Moonshot Project: Kritical Hit Written Report V1.0

Author	Paul NOWAK
Created	June 9th 2025
Last Modified	June 9th 2025
Document Deadline	June 9th 2025

Table of contents

▼ Details

2. Monthly Progress Reports

2023

February 2023

In December 2022, my Moonshot project named **KriticalHit** was validated by the Algosup team, who gave me the green light to start working on it.

At the beginning of the year 2023, I started writing weekly reports about my activities on my computer. These are document summaries of my accomplishments, problems, observations, and goals that I need to achieve. I try to update them each day, if I manage to find time to work on the development of KriticalHit.

Indeed, I wanted to leave a trace of any kind of work I was doing, in case I need to check older files for future work.

Currently, I'm in the **pre-production phase** of KriticalHit, which means I am focusing on the planning and research aspects of the project.

For now, I'm still trying to figure out how I will organize myself (tools to use, main tasks to accomplish) while researching valuable information related to my work. To have a good start, I need to clearly define the rules for how my product will work, so I can begin working on the functional and technical specifications.

Now, I'm planning to use a **project management software** to organize all the data, tasks, documents, and progress of my work. I've done some research on the best options, and I'm currently hesitating between **Trello** and **Wrike**.

Meanwhile, I've started researching **game engines** in general, such as **GameMaker** (a 2D game engine), to get an idea of how I could build my software.

François Gutherz from **Harfang3D** gave me advice about how they developed Harfang3D, so I now have several leads on how I could design my own game engine.

Finally, after gathering everything I need to manage my project and conducting my initial research, I will begin studying in more depth how to build KriticalHit and make a final decision:

> **Will I build this software from scratch, or base it on an existing one?**

March 2023

I pursued my research about different game engines, like **Construct** or **GameMaker**, to get an idea of how my software would look like.

Indeed, I've spoken to **Konstantinos Dimopoulos**, my Game Design teacher, about my project. I wanted to ask him for advice on how to make my engine as interactive and user-friendly as possible. While proposing some applications, he told me that he will contact me again if he finds any interesting **UI mock-up providers**.

Then, a comrade advised me to use **ClickUp** as a project management tool. So, I created a project for **Kritical Hit** where I've listed all my tasks, sorted them into different domains (**Planning**, **Research**, **Design**, **Development**, **Testing**, and **Deployment & Maintenance**) and by importance.

ClickUp Organization

I also created a **Gantt diagram** to visualize my tasks and tried to plan how much time I would require to solve each of them. This view is very likely to change in the next weeks.

After creating document templates for project management (continuous improvement and brainstorming subjects), I also started to follow a **C++ tutorial course** on YouTube by "**The Cherno**."

Finally, as I was still trying to figure out how to build my software (from scratch or based on another engine), after much research, I realized I would need to **list all the features** I want for my software. That way, I can decide what my engine will need and what it won't need.

So, I've created an **Excel Spreadsheet** to list all the features I would need, rank how important they are, and break them down into smaller tasks as much as possible.

It would include things like:

- *Will the main menu have a LEAVE button?*
- *Will there be a confirmation window displayed when we click it?*
- *What about the "save template" button? Will it have no confirmation window?*

As many possibilities could be addressed, this spreadsheet will help me write my **functional specifications**. Once I decide everything I need, I'll have a clearer view of what kind of engine I must build.

April 2023

This month, I managed to complete my **Features List** for my **KriticalHit** project. Indeed, it contains details about each feature's description, process, degree of importance, and requirements.

KriticalHit Features list.xlsx

In the process, **Konstantinos Dimopoulos** advised me not to detail the features too much and to focus more on the **core aspects**. Now, I need to validate them and decide which ones I will implement based on their importance.

I managed to define the **milestones** to accomplish my project:

- Project Scope decided
- UI Prototype finished
- Software Development started
- Quality Assurance Testing
- User Acceptance Testing
- Project Delivery

Although I seem to have completed my task list, I still need to confirm the **time required** for each of them, as I struggle to make accurate estimations.

In fact, I believe I should inquire more about the **average time required** for similar objectives and milestones.

By the way, **Konstantinos Dimopoulos** suggested I use **Figma** as a prototype design tool. For that, I just need to watch some tutorials about it and explore the kinds of **UI mock-ups** I could create.

May 2023

First, I finally finished detailing my **Features List** for **KriticalHit**. I also selected the ones I want to implement based on their **importance** and for **time management** purposes.

Then, I continued to look for a tool to help me create an interface and learned that **design-first prototype UIs** could cause backtracking issues. So, I decided to search for a tool to make a **code-based prototype** instead.

I found **UXPin**, a wonderful prototyping tool that includes built-in UI elements to help create an interface with a **code-based approach**.

Indeed, it’s possible to **export the prototype in full HTML**, but it isn’t integrated with the C++ language yet.

However, I discovered the possibility of using a **C++ web framework** that handles HTML elements, and I found **Wt**:

> Wt is a C++ library for creating web applications using HTML elements, and I would be interested in using it.

By the way, I decided to make **Kritical Hit** a **Desktop Application**, written in **C++**, built with **dynamic libraries** for performance, and using an **HTML interface**.

At the end of the month, I was finally able to **reorganize my tasks** and estimate how much time they would require. Of course, there’s no guarantee that the schedule will be 100% followed, but I’ll try my best to stick to it.

Indeed, it’s likely that I’ll **update it regularly** in the next few months.

June 2023

At the start of the month, I created an account to test **UXPin**’s features through a **one-month free trial**.

I created a **test prototype** where I experimented with different symbols (circles, textboxes, etc.) and tried various interactions, such as:

- Moving an icon from one place to another by clicking on it
- Changing an icon’s color while hovering the mouse

Furthermore, UXPin allows you to add **states** to elements to create more unique interactions. I was able to create a **hidden menu** that scrolls in from the left when clicking on an icon.

To get more information about the **Wt framework**, I had a phone call with a manager named **Wim Dumon**. I initially intended to use Wt to create a **C++ desktop application** for better performance.

However, he explained that Wt is more focused on **web applications**, and advised me to use the **Qt framework**, which is more suitable for **desktop applications**.

Then, I did some research on a framework called **ORK**, which helps with creating **RPG games** in Unity.

I discovered that there are several types of **RPG battle systems**, and I should choose the ones most suitable for implementation in **KriticalHit**.

Indeed, I plan to **study ORK’s source code** so I can understand its architecture and potentially reuse parts of it in my own software.

Finally, I started drawing various **concept arts** for my application.

July 2023

First of all, I started to research the **Qt Framework**: it’s a cross-platform software development framework that contains a comprehensive set of highly intuitive and modularized C++ library classes.

Indeed, it produces highly readable, easily maintainable, and reusable code with high runtime performance and a small footprint.

However, the full software is very expensive, and the free trial is only for 10 days. Although there is a free open-source version for smaller projects, I have yet to determine if it’s sufficient for my Moonshot project.

I found a tutorial on YouTube (created by Sciber) that allows us to create a small project with Qt. I tried to follow it, but I only managed to download a version of Qt used for designing and not for creating.

So, I sent a message to Sciber, who promised to help solve the issue for me.

Meanwhile, I've also decided to research **WebView2**: it's a free service created by Microsoft Edge that allows us to embed web technologies (HTML, etc.) in a native C++ app.

I created a post on a Microsoft forum where I received a tutorial and a sample project to learn how to use WebView2. Then, I cloned the project from a GitHub repository and started following the tutorial.

Perhaps learning more about C++ again would help me continue this tutorial.

Finally, I continued to draw my concept arts for **Kritical Hit**. I also plan to detail more about what each feature can do.

August 2023

This month, I tried to inquire more about **WebView2** and **Qt**.

First, I followed a tutorial for WebView2 which allowed me to open a window using Microsoft Edge.

I also discovered another tutorial showing how to embed web content into a native application. Indeed, it contains some use cases highlighting communication in WebView2, like sending messages from the host app to WebView2, for example.

Some lines of code from certain files use web code, but I have yet to manage modifying it so the HTML code can be changed during debugging.

Then, I'm still searching for **Qt Creator**, and I also created a post on the Qt forum for help.

As I'm still training in C++, I plan to create a simple text-based RPG program for practice, but also potentially to test possibilities for my app's features.

Finally, I've been detailing sub-features for my app in my personal notes, and I plan to start writing my functional and/or technical specifications.

September 2023

To begin with, I finally managed to try the Qt tutorial I found by reinstalling the application again. Indeed, I was able to create a small application with a simple calculator.

Then, I managed to contact M. Gutherz, my current computer science teacher, who is familiar with the Qt framework. I emphasized that I was looking for a GUI with good interactivity, so he advised me about **QML**, a GUI framework for Qt that offers a good trade-off.

Finally, I decided to ask Markus Adams, a teacher specialized in project management, to better estimate the time required for each of my tasks.

October & November 2023

I figured out that I often underestimate the time required for my tasks in my Gantt chart, which can be found on my ClickUp account. Thanks to Markus Adams' advice, a teacher specializing in project management, I decided to reorganize my tasks.

So, I decided to work on a project charter where I reworked my task list and estimated the time required for each.

Among the new objectives, I plan to make a roadmap on how to build my software step by step and gather all the information I need for writing my functional requirements.

Although, I still plan to run tests using the Qt framework and try creating software with a user-friendly interface.

December 2023 & January 2024

I decided to start a new list of tasks for my Moonshot project. In fact, it contains more details about existing tasks, such as subtasks and dependencies, and they are organized into 10 phases: Planning/Initiation, User Research, Technical Requirements, Interface Design, Battle System Mechanics, Programming Setup, Code Creation, Docs and Tutorials, Testing & Q.A., and Software Launch.

ClickUp Project Link Note: This link is now outdated.

Furthermore, I created a new Gantt diagram to schedule the tasks and estimate how much time they would require. I also plan to check their progress each week and verify if the deadlines are being met.

To finish the month of January, I have to start writing a new project charter (in a README file), set the milestones and KPIs, and also decide on the MVP (Minimum Viable Product).

Year-End Review

The first year was primarily focused on project planning, research, and foundational work:

Key Achievements:

- Project validation by ALGOSUP in December 2022
- Established project management system using ClickUp
- Completed initial features list and research on game engines
- Made key technology decisions (C++, Qt Framework)
- Created first concept arts
- Started learning required technical skills (C++, Qt)

Main Challenges:

- Difficulty estimating task durations accurately
- Uncertainty about whether to build from scratch or use existing engine
- Technical tool selection process took longer than expected
- Frequent schedule reorganization needed

Overall Status: The project remained in pre-production phase throughout 2023, with focus on planning and research rather than implementation. While this meant slower visible progress, it established important foundations.

2024

February 2024

This month, I managed to finish all the tasks required to achieve Phase 2: User Research. In fact, I mostly worked on setting up everything needed for the creation of the functional specifications.

For example, I defined a rough plan to make it, identified the targeted audience and market, and created several user personas.

Although I made some progress, I still have trouble finishing my tasks on time.

Indeed, I need to complete Phase 1 by starting to write the functional specifications. I must also decide on some technical details, start working on the UI, and imagine different scenarios for RPG battles.

March 2024

I started to write the functional specifications for my Kritical Hit project.

As a help, I created a navigation roadmap for preparing the UI concept arts, conceived different battle test scenarios to decide the RPG features to implement, and started some research to prepare the technical specifications.

However, I had to delay many of my tasks and reorganize my task schedule because the number of tasks per week was too high. In fact, I believe I will need more time to achieve my Moonshot project.

I plan to complete the functional specifications this month and to monitor my progress on a weekly basis.

April & May 2024

I’m still in the process of writing the functional specifications, as I need to carefully apply the MoSCoW matrix to the complete list of features.

I had to reorganize my schedule again, but I expect to finish the Functional Specifications in June.

Even though I believe my project will be done in C++, I plan to do some research to see if it could also be done in C#. If so, I might need another framework other than Qt.

Now, I have decided to create concept arts on Figma, working on the login and menu screens.

June 2024 & July 2024 & August 2024

During my vacation, I had to modify the summary of the functional specifications document. I decided to provide more details about the UI interaction, which will be crucial for this project.

To complete that section, I continued working on the concept art, including the Battle template screen, which will be the core of the software.

Additionally, I finally completed the MoSCoW matrix. My plan is to reorganize my schedule based on the importance of the features.

There are five types of features that must be implemented:

- Template Creation
- Character Creation
- Stats Creation
- Skills Creation
- Status Creation

Lastly, I created a Minimum Viable Product (MVP) of the app, where the first version contains only the app skeleton and the Template Creation menu.

September 2024

Recently, I presented the progress of my current Moonshot Project to some of my comrades. Indeed, I mentioned that I might need another framework other than Qt if I plan to build the app using C# instead of C++.

Some of them found my project too ambitious, especially since I have never developed new software by myself, and advised me to make it an extension of a game engine like Unity instead.

Furthermore, they insisted that I should finish the specification documents and start coding as soon as possible, using the Agile methodology for better organization.

Finally, they proposed setting up a meeting with Franck, as I need to review the target problem of my Moonshot Project.

October 2024

Finally, I started the creation of a prototype for Kritical Hit, focused on giving a preview of the targeted User Interface.

Indeed, it's based on Phase 1 of my Minimum Viable Product, including the navigation within the app and the template menu, and it contains the very first code files of my Moonshot project, placed within a new branch named "Prototype".

Although it's not ready to be tested yet after a total of 6 hours spent working on the prototype this month, I believe I will be able to finish it this November.

SPRINT: Honestly, I believe I didn't spend enough time on my prototype, which resulted in some misunderstanding of several Qt tools.

Even if I'm mostly satisfied with several of the framework's features to make my UI, I haven't followed any complete tutorial videos because I have yet to test the various features I need.

Additionally, some of my tasks related to the technical specifications documents are overdue.

So, I need to update my ClickUp document, following only the order of tasks to be completed.

November 2024

So far, I was able to build my first prototype and have a few people review it. The feedback I received often mentioned the overuse of colors and the lack of modernity in the design.

However, I found out that my prototype only showed a single feature, which simply introduced how my app's interface worked, and never revealed the true potential of Kritical Hit.

Indeed, I need to build a new prototype showcasing the expected features, such as the simulation menu and the rules menu. I will use most of the criticism I gathered when testing my first prototype to build the new one.

However, I will have to complete this before the official 0.1 version of Kritical Hit, which will contain a cleaner version of the interface.

SPRINT: Although I managed to build my first prototype, I didn't complete any official version on time. Furthermore, I haven't started writing my Technical Specifications yet.

Another problem: some reviewers took too much time to provide feedback, and I lost time waiting before starting work on the next prototype.

What I need to change is to find a better way to select future testers, such as asking them in advance, and to move forward to the next update of Kritical Hit while the current version is still being tested.

Additionally, I need to conduct some technical research about potential Qt resources to help me build my app and gather knowledge for my Technical Specifications.

December 2024

I started working both on the 0.1 version of Kritical Hit and on writing my technical specifications.

Thanks to various tutorials and research, I was able to start getting used to using Qt Creator, but I still have a lot to learn.

I managed to integrate the login system for accessing the main menu, and I built my first two pages while trying to address the criticism from my first prototype.

Furthermore, I started writing my technical specifications by planning and gathering information about the potential Qt Creator modules required for our project.

SPRINT: My progress is slow, and I need to work more regularly.

Indeed, I believe I didn't focus enough on the technical specifications while working on the unfinished version 0.1.

Next time, I need to balance my Moonshot Project working time between coding and writing documentation, as I haven't even started my test plan yet.

In the end, I might need a mentor for my Moonshot project to provide advice both on my organization and my decisions.

Year-End Review

The second year saw transition from planning to initial implementation:

Key Achievements:

- Completed user research phase
- Created functional specifications
- Developed first UI prototype
- Started actual coding with Qt Creator
- Refined project scope (focusing on Pokémon-style battles)
- Established MVP requirements
- Began technical specifications

Main Challenges:

- Continued issues with time management
- Project scope proved too ambitious initially
- Documentation fell behind implementation
- Prototype feedback highlighted need for design improvements
- Difficulty balancing coding vs documentation work

Overall Status: 2024 marked important transition from planning to implementation, though progress was slower than hoped. The project scope was successfully narrowed to something more achievable.

2025

January 2025

The first month of the year was focused on database research.

Indeed, I discovered Power Designer to create a conceptual data model based on my needs and generate a database.

Then, I installed DB Browser to use the SQLite database, and I was able to create a prototype database test where I can perform different query operations: GET, UPDATE, INSERT, and DELETE.

SPRINT: Unfortunately, I focused too much on trying the database.

I need to work more regularly on coding as well as on the various documents required for the project. So, starting next month, I need to balance coding and document writing.

I should also implement sprints each week, so I need to update the monthly report template.

Finally, I plan to go to the ALGOSUP library to check resources about the Qt Framework, and I need to have a meeting with Franck to discuss the Moonshot project further.

February 2025

That month, I was especially focused on the project code. Indeed, I took the opportunity to build new pages, `.ui` files, and new classes for my app.

Many changes were made during the vacation, as I realized the number of features I wanted to implement was too ambitious. Instead of creating an RPG simulator inspired by various RPG games, I decided to focus on the Pokémon battle mechanics.

That’s why I changed my priorities and decided to work on the simulation menu as soon as possible. In fact, I want to take an approach focused on implementing simpler features.

I’m satisfied with my progress, and I plan to have a first released version of my project ready by April 2025.

March 2025

Despite medical conditions, I was able to make progress in the development of my project.

I reached a point where the simulation menu is closer to the original Pokémon games, with two attack actions possible and a Quit menu.

However, I focused a lot on creating the Simulation menu, so I need to reorganize my task schedule, especially the documentation tasks. Updating my MVP will require me to review the current version of my Functional Specifications.

As I plan to start working on the Rules menu, I should consider the interface design of the other menus and seek advice from other students regarding organization.

April 2025

I managed to create and connect the Rules menu to my Simulation menu, allowing activation or deactivation of certain features, such as specific capacities or the PP system.

My simulation can now include healing and buffing moves, and I created new external files to set up the battle. However, I still encountered some issues, such as a crashing debug mode and a simple but defective AI entity.

I’m also close to finishing a new version of my Moonshot Project, but I need to consider whether I’ll be able to complete the next version by May.

Furthermore, I need to work more regularly on the documentation, starting with revising the functional specifications as soon as possible, followed by the other documents.

In any case, I plan to fix the AI issue and see if I can continue adding new features in May.

May 2025

May 2025 marked the end of the code development phase for Kritical Hit, at least for the first version.

I had to stop adding new features in order to focus on completing the remaining documentation.

