started/accepted it.

## 5.4 Entity-Relation Diagram



A user creates an account and player (playable character) through the account creation process

Player represents the in-game character and the display name is unique and cannot be changed

All players' stored items are distinguished by their id, whether they are in the bank or inventory, and who the item belongs to

If the player wishes to purchase membership for extra perks, their billing information and membership status is stored

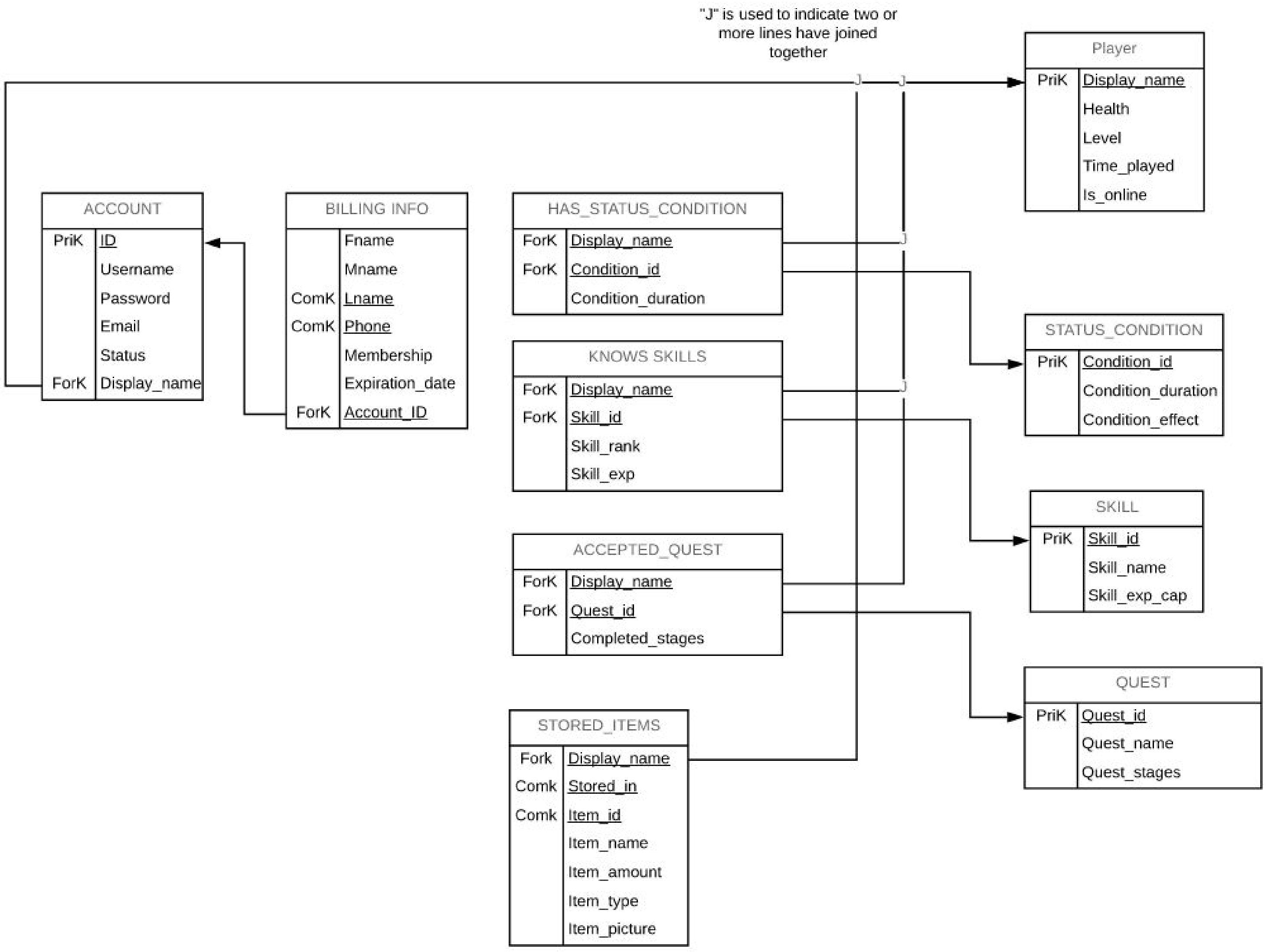
Quests are coded by having stages or checkpoints. Once the player has completed all stages, the quest is considered complete

Skills are leveled (improved) by performing actions in-game and the player's progress with those skills is recorded and saved

Status conditions are general modifiers that affect the player for a certain period of time (i.e. buffs, powers, poison, etc)