Since I was very late on the document due June 9th, I decided to use ChatGPT and NinjaAI to help speed up the process.

I rewrote the functional specifications and completed the test plan, but I still need to finish the test cases, bug tracking, and technical specifications. Additionally, I must review everything and eventually merge the documents into a single file.

The following month will be dedicated primarily to preparing for the oral presentation, where I need to develop a compelling 40-minute storytelling.

Partial Year Review (Through May)

The final months focused on completion and documentation:

Key Achievements:

- Implemented database functionality
- Created working battle simulation
- Added rules menu and game mechanics
- Developed AI system (though with issues)
- Started completing required documentation

Main Challenges:

- Time pressure to complete documentation
- Technical issues with AI and debugging
- Balancing feature completion with documentation needs
- Medical conditions affecting work in March

Overall Status: Project entered final phase with focus on completing minimum viable product and required documentation, though time pressure became significant factor.

Final Review

The project evolved from an ambitious RPG battle simulator to a more focused Pokémon-style battle system, with steady if slower than planned progress toward completion. Each year showed distinct phases: planning (2023), implementation (2024), and completion/documentation (2025).

3. Functional Specifications

3.1. Overview

3.1.1. Document Purpose

This document has been written to provide the functional specification of a Moonshot Project named Kritical Hit. The Moonshot Project is a final evaluation imposed by the ALGOSUP school staff to create our unique and professional project through the whole scolarity and validate our Master's level in Software development.

3.1.2. Context

Video games have long been a cornerstone of the entertainment industry, generating approximately $180 billion annually. The market continues to evolve rapidly, driven by advancements in both hardware and software—particularly through the rise of powerful game engines. While these engines are also utilized across various industries, they have revolutionized game design by enabling the creation of more complex and sophisticated interactive experiences.

Despite these innovations, developing a video game remains a time-consuming and detail-oriented task for software engineers. With the rise of new technologies and changing audience expectations, modern games are expected to meet high standards in performance, quality, and depth to remain competitive and marketable.

Among all genres, role-playing games (RPGs) hold a special place in the industry. They offer immersive storytelling, rich character progression, and strategic battle systems that attract a wide and loyal audience.

A prime example is the Pokémon franchise, which exemplifies the global appeal and commercial success of RPGs. As of 2023, Pokémon is the highest-grossing entertainment media franchise in the world, having earned over 150 billion dollars in total revenue. This includes 30 billion dollars from video games alone and over 100 billion dollars from licensed merchandise. The core games in the Pokémon series follow a classic RPG structure, where players assume the roles of characters who captures, trains, and battles Pokémon with the ultimate goal of becoming the Pokémon Champion.

Building such RPGs often requires dedicated game design tools, particularly for crafting engaging battle mechanics and managing complex character systems. However, current game engine software tools often fall short—they may either lack essential resources or present interfaces that are not welcoming to new developers. Moreover, extending their functionality by adding plugins can be too time-consuming or technically demanding for amateurs or solo designers.

This highlights a critical gap in the tools available for designing high-quality RPG systems, especially for developers inspired by franchises like Pokémon but lacking the means or technical expertise to implement similarly rich gameplay experiences.

3.1.3. RPG Domain

An RPG (Role-Play Game) is a game where the player embodies a fictional character that will evolve, often alongside other characters, where he will accomplish diverse quests, fight enemies and explore a more or less imaginary world. Indeed, RPGs are based on a point system and experience level that can be increased and allow the player to get stronger and receive new abilities.

Inspired by several sources like the Dungeon and Dragons pen-and-paper games and fantasy writings by authors like J. R. R. Tolkien, RPGs bring a whole new concept of entertainment with the success of Franchises like Pokemon, Final Fantasy, and Dragon Quest. Throughout time, they have distinguished themselves in subgenres like Action RPG (including action gameplay) and MMORPG (online role-playing).

Each player chooses a character, whether completely create him or embodying a defined one (like Cloud from Final Fantasy VII), and have the freedom to change its appearance (clothes, weapons, magic powers) and its stats (strength, agility, vitality...).

3.2. Product Goal

3.2.1. Project Scope

To support both amateur game design and professional developers, the goal of this project is to design a desktop application functioning similarly to a lightweight game engine.

This software will feature an intuitive, user-friendly UI to help users define and configure battle system rules for RPGs in development. It aims to foster creativity and give users the flexibility to create a wide variety of RPG battle templates with ease.

Inspired by iconic titles like Pokémon, the application will include a simulation interface that replicates the feel of a classic Pokémon battle, allowing users to test their systems familiarly and engagingly.

The following are the key features envisioned for the final product:

- The software will allow the user to navigate through the different menus with an understandable and colorful interface, similar to the game Super Mario Maker.
- Users will be able to simulate a turn-based Pokémon-style battle between two sides, each selecting one of four available Pokémon before the fight begins.
- The system will include an integrated battle interface reminiscent of the original Pokémon games, allowing users to interactively test how their fight logic and rules behave in a turn-based simulation.
- A rules menu will allow users to enable or disable specific mechanics—such as turn order logic, stat caps, or critical hit chance—before starting a new simulation.
- Battle moves inspired by Pokémon will be available, including damaging attacks, healing actions, buffs (stat increases), and nerfs (stat reductions), offering a diverse range of effects to test.
- User can also create and load several many battle templates in the application's database to save a variety of battle rules.

3.2.2. Constraints

First and foremost, this project will be developed by a single individual with no prior experience in creating complete software independently. As part of the Moonshot Project, it must be completed alone, without active assistance from other students, teachers, or professionals.

Due to this context, no financial investments are possible. The project will rely entirely on a limited local budget, as well as the available hardware and software tools.

Additionally, the product will be developed under time constraints, as the contributor must balance this project alongside ongoing academic commitments. Although the Moonshot Project is required to validate the contributor’s degree, it does not follow a fixed deadline—task scheduling must therefore remain flexible and adapt to the progression of the contributor’s studies.

Finally, as the project now focuses specifically on implementing gameplay mechanics inspired by Pokémon battles, this game design choice inherently limits the versatility and creative freedom users may have when testing other types of combat systems. At least in the planned final version, the simulation will be tailored to turn-based, Pokémon-style encounters, potentially reducing support for broader RPG battle variations.

Technical, Database, and Performance Constraints

The development environment being used is unfamiliar to the contributor, who is working with this tool for the first time. As such, the contributor may face difficulties understanding how the environment handles memory, manages technical specifications, and deals with performance and database interactions. These challenges could impact the stability and efficiency of the application during development.

The project will be developed and tested exclusively on a single computer. Any technical limitations of this machine—including available RAM, processor capability, and storage—may directly influence development speed, build size, and runtime performance. Hardware issues or system-specific bugs may also go unnoticed due to the absence of cross-platform testing or deployment.

Moreover, the contributor lacks experience in designing, configuring, and integrating databases within applications. This unfamiliarity may lead to inefficient database schema structures, suboptimal query handling, and potentially poor data management practices. Such issues could hinder the application's performance, especially if real-time data access or scalability becomes necessary during later stages of development.

3.2.3. Risks and Assumptions

3.2.3.1 Risks

Risk	Impact	Mitigation
Insufficient app responsive speed	The computer's performance might not be fast enough to allow the user to test the app properly	Extended research about the app's speed and responsiveness must be performed
User-friendly interface bad outcomes	Highlighting the importance of testing UI prototypes to balance usability and simplicity	

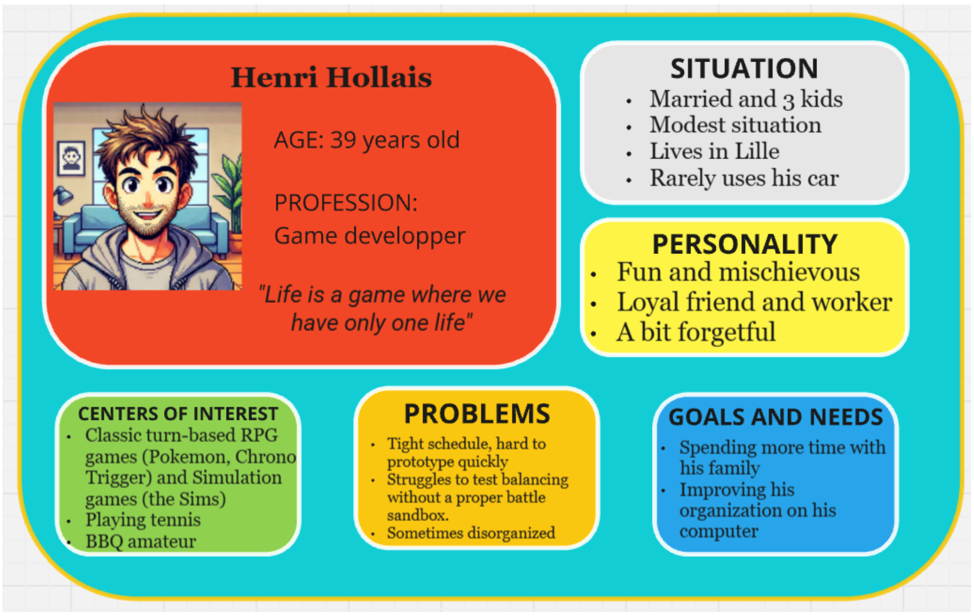
Risk	Impact	Mitigation
Complex UI effects	Some planned UI widgets may be too difficult and complex to implement in the software and could decrease its responsiveness	Creation of UI prototypes
Heavy database	The software's response could be slow due to the required amount of database stored in backend	Further studies of database app storage

3.2.3.2 Assumptions

Assumption	Impact	Mitigation
2D character models	To simulate RPG battle, 2D sprites should be represented as they require less space in the memory than 3D ones	Finding how to integrate 2D sprites in an app and make them react
Tree representations	It could bring visibility to the user to have a clearer view on certain features such as stats	Research on tree-like representation UI models
Interactive tutorial	A tutorial could be added to help the user get familiarized with the product	Search how to implement a tutorial in an app
Battle systems RPG	RPG battle systems are various and unique, so we need to make sure the user gets enough creativity for his taste	Several battle systems must be tested out

3.3. User Personas

3.3.1. Henri Hollais



Henri Hollais

AGE: 39 years old

PROFESSION: Game developper

"Life is a game where we have only one life"

SITUATION
- Married and 3 kids
- Modest situation
- Lives in Lille
- Rarely uses his car

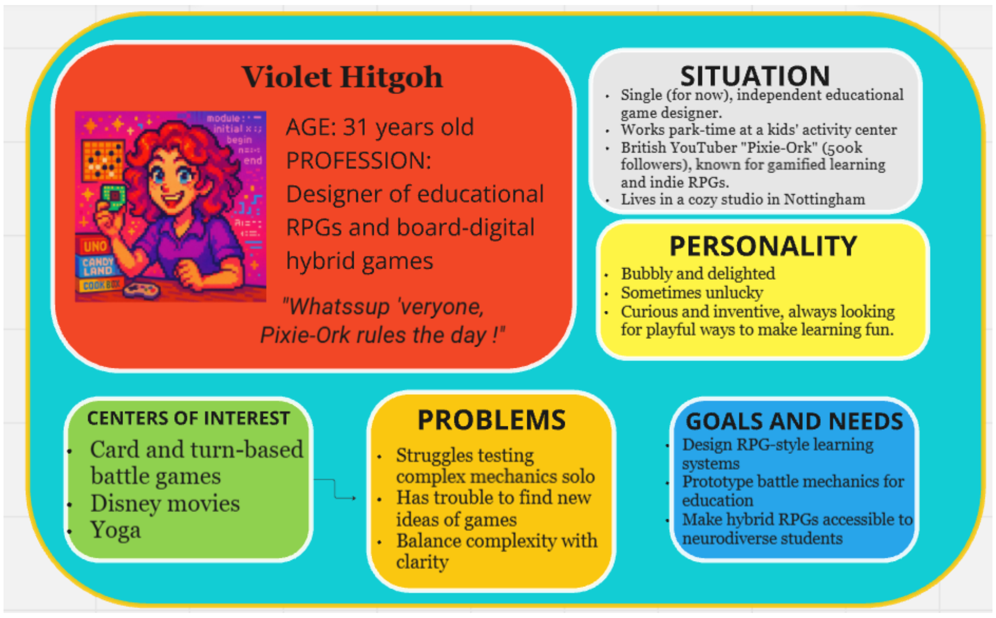
PERSONALITY
- Fun and mischievous
- Loyal friend and worker
- A bit forgetful

CENTERS OF INTEREST
- Classic turn-based RPG games (Pokemon, Chrono Trigger) and Simulation games (the Sims)
- Playing tennis
- BBQ amateur

PROBLEMS
- Tight schedule, hard to prototype quickly
- Struggles to test balancing without a proper battle sandbox.
- Sometimes disorganized

GOALS AND NEEDS
- Spending more time with his family
- Improving his organization on his computer

3.3.2. Violet Hitgoh



Violet Hitgoh

AGE: 31 years old

PROFESSION: Designer of educational RPGs and board-digital hybrid games

"Whatssup 'veryone, Pixie-Ork rules the day !"

SITUATION

- Single (for now), independent educational game designer.
- Works park-time at a kids' activity center
- British YouTuber "Pixie-Ork" (500k followers), known for gamified learning and indie RPGs.
- Lives in a cozy studio in Nottingham

PERSONALITY

- Bubbly and delighted
- Sometimes unlucky
- Curious and inventive, always looking for playful ways to make learning fun.

CENTERS OF INTEREST

- Card and turn-based battle games
- Disney movies
- Yoga

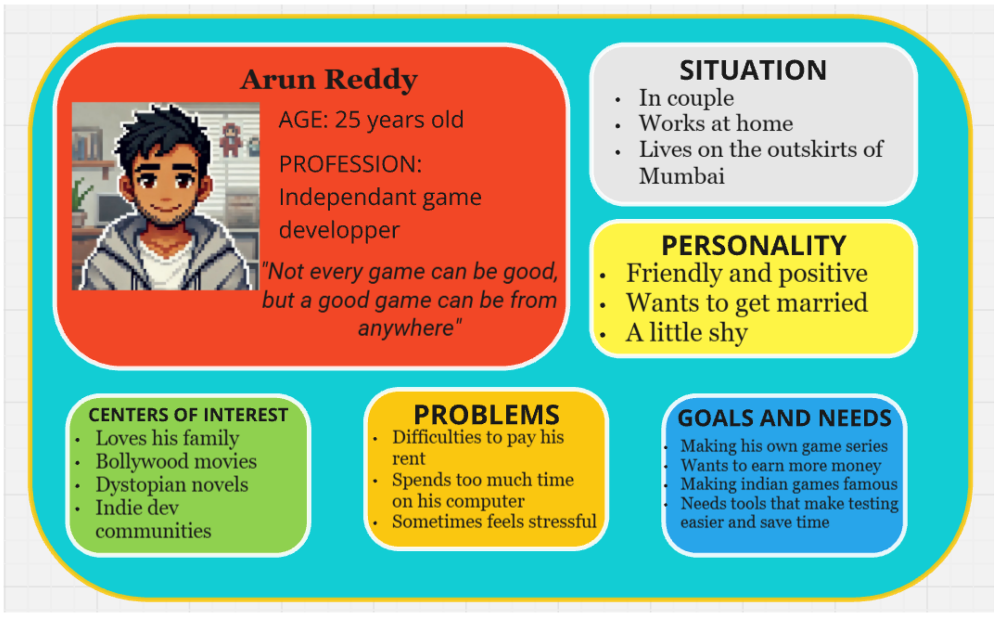
PROBLEMS

- Struggles testing complex mechanics solo
- Has trouble to find new ideas of games
- Balance complexity with clarity

GOALS AND NEEDS

- Design RPG-style learning systems
- Prototype battle mechanics for education
- Make hybrid RPGs accessible to neurodiverse students

3.3.3. Arun Reddy



Arun Reddy

AGE: 25 years old

PROFESSION: Independant game developper

"Not every game can be good, but a good game can be from anywhere"

SITUATION

- In couple
- Works at home
- Lives on the outskirts of Mumbai

PERSONALITY

- Friendly and positive
- Wants to get married
- A little shy

CENTERS OF INTEREST

- Loves his family
- Bollywood movies
- Dystopian novels
- Indie dev communities

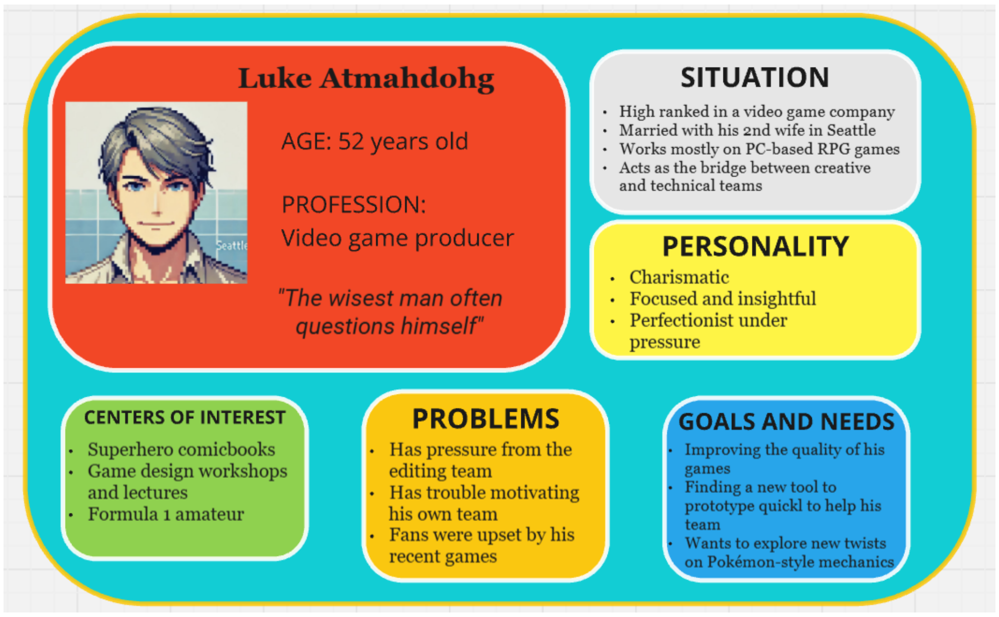
PROBLEMS

- Difficulties to pay his rent
- Spends too much time on his computer
- Sometimes feels stressful

GOALS AND NEEDS

- Making his own game series
- Wants to earn more money
- Making indian games famous
- Needs tools that make testing easier and save time

3.3.4. Luke Atmadohg



Luke Atmahdohg

AGE: 52 years old

PROFESSION: Video game producer

"The wisest man often questions himself"

SITUATION

- High ranked in a video game company
- Married with his 2nd wife in Seattle
- Works mostly on PC-based RPG games
- Acts as the bridge between creative and technical teams

PERSONALITY

- Charismatic
- Focused and insightful
- Perfectionist under pressure

CENTERS OF INTEREST

- Superhero comicbooks
- Game design workshops and lectures
- Formula 1 amateur

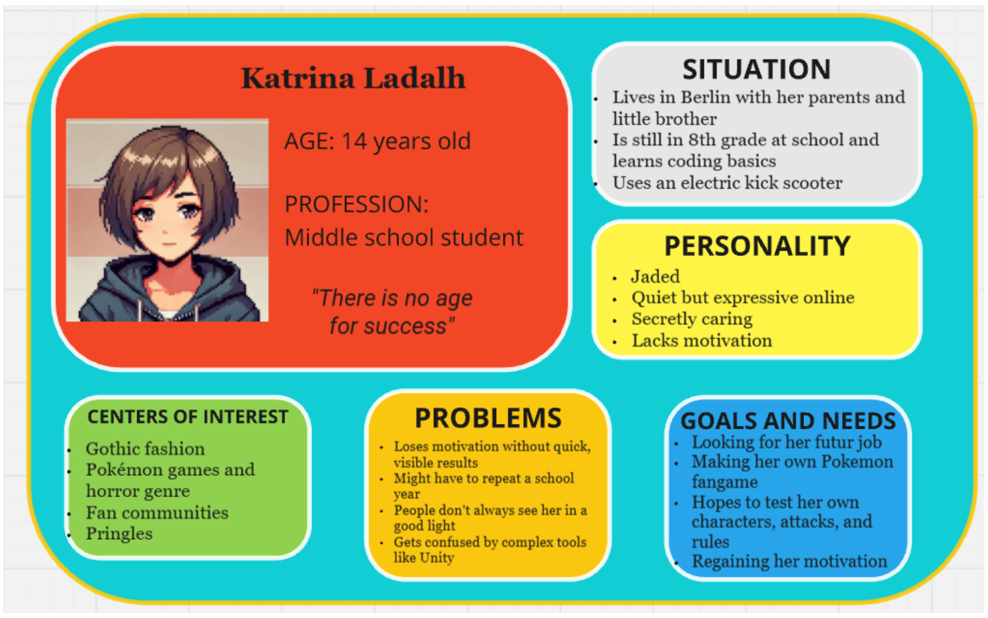
PROBLEMS

- Has pressure from the editing team
- Has trouble motivating his own team
- Fans were upset by his recent games

GOALS AND NEEDS

- Improving the quality of his games
- Finding a new tool to prototype quickl to help his team
- Wants to explore new twists on Pokémon-style mechanics

3.3.5. Katrina Ladalh



Katrina Ladalh

AGE: 14 years old

PROFESSION: Middle school student

"There is no age for success"

SITUATION
- Lives in Berlin with her parents and little brother
- Is still in 8th grade at school and learns coding basics
- Uses an electric kick scooter

PERSONALITY
- Jaded
- Quiet but expressive online
- Secretly caring
- Lacks motivation

CENTERS OF INTEREST
- Gothic fashion
- Pokémon games and horror genre
- Fan communities
- Pringles

PROBLEMS
- Loses motivation without quick, visible results
- Might have to repeat a school year
- People don't always see her in a good light
- Gets confused by complex tools like Unity

GOALS AND NEEDS
- Looking for her futur job
- Making her own Pokemon fangame
- Hopes to test her own characters, attacks, and rules
- Regaining her motivation

3.4. List of Features

The software includes a wide range of features, many of which may be difficult to implement during development.

Therefore, the MoSCoW matrix method will be used to prioritize each feature and sub-feature based on their likelihood of being implemented.

- **MUST-HAVE**: These features are mandatory and non-negotiable needs for this project.
- *SHOULD-HAVE*: These features are essential to the product, but they don't represent a vital part.
- COULD-HAVE: These features aren't necessary to the core product and have a much smaller impact if left out, but they are considered "nice to have".

3.4.1. User-Friendly Application

These features were designed to ensure the interactive appeal of *Kritical Hit* and allowed users to navigate seamlessly through the various menus.

3.4.1.1 User Login System

Feature	Description	Priority
User Login	Implement a login system for users	COULD-HAVE
↳ **Validate Credentials**	Ensure user credentials are checked properly	COULD-HAVE
↳ **Error Messages**	Display error messages when login fails	COULD-HAVE

3.4.1.2 Main Menu Page

Feature	Description	Priority
Main Menu UI	Build the main menu page	**MUST-HAVE**
↳ **Go to Simulation Page**	Allow navigation to the battle simulation interface	**MUST-HAVE**
↳ **Go to Rules Menu**	Allow access to the rules customization menu	**MUST-HAVE**
↳ **Go to Character Selection**	Navigate to Pokémon/ character selection screen	COULD-HAVE
↳ **Go to Damage Calculator**	Navigate to Damage Calculator editor screen	COULD-HAVE
↳ **Logout Button**	Add a logout option to exit the user session	COULD-HAVE

3.4.1.3 Responsive Design

Feature	Description	Priority

Feature	Description	Priority
Full-Screen Mode	Enable full-screen display for better immersion	COULD-HAVE
** Responsive Layout**	Ensure each page adapts to various screen sizes	COULD-HAVE

3.4.2. Battle Interface and Flow Simulation

These features focus on delivering a functional and immersive battle experience, simulating turn-based combat flow with responsive UI and basic enemy AI.

3.4.2.1 Battle Interface Design

Feature	Description	Priority
Design Battle Interface	Create the UI for the battle screen	**MUST-HAVE**
↳ **Show Pokémon & HP Bars**	Display both Pokémon on-screen with visual HP bars	**MUST-HAVE**
↳ **Display Battle Text**	Show relevant messages and narration during battle	*SHOULD-HAVE*
↳ ↳ "*What will you do?*" Prompt	Display the classic player turn prompt	*SHOULD-HAVE*
↳ ↳ * Move Usage Text*	Show messages like "Pikachu uses Thunderbolt!"	*SHOULD-HAVE*
↳ ↳ *Healing Text*	Display messages for healing moves (e.g. "Bulbasaur heals!")	COULD-HAVE
↳ ↳ * Buff Text*	Display stat increase messages (e.g. "Attack rose!")	COULD-HAVE
↳ ↳ * Nerf Text*	Display stat decrease messages (e.g. "Defense fell!")	COULD-HAVE
↳ **Implement Move Buttons**	Show four move buttons the player can click	**MUST-HAVE**
↳ **Update UI Elements**	Dynamically update HP bars, text, and states after actions	**MUST-HAVE**

3.4.2.2 Enemy AI System

Feature	Description	Priority
Enemy AI	Automate opponent's decisions during battle	**MUST-HAVE**
↳ **Random Move Selection**	Enemy chooses a move randomly	**MUST-HAVE**
↳ **Check for PP**	Prevent enemy from choosing moves with 0 PP	*SHOULD-HAVE*
↳ ** HP-Based Decisions**	Make smarter move choices based on remaining HP	COULD-HAVE

3.4.2.3 Quitting the Battle

Feature	Description	Priority
Quit Battle Handling	Manage quitting mid-battle	*SHOULD-HAVE*
↳ **Confirmation Popup**	Show confirmation before exiting the battle	*SHOULD-HAVE*
↳ **Return to Main Menu**	Navigate back to the main menu after quitting	*SHOULD-HAVE*

3.4.2.4 Battle End Conditions

Feature	Description	Priority
Detect Battle End	Determine when the battle is over	**MUST-HAVE**
↳ **Pokémon Fainting**	Detect when a Pokémon's HP reaches 0	**MUST-HAVE**
↳ **Display Outcome Message**	Show win/lose message and return to the main menu	**MUST-HAVE**

3.4.3. Battle Mechanics System

This section defines the core logic behind battles, focusing on turn resolution, move execution, and stat-based outcomes to ensure strategic depth.

3.4.3.1 Combat Turn System

Feature	Description	Priority
** Turn-Based Combat**	Two Pokémon engage in a turn-based fight	**MUST-HAVE**
** Stat-Based Turn Order & Outcome**	Uses stats (HP, Attack, Defense, Speed) to resolve turns	**MUST-HAVE**

3.4.3.2 Move Selection & Usage

Feature	Description	Priority
List Available Moves	Display list of usable moves for the player's Pokemon	**MUST-HAVE**
** PP System**	Manages move usage based on remaining PP	*SHOULD-HAVE*
↳ Reduce PP on Use	Deducts 1 PP when a move is used	*SHOULD-HAVE*
↳ Block When PP = 0	Prevents using moves that have no PP remaining	*SHOULD-HAVE*
↳ Trigger Game Over	Ends game if no available moves remain	COULD-HAVE
** Accuracy and Evasion**	Uses an algorithm to determine how likely each move can succeed in hitting the target, depending on the user's accuracy and the target's evasiveness.	*SHOULD-HAVE*
↳ Setting move accuracy	Set an accuracy for each move and deals with move failure.	*SHOULD-HAVE*
↳ Boosting Accuracy/ Evasiveness	Creates moves allowing to buff and/or nerf the general Accuracy and Evasion of a Pokemon	COULD-HAVE

3.4.3.3 Move Effects System

Feature	Description	Priority
** Move Effects System**	Handles the outcome of moves used in battle	**MUST-HAVE**
** Damage-Dealing Moves**	Executes damage-based calculations during combat	**MUST-HAVE**
↳ Apply Base Damage Formula	Uses a formula to compute base damage output	**MUST-HAVE**
↳ Critical Hit Mechanic	Applies critical hit chance and multiplier	COULD-HAVE
↳ STAB Bonus	Applies Same-Type Attack Bonus if move type matches user's type	COULD-HAVE
↳ Type Effectiveness	Adjusts damage based on move vs. opponent type (e.g., fire > grass)	COULD-HAVE
↳ Implement Limited Types	Includes at least 4 basic types: Fire, Water, Grass, Electric	COULD-HAVE
Healing Move	Heals 50% of user's base HP	*SHOULD-HAVE*
Buffing Stat Move	Increases a selected stat (e.g., Attack, Defense)	*SHOULD-HAVE*
Nerfing Stat Move	Decreases opponent's stat (e.g., Speed, Defense)	*SHOULD-HAVE*

3.4.4. Set Up Battle

This section covers how players configure battles by selecting Pokémon, assigning moves, defining rules, and customizing templates or calculation systems.

3.4.4.1 Set Up Characters

Feature	Description	Priority
Create Selectable Pokémon	List of 6 available Pokémon to choose from	*SHOULD-HAVE*
Load/Select Pokémon	Load data from database or list	*SHOULD-HAVE*
Display Pokémon Data	Show name, stats, and types of each Pokémon	COULD-HAVE

3.4.4.2 Assign Capacities (Movesets)

Feature	Description	Priority
Set Up Capacities Database	Create a list/ database of all possible moves	**MUST-HAVE**
Assign Moveset to Pokémon	Assign 4 moves to each Pokémon	**MUST-HAVE**
↳ Same Moveset for All	All Pokémon share the same 4 moves	**MUST-HAVE**
↳ Different Movesets	Each Pokémon has unique moves	*SHOULD-HAVE*
Assign Move Types	Ensure each move has a type (e.g., Fire, Water, Electric, etc.)	COULD-HAVE

3.4.4.3 Choose Rules

Feature	Description	Priority
Select Ruleset	Choose which special rules will apply in battle	**MUST-HAVE**
↳ No Healing Move Rule	Prevents healing moves from being used	**MUST-HAVE**
↳ No Buffing Move Rule	Disables stat-increasing moves	**MUST-HAVE**
↳ No Nerfing Move Rule	Disables stat-decreasing moves	*SHOULD-HAVE*
↳ No PP System Rule	Turns off PP management	*SHOULD-HAVE*
↳ No Type Table Rule	Disables type effectiveness system	COULD-HAVE
↳ Different Critical Hit Coefficient	Adjusts multiplier for Critical Hit Bonus	COULD-HAVE
↳ Different STAB Coefficient	Adjusts multiplier for Same-Type Attack Bonus	COULD-HAVE
Save Ruleset	Store selected rules for use during the battle	**MUST-HAVE**

3.4.4.4 Choose Characters for Battle

Feature	Description	Priority
Select Player Pokémon	Choose which Pokémon the player will use	*SHOULD-HAVE*
Select Opponent Pokémon	Choose which Pokémon the opponent will use	*SHOULD-HAVE*
Confirm Selection	Display final choice and prompt confirmation	*SHOULD-HAVE*

3.4.4.5 Setting Up Battle Template

Feature	Description	Priority
Create New Battle Template	Generate a new battle template that adds a custom rule set to the database.	COULD-HAVE
Load Battle Template	Retrieve and use an existing battle template from the database.	COULD-HAVE
Delete Battle Template	Remove a selected battle template from the database.	COULD-HAVE

3.4.4.6 Changing Damage Calculator

Feature	Description	Priority
Edit Damage Calculator	Open and edit the logic behind damage calculations.	COULD-HAVE
↳ **Adjust Coefficients**	Modify individual calculation values.	COULD-HAVE
↳ **Replace Formula**	Overwrite the entire calculation formula.	COULD-HAVE

3.5. Product Details

3.5.1. Minimum Viable Product

The project will be developed progressively through a Minimum Viable Product (MVP) approach, organized into distinct phases. In other words, each phase corresponds to a specific stage of development based on the features implemented and the version released. Furthermore, every phase is expected to be functional and designed with a user-friendly interface.

Phase	Added Features	Performance Benchmarks	Version
Phase 1	App skeleton, login menu, main template menu, basic simulation with 2 preset Pokémon (same attacks & stats)	Simulation loads in under 1 second; attacks execute within 100 ms and update both HP bars.	0.2
Phase 2	Custom Pokémon stats, attack order system, classic-style damage calculator, attack delay system	Delay between attacks below 2 seconds; each character's HP bar updates within 100 ms of their turn.	0.5
Phase 3	Rules menu, healing and buffing moves, PP system, support for 4 attacks per Pokémon	Rules menu opens in under 1 second; updates occur with a 100 ms interval; exiting menu takes less than 200 ms, even after multiple updates.	1.0
Phase 4	Nerfing moves, critical hits, 4-type system with effectiveness chart, custom movesets	Simulation menu loads in under 1 second, even with 8 capacity objects saved in the database.	1.5
Phase 5	Character selection menu, expanded stats (evasion, accuracy), support for 6 types	Character selection opens in under 1 second; interface updates occur every 100 ms; exiting takes less than 200 ms.	2.0
Phase 6	Save/load custom battle templates, 8-type system with immunities, damage calculator menu	Calculator menu opens in under 1 second; updates every 100 ms; exiting within 200 ms; app remains responsive with 5 battle templates saved; new template loads < 500 ms.	2.5

Two specific versions of the Kritical Hit project will be presented to the jury: version 1.0 during the first oral exam scheduled for June 24th, 2025, and version 2.0 — or possibly 2.5 — during the second oral presentation (date TBD).

Data Migration and Backward Compatibility

As new features are introduced across MVP phases, particular attention will be paid to ensuring that saved data, configurations, and battle templates remain functional and forward-compatible. Specific strategies include:

- **Phase 3 → Phase 4**: Entity objects, Capacity objects, and their classes will be updated to integrate new type attributes without breaking existing data.
- **Phase 4 → Phase 5**: The transition from a 4-type to a 6-type system will be handled by updating the effectiveness structure. Existing custom moves and Pokémon will adapt to new type definitions.
- **Phase 5 → Phase 6**: Templates created in previous versions (including pre-save/load support) will remain usable. The system will support legacy loading with internal migration logic to upgrade their data format silently.
- Across versions, older configurations will not require user intervention to remain usable in newer versions unless critical incompatibilities are detected.

3.5.2. Non-Functional Requirements

- **Accessibility**: The User Interface of this product must be accessible to users unfamiliar with RPG games or software. Kritical Hit's design must maintain an intuitive experience, ensuring that at least 95% of users report satisfaction during usability testing.
- **Configurability**: The software must allow customizable settings for all available features. Users should be able to tailor configurations (e.g., battle parameters, UI layout) to suit personal or project needs.
- **Flexibility**: The software must support the easy integration of new features to enable creative expansion of battle systems. Updates to core functionalities should not require significant rework or refactoring by the development team.
- **Responsiveness**: Interface feedback must be fast. Button clicks should respond within 100 ms, menu navigation should not exceed 200 ms, and loading a new battle template must occur within 500 ms. The simulation menu — the application's core — must support real-time updates and quick interface reactions to maintain usability.
- **Re-usability**: The software's interface design should be modular and well-structured, allowing it to be reused in future software projects requiring accessible and intuitive system creation interfaces.
- **Scalability**: The software must be capable of handling a growing collection of battle system templates with stable performance. While exact data size thresholds are to be determined, the system must remain responsive as data accumulates.
- **Security**: Any personal data saved by the user must be protected from unauthorized access or tampering. Template files must be safeguarded from unintended overwrites, deletion, or corruption.

- **Usability**: The interface must be easy to understand and visually appealing, enabling users to navigate menus and complete tasks without extensive documentation. Visual feedback (e.g., hover effects) must enhance clarity without clutter.
- **Reliability**: Users must always be able to load their most recently saved battle templates. While automated backup isn't a current priority, manual saves should be fully reliable and restorable.
- **Compatibility**: The application is primarily developed for Windows desktop systems. It must remain operable on typical Windows school/university computers and should support future testing on alternative platforms (e.g., Mac OS).