*Note: Please zoom in to see the ER Diagram in better quality (CTRL + Mouse Wheel).*

## 5.5 Relational Model Diagram



## 5.6 Domain Constraints

- Database integers are whole numbers between -2,147,483,647 and 2,147,483,647.
- All IDs are integers greater than 0.
- All names are non-empty strings.
- ACCOUNT:
  - Status is any string from {“Banned”, “Permitted”}
  - Email is any string that follows the pattern:
    - [1+ alphanumeric characters]@[1+ alphanumeric characters].[TLD]

  - Password is any non-empty strings.
- BILLING_INFO:
  - Phone is any string that follows the pattern:
    - [3 numbers]-[3 numbers]-[4 numbers]
  - Days_left is any integer between 0 and 366.
  - Status is any string from {“Yearly”, “Monthly”, “Non-member”}
- PLAYER:
  - Is_online is any boolean value.
  - Health and level is any integer between 0 and 100.
  - Timeplayed is any integer greater than 0.
- QUEST:
  - Completion_percentage is any integer between 0 and 100.
- ACCEPTED_QUEST:
  - Completion_stage is any integer greater than or equal to zero.
- STATUS_CONDITION:
  - Condition_duration is any positive integer.
  - Condition_effect is any string from {“Poison”, “Confusion”, “Weak”, “Fear”}
- HAS_STATUS_CONDITION:
  - Condition_duration is any positive integer.
- SKILL:
  - Skill_exp_cap is any positive integer.
- KNOWS_SKILLS:
  - Skill_rank is any integer between -1 and 100.
  - Skill_exp is any positive integer.
- STORED_ITEMS:
  - Stored_in can only be {“bank”, “inventory”}.
  - Item_picture is an alphanumeric string referencing a picture resource.

## 5.7 Database Assumptions

- One account will be able to have one billing information.
- A player will be able to accept multiple quests, and learn multiple skills.
- A player can be affected by multiple status effects.
- A player does not have to own any items but a stored item must be owned by a player.
- A Quest can exist without being accepted by a player
- Multiple players can be affected by the same condition, condition_duration is used in game to track how long condition lasts.
- Quest requirements are checked with in game code in by comparing to player information.
- All players will start with at least one skill, but may not be able to use them immediately. In that case, they will be at rank -1.

- Items are coded in-game, STORED_ITEMS is only used for keeping track of which player has which item. We don't include an ITEM entity because that is dealt within the actual game coding.
- STORED_ITEMS attribute stored_in refers to whether the item is stored in the bank or the inventory.
- Multiple players may have the same condition with an identical duration.
- The STORED_ITEM's picture id does not mean to store an picture, but a reference to a picture id used for the item.
- Quest_stages refers to how many stages in the quest the player must complete and used as reference on whether quest has been completed, Completion_stage = 2 means player is working on stage 3, if Quest_stages = 3 and Completion_stage = 3 then quest has been completed.
- A user may only create one character (PLAYER) per account.
- When a user creates an ACCOUNT, the PLAYER is automatically instantiated into the system (as part of the account creation requirements).

## 5.8 Change Documentation

Throughout the project our database has evolved as we delved deeper into its inner workings. While we created the actual database and set up constraints and keys, we found some problems with our original design that needed to be addressed.

For example, we only had one integer that tracked health. While we were working with the database, we realized we needed to store an integer based on their levels that determined their health while also keeping track of their current health. Now we can track the player’s max health and current health seamlessly.

Another larger example comes from the STORED_ITEM schema. Originally, we had planned to have two different schemas, one for bank and another for inventory, and have it all connect from an ITEM schema. In the end, we realized that it would be more efficient to converge them into one shared STORED_ITEM list and have an attribute that defines whether it is stored in the bank or in their inventory.

We also decided to allow items to be upgraded to add the option of item expansion to allow for some more variety. In doing so, we soon realized that we would have to update our primary key for that schema as well. Originally, we defined a stored item from the owner, id, and where it was stored. With the addition of item upgrades, trying to add an upgraded item to a default item would cause an entity constraint violation. To alleviate this problem, we included the item’s upgrade value in the primary key so items of different levels of upgrades could be stored together in the same place.

We have limited our database to having only 10 skills within the SKILL table in order to limit the amount of data that's created in our HAS_SKILL table. Each user would essentially have 10 tuples within the HAS_SKILL table and any more can possibly cause the database to have too many entries. Eg: 10,000 users with 10 skills each = 100,000 HAS_SKILL entries.

# 6 Tooling Assessment

## 6.1 DBMS

We are using **MySQL** as our database as well as using **MySQL Workbench** in order to manage our database. MySQL Workbench is extremely intuitive and allows multiple users to login into the database and manage it with different degrees of permissions.