3.5.3. Acceptance Criteria

To be considered functionally complete and successful, **Kritical Hit** must meet the following **acceptance criteria**:

- **UI Responsiveness**: All user interface actions must respond within strict thresholds:
    - **Button interactions** (click, hover, selection): under **100 ms**
    - **Menu navigation**: under **100 ms**, or **200 ms** when loading user data or applying internal settings
    - **Visual feedback effects** (hover glow, color change): perceived as **instantaneous**
- **Startup Time**: The application must **fully launch and display the login menu within 2 seconds** of execution on a standard desktop system.
- **Simulation Experience**: The built-in simulator must respond fluidly and offer the feel of a real-time RPG combat system.
    - **Battle logs** must update progressively with **no freeze or lag** during execution
    - Visual transitions such as **HP bar decreases** must feel natural
    - A **minimum effective refresh rate** (approx. **30–60 Hz**) should be maintained to ensure clarity and consistency, especially if progressive UI animations (e.g., log updates or stat effects) are implemented
- **Data Persistence**: All user-created battle system templates must be **accurately saved and fully restored** after application restarts
    - A saved project must be **reloaded identically** upon reopening the app
    - **No data loss or corruption** is acceptable during normal use
    - Crashes or shutdowns must not erase the **latest saved state**
- **Crash Tolerance**: The application must **not crash** during:
    - Navigation between menus, especially transitions into the **simulation interface**
    - Execution of simulations with any template configuration
    - Normal editing and saving of **custom battle rules**
- **Bug Tolerance**: Minor, **non-blocking UI visual glitches** (e.g., misplaced button, flicker) are acceptable as long as they:
    - Do **not prevent functional use** of the affected feature
    - Do **not interfere with user comprehension** or smooth navigation
- **Cross-Platform Consistency**: On supported platforms (e.g., **Windows**, **macOS**), the software must:
    - **Maintain the same structure and functionality**
    - **Tolerate minor layout or visual differences**, provided they do not affect usability or access to any feature

3.5.4. Out of Scope

The following features were originally considered but will not be part of the final product:

- **General-purpose 2D game simulation**: including a level editor, event manager, and support for various gameplay modes beyond RPG.
- **Advanced RPG systems**: such as real-time battle mechanics, combo attack linking, and versatile simulation styles similar to games like Final Fantasy or EarthBound.
- **Complex customization tools**:
    - Stat editors, creating new stats and updating existing ones.
    - Skill and ability creators with emblem and element tagging.
    - Status effect and hazard systems (e.g., weather effects).
    - Equipment and inventory management, including bonuses/maluses, item crafting, and achievement trees.
- **Additional utilities**:

- Experience point calculators.
- In-app simulation video recording and saving.

These were excluded to maintain a clear scope focused on creating a functional, user-friendly turn-based RPG battle simulator.

3.6. Technical Stack & Requirements

3.6.1. Development Environment

The development of this desktop application relies primarily on **Qt Creator** as the main integrated development environment. It enables the complete design and implementation of both the application's logic and its graphical interface. **Qt Creator** provides built-in tools to manage project structure, integrate interface elements, preview layouts, test functionality, and deploy executable versions using appropriate development kits.

The project is developed using the **C++ programming language**, supported natively by Qt Creator for building responsive and visually structured desktop applications.

For version control and progress tracking, the project is maintained through **GitHub**, using both the GitHub web platform and its dedicated **GitHub Desktop** application to manage changes, synchronize versions, and ensure code history is safely stored.

Other supporting tools are occasionally used:

- **Visual Studio Code**: for drafting technical documentation.
- **Microsoft Office Suite**: for writing monthly reports and documenting progress.
- **DB Browser for SQLite**: to manually manage and inspect database tables used within the app.
- **ChatGPT**: as a support assistant for rewriting documentation segments and resolving technical uncertainties.

When a version of the app is packaged for testing, end users simply download the release archive, extract it, and launch the application via the included executable — **no installation of additional tools or libraries is required**.

The choice of **Qt Creator** was guided by its strong support for building visually structured interfaces, its integration with **C++**, and its practical suite of tools for managing, debugging, and deploying desktop applications.

3.6.2. Database Overview

The application currently uses a local relational database (SQLite) to support core data management. SQLite was chosen for its ease of use, low setup overhead, and compatibility with tutorial material used during development. It allows for straightforward handling of simple data types like strings and numbers.

Although SQLite is sufficient for the current single-user structure, the system remains open to migrating to more scalable solutions (e.g., PostgreSQL or MySQL) in future versions if features like user accounts or shared data become necessary.

Type of Database

- **SQLite (Relational)**
- Optimized for small-scale, local applications
- Easy to manage within Qt Creator integration
- May be upgraded later if larger or concurrent data handling is needed

Main Data Entities to Store

The application revolves around a few fundamental types of gameplay data:

- **Entities**: Characters with stats and attributes, created specifically for one Battle Template
- **Capacities**: Skills or abilities used by Entities during battles (e.g., "Tackle", "Sword Dance")
- **Battle Templates**: Defined setups that include two selected Entities and their four associated Capacities

Entities are currently stored in the code and are tied to their specific battle template. Capacities are more flexible — they can be reused or remain in a "library" without being linked.

Basic Relationships Between Data

- Each **Battle Template** contains **two Entities**
- Each **Entity** has **four Capacities**
- **Capacities** may be shared across multiple Entities
- A **Capacity** can exist independently (i.e., not yet assigned)

- **Entities** are **not shared** across Battle Templates — they are unique to the template

Estimated Data Volume

Data Type	Expected Quantity (per user)	Notes
Battle Templates	2–5	Most users will only create a few test templates
Entities	2 per template	Created fresh for each template
Capacities	Shared pool of 10–20	Some may be unlinked ("library" style)
Simulation Logs	0	No logs are stored in this version
Media Assets	Few dozen images/fonts	Stored externally, not in the database

Large media files like sounds or images are handled outside the database system. Future updates may consider audio support, but this is out of scope for now.

3.6.3. System Requirements

This section outlines the system requirements for running the application in its current development phase. Since the project is still under development, some specifications are subject to change as features evolve or performance is optimized.

Minimum Hardware Requirements

- **Target Platform**: Standard university-issued laptop computers.
- **Tested Hardware Example**:
  - **CPU**: Intel® Core™ i7-1065G7 @ 1.30GHz (up to 1.50 GHz)
  - **RAM**: 16 GB installed (6 GB used during app tests)
  - **Operating System**: Windows 11 Pro (64-bit)
- **Recommended RAM**: 6 GB minimum for stable use, 8 GB or more for headroom.
- **Graphics**: No dedicated GPU required. The application uses 2D pixel graphics with minimal animation and does not rely on GPU acceleration.
- **Storage**: ~100–200 MB estimated for application binaries and assets.

Operating System Compatibility

- **Primary OS Tested**: Windows 11 Pro (64-bit)
- **Compatibility**: May be compatible with other Windows versions and macOS, though not yet tested.
- **Architecture**: 64-bit systems recommended.

Display Requirements

- **Display Scaling**: Developed and tested at **150%** display scaling (Windows default for many laptops).
- **UI Style**: Considered **compact**, featuring a clean background, large buttons, and user-friendly navigation similar to *Super Mario Maker*.
- **Minimum Resolution**: Not formally defined. The current version runs in a windowed mode. Tentative recommendation: **1280×720** minimum for future fullscreen support.

Performance Considerations

- **Stability**: No slowdowns, crashes, or freezing observed in the current Phase 3 version.
- **Known Issues**: A prior crash occurred during simulation testing with excessive static `Capacity` object creation in `.h` files. This issue is under investigation and does not affect the current version.
- **Multi-App Use**: No noticeable performance issues when used alongside apps such as Chrome, VSCode, or GitHub Desktop. Not yet tested alongside media players or heavy background processes.

Storage Requirements

- **Installation Footprint**: Lightweight. Primary space used for image assets and font files.
- **Asset Handling**:
  - Images are stored in an external folder within the project directory and loaded at runtime.
  - Fonts are stored in the build folder and referenced accordingly.
- **Data Import/Export**: Not currently supported. Templates are internally managed and may be stored in a database in future versions.

Network Requirements

- **Connectivity**: No internet or local network is required for current functionality.
- **Future Plans**: Potential for online template sharing or remote battle functionality in later development phases.

3.7. Testing Strategy Overview

3.7.1. Testing Objectives

The primary objective of the testing process is to ensure that the application meets its defined requirements, behaves reliably across common scenarios, and delivers a smooth and satisfying user experience. The testing approach includes both functional and non-functional goals.

User Acceptance Criteria

- The user can navigate the application without critical bugs or crashes.
- All major features (battle system setup, simulation menu, and user interface) must behave as expected under normal use.
- Templates created by users must be retrievable, modifiable, and saved correctly.
- The interface must be user-friendly, intuitive, and provide clear visual feedback when interacted with.
- External users must be able to test the application with minimal instruction and complete core tasks (e.g., simulate a battle, configure a set of rules).
- Testing participants will complete a feedback form to assess satisfaction and usability.
- The final version must be stable enough to allow informal external testing.

Performance Goals

- **Application launch time**: under 2 seconds on a typical test machine.
- **UI responsiveness**: under 100 ms for button interactions and 200 ms for menu transitions.
- **Simulation behavior**: runs without freezing or crashing during battle processing.
- **Database operations**: retrieving, updating, and deleting battle templates must be completed under 500 ms from the user's point of view.
- The app must remain usable even when other standard software (browser, text editor, etc.) is running in the background.

Compatibility Requirements

- The application must run on **Windows 11 Pro** systems (primary target).
- Future testing on **macOS** is planned to ensure compatibility and layout consistency.
- Basic testing should be done at **150% display scaling**, with UI elements readable and accessible.
- A mouse-based interface is assumed; no touchscreen or special hardware is required.
- The application must function correctly in offline mode with no network dependency.

3.7.2. Testing Scope

The purpose of this testing scope is to define the functional areas and system behaviors that will be evaluated during the testing process. Testing will focus on ensuring a smooth user experience, correct battle mechanics, and stable data handling across core features.

Core Features Requiring Testing

- **Simulation Menu**: The most critical component of the application. The simulation must run without crashes, bugs, or severe slowdowns. The player must be able to execute a full battle scenario under different rule settings.
- **Battle Configuration (Rules Menu)**: Must allow users to set up valid battle conditions before launching a simulation. Includes testing default rule sets and user-defined rule configurations.
- **Character Menu**: Must allow for entity selection and association with capacities for simulation.
- **Damage Calculator Menu**: If integrated, must retrieve and apply correct damage values based on user selections and settings.
- **Battle Template System**: Must support creating, saving, loading, and updating templates. Functionality must work correctly with both predefined and user-customized values.
- All components depending on **database interaction** must be tested with insertion, modification, and deletion operations to ensure data integrity.

User Interface Testing

- **Buttons and Menus**: All interactive UI components (e.g., buttons, tabs, and navigation menus) must respond correctly to user input.
- **Navigation**: Users must be able to switch between menus and pages easily without confusion or error.
- **Simulation Interface**: Must be clear, intuitive, and capable of displaying battle progress in real-time.
- **Visual Feedback**: Basic visual effects such as hover and click animations will be tested for completeness but are not prioritized over core functionality.
- **Fullscreen Display**: Will be tested as an optional enhancement if implemented.

Battle System Validation

- **Win/Loss Conditions**:
  - A battle ends when one entity's HP reaches 0.
  - If both entities are unable to attack due to depleted PP, the winner is determined by who has the higher remaining HP.
  - A turn limit may be introduced as a configurable rule (e.g., 10 or 20 turns), though it is not required by default.
- **Edge Cases**:
  - Entities with extremely high stat values (e.g., up to 999) will be tested for correct handling.
  - Entities must always have at least one capacity assigned. Configurations with fewer than 4 capacities may be allowed but must be tested to prevent simulation crashes.
  - Invalid setups (e.g., entities with 0 in any stat) are considered out of scope and must be prevented by input validation.
- **Battle Log Accuracy**:
  - Battle progress must be displayed clearly during simulation using a real-time text log.
  - Each log entry must correctly describe which entity is acting, which move is being used, and the resulting outcome.

3.8. User Interface & User Experience

3.8.1. Importance

One of the most essential features of **Kritical Hit** is the possibility to appeal to a wide range of users—whether they are experienced with software tools, familiar with Pokémon games, or completely new to video games in general.

Traditional game engines often come with powerful but complex user interfaces designed for maximum flexibility. While these interfaces allow developers to create virtually any kind of game, they typically have a steep learning curve. Mastering them can be time-consuming, and only a limited number of users manage to do so fully.

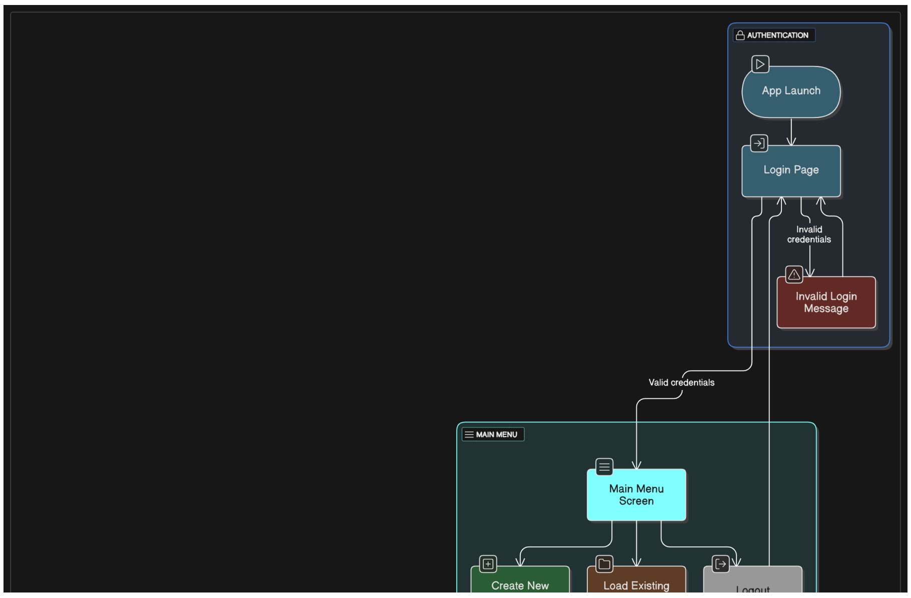
By focusing specifically on the **battle system** aspect of RPGs, **Kritical Hit** has the opportunity to adopt a more game-inspired user interface. This approach can take cues from Nintendo titles such as *Super Mario Maker* and *Super Smash Bros. Ultimate*. For instance, *Super Mario Maker* is often praised for its intuitive UX design, which empowers users to build levels creatively using tools that are simple, visual, and easy to understand.

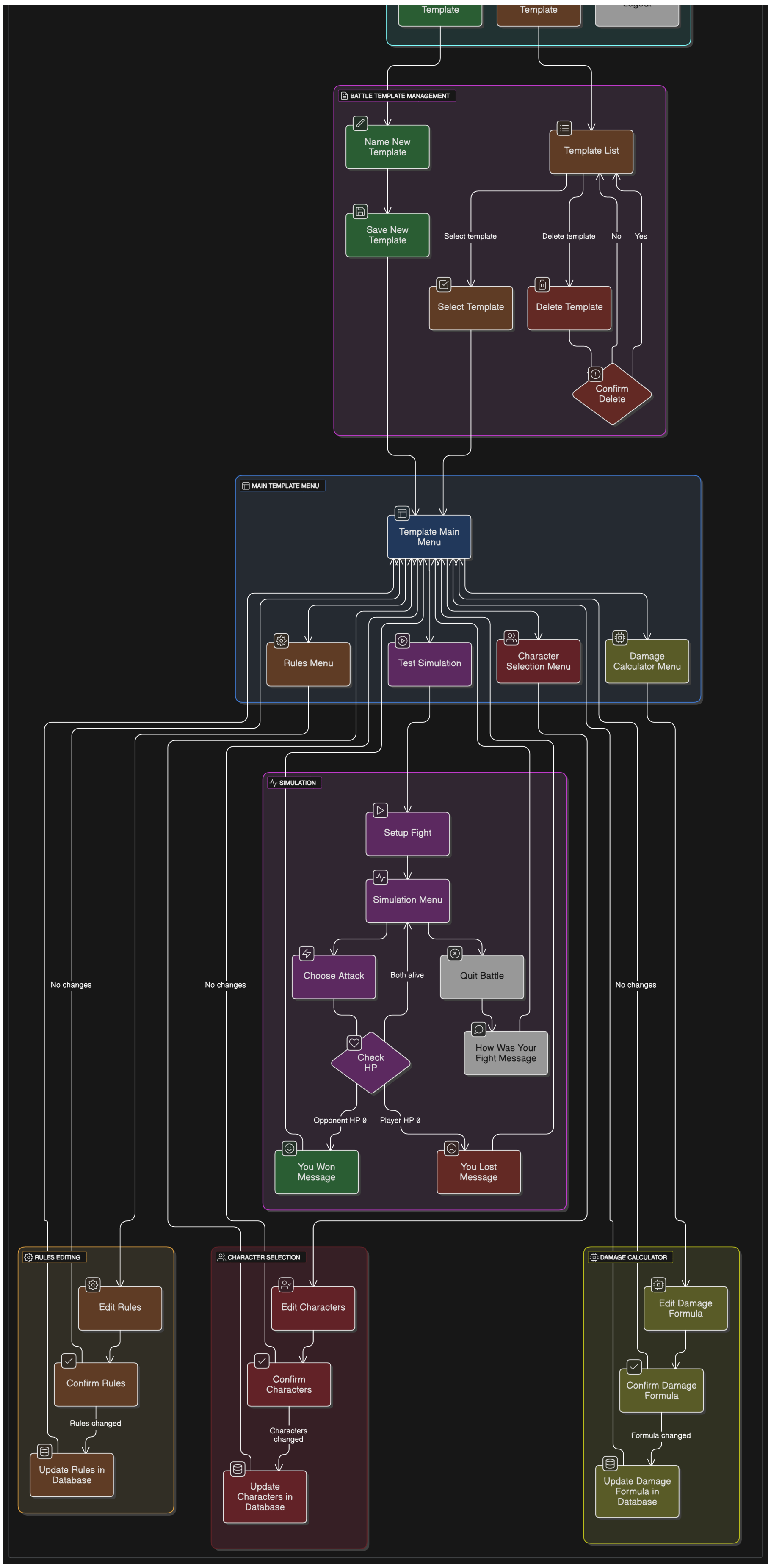
Similarly, the **Pokémon** series provides a great example of effective UI/UX in action. The battle interface in Pokémon games is distinct from the overworld and serves as the franchise's core gameplay component. Despite being rooted in traditional RPG mechanics, Pokémon stands out for its streamlined and accessible interface—most notably the use of a four-move menu. This clear, concise system ensures that players have all the critical information they need at a glance, enabling them to focus on strategy without feeling overwhelmed.

Kritical Hit can replicate this level of UI flexibility and clarity thanks to **Qt Creator's built-in UI design tools**, which allow for the creation of clean, modular, and intuitive interfaces without requiring extensive UI programming knowledge. This empowers designers to prototype and build user-friendly layouts that reflect familiar gaming interfaces while maintaining high usability across a wide audience.

3.8.2. User Flowchart

The User Flowchart, created using the *AI Flowchart Generator* tool, illustrates the various paths a user must follow to test different features while navigating through the application.



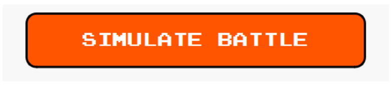


Note: This flowchart is based on the 2.5 version of the Kritical Hit project.

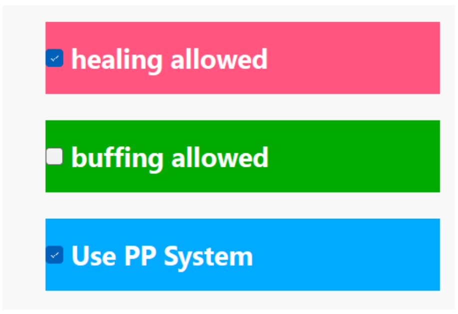
3.8.3. UI Elements

Thanks to Qt Creator's built-in UI tools, we can easily integrate essential interface elements that help users set up and simulate a Pokémon battle. These elements are not only simple to add but also fully customizable through stylesheets—allowing adjustments to colors, sizes, font families, and more—to ensure they remain visually distinct and intuitive to use.

Core Components

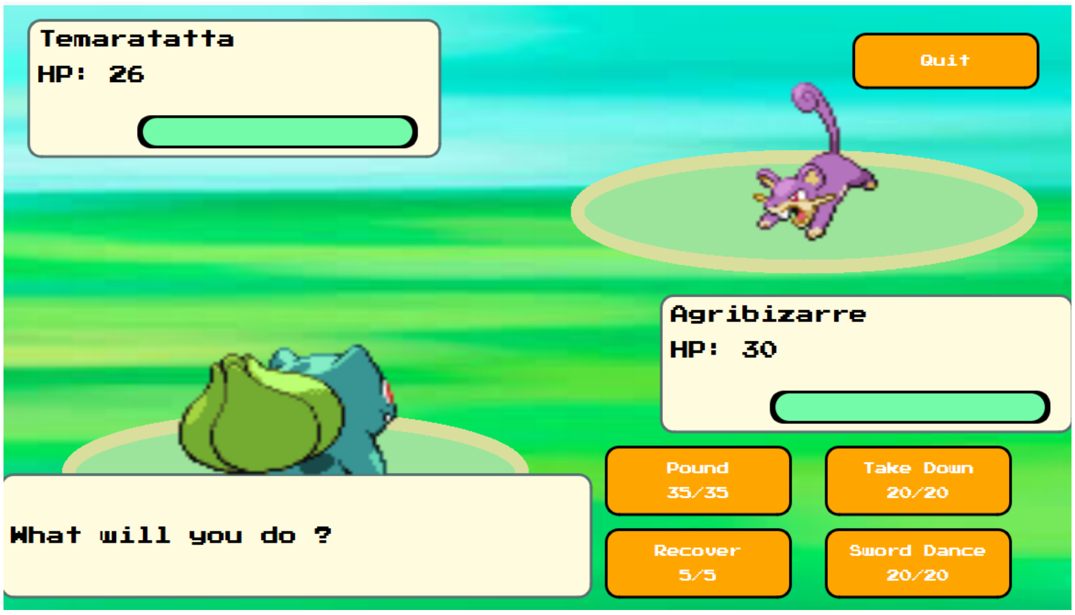


- **PushButtons**: Styled with rounded shapes and sometimes icons, these enable user actions and app navigation.



- **Checkboxes**: Allow users to toggle rules or conditions during battle setup.

Here's the simulation interface that also contains important UI components:



- **Battle Sprites**: 2D characters placed on elliptical battle platforms to separate them from the background.
- **Pokémon Info Panels**: Display entity names and dynamic HP bars that update during battle.
- **Battle Log**: Turn-by-turn summary of the ongoing battle (e.g., actions taken, HP changes).

3.8.4. Accessibility and Responsiveness

- The application targets a **general audience**, including casual players and developers. Its interface design emphasizes **clarity, large buttons**, and familiar gaming structures.
- **Keyboard navigation** may be supported in some screens. Mouse-based interaction is prioritized.
- While the app is desktop-focused, future consideration may be given to **touchscreens or tablets**. No minimum touch target size is currently defined.
- Display scaling at **150%** is supported and tested under Windows 11.

3.8.5. Visual Feedback and Error Handling

The app is designed to provide clear, simple messages and transitions that help the user understand what's happening:

- **Error messages** guide users when input is missing or incorrect:
  - *"Wrong email or password. Please try again."*
  - *"Please, write a valid email/password."*
- **End-of-battle feedback** offers a personalized, game-inspired experience:
  - *"Congratulations! You won the battle! What were your thoughts on this fight?"*

    - *"Game Over! What were your thoughts on this fight?"*
- **Quit confirmation**:
    - *"Are you sure you want to leave?"*
    - *"What were your thoughts on this fight?"*

A **loading indicator** may be added to visualize progress during processes such as simulation startup or data loading.

4. Technical Specifications

4.1. Overview

4.1.1. Document Purpose

This document has been created to provide the technical specification of a Moonshot Project named Kritical Hit. In fact, the Moonshot Project is a final evaluation imposed by the ALGOSUP school to create our own unique and professional project through the whole scolarity and to validate our Master level in Software development.

Furthemore, it's a complementary document to the Functionnal Specifications while showcasing the technical details required for the project and the strategies planned to accomplish it.

4.1.2. Project Presentation

Kritical Hit is a Desktop Application and Game Development assistant tool designed to support the creation of Combat Design systems for RPG games, with a particular emphasis on the mechanics found in Pokémon-style gameplay.

The application will feature a **user-friendly interface** aimed at providing intuitive navigation across all tools and functionalities. Special attention will be given to ensuring accessibility for both novice and experienced developers. Users will easily access core modules such as battle simulation, character setup, and rule customization through clearly organized menus and visually guided workflows.

A key component of the interface will be the **Simulation Menu**, where users can test and visualize the behavior of an RPG game's combat system in real time. This simulation environment enables immediate feedback and iterative design, allowing developers to fine-tune their systems effectively.

Once users are familiar with the simulation tools, they can dive into the **template customization** system. Kritical Hit will support the creation, modification, and storage of multiple battle templates. These templates include settings for battle rules, character selection, and other RPG assets. The internal database will allow templates to be saved, loaded, and shared, making it easy to manage different combat configurations and test scenarios.

This structure ensures that developers can first explore, test, and refine ideas through the interface and simulation tools, before committing to more advanced template editing.

4.1.3. System Overview

The software will be developed as a **desktop application** to guarantee high performance and responsiveness, especially given the significant volume of data involved in RPG combat system design and simulation. This local setup ensures smooth functionality without relying on internet speed, which is essential for maintaining consistency during real-time simulations and UI interactions.

Kritical Hit will be developed using **C++** in combination with the **Qt framework**, leveraging its robust support for graphical user interface (GUI) development. **Qt Creator**, a dedicated IDE for Qt applications, will be used to manage the design and implementation of the interface and underlying logic.

Qt offers cross-platform capabilities and a comprehensive set of software libraries and APIs tailored for scalable desktop and embedded applications. This makes it a strong fit for building an application that is both **modular and maintainable**, while supporting rich UI features and responsive simulation tools essential for Kritical Hit's goals.

The choice of C++ and Qt ensures that the application can handle complex data structures and intensive simulation processes, all while delivering a seamless and user-friendly interface experience.

4.2. System Architecture

4.2.1. App Architecture

The application was developed using the **Qt Framework**, specifically as a **Qt Widgets Application**. This format allows for a GUI based on `.ui` files designed visually with **Qt Designer**, while logic and behavior are implemented through **C++ source and header files**.

The project is configured to use:

- **Qt version**: 6.5.2
- **Compiler**: MinGW 64-bit
- **Build system**: `qmake` (Qt's original build tool)

> Although Qt supports both `qmake` and `CMake`, the `qmake` system was selected for its simplicity and ease of use, especially for a first-time Qt-based project. Cross-platform portability was not a priority at this stage, making `qmake` a suitable choice.

The main project file `KriticalHit_App.pro` contains all configuration data required for building the application, including file references, compiler flags, and module dependencies.

File Structure Overview

The following is the actual **on-disk file organization**, presented as a tree structure:

```
/Kritical-Hit
├── .git/                       # Git version control data
├── Dev/
│   ├── KriticalHit_App/        # Main application source directory
│   │   ├── Images/             # Asset folder for image files
│   │   ├── *.cpp               # Source code implementing application logic
│   │   ├── *.h                 # Header files declaring classes and interfaces
│   │   ├── *.ui                # UI layout files created with Qt Designer
│   │   ├── *.ts                # Translation file for internationalization
│   │   ├── *.pro               # qmake project file
│   │   ├── License.txt         # MIT License
│   └── build-KriticalHit_App-Desktop_Qt_6_5_2_MinGW_64_bit-Debug/
│       ├── debug/, release/    # Compiled binaries
│       ├── *.ttf               # Custom font files
│       ├── *.db                # Local database used by the application
│       ├── Makefile.*          # Build instructions generated by qmake
│       ├── ui_*.h              # Auto-generated headers from .ui files
├── Documents/                  # Specifications, test plans, project docs
├── Prototypes/                 # Early Qt prototypes and feature experiments
└── README.md                   # Project overview and setup instructions
```

4.2.2. Modules and Components

The **Qt Framework** is built around a set of modular libraries that provide specialized, cross-platform functionality. These **Qt modules** come in both source and binary form and are widely applicable across different Qt applications.

Modules with specific functionality (such as testing or database access) are often considered *add-on modules*, even when supported across all platforms.

Below are the main modules used in this project:

Module Name	Description
Qt Core	Provides the core functionality: event loops, signals/slots, object trees, and property management.
Qt GUI	Offers classes for windowing, 2D graphics, OpenGL, fonts, and basic imaging.
Qt Test	Enables unit testing with tools like `QTest`, `QSignalSpy`, and model testers.
Qt SQL	Supports SQL-based database integration with various database backends.

Some of these modules contain classes with specialized utilities, such as:

- `QDebug` – for easy debugging output
- `QApplication` – manages the GUI application's main control flow
- `QWidget` – base class for all UI objects

In addition to Qt modules, standard **C++ libraries** are also used in this project, including:

- `<string>` – for string management
- `<random>` – for generating random numbers (e.g., simulating attack variability)

Custom Functional Modules

To implement game-specific logic, several **custom functional modules** were developed. These contain the core functionality of the simulation:

- `Battle`: Manages the combat system between two characters, including turn-based logic and damage calculation.
- `Entity`: Represents a character (e.g., a Pokémon-like fighter), including its base stats, level, name, and a set of up to 4 `Capacity` objects.
- `Capacity`: Defines a move or ability (name, power, PP, type, etc.) used during battles.

Additional helper modules coordinate setup and data integration:

- `Setup`: Prepares the battle configuration, rules, and selected characters/movesets.
- `Database`: Manages the database connection and queries to store or retrieve character and template data.

User Interface Structure

The project uses several `.ui` files (with associated `.cpp` and `.h` files) to design the user interface:

- **Main Window Menu**: Landing screen where users log in with email and password.
- **Simulation Menu**: Main interface where battles are run and simulated.
- **Template Main Menu**: Central hub that links to other menus and features.
- **Rules Menu**: Allows the user to view, edit, and save the rules applied to battles.

Planned UI Menus for Future Versions:

- **New Template Menu**: Allows users to create and customize new battle templates.
- **Template Gallery Menu**: Displays existing templates stored in the database.
- **Character Selection Menu**: Interface to pick 2 characters from a pool of 6.
- **Damage Calculator Menu**: Enables users to modify the damage formula based on stats and selected rules.

4.2.3. Other External IT Tools

The development and documentation of the project were supported by a variety of external IT tools. These tools cover a broad range of use cases, from code editing and version control to project management, design, and AI assistance.

Tool Name	Description	Used For
Visual Studio Code	Lightweight yet powerful source code editor with built-in support for JavaScript, TypeScript, Node.js, and extensions for C++, Python, Java, and more.	Writing code, editing specification documents, and working with external prototypes.
GitHub / GitHub Desktop	GitHub is a cloud-based platform for hosting and managing code repositories with Git version control. GitHub Desktop provides a user-friendly interface.	Repository management, group collaboration, version control, creating issues and pull requests.
DB Browser	A visual tool to create, design, and manage SQLite database files, supporting browsing, querying, and modifying database tables without needing SQL command line.	Managing database connections, executing queries, modifying tables, and inspecting database contents.
ChatGPT (OpenAI)	An advanced AI chatbot built on GPT-4o, capable of answering questions, generating content, reviewing text, and even creating code.	Spelling checks, rewriting content, AI image generation for personas, brainstorming and ideation.
ClickUp	A flexible project management tool for organizing tasks, tracking progress, and managing time with boards, lists, and timelines.	Task and time management, sprint planning, collaborative project tracking.
Microsoft Office 365 (Online)	Web-based versions of Word, Excel, PowerPoint, and OneNote for collaborative editing and document management.	Writing weekly reports, project documentation, surveys (via Microsoft Forms).
NinjaAI	AI assistant specialized in rewriting and improving documents, reports, and written content with an emphasis on clarity and quality.	Reformulating technical content, improving documentation clarity, rewriting reports.
Draw.io (diagrams.net)	A free online diagram tool for creating flowcharts, UML diagrams, ER diagrams, and other graph-based visualizations.	Designing workflows, technical diagrams, and logic flows.
Figma	A collaborative design platform for UI/UX design, wireframes, and prototyping, enabling teams to work together in real time.	UI prototyping, concept art, wireframes for menus and interactions.

Tool Name	Description	Used For
Miro	An online collaborative whiteboard platform for ideation, project planning, and team brainstorming.	Creating personas, team discussions, collaborative design thinking.
Eraser.io	An AI-powered diagram generator that transforms text prompts into diagrams, charts, and visualizations.	Quickly generating UMLs, flowcharts, system architecture, and technical visuals based on text input.

4.3. Technologies Used

4.3.1. Front End

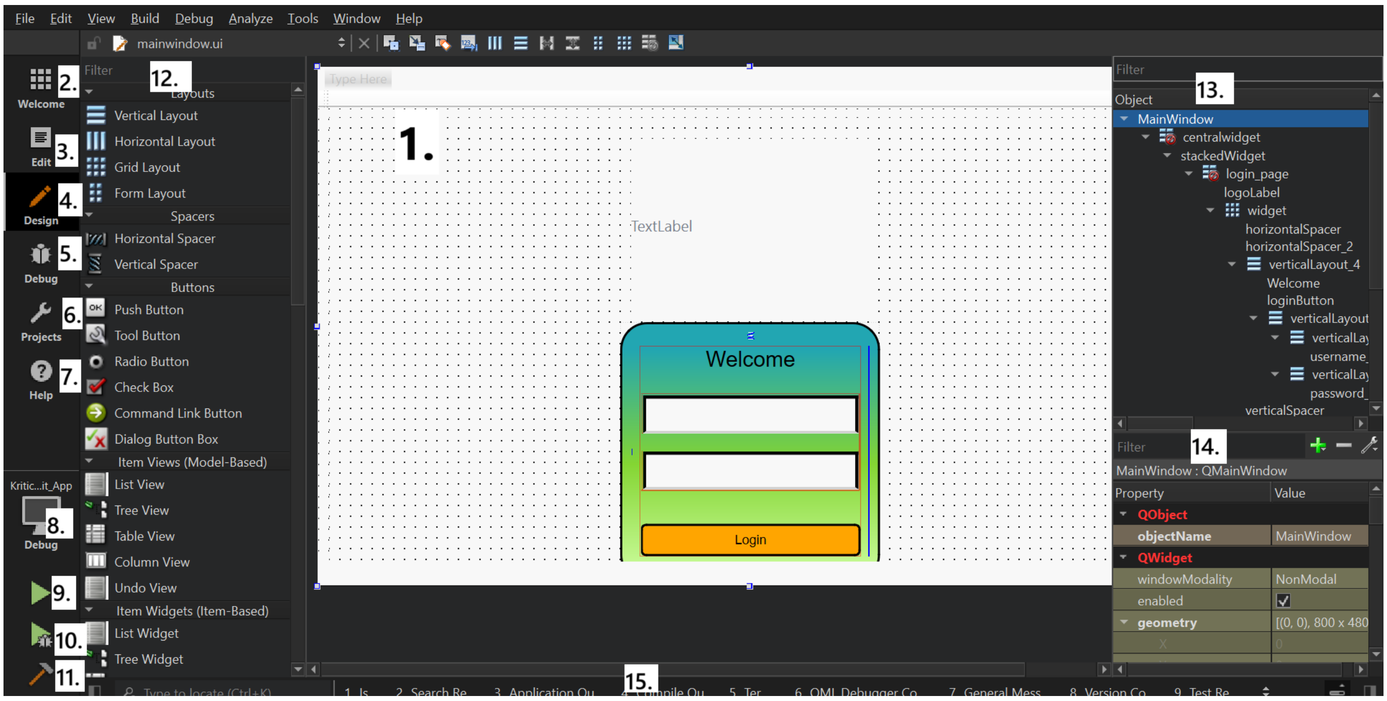
Kritical Hit's front end is almost entirely developed using the **Qt framework**, which provides both the core libraries and the Qt Creator IDE for designing and building applications.

Qt allows for both visual design and low-level programming through `.ui` (user interface) files and C++/Python back-end integration. It is ideal for desktop applications with custom UI logic and cross-platform capabilities.

> 💡 To open a project, double-click the `.pro` file. Qt Creator launches and lets you work through different development modes.

4.3.1.1 Qt Creator IDE Overview

Qt Creator is the official IDE for developing Qt-based applications. Here's an example screenshot of Kritical Hit's interface in **Design Mode**:



The Qt Creator interface in **Design Mode** is composed of several tools and panes. Here's a quick breakdown of the most important ones:

Component	Description
1. Form Editor	Drag-and-drop interface to design windows using Qt Widgets.
2. Welcome Mode	Start page for opening projects, viewing examples, or tutorials.
3. Edit Mode	Modify source code and project files (`.cpp`, `.h`, `.ui`).
4. Design Mode	Visual editing for `.ui` files with live layout previews.
5. Debug Mode	Analyze runtime behavior, memory, and breakpoints.
6. Projects Mode	Configure how your app is built and run.
7. Help Mode	Access Qt framework and Qt Creator documentation.
8. Kit Selector	Select the target platform (Debug/Release, Desktop/Embedded).
9. Run Button	Build and run the application.

Component	Description
10. Debug Button	Run debugger with optional breakpoints.
11. Build Button	Compile the application.
12. Widget Box	UI component toolbox: Buttons, Layouts, Containers, etc.
13. Object Inspector	Hierarchical list of all widgets in the current form.
14. Property Editor	Modify widget properties like text, visibility, font, etc.
15. Output View	Shows logs from build, debug output, and app status messages.

4.3.1.2 Design Mode and Widget Uses

Design Mode is the core feature of Qt Creator. It allows users to drag and drop UI elements directly into the form editor, making interface design intuitive and visual.