## 6.2 User Interface Framework

We will be creating our **user interface** with **Java** and utilizing the default MySQL DB connector classes (https://www.oracle.com/technetwork/java/javase/jdbc/index.html). The DB connector interfaces extremely well with Java's swing framework which will allow us to create buttons, and forms for effective test cases and queries.

## 6.3 Cloud Hosting Platform

Our **hosting platform** will be a cloud based service that preferably has a student discount in order to use a free low-usage tier. Amazon AWS - RDS (Relational database service) https://aws.amazon.com/rds/ seems like a perfect choice for hosting the MySQL database as it's free for students (up to 750 hours a month, which equates to roughly 30 days when running 24/7).

## 6.4 Data Generation

We have **generated our data** using a free service called Mockaroo (https://mockaroo.com/). The service generates random data for many predefined formats. If the format is needed is not in their library, we can easily create our own format based on defined rules we make.

Unfortunately, some tables data had to be generated by hand as there were foreign key constraints that had to be satisfied in order to be imported into the database. A great example of this issue is with the STORED_ITEM table requiring a foreign key from the PLAYER(Display_Name). We could not see a feasible way to automatically connect the owner_name field with the rest of the fields.

# 7 Test Data

## 7.1 Sample Data

We generated the vast majority of our test data using Mockaroo. We filled in the required fields for our table's attributes and generated a series of mySQL statements through the automatic formulas. For most of our relations, our data was produced from copying and pasting from that generated file and seeing if the execution performed smoothly.

However, Mockaroo has some limitations that required us to find various workarounds for. For example, since our username must be unique, we had to find a way to generate several hundred different distinct usernames. We accomplished this by finding a generic text file full of a list of usernames online and feeding them into Mockaroo for our sample data.

Another significant difficulty that we had concerned the STORED_ITEM schema. Early in the project, we made the decision not to include an ITEM database since that could be more conveniently created and catalogued in the client side. However, since we have not yet built the code for automated item creation and assignment, we needed a temporary list of items to manually create mySQL statements from.

We created a list of twenty items on a google spreadsheet found here to use for future testing. Using that database as a reference, we were able to manually create a small set of queries that tested the logic and inner workings of our STORED_ITEMS database. In the future, a much more comprehensive sample size can be produced after UI implementation and the addition of random generation.

## 7.2 Schema/Constraint Testing

While adding in sample data for our database, we made sure to test various parts of our relations to make sure everything worked properly. After adding in successfully test data, we essentially tried to break our table by adding in incorrect values and relationships. Every time we were able to corrupt the data, we dropped and added the table and added in a new constraint to prevent it from happening again.

On the inverse side, we wanted to make sure there were no errors with current constraints and relationships as well. We also submitted test queries designed to break constraints such as setting multiple players for one account and trying to duplicate items in STORED_ITEM. Most of the time, these constraints worked as expected. However, some of the problems will have to be dealt with on the client side. For example, when a player tries to add gold to their bank when they already have gold in it, it should be done with an update statement through client-side logic instead of an insert statement.