One of the primary UI elements is the **PushButton** widget, which provides a clickable command button.

When you select a PushButton and place it on the form editor, the **Object Inspector** automatically updates by adding the new widget as a child of the window widget.

To customize the button's appearance, right-click it and select **Change StyleSheet**. This lets you modify properties like font family, background color, and dimensions. You can also apply stylesheets to other widgets, including the window itself. This approach allows styling multiple widgets of the same family using inheritance and C++ classes.

To set up the button to perform an action, right-click the button and choose: *Go to Slot* -> *QAbstractButton* -> *clicked*.

Qt Creator will automatically generate a new function within the window widget class that executes when the button is clicked. For example:

```
void TemplateMainMenu::on_simulation_Button_clicked()
{
    ui->stackedWidget3->setCurrentIndex(1);
}
```

Additionally, the window widget's header file is updated to include this new function as a **private slot**.

4.3.1.3 Edit Mode, Project Structure, and Development Tools

When working in **Edit Mode**, Qt Creator organizes files into virtual categories for better readability. This organization does not necessarily reflect the actual file system structure but helps developers quickly find relevant files.

Category	Extension	Purpose
Headers	`.h`	Define class interfaces, variables, constants, and method declarations.
Sources	`.cpp`	Implement the functionality of methods and logic declared in headers.
Forms	`.ui`	XML-based layout files created via Design Mode, defining layout and widget properties.

> ⚙️ `.ui` files are converted into auto-generated C++ code during compilation.

Qt Creator also provides many features that enhance development productivity:

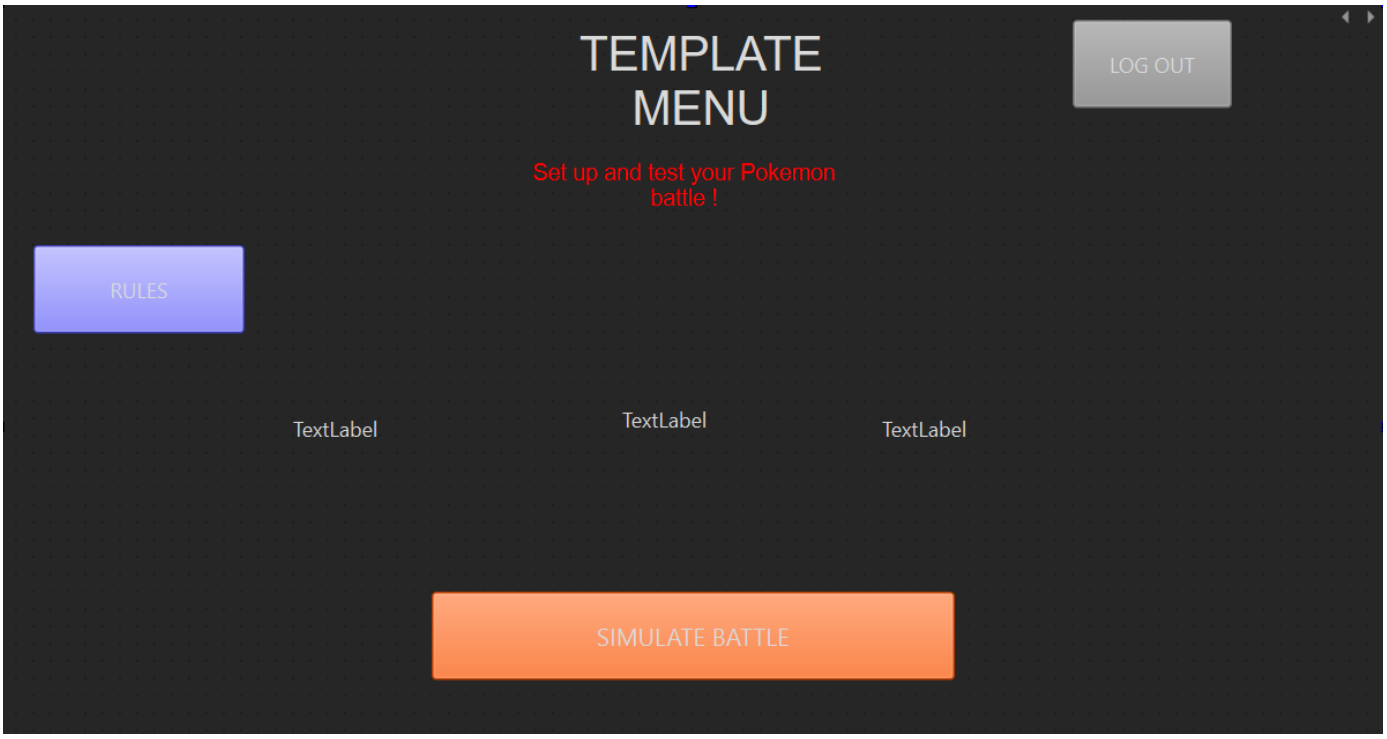
- **Code completion**: Suggests class names, functions, and parameters as you type.
- **Semantic highlighting**: Uses color coding to distinguish types, variables, and functions.
- **Syntax error checking**: Displays inline errors while coding to catch mistakes early.
- **Documentation tooltips**: Hover over keywords or functions to view quick descriptions.
- **Live UI preview**: Changes made to `.ui` files instantly update the design view.
- **Git integration**: Commit, push, and track changes directly within the IDE.
- **Keyboard shortcuts**: Enable fast navigation and editing (can be customized or expanded later).

4.3.1.4 Signal & Slot System

Qt Creator uses a **Signals and Slots** mechanism to enable seamless communication between objects. This powerful event-driven model allows different widgets and windows to interact efficiently. Slots are essentially private functions tied to widget events, while signals notify other parts of the program when something happens, enabling connections between widgets across different windows.

To illustrate, consider the **QStackedWidget** class, which manages a stack of widgets where only one is visible at a time. It works with a system of pages and indexes to display different views within the same container.

For example, if you design **Page 1** with certain UI elements and switch to **Page 2** in Design Mode, Page 2 will initially appear empty until you add widgets to it. You can add a new page by right-clicking the `QStackedWidget` object in the Object Inspector.



Use the two arrows at the top-right corner of the stacked widget editor to navigate between pages. This lets you update and organize the widget hierarchy across different pages.

You can also programmatically control page navigation. For instance, when the stacked widget reaches a certain page, you can trigger it to switch to another widget or window.

Example: Navigating Between Menus Using Signals and Slots

Suppose you want to switch from the **Main Template Menu** to a **Simulation Menu**:

1. In `templatemainmenu.h`, include the header for the simulation menu:

```
#include "simulationmenu.h"
```

2. Declare a private member instance of `SimulationMenu` and a slot function to handle menu switching:

```
private:
    SimulationMenu _simInfo;
private slots:
    void moveTemplateMenu();
```

3. In `simulationmenu.h`, declare a signal to indicate when the battle is finished:

```
signals:
    void battleFinished();
```

4. In `templatemainmenu.cpp` constructor, add the simulation menu widget to the stacked widget and connect the signal to the slot:

   ```
   ui->stackedWidget3->insertWidget(1, &_simInfo);
   connect(&_simInfo, SIGNAL(battleFinished()), this, SLOT(moveTemplateMenu()));
   ```

Now, when the simulation button on the main template menu is clicked, the stacked widget index updates to show the simulation menu. When the battle ends (in `simulation.cpp`), you emit the `battleFinished()` signal:

```
emit battleFinished();
```

This triggers *moveTemplateMenu()*, which resets the stacked widget index and returns the user to the main template menu page.

This system of signals and slots combined with stacked widgets provides a flexible and maintainable way to manage complex UI navigation and event handling in Qt applications.

4.3.2. Database Specifications

Overview

The application utilizes a **SQLite database** as its primary data storage solution.
While the project does not implement a traditional backend or API architecture, the database serves as a *crucial component* for managing game rules and settings.

Current Database Implementation

Schema

The current database schema is defined as:

```
CREATE TABLE RULES_SET1 (
    rulesID INTEGER PRIMARY KEY NOT NULL,
    healingAllowed INTEGER NOT NULL,  -- Boolean (0/1)
    buffingAllowed INTEGER NOT NULL,  -- Boolean (0/1)
    PPSystem INTEGER NOT NULL         -- Boolean (0/1)
);
```

Error Handling

The application implements **basic error handling** through:

- Return value checking for database operations
- Console logging via `qDebug()` for error tracking
- Graceful failure handling with default values
- Direct SQL error reporting through `QSqlError`

Database Access

Database operations are performed using **raw SQL queries** through Qt's SQL modules.
Example:

```
QSqlQuery query(rules_DB);
query.prepare("UPDATE RULES_SET1 SET buffingAllowed = :val WHERE rulesID = 111");
query.bindValue(":val", value ? 1 : 0);
```

Setup Instructions

1. Database Creation

- Use *DB Browser for SQLite* to create the initial database
- Place the database file in the **Qt Project build folder**

2. Project Configuration

- Add SQL module to project file (`.pro`):

```
QT += sql
```

- Include required headers in `mainwindow.h`:

```
#include <QSqlDatabase>
#include <QSqlQuery>
#include <QSqlError>
#include <QDir>
```

3. Connection Setup

- Add database connection variable to `mainwindow.h`:

```
private:
    QSqlDatabase DB_Connection;
```

- Initialize connection in `mainwindow.cpp`:

```
QDir databasePath;
QString path = databasePath.currentPath()+"/NAME.db";
DB_Connection = QSqlDatabase::addDatabase("QSQLITE");
DB_Connection.setDatabaseName(path);
```

Future Enhancements

The database structure will be expanded to include:

- **Battle template storage**
- **Entity data management**
- **Capacity object storage**

This expansion will improve memory efficiency by *moving data from runtime memory to persistent storage*.

Technical Notes

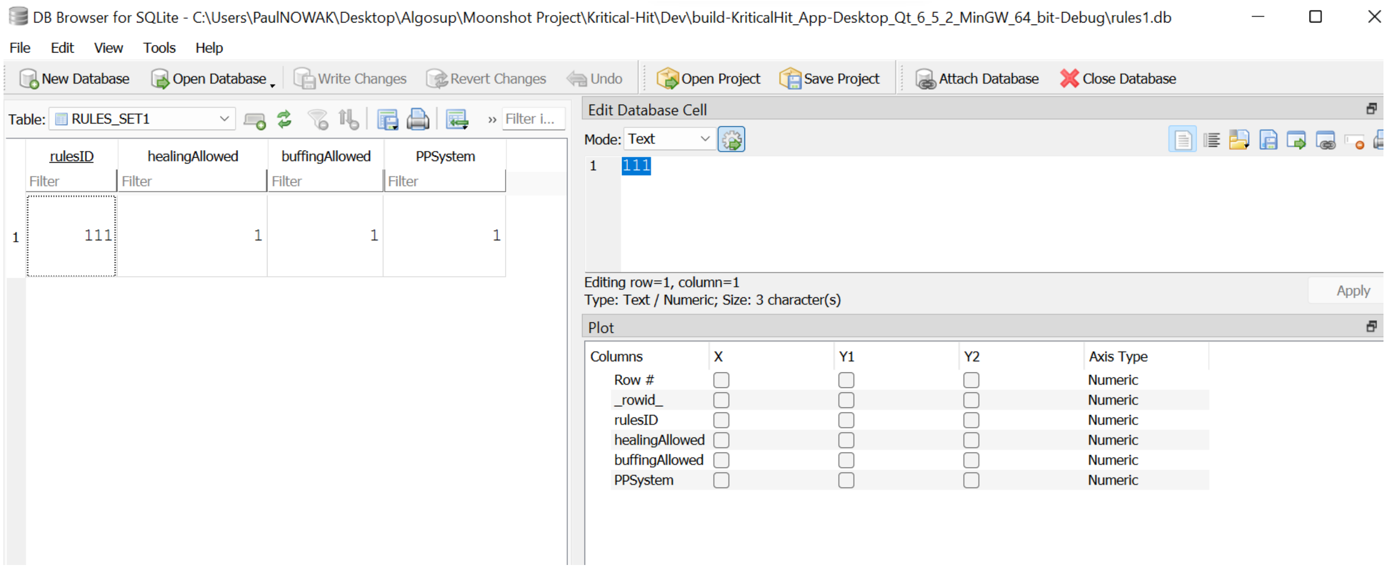
- SQLite was chosen for its *simplicity and easy integration* with Qt
- Boolean values are stored as **integers (0/1)** due to SQLite limitations
- Database file must reside in the **build folder** for runtime access
- A **single database connection** is maintained throughout the application lifecycle

---

Database Management

DB Browser for SQLite is used as the primary tool for:

- Table creation and modification
- Data viewing and editing
- SQL query execution
- Database structure visualization



DB Browser for SQLite - C:\Users\PaulNOWAK\Desktop\Algosup\Moonshot Project\Kritical-Hit\Dev\build-KriticalHit_App-Desktop_Qt_6_5_2_MinGW_64_bit-Debug\rules1.db
File Edit View Tools Help
New Database
Open Database
Write Changes
Revert Changes
Undo
Open Project
Save Project
Attach Database
Close Database
Edit Database Cell
Plot
Apply

Table: RULES_SET1

	rulesID	healingAllowed	buffingAllowed	PPSystem
1	111	1	1	1

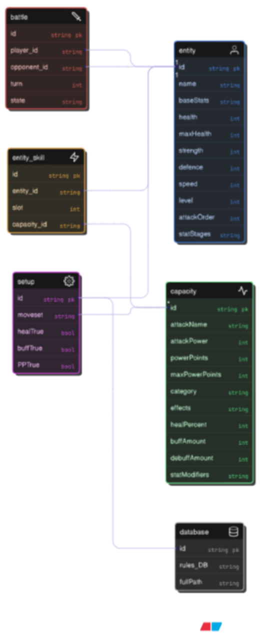
Mode: Text

1 111

Editing row=1, column=1
Type: Text / Numeric; Size: 3 character(s)

Columns	X	Y1	Y2	Axis Type
Row #	☐	☐	☐	Numeric
rowid	☐	☐	☐	Numeric
rulesID	☐	☐	☐	Numeric
healingAllowed	☐	☐	☐	Numeric
buffingAllowed	☐	☐	☐	Numeric
PPSystem	☐	☐	☐	Numeric

The class diagram below illustrates how the database integration fits into the overall application architecture:



4.3.3. Technical Constraints and Limitations

Development Environment Specifications

The development environment consists of a Windows 11 Pro (64-bit) system powered by an Intel® Core™ i7-1065G7 processor running at 1.30GHz with boost capabilities up to 1.50 GHz, supported by 16 GB of installed RAM, of which approximately 6 GB is utilized during application testing phases.

Current Technical Limitations

The application's development faces several significant constraints, particularly in memory management within the moveLibrary.h component, where attempts to expand beyond approximately 15 capacity objects result in application crashes, indicating a critical need for optimization through improved database implementation strategies.

Development is currently conducted using Qt version 6.5.2 with QMake as the build system, which presents its own set of limitations including reduced flexibility for large-scale projects, lack of support for complex build logic, and challenges in cross-platform dependency management that would be better addressed through CMake, especially for Qt 6+ projects.

Database Implementation Constraints

While the current SQLite implementation adequately handles basic rule storage, the database architecture requires significant optimization to manage larger datasets effectively. The setup and configuration process for database tables has proven more complex than anticipated, suggesting a need for streamlined database management procedures and improved architectural design to support future scalability requirements.

Development Environment Challenges

A significant technical constraint involves the debugging capabilities of the development environment, where disk management issues currently prevent the effective use of Qt Creator's integrated debugging tools, particularly when accessing the simulation menu. This limitation substantially impacts the development workflow and testing capabilities.

Testing and Deployment Limitations

The development and testing processes are currently restricted to a single Windows-based machine, which introduces potential risks in terms of unidentified platform-specific issues and cross-platform compatibility concerns. This single-environment testing limitation could potentially impact the application's reliability across different operating systems and hardware configurations.

Build System Considerations

The current utilization of QMake as the build system introduces several operational constraints, including the necessity for manual build process updates and limited integration capabilities with external tools. These limitations particularly affect the project's scalability and maintenance, suggesting that a future migration to CMake might be beneficial for more advanced development requirements.

Performance Monitoring Constraints

The current development environment lacks comprehensive tools for monitoring and optimizing memory usage, which has led to difficulties in identifying and resolving performance bottlenecks, particularly in relation to database operations and object management within the application's core functionality.

Future Considerations

These identified constraints and limitations are being actively addressed through planned improvements in database implementation, memory management strategies, and potential build system upgrades, with a focus on enhancing the application's stability, scalability, and overall performance characteristics.

4.3.4. Non-Functional Requirements - Technical Implementation Details

Performance Monitoring

Current Status • No active performance monitoring tools implemented • Response time requirements (100ms for clicks, 200ms for navigation) not currently measured • Planning to implement QT Test module for performance testing and monitoring

Planned Optimizations • Database architecture redesign to support larger datasets • Implementation of performance benchmarking tools • Integration of QT Test module for systematic testing

Configuration Management

Interface Implementation • UI layouts managed through .ui files • Widget placement, styling, and hierarchy controlled via Qt Designer • Custom font implementation (PressStart2P-vaV7.ttf) through stylesheet configuration

Current Limitations • Basic authentication system using simple if-else statements • Limited user settings persistence • No comprehensive configuration file structure

Security Implementation

Current Status • Basic file operations without specific security measures • Planned implementation of private database access • No current mechanism for preventing data corruption • Security measures for file operations to be implemented

Future Security Enhancements • Implementation of file locking mechanisms • Development of data corruption prevention strategies • Enhanced database security protocols

System Compatibility

Technical Requirements • Qt Version: 6.5.2 • Operating System: Windows 11 Pro • Hardware Specifications:

- Processor: Intel® Core™ i7-1065G7 or equivalent
- RAM: Minimum 16 GB recommended
- Storage: Sufficient for application and database growth

Development Approach • Platform-independent code implementation • No OS-specific dependencies • Designed for compatibility with standard Windows 11 Pro systems

Planned Technical Improvements

Short-term Goals • Implementation of performance monitoring tools • Development of robust configuration system • Enhancement of file operation security

Long-term Goals • Comprehensive testing framework implementation • Database optimization and security enhancement • Cross-platform compatibility testing • Robust error handling and data corruption prevention

These technical specifications aim to support the non-functional requirements while ensuring system reliability, security, and performance optimization.

4.4. Application Features

4.4.1. Battle System Rules

The battle system is inspired by the traditional Pokémon mechanics and defines how turn-based encounters between creatures are resolved. This system aims to simulate strategic one-on-one combat with various tactical elements.

General Rule

- The primary objective of a battle is to reduce the opponent Pokémon's HP (Hit Points) to 0, resulting in a knockout (KO).

Pokémon Attributes

Each Pokémon possesses the following characteristics:

- **Level**: A numerical indicator of experience and power. It affects the damage calculation.
- **Stats (6 total)**:
  - **HP (Hit Points)**: Determines how much damage the Pokémon can receive before fainting.
  - **Attack**: Influences the damage dealt by physical moves.
  - **Defense**: Reduces incoming damage from physical attacks.
  - **Special Attack**: Influences the damage of special (non-physical) moves.
  - **Special Defense**: Reduces incoming damage from special moves.
  - **Speed**: Determines the order of actions in a turn; the faster Pokémon attacks first.

Typing System

- Each Pokémon can have **one or two types** (e.g., Fire, Water, Grass), and each move also has a type.
- **Type Effectiveness**:
  - Super effective: 2× damage
  - Not very effective: 0.5× damage
  - No effect (immunity): 0× damage
- **Same-Type Attack Bonus (STAB)**:
  - If a move's type matches one of the user's types, a **1.5× damage bonus** is applied.
- Understanding the **type chart** is critical for maximizing damage output.

Moves and Attacks

- A Pokémon can know **up to four different attacks**.
- Most attacks have:
  - **Power**: A base value that contributes to the damage dealt.
  - **PP (Power Points)**: Indicates how many times a move can be used.
    - If a move's PP is depleted, it cannot be used.

        - If all moves have 0 PP, the Pokémon is considered to have no usable moves and may lose automatically (depending on system rules).
- Some moves are **non-offensive**:
    - **Healing moves** restore a portion of the user's HP, usually based on a fixed percentage or stat-based formula.
    - **Stat-altering moves** can **buff (increase)** or **nerf (decrease)** stats between **-6 and +6 stages**. These changes are persistent until overridden.

Damage Calculation Formula

The damage dealt by an offensive move is calculated using the following base formula:

```
BaseDamage = (((Level * 0.4 + 2) * Attack * Power) / (Defense * 50)) + 2
```

Where:

- `Level`: Level of the attacking Pokémon
- `Attack`: The attacker's stat (Attack or Special Attack, depending on move type)
- `Power`: The base power of the move
- `Defense`: The target's stat (Defense or Special Defense, based on move)
- The `+2` ensures minimum damage output

This base damage is then adjusted with the following multipliers:

```
BaseDamage = (((Level * 0.4 + 2) * Attack * Power) / (Defense * 50)) + 2
```

Multipliers:

- `STAB`: 1.5 if the move type matches one of the attacker's types, otherwise 1
- `Effectiveness`: 2 for super effective, 0.5 for not very effective, 0 for immune
- `CriticalHit`: 1.5 if a critical hit occurs (12.5% or 1/8 chance), otherwise 1

Battle Flow

- Battles are **turn-based**, and **each side performs one action per turn**.
- The **faster Pokémon (higher Speed)** acts first.
- At the end of a turn, the player is prompted to choose the next action:
    - Select a move
    - Quit the battle (if permitted by the rules)

Strategy Considerations

- Players must find the best strategy to **defeat opponents quickly** or **survive longer**, depending on the situation.
- Understanding typing, move selection, stat advantages, and turn order is key to success.

Current Implementation

- The current version includes **2 distinct Pokémon** with different stats.
- Only the Attack, Defence, Speed and HP stats are used.
- All Pokémon use the **same moveset**.
- **Typing system is currently disabled** in this version. All damage calculations are neutral (no STAB, type effectiveness, or immunities applied).

4.4.2. Battle Simulation Core

Overview

The battle simulation core manages the turn-based combat system through the SimulationMenu widget. Here's a detailed breakdown of its key components and functionality.

Widget Initialization and Setup

The SimulationMenu widget is accessed via a pushbutton from the template main menu. During the showEvent(), the widget:

- Initializes the battle through **initializeBattle()**
- Sets up character displays and HP bars
- Configures the battle scene with graphical elements
- Starts a 2-second timer before enabling player input

Core Battle Functions

Battle Flow Control

- **initializeBattle()**: Creates characters, movesets, and battle instance using the Setup class
- **playerTurn(int move)**: Handles player move execution and result processing
- **enemyTurn()**: Manages AI opponent moves using random selection (25% chance per move)
- **secondCharacterPerform(bool isPlayer, int move)**: Coordinates second character's turn
- **goToNextTurn()**: Advances turn counter and updates battle state

Move Resolution System

- **handleMoveResult()**: Processes move outcomes including:
    - Damage calculation and HP updates
    - Healing effects
    - Stat modifications
    - Battle continuation checks
    - PP (Power Points) management

Status Management

- Dynamic status messages display battle events through:
    - showStatusMessage() for fixed states
    - showDynamicStatusMessage() for action-specific updates
- Battle state tracking using the **MoveResultState** struct:

```
struct MoveResultState {
    bool continueBattle;
    bool hasHealing;
    bool hasBuffing;
    Entity* character;
    std::shared_ptr<capacity> moveUsed;
};
```

UI and Control Management

- Attack buttons are disabled during move animations
- Battle completion triggers **battleFinished()** signal
- Memory management includes proper cleanup of battle objects
- HP bars update in real-time using percentage calculations

4.4.3. Main Battle Functions

Core Combat Mechanics

- **calculateDamage()**: Computes damage using the formula:

```
damage = (((level * 0.4 + 2) * attack * power) / (defense * 50)) + 2
```

- **performMove()**: Executes moves and applies their effects
- **applyEffect()**: Processes different effect types (Attack, Heal, Buff)

State Management

- Battle progression tracked through **BattleState** enum:

```
enum class BattleState {
    Start,
    WaitingForPlayer,
    Animating,
    Finished
};
```

- Turn order determined by **checkAttackOrder()** based on speed stats

Effect System

- **EffectResult** struct tracks move outcomes:

```
struct EffectResult {
    int damageDealt;
    int hpHealed;
    short int attackBoost;
    short int defenceBoost;
    short int speedBoost;
};
```

- Stat modifications handled through **getStatMultiplier()**:
  - Positive stages: (2 + stage) / 2
  - Negative stages: 2 / (2 - stage)
  - Clamped between -6 and +6

AI Implementation

- Enemy moves selected using **randomMoveIndex()**
- Utilizes C++'s std::mt19937 random number generator
- Each move has equal 25% selection probability

4.4.4. Rules Implementation

The rules system implementation involves several key components working together:

Database Initialization

The database connection is initially established in **templateMainMenu.cpp** when its constructor is called. This is a crucial first step as all rule operations depend on having an active database connection.

Navigation to Rules Menu

When a user clicks the rules button in the template main menu, the stackedWidget index is updated to display the rules menu widget. This transition is handled through Qt's widget stacking system.

Rules State Management

The rulesMenu class interacts with the database class to manage rule states:

1. When the rules menu is displayed, the `showEvent` method is triggered and:

- Calls `database::getAllRules()` to retrieve the current state of all rules
- Updates the UI checkboxes to reflect the current rule states:

  ```
  QMap<QString, bool> rules = database::getAllRules();
  ui->healing_checkBox->setChecked(rules["healingAllowed"]);
  ui->buffing_checkBox->setChecked(rules["buffingAllowed"]);
  ui->PP_checkBox->setChecked(rules["PPSystem"]);
  ```

2. When the user clicks the Confirm button:

- The current state of all checkboxes is captured
- `database::setAllRules()` is called to update all rules simultaneously:

  ```
  bool newHealValue = ui->healing_checkBox->isChecked();
  bool newBuffValue = ui->buffing_checkBox->isChecked();
  bool newPPValue = ui->PP_checkBox->isChecked();
  database::setAllRules(newHealValue, newBuffValue, newPPValue);
  ```

- The `rulesConfirmed` signal is emitted to trigger the return to the template main menu

This implementation ensures efficient rule state management with minimal database operations by using batch updates instead of individual rule updates. The signal-slot mechanism handles the navigation flow, maintaining a clean separation between the UI and business logic.

4.4.5. Characters Implementation

Entity Class Overview

The Entity class serves as the foundation for all characters in the game, implementing core functionality for managing character stats and abilities. It represents any creature that can participate in battles, with comprehensive stat management and skill systems.

Stats Management System

Base Stats

The base stats are stored in a vector `_baseStats` and represent the character's innate, unchangeable attributes:

```
void Entity::setBaseStats(int health, int strength, int defence, int speed) {
    _baseStats[0] = health;     // Base HP
    _baseStats[1] = strength;   // Base Strength
    _baseStats[2] = defence;    // Base Defence
    _baseStats[3] = speed;      // Base Speed
}
```

Dynamic Stats

While base stats remain constant, the actual battle stats (`_health`, `_strength`, `_defence`, `_speed`) can be modified during gameplay:

• Initially set to match their corresponding base stats:

```
_maxHealth = _baseStats[0];
_health = _baseStats[0];
_strength = _baseStats[1];
_defence = _baseStats[2];
_speed = _baseStats[3];
```

• Can be temporarily modified through buffs/nerfs using the stat stage system:

```
void Entity::setStatStage(StatType stat, int stage) {
    if (stage > 6) stage = 6;       // Cap maximum buff
    if (stage < -6) stage = -6;     // Cap maximum nerf
    _statStages[static_cast<int>(stat)] = stage;
}
```

Health Management

The `checkHealth()` function ensures health values remain within valid bounds:

```
void Entity::checkHealth() {
    if (getHealth() > getMaxHealth()) {
```

```
        setHealth(getMaxHealth());    // Prevent overhealing
    } else if (getHealth() < 0) {
        setHealth(0);                 // Prevent negative health
    }
}
```

This function is crucial for maintaining game balance by: • Preventing health from exceeding maximum health after healing • Ensuring health doesn't go below zero when taking damage • Maintaining consistent health state throughout battle

This implementation creates a robust system where characters have permanent base attributes while allowing for dynamic stat modifications during gameplay, all while maintaining proper boundaries for health values.

4.4.6. Capacities Implementation

Core Structure and Enumerations

The capacity system is built around several key enumerations that define the fundamental types of moves and effects:

```
enum class MoveCategory {
    Physical,
    Special,
    Status
};

enum class EffectType {
    Attack,
    Buff,
    Debuff,
    Heal
};

enum class StatType {
    Strength,
    Defence,
    Speed
};
```

Stat Modification System

The `StatModifier` struct provides a robust way to define stat changes:

```
struct StatModifier {
    StatType stat;     // Which stat to modify
    int amount;        // Modification magnitude (+/-)
};
```

This structure is crucial for implementing buff/debuff mechanics, allowing precise control over: • Which stat is being modified (Strength/Defence/Speed) • The magnitude and direction of the modification • Multiple modifications from a single capacity

Capacity Class Implementation

The capacity class maintains several key attributes:

```
protected:
    std::string _attackName;
    int _attackPower;
    int _powerPoints;
    int _maxPowerPoints;
    MoveCategory _category;
    std::vector<EffectType> _effects;
    std::vector<StatModifier> _statModifiers;
```

Power Point (PP) Management

The PP system is implemented through the `useCapacity()` function:

```
bool capacity::useCapacity() {
    if (_powerPoints > 0) {
        _powerPoints--;
        return true;
    }
    return false;
}
```

This ensures: • Moves can only be used if PP is available • PP decrements after each use • Moves become unusable when PP reaches 0

Integration with Entity System

Each Entity maintains a moveset of 4 capacities:

```
std::array<capacity, 4> _skillList;
```

This allows: • Each entity to have a unique set of moves • Moves to be accessed and modified individually • PP tracking for each move separately

Stat Modification Tracking

The system includes comprehensive stat tracking:

```
std::vector<int> capacity::getStatChangeSummary() const {
    std::vector<int> summary(3, 0);  // Tracks all stat changes
    for (const auto& modifier : _statModifiers) {
        switch (modifier.stat) {
            case StatType::Strength: summary[0] += modifier.amount; break;
            case StatType::Defence:  summary[1] += modifier.amount; break;
            case StatType::Speed:    summary[2] += modifier.amount; break;
        }
    }
    return summary;
}
```

This implementation provides a robust foundation for complex battle mechanics while maintaining clear separation of concerns and efficient resource management.

4.4.7. Setting Up Battle

Rule Management System

The Setup class serves as the battle initialization controller, managing game rules and move configurations. It interfaces with the database to retrieve rule states:

```
Setup::Setup() {
    healTrue = database::getHealingRule();
    buffTrue = database::getBuffingRule();
    PPTrue = database::getPPRule();

    initializeMoveset();
    filterMoveset();
}
```

Rule Implementation

Three core gameplay rules are managed:

1. `healingAllowed` (healTrue): Controls whether healing moves are permitted
2. `buffingAllowed` (buffTrue): Determines if stat-boosting moves are allowed
3. `PPSystem` (PPTrue): Enables/disables the Power Point system for move usage

Moveset Management

The moveset system implements rule-based move filtering through two key functions:

```
void Setup::initializeMoveset() {
    moveset = {
        MoveLibrary::Pound,
        MoveLibrary::TakeDown,
        MoveLibrary::Recover,    // Healing move
        MoveLibrary::SwordDance  // Buff move
    };
}

void Setup::filterMoveset() {
    for (auto& move : moveset) {
        const std::vector<EffectType>& effects = move.getEffects();

        // Rule-based move replacement
        if (!healTrue && hasHealEffect(effects)) {
            move = MoveLibrary::VineWhip;
        }
        else if (!buffTrue && hasBuffEffect(effects)) {
            move = MoveLibrary::Bite;
        }
    }
}
```

PP System Integration

The PP system is integrated into the battle mechanics through the attack button handlers:

```
void SimulationMenu::on_attackButton_1_clicked() {
    if (battleSetup->getPPRule() == false || player->getNewSkill(0).useCapacity()) {
        newCheckAttack(0);
    } else {
        QMessageBox::warning(this, "No PP Left", "This move cannot be used anymore.");
    }
}
```

This implementation: • Checks if PP system is enabled • Verifies PP availability before allowing move usage • Provides user feedback when moves are depleted • Maintains move usage tracking throughout the battle

The Setup class is always instantiated before battle initiation, ensuring all rules and movesets are properly configured before combat begins. This creates a robust foundation for rule-based gameplay mechanics while maintaining clean separation of concerns.

4.4.8. Future Features

Planned Database Extensions

The current implementation lays the groundwork for several advanced features that would require more sophisticated database structures:

Character Selection System

```
-- Character Base Table
CREATE TABLE Characters (
    characterID INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    baseHP INTEGER,
    baseStrength INTEGER,
    baseDefense INTEGER,
    baseSpeed INTEGER
);

-- Team Configuration Table
CREATE TABLE Teams (
    teamID INTEGER PRIMARY KEY,
    userID INTEGER,
    slot1_characterID INTEGER,
    slot2_characterID INTEGER,
    FOREIGN KEY (slot1_characterID) REFERENCES Characters(characterID),
    FOREIGN KEY (slot2_characterID) REFERENCES Characters(characterID)
);

-- Character Movesets Table
CREATE TABLE CharacterMovesets (
    characterID INTEGER,
    moveID INTEGER,
    slot INTEGER CHECK (slot BETWEEN 1 AND 4),
    PRIMARY KEY (characterID, slot),
    FOREIGN KEY (characterID) REFERENCES Characters(characterID)
);
```

Damage Calculator Database

```
-- Damage Formula Configuration
CREATE TABLE DamageCalculator (
    calculatorID INTEGER PRIMARY KEY,
    formulaVersion TEXT,
    baseMultiplier FLOAT,
    criticalHitModifier FLOAT,
    effectivenessModifier FLOAT
);

-- or for text-based algorithm storage:
CREATE TABLE DamageAlgorithms (
    algorithmID INTEGER PRIMARY KEY,
    algorithmText TEXT,
    inputParameters TEXT,
    lastModified TIMESTAMP
);
```

Battle Template System

```
-- Master Template Table
CREATE TABLE BattleTemplates (
    templateID INTEGER PRIMARY KEY,
    templateName TEXT,
    rulesSetID INTEGER,
    characterSetID INTEGER,
    calculatorID INTEGER,
    FOREIGN KEY (rulesSetID) REFERENCES RULES_SET1(rulesID),
    FOREIGN KEY (characterSetID) REFERENCES Teams(teamID),
    FOREIGN KEY (calculatorID) REFERENCES DamageCalculator(calculatorID)
);
```

Integration Points

The system would require several key integration components:

1. Character Management:

```
class CharacterManager {
    QSqlDatabase characterDB;
    std::vector<Entity*> availableCharacters;

    void loadCharacterFromDB(int characterID);
    void updateMovesetWithRules(Entity* character, const RuleSet& rules);
};
```

2. Dynamic Damage Calculation:

```
class DamageCalculator {
    QSqlDatabase formulaDB;

    float calculateDamage(const Attack& attack, const Entity& attacker, const Entity& defender);
    void loadFormulaParameters();
};
```

3. Template Management:

```
class BattleTemplateManager {
    QSqlDatabase templateDB;

    void loadTemplate(int templateID);
    void saveTemplate(const BattleConfiguration& config);
    void applyTemplateRules(int templateID);
};
```

These planned features would significantly expand the system's capabilities while maintaining the current architecture's modularity and rule-based approach. The database structure would support complex relationships between different game elements while allowing for flexible configuration and easy expansion.

4.5. Product Deployment

4.5.1. Deployment Environment

Build Configuration

1. Project Configuration:

- Switch build configuration from Debug to Release mode in Qt Creator
- Optimize compilation settings for release deployment
- Remove debug symbols and assertions

Package Creation

```
# Directory structure for deployment
KriticalHit/
├── KriticalHit.exe        # Main executable
├── rules1.db             # Database file
├── qt6core.dll          # Qt dependencies
├── qt6gui.dll
├── qt6widgets.dll
└── platforms/           # Qt platform plugins
    └── qwindows.dll
```

Distribution Process

1. Create deployment package:

- Compress project folder and build folder into ZIP archive

- Include all necessary DLL files and dependencies
- Ensure database file is properly packaged

2. User Installation:

- Extract ZIP file to desired location
- Run KriticalHit.exe directly
- No additional framework installation required
- All dependencies included in package

3. Documentation:

- Installation instructions available in README.md
- Download links provided in project repository
- System requirements listed in documentation

4.5.2. Release Schedule

Version	Release Date	Status	Key Features
1.0	June 9, 2025	☑ Confirmed	- Basic battle system - Rule customization - Initial moveset
1.5	TBD	🔚 Planned	- Type System - Nerfing Moves - Custom movesets
2.0	TBD	🔚 Planned	- Character Selection - Expanded Stats
2.5	TBD	🔚 Planned	- Battle Template Management - Damage Calculator Menu

Each release will undergo thorough testing and quality assurance before deployment. Version numbers reflect significant feature additions and improvements to the core gameplay experience.