# 8 Normalization

## 8.1 Initial Assessment

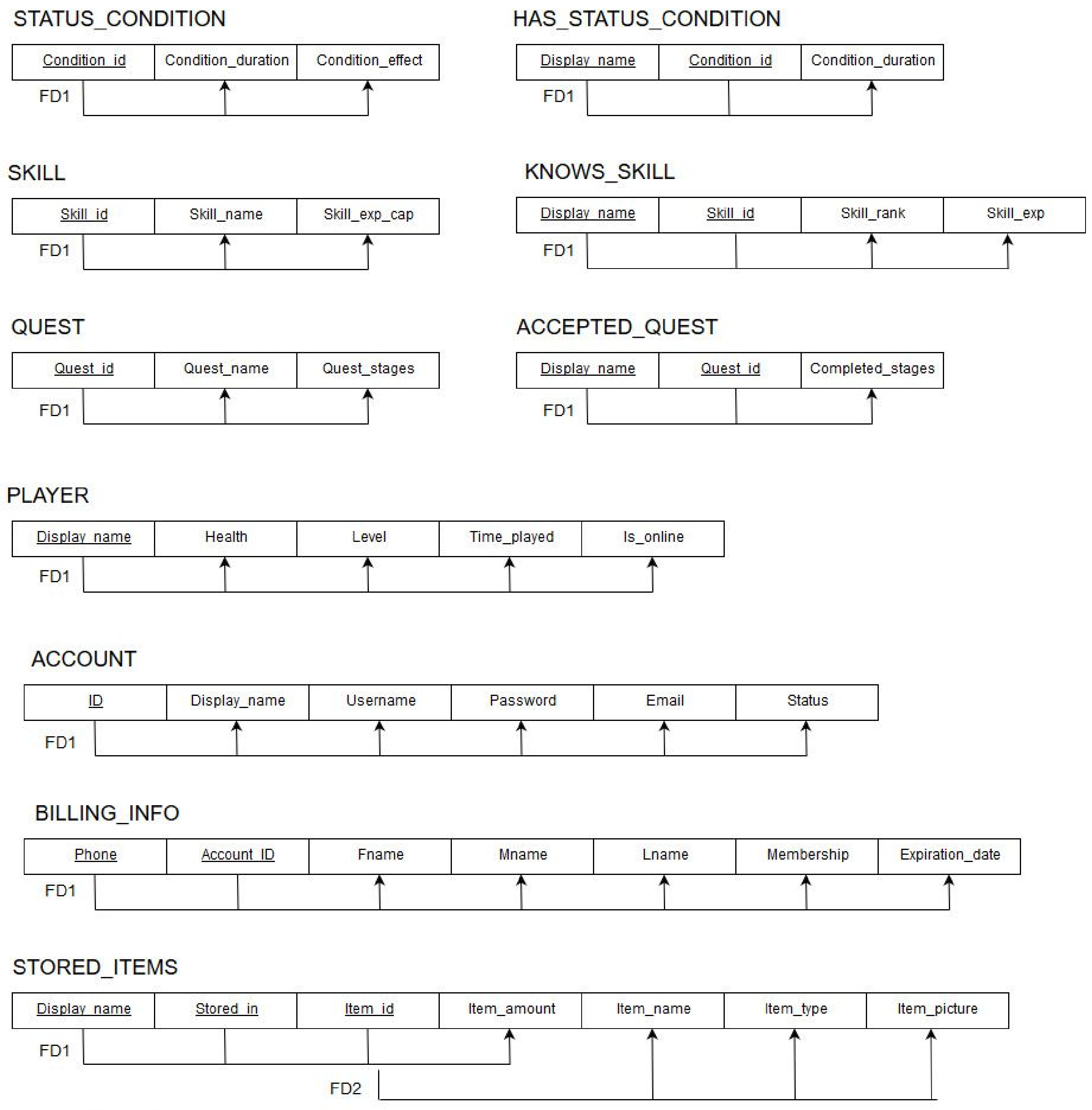
We did really well with creating the tables at the beginning of the quarter without knowing about normalization. All but one of our tables (STORED_ITEMS) is already in the 3rd normal form. These tables have dependencies that are fairly straight forward. There are no multi-valued attributes, so they are in the 1st form. Most tables have a single primary key attribute, so we are not concerned with partial key dependencies. And all the non-prime attributes are dependent on the primary key, so these tables are in the 2nd form. As well, there are in the 3rd normal form because there are no transitive dependencies.  Here are all the tables, except STORED_ITEMS, and dependencies:

- **STATUS_CONDITION**: {Condition_id}-->{Condition_duration, Condition_effect}
    - This Condition_duration is the time that all conditions of this effect last for.
- **HAS_STATUS_CONDITION**: {Display_name, Condition_id}-->{Condition_duration}
    - This Condition_duration is the time left for the player to have that condition. For example, a player will be in the poisoned state for 10 more seconds.
- **SKILL**: {Skill_id}-->{Skill_name, Skill_exp_cap}
- **KNOWS_SKILL**: {Display_name, Skill_id}-->{Skill_rank, Skill_exp}
    - Skill_rank cannot be derived entirely from Skill_exp because a player's rank can be boosted temporarily.
- **QUEST**: {Quest_id}-->{Quest_name, Quest_stages}
- **ACCEPTED_QUEST**: {Display_name, Quest_id}-->{Completed_stages}
- **PLAYER**: {Display_name}-->{Health, Level, Time_played, Is_online}
- **ACCOUNT**: {ID}-->{Display_name, Username, Password, Email, Status}
- **BILLING_INFO**: {Phone, Account_ID}-->{Fname, Mname, Lname, Membership, Expiration_date}

STORED_ITEMS is only in the 1st normal form. It's attributes are single and atomic. It is not in the 2nd normal form, however, because some of the non-primary attributes are not fully dependent on the primary key. The primary key only uniquely identifies item_amount because a player can only have one amount of an item in a container ("Bank" or "Inventory"). This leaves {Item_name, Item_type, Item_picture} which are functionally dependent on the primary key {Display_name, Stored_in, Item_id}, but not fully. The dependency still holds if we remove {Display_name, Stored_in} from the left side. Here are the full dependencies for STORED_ITEMS:

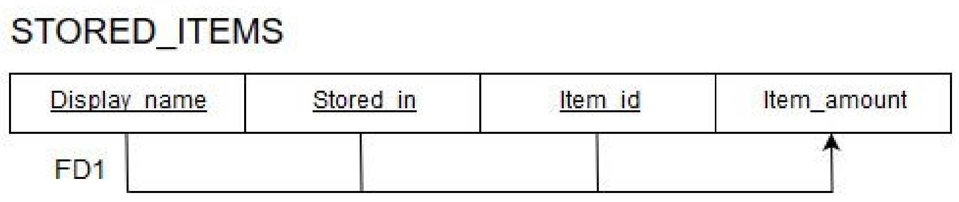
- **FD1**: {Display_name, Stored_in, Item_id}-->{Item_amount}
    - Item_amount is uniquely identified by a player(Display_name) who stores an item(Item_id) in a certain container(Stored_in).

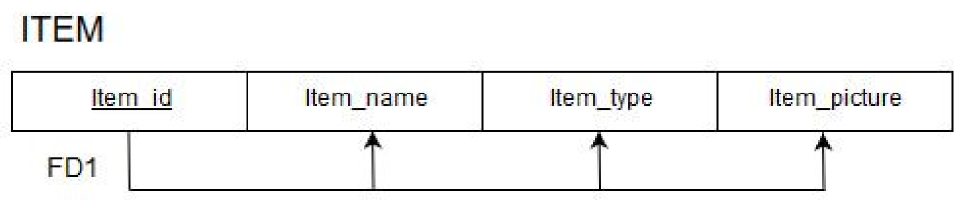
- **FD2**: {Item_id}-->{Item_name, Item_type, Item_picture}


STATUS_CONDITION
Condition_id | Condition_duration | Condition_effect
FD1
HAS_STATUS_CONDITION
Display_name | Condition_id | Condition_duration
FD1
SKILL
Skill_id | Skill_name | Skill_exp_cap
FD1
KNOWS_SKILL
Display_name | Skill_id | Skill_rank | Skill_exp
FD1
QUEST
Quest_id | Quest_name | Quest_stages
FD1
ACCEPTED_QUEST
Display_name | Quest_id | Completed_stages
FD1
PLAYER
Display_name | Health | Level | Time_played | Is_online
FD1
ACCOUNT
ID | Display_name | Username | Password | Email | Status
FD1
BILLING_INFO
Phone | Account_ID | Fname | Mname | Lname | Membership | Expiration_date
FD1
STORED_ITEMS
Display_name | Stored_in | Item_id | Item_amount | Item_name | Item_type | Item_picture
FD1
FD2

## 8.2 Decomposition

We only need to decompose STORED_ITEMS. The problem with this table is that it groups together items and their details with the container, which makes sense at face value. But, it is necessary to separate the container from the item's details. This was done by breaking it into two tables:





Now we have the following dependencies:

- **STORED_ITEMS**: {Display_name, Stored_in, Item_id}-->{Item_amount}
- **ITEM**: {Item_id}-->{Item_name, Item_type, Item_picture}

These tables are in the 2nd normal form now because we removed the partial key dependency. {Item_name, Item_type, Item_picture} are fully dependent on the primary {Item_id}. And since there are no transitive dependencies, these tables are in the 3rd normal form too.

# 9 Query Statements

## 9.1 Query Table

**Admin View:**

| **Query** | **Purpose** |
|---|---|
| select P.Display_name, P.Time_played<br>from PLAYER P<br>join ACCOUNT A on A.Player_name = P.Display_name<br>where A.Status = 'Banned'<br>order by P.Time_played desc; | Show most loyal users that are banned. An admin could review their case since it is unlikely they intentionally tried to get banned. |
| select Q.Quest_name, count(*)<br>from QUEST Q<br>join ACCEPTED_QUEST AQ on Q.Quest_id = AQ.Quest_id<br>where AQ.Completed_stages = | Show number of players that have completed each quest. This will allow admins to get a sense of the quest difficulty. If very few people have completed a certain quest, perhaps it is too difficult. |

| | |
|---|---|
| Q.Quest_stages<br>group by Q.Quest_id<br>order by count(*); | |
| select A.Player_name, A.Email, B.Expiration_date<br>from ACCOUNT A<br>join BILLING_INFO B on A.Id = B.Account_id<br>where month(curdate()) = month(B.Expiration_date)<br>and year(curdate()) = year(B.Expiration_date); | Find players and their emails whose membership expires that month. This can used by the company at the first of every month to send email reminders for renewing their membership. |
| select A.Player_name, A.Email, max(S.Item_amount)<br>from ACCOUNT A<br>join PLAYER P on A.Player_name = P.Display_name<br>join STORED_ITEM S on P.Display_name = S.Owner_name<br>where S.Item_name = 'Golden Chefs Hat'<br>group by S.Item_name; | Get the email of the player who has the most Golden Chef Hats. The company can congratulate this player by email for having the most of the rarest item in the game. |
| select P.Display_name, sum(S.Skill_exp), P.Time_played<br>from PLAYER P<br>join KNOWS_SKILL S on P.Display_name = S.Player_name<br>group by P.Display_name<br>having (P.Time_played * 80000) < sum(S.Skill_exp); | This detects for players that have exorbitant amounts of skill experience and very few hours of time played. Being in this list likely means the player is cheating. With our test data, we found a player who had about 1 million experience and 0 hours played. |

**Player View**: *Note: These queries use a variable in the UI code rather than ‘playername’.

| **Query** | **Purpose** |
|---|---|
| select *<br>from ACCOUNT<br>where Player_name = ‘playername’; | Displays player account information for specific player into the display table |
| select Player_name, Skill_name, Skill_rank, Skill_exp<br>from KNOWS_SKILL JOIN SKILL on KNOWS_SKILL.Skill_id = SKILL.Skill_id<br>where Player_name = ‘playername’; | Display skills player learned with its rank and experience |

| | |
|---|---|
| select Owner_name, Stored_in, Item_name, Item_amount, Item_type, Item_picture, Item_upgrade<br>from STORED_ITEM where Owner_name = ‘playername’<br>ORDER BY Stored_in,Item_name; | Display player items in inventory and bank |
| select Player_name, Quest_name, Completed_stages, Quest_stages<br>from ACCEPTED_QUEST join QUEST on QUEST.Quest_id = ACCEPTED_QUEST.Quest_id<br>where Player_name = ‘playername’; | Display quests player is on along with their progress as completed stages |
| select Player_name, Condition_effect, HAS_STATUS_CONDITION.Condition_duration<br>from HAS_STATUS_CONDITION join STATUS_CONDITION on HAS_STATUS_CONDITION.Condition_id = STATUS_CONDITION.Condition_id<br>where Player_name = ‘playername’; | Display conditions that is affecting the player and current duration. |

# 10 Work Distribution/Scheduling

## 10.1 Deliverables Schedule

The schedule progression will follow along with the assignment schedule. The target date is our goal date to complete the deliverable.. Our target date for each deliverable is three days before the due date, except for the final iteration. For the final iteration, we have distributed the sections over the course of ten days.

| **Deliverable** | **Target Date** |
|---|---|
| Proposal | Jan 15th |
| Update Proposal | Jan 28th |
| ER schematizing to completion | Jan 28th |
| RM schematizing to completion and beyond | Jan 29th |

| | |
|---|---|
| Big Picture and Tooling: all schematizations are completed and ready to be delivered | Feb 3rd |
| Set up AWS Accounts and verify that the server is working and can store data | Feb 9th |
| Tooling Assessment: Final verification of all tools needed to complete project such as AWS, mySQL database/workbench | Feb 12th |
| Put data tables into mySQL | March 5th |
| Write scripts to populate data in mySQL database using dummy data | March 6th |
| SQL Query Statements | March 7th |
| Normalization of data AKA all data is not redundant and in the right place by this date | March 9th |
| Work on and complete the poster outside of school hours | March 11th |
| Prepare the demo and everything should work in it by this date | March 12th |
| Final Report everything is done to completion and no more work needed | March 14th |
| Official Pizza Party | March 15th |

## 10.2 Distribution of Work

We see it fit that in order to hold everybody accountable for the project, **we will work together on every portion of the deliverables listed above**. This will ensure each team member understands what has been done and what needs to be done next in order to successfully meet the next deadline.

# 11 Updates

## 11.1 Iteration 1 → Iteration 2

Initially, we assumed that our database was going to keep track of the conditions monsters can also receive. We have decided to remove that as we don't keep track of any monster entities and as such shouldn't be inside this database. We also modified some of our entities and their relationships.

The major change involves how we finally decided to implement the project. Originally, we talked about using C++ and a .net framework and even played around with the idea of hosting in a browser-based environment. However, we quickly realized none of us had sufficient experience. The premise and idea behind having a browser-based, C++ powered game was interesting yet remained unrealistic since we would have to learn all of the implementation details and SQL at the same time.

In the end, we settled on a language everyone was comfortable with: Java. One of our team members also had experience with the Java framework used to implement the GUI, so it was a perfect fit.

## 11.2 Iteration 2 → Iteration 3

We were planning to connect other players to one another by being able to add each other to the friends list. We found out that this would become a lot more complicated and time consuming than we could handle and ended up scrapping this idea in about iteration 2 or 3. We changed a lot of other small aspects in our attributes and entities. For example, instead of having most information in the QUEST table, we expanded the ACCEPTED_QUEST table to better track players' progress in specific quests.

The transition between this iteration and the previous iteration indicates how our knowledge of databases has improved. We were able to remove some unnecessary attributes in the entities that didn't need external storage, but could be acquired through static client-side code. Most if not all of our database at this point only has necessary data that must be stored and static data that is more efficient to have in SQL form rather than based client-side.

## 11.2 Iteration 3 → Iteration 4

In our billing table, we had noticed that the phone number was only returning 3 digits instead of the standard 10 digit phone number. We realized that the way in which we formatted the phone number in the insert statement was the cause of the issue. We did not account for the spaces within the phone number field and because of that it only grabbed the first three. We had fixed this and successfully updated our database with the correct data.

After creating and hosting our database in iteration 3, we started to write the sql queries for UI functionalities, once basic framework for the GUI is completed all we had to do was merge the queries into the GUI controls and connect to the database hosted on AWS. During the creation of the GUI we separated the view screens for players and administrators, allowing players to only view their own information while administrators can see the information for all players. To implement the player view, we have the user enter their username in a text box prior to clicking the button for player view, this will ensure everything they see is their own information and the player view window will only show up if a correct name is entered.

We showed the process of normalizing the tables. It was fairly simple, since only one table was not already in the 3rd normal form. We choose not to update our sql scripts, since it was only one table and it would have been a lot of work to update the data scripts. Plus, we were all working synchronously on different aspects of the project.

# 12 Project Evaluation

## 12.1 Database Design And Results

Refer to **Section 5** for design, results, and the evolution of our design.

## 12.2 General Evaluation

Overall the effort spent was significant. Designing, creating, or testing a database is not something you can quickly do in a few hours. There is a lot of thought that goes behind every single design detail and test. If some perspectives are overlooked it could cost a company significant amount of money. The below paragraphs goes into more detail about some decisions we made that ended up helping us along with a few that dragged us down a little bit.

The first biggest part of the project for our group included the design of the entity relationship diagram and the relational model. We wanted to make sure we got it right the first time and focused on creating the highest quality design in order to accelerate the rest of the project. Thinking back on it, we are extremely happy that we had most of the design locked down as major changes in the design for subsequent iteration could have been extremely time consuming.

The second biggest part of our project was the implementing the database, data generation, and coding the UI.
We had some technical difficulties with our team initially being able to login to the Amazon Relational Database Server in order execute the database creation query and implementation of the data. We ended up figuring out that the issue had something to do with Amazon's security policies that blocks all IP addresses except localhost from connecting to the database. We were able to whitelist all IP's and that seemed to have fixed all the problems we have encountered.

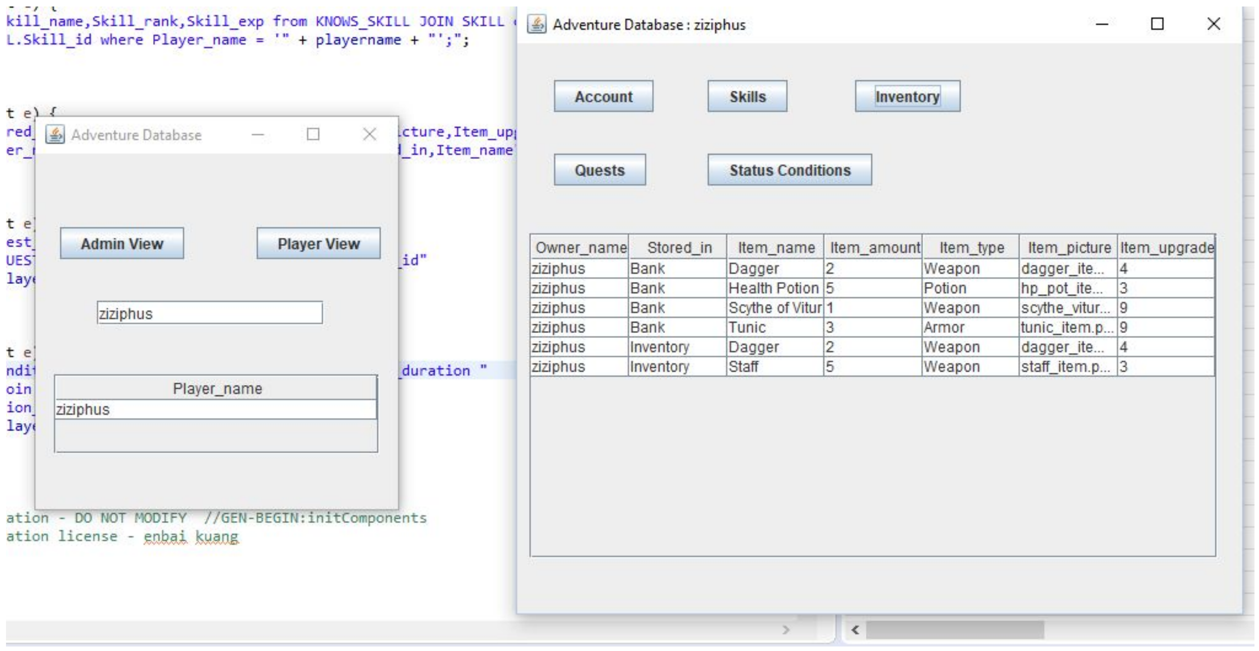
The UI has taken a considerable amount of time due to our team never using the Swing Framework before. We eventually opted to use a program called JFormDesigner which allows anybody to quickly and easily drag and drop UI elements onto a JFrame.

If we had another chance at creating a database we would probably start on the user interface at an earlier time. This would allow us to visualize all of the abilities players should have and each actions that a user or admin can take. As well, we would be able to spend more time creating a higher quality user interface that has more advanced features.

We had only generated special data for the STORED_ITEMS table as it required specific user names in order to connect it to the item that has been stored. This created a little bit of an issue as we had to use names that we used in the past which Mockroo did not support. Although, it took longer than we wanted to it insured that we had correctly working data that our table supported.

## 12.3 Feedback from Iteration 3

Our grader gave us feedback that we should include details about how our UI will work for the current database and how you intend to represent your data. Below, we've inserted pictures of our UI along with how the data would be shown for the admin and how different the information would be shown if you're a user.



```
kill_name,Skill_rank,Skill_exp from KNOWS_SKILL JOIN SKILL
L.Skill_id where Player_name = '" + playername + "';";

t e) {
red_                                    cture,Item_up
er_                                     _in,Item_name

t e
est_
UES                                     _id"
lay

t e
ndi                                     _duration "
oin
ion_
lay

ation - DO NOT MODIFY  //GEN-BEGIN:initComponents
ation license - enbai kuang
```

| Player_name |
|---|
| ziziphus |

| Owner_name | Stored_in | Item_name | Item_amount | Item_type | Item_picture | Item_upgrade |
|---|---|---|---|---|---|---|
| ziziphus | Bank | Dagger | 2 | Weapon | dagger_ite... | 4 |
| ziziphus | Bank | Health Potion | 5 | Potion | hp_pot_ite... | 3 |
| ziziphus | Bank | Scythe of Vitur | 1 | Weapon | scythe_vitur... | 9 |
| ziziphus | Bank | Tunic | 3 | Armor | tunic_item.p... | 9 |
| ziziphus | Inventory | Dagger | 2 | Weapon | dagger_ite... | 4 |
| ziziphus | Inventory | Staff | 5 | Weapon | staff_item.p... | 3 |

The grader has also asked us to describe more about where the data comes from, how its maintained, and how it would work in a real implementation. Currently, the data consists of mock data created by Mockaroo that represents an example of how it would operate in a real environment. In the actual implementation, the data would come from the game client itself.

For example, when the player kills an enemy the monster may drop an item onto the ground. When the player picks up an item, the code would execute either a SQL insert statement or an update statement depending what's already in the player's inventory. If the player picks up an unstackable item such as a +7 Longsword, it would be added into the STORED_ITEMS table under the player's name. If it was a stackable item that the player already had, such as 73 gold, the tuple that describes the player's current gold in inventory would be updated.

This same process parallels how most data is changed and stored in the game. When a player stored an item from their inventory to the bank, the attribute Inventory will change to Bank to show the change of location. Every time the player successfully picks an NPC's pocket, their current Thieving experience is increased through a client-side SQL statement call. Basically, the code in the game client automatically dictates what SQL statements are executed, allowing for hands-off automation for data storage and maintenance.

## 13 References

Mark Brocato. *Mockaroo.* 2019. http://www.mockaroo.com. Accessed April 24th 2019.
"Package Java.sql." Java.sql (Java SE 10 & JDK 10 ), http://www.docs.oracle.com/javase/10/docs/api/java/sql/package-summary.html . Accessed March 14th 2019.