5. Test Plan

5.1. Introduction

5.1.1. Document Purpose

This test plan document outlines the testing strategy and methodology for the KriticalHit project, a desktop application for simulating RPG battle systems. The document serves multiple purposes:

- Establish a structured approach to verify that the application meets its defined requirements
- Define testing procedures for both developer testing and external user testing
- Provide a framework for identifying and documenting potential issues
- Ensure the quality and reliability of core features before release

The testing approach will primarily rely on:

- Manual testing by the developer
- Automated unit testing through Qt Creator's framework
- Beta testing phase with external users (post v1.0)
- Performance monitoring through Qt Creator's analysis tools

Success criteria will be measured through:

- Feature functionality verification
- Performance metrics (response times under 100-200ms)
- User satisfaction surveys
- Bug severity assessment

5.1.2. Objectives

Version Targets

- Version 1.0 release date: June 9th, 2025
- One primary testing cycle per development phase
- Additional testing following significant feature implementations
- Beta testing phase immediately preceding v1.0 release

Core Testing Priorities

1. Battle Simulation System
    - Combat mechanics
    - Turn resolution
    - Win/loss conditions
2. User Interface
    - Menu navigation
    - Visual feedback
    - Response times (≤100ms for interactions)
3. Rules Configuration
    - Database updates
    - Rule application
    - System integration

Testing Environment

- Primary development and testing on Qt Creator
- Unit testing through Qt's testing framework
- Performance analysis using Qt's built-in tools
- Manual state monitoring through debug outputs

Quality Metrics

Success will be determined by:

- Core features functioning without critical bugs
- UI response times meeting target thresholds
- Positive user feedback during beta testing
- Stable database operations for rule management

Testing will be conducted iteratively, with priority given to critical features essential for the v1.0 release. External testing will be introduced in later versions, incorporating feedback from both technical users (computer science students) and non-technical users (family members) to ensure broad usability.

5.2. Testing Strategy

5.2.1. Functional Testing

Unit Testing

Unit tests will focus on core functionality components using Qt's testing framework. Primary test targets include:

1. Battle Mechanics Components:
    - Damage calculation system
        - Base damage computation
        - STAB bonus application
        - Critical hit calculations
    - Turn order determination
    - HP management system
2. Class Functionality:
    - Entity class creation and management
    - Capacity class implementation
    - Battle class operations

3. Rule System:

    - Individual rule activation/deactivation
    - Rule interaction verification
    - Rule persistence testing

4. UI Component Testing:

    - Button functionality
    - Display update mechanisms
    - Battle text generation

Integration Testing

Priority will be given to testing critical component interactions:

1. Battle System Integration:

    - Complete attack sequence testing
        - Move selection → damage calculation → HP update
        - Multiple modifier application
        - Move effect processing

2. Database Integration:

    - Rule configuration persistence
    - Battle template management
    - Character/move data retrieval
    - Real-time database updates during battles

3. UI-Backend Integration:

    - Menu navigation with state preservation
    - Battle state visualization
    - Real-time data display updates

System Testing

Main test scenarios include:

1. Complete Battle Flows:

    - Default configuration battles
    - Custom rule set battles
    - Modified character loadouts
    - Alternative damage calculator implementations

2. Battle Conditions:

    - Pre-scripted AI scenarios for specific testing
    - Various rule combinations
    - Different character configurations

3. Edge Case Testing:

    - Maximum stat values (999)
    - Incomplete movesets (1-3 moves)
    - PP depletion scenarios
    - Multiple rule interactions

5.2.2. Non-Functional Testing

Performance Testing

1. Response Time Targets:

    - UI interactions: ≤100ms
    - Database operations: 100-500ms

- Menu transitions: ≤200ms

2. Load Testing:

   - Multiple battle template management (up to 5)
   - Continuous battle execution
   - Rapid UI interaction sequences

3. Memory Management:

   - Battle system resource monitoring
   - Database operation memory tracking
   - Long-session stability testing

Usability Testing

1. Interface Elements:

   - Button responsiveness
   - HP bar visibility and updates
   - Battle text readability
   - Checkbox clarity and function

2. Visual Feedback Criteria:

   - Response timing
   - State change visibility
   - Color scheme consistency
   - Font and image clarity
   - Glitch identification and tracking

Compatibility Testing

- Primary: Windows 11 Pro environment
- Future: macOS compatibility assessment
- Display scaling verification (150%)
- Hardware variation testing

5.2.3. Regression Testing

Testing Schedule:

- Major version releases (v1.0, v2.0)
- Feature addition verification
- Bug fix validation

Testing Summary Table:

Test Batch	Date	Tests Performed	Bugs Found	Tests Completed
Pre-v1.0	June 9th 2025	46	1	45
Pre-v1.5	TBD	0	0	0
Pre-v2.0	TBD	0	0	0

Progress Tracking:

- Qt debugger for performance monitoring
- Manual test case logging
- Bug report documentation
- Feature completion verification

5.3. Testing Process

5.3.1. Environment Setup

Software Configuration

- Qt Creator 11.0.2 (Community)
- DB Browser for SQLite Version 3.13.1
- Windows 11 Pro (Primary OS)

Testing Environment Requirements

1. Debug Mode Configuration:
    - Qt Creator in debug mode
    - Minimal running applications
    - DB Browser for database monitoring
2. Database Management:
    - Dedicated test tables in main database
    - Separate test data sets
    - Database state verification between tests

5.3.2. Test Development Structure

Test Organization

1. Source Code Testing:
    - Unit test files
    - Integration test files
    - All tests contained in main code folder
2. Test Documentation:
    - Test case templates
    - Bug report templates
    - External tester instructions

Test Prioritization

1. Critical Features (High Priority):
    - Battle system functionality
    - Database operations
    - Core UI components
2. Secondary Features:
    - Enhanced UI elements
    - Optional features
    - Performance optimizations

5.3.3. Test Execution Flow

1. Development Testing Phase:
    - Unit testing after each feature implementation
    - Integration testing for connected components
    - Commit-triggered test execution
    - Pre-release comprehensive testing
2. Testing Schedule:
    - New feature testing: 1-2 hours per feature
    - Release testing: Comprehensive test suite
    - Regular testing during development cycles

5.3.4. External Testing Protocol

Tester Management

- Target: 5-10 external testers

- Mix of technical and non-technical users
- Scheduled testing sessions

Testing Materials

1. Distribution Package:
   - Executable in ZIP format
   - Installation instructions
   - Testing guidelines
2. Feedback Collection:
   - Google Forms survey
   - Structured questionnaire
   - User experience evaluation

5.3.5. Bug Tracking System

Bug Documentation

1. Bug Report Elements:
   - Bug identifier
   - Description
   - Related test case
   - Discovery conditions
   - Discovery date
   - Current status
   - Resolution date
   - Fix methodology
2. Priority Classification:
   - Critical: Affects core functionality
   - High: Impacts main features
   - Medium: Affects secondary features
   - Low: Minor visual/cosmetic issues

Bug Resolution Flow

1. Discovery and Documentation
2. Priority Assessment
3. Resolution Planning
4. Fix Implementation
5. Verification Testing
6. Resolution Documentation

5.3.6. Test Execution Checklist

1. Pre-Test Setup:
   - Environment configuration
   - Database preparation
   - Test case documentation ready
2. Testing Sequence:
   - Unit tests execution
   - Integration testing
   - System testing
   - Performance verification
   - External user testing (when applicable)
3. Post-Test Activities:
   - Results documentation

- Bug reporting
- Fix prioritization
- Test report generation

5.4. Main Features to Test

5.4.1. Battle System Configuration

Database Operations Testing

1. CRUD Operations Validation

    - Create Operations:
        - Battle Template creation
        - Battle instance creation
    - Read Operations:
        - Battle Template retrieval
        - Entity data access
        - Capacity information
        - Damage Calculator configuration
        - RuleSet verification
    - Update Operations:
        - Battle Template modification
        - Damage Calculator adjustments
        - RuleSet changes
    - Delete Operations:
        - Battle Template removal
        - Battle instance cleanup

2. Character Data Validation

    - Stat Range Verification:
        - HP: 0 to max HP
        - Level: 1 to 100
        - Base Stats: 1 to 999 (Attack, Defense, Speed)
    - Stat Modification Tracking:
        - Buff/Nerf limits (-6 to +6 from base)
        - STAB bonus calculation
    - Move Management:
        - PP tracking per move
        - Move availability verification
    - Display Verification:
        - Character name
        - Current HP
        - Four moves display

3. Rule Configuration Testing

    - Minimum 8 rule combinations
    - Rule interaction verification
    - Rule persistence testing

Battle Mechanics Testing

1. Damage Calculation Validation Components to test:

    - Attacker Level
    - Attack Stat
    - Move Power
    - Defender's Defense
    - STAB Bonus
    - Critical Hit Bonus

2. Turn Order System

- Speed-based priority
- Tie-break random selection (50/50)
- Turn execution order

3. Win/Loss Conditions

   - HP depletion (0 HP)
   - PP depletion scenario
   - Optional turn limit rule

5.4.2. Simulation Interface

Core Functionality Testing

1. Battle Initialization

   - Screen transition (100-200ms)
   - Initial state setup
   - Character display
   - Move selection interface

2. Turn Execution Validation

   - Player move selection
   - AI move selection
   - Action resolution
   - State updates
   - Continuation check

3. HP Management

   - HP bar visual updates
   - Numerical HP display
   - Damage/healing reflection

Performance Metrics

1. Response Time Testing

   - Menu transitions: ≤200ms
   - Button interactions: ≤100ms
   - Battle state updates: ≤100ms

2. Resource Management

   - Memory usage monitoring
   - System resource tracking
   - Performance optimization verification

5.4.3. User Interface

Navigation Testing

1. Menu Flow Validation Primary Transitions:

   - Login ↔ Main Menu
   - Main Template Menu ↔ Simulation Menu
   - Main Template Menu ↔ Rules Menu
   - Main Template Menu ↔ Character Menu
   - Main Template Menu ↔ Damage Calculator
   - Main Menu ↔ Template Gallery
   - Main Menu ↔ New Template Menu
   - Template Gallery ↔ Main Template Menu
   - New Template Menu ↔ Main Template Menu

2. Error Handling

- Login validation messages
- Input verification
- Error message display
- Error recovery flow

Visual Elements Testing

1. UI Component Validation
    - Button scaling verification
    - Menu layout consistency
    - Element positioning
    - Visual hierarchy
2. Display Testing
    - Screen resolution adaptation
    - Element scaling
    - Text readability
    - Interface clarity

5.5 Out of Scope Features

5.5.1. Feature Exclusion Overview

Currently Excluded Features

1. General-Purpose 2D Game Simulation
    - Level editor functionality
    - Event management system
    - Multiple gameplay mode support
2. Advanced RPG Systems
    - Real-time battle mechanics
    - Combo attack system
    - Alternative battle styles (Final Fantasy, EarthBound)
3. Complex Customization Tools
    - Stat editor interface
    - Advanced skill creation system
    - Status effect management
    - Equipment and inventory systems
4. Additional Utilities
    - Experience point calculations
    - Battle recording functionality
    - Multi-user account system

Exclusion Rationale

- Time constraints for initial development
- Focus on Pokémon-style battle system expertise
- Maintaining clear, manageable project scope
- Prioritizing core battle mechanics quality

5.5.2. Future Implementation Possibilities

Planned Future Features (Post v2.5)

1. Battle System Expansions
    - Complete type effectiveness table
    - Enhanced stats management

    - Comprehensive skills system
    - Status effect implementation

2. User Experience Improvements
    - Battle template code export
    - Multi-user support system
    - Enhanced customization options

Extensibility Considerations

- Current type system supports future expansion
- Character system allows additional Pokémon
- Move system can accommodate new attacks
- User feedback will guide priority features

5.5.3. Testing Impact Analysis

Current Testing Boundaries

1. Scope Limitation Benefits
    - Focused testing approach
    - Concentrated quality assurance
    - Streamlined validation process
2. Feature Dependencies
    - Type table → Current type system
    - Stats menu ↔ Damage calculator
    - Character menu → Multiple feature dependencies
        - Stats system
        - Rules configuration
        - Skills management
        - Status effects

Future Testing Considerations

1. Integration Requirements
    - Regression testing for existing features
    - Compatibility verification
    - New feature integration testing
2. Test Case Evolution
    - Current test case adaptation
    - New test case development
    - Integration test expansion

5.5.4. Documentation and Communication

Feature Status Communication

1. Update Channels
    - GitHub repository updates
    - Social media announcements
    - Email notifications
    - Documentation updates
2. Bug Report Management
    - Classification of out-of-scope reports
    - Feature request tracking
    - Future implementation consideration

3. Boundary Documentation
    - Functional specification updates
    - Clear feature limitation documentation
    - Future roadmap maintenance

5.5.5. Testing Preparation for Future Features

1. Current System Preparation
    - Modular test structure
    - Extensible test frameworks
    - Clear documentation of test boundaries
2. Future Testing Strategy
    - Test case template preparation
    - Integration test planning
    - Performance benchmark updates
3. Quality Assurance Evolution
    - Test coverage expansion plans
    - Testing tool adaptation strategy
    - Documentation update procedures

5.6. Hardware Requirements

5.6.1. Testing Environment Specifications

Primary Test Environment

1. Hardware Configuration
    - System: Lenovo Windows 11 Pro (64-bit)
    - CPU: Intel® Core™ i7-1065G7 @ 1.30GHz (up to 1.50 GHz)
    - RAM: 16 GB installed
        - Minimum required: 6 GB
        - Recommended: 8 GB or more
    - Storage: 100-200 MB total space
        - Application binaries
        - Asset files
        - Test data storage
2. Display Requirements
    - Minimum resolution: 1280×720
    - Display scaling: 150% support
    - No dedicated GPU required
    - 2D graphics rendering capability
3. Performance Monitoring Tools
    - Windows Task Manager for CPU/Memory tracking
    - Qt Creator performance analysis tools
    - System resource monitors
    - Display response monitoring

5.6.2. Test Environment Variations

Secondary Test Configurations

1. Operating Systems
    - Windows 11 Pro (Primary)
    - Future macOS testing planned
    - Additional Windows versions TBD

2. Display Testing Matrix
    - Multiple screen resolutions
        - Starting from 1280×720
        - Various aspect ratios
    - Different laptop display sizes
    - Scaling configurations

5.6.3. Input Device Testing

Required Input Devices

1. Mouse/Touchpad Requirements
    - Basic functionality testing
    - UI element interaction verification
    - Cursor response validation
    - Click recognition testing
2. Keyboard Testing Requirements
    - Character input validation (login)
    - Special key functionality
        - Arrow keys
        - Space bar
        - Escape key
    - Input response verification

Out of Scope Input Testing

- Touchscreen functionality
- Alternative input methods
- Specialized gaming peripherals

5.6.4. Performance Testing Tools

Resource Monitoring

1. CPU Usage Tracking
    - Windows Task Manager
    - Qt Creator profiling tools
    - Performance baseline establishment
    - Usage pattern analysis
2. Memory Management
    - RAM usage monitoring
    - Memory leak detection
    - Resource allocation tracking
    - Peak usage measurement
3. Display Performance
    - UI responsiveness testing
    - Frame rate monitoring
    - Screen scaling verification
    - Visual clarity assessment

5.6.5. Testing Metrics

Performance Benchmarks

1. Response Times
    - UI interaction: ≤100ms
    - Menu transitions: ≤200ms

- Battle system updates: real-time

2. Resource Usage Limits

    - RAM: 6 GB minimum
    - Storage: 200 MB maximum
    - CPU: Baseline measurements TBD

3. Display Requirements

    - Minimum resolution support
    - Scaling functionality
    - Visual element clarity
    - UI consistency across configurations

5.7. Environment Requirements

5.7.1. Development Environment Setup

Core Development Tools

1. Qt Environment

    - Qt Version: 6.5.2
    - Qt Creator: 11.0.2 Community
    - Primary development IDE
    - Testing framework integration

2. Documentation Tools

    - Visual Studio Code
        - Test case documentation
        - Bug tracking documentation
        - General documentation management

3. Version Control System

    - GitHub Desktop
        - Commit management
        - Branch switching
        - Branch merging
        - Push operations

4. Database Management

    - SQLite Browser
        - Database table verification
        - Data reading/updating
        - Test data validation

5.7.2. Testing Framework Configuration

Testing Tools Integration

1. Qt Testing Framework

    - Framework familiarization required
    - Unit test implementation
    - Integration test support
    - Performance test capability

2. Database Testing

    - SQLite Browser for data verification
    - Table structure validation
    - Data integrity checking
    - Query testing

3. Performance Monitoring
    - Qt Creator built-in tools
    - System resource monitoring
    - Response time tracking

5.7.3. Version Control Strategy

Repository Management

1. Commit Strategy
    - Frequency: 1-5 commits daily
    - Feature-based commits
    - Bug fix documentation
    - Test case updates
2. Branch Structure
    - Main branch (stable releases)
    - Development branch (active development)
    - Feature branches (specific implementations)
    - Documentation branch
3. Testing Documentation
    - Test case tracking
    - Results documentation
    - Change verification
    - Bug tracking

5.7.4. Test Environment Management

Environment Setup

1. Testing Conditions
    - Quiet, controlled environment
    - Minimal background applications
    - Required tools activated as needed
    - Resource monitoring active
2. Cross-Platform Consistency
    - Machine-specific conditions documented
    - Code adaptations for different platforms
    - Configuration standardization
    - Test case adaptation as needed
3. Testing Workflow
    - Environment preparation checklist
    - Tool initialization sequence
    - Test execution protocol
    - Results documentation process

5.7.5. Quality Assurance Procedures

Testing Protocol

1. Pre-Test Setup
    - Environment verification
    - Tool availability check
    - Database state confirmation
    - Documentation preparation
2. Test Execution

  - Systematic test case execution
  - Result recording
  - Issue documentation
  - Performance monitoring

3. Post-Test Procedures

  - Results validation
  - Documentation update
  - Environment restoration
  - Issue tracking update

5.8. External User Testing

5.8.1. Participant Selection

Tester Demographics

1. Target Audience

  - Age range: 8+ years
  - Mix of technical and non-technical users
  - Basic computer literacy required
  - RPG gaming familiarity needed

2. Tester Categories

  - Technical Users
    - Computer science students
    - Software engineering background
  - Non-Technical Users
    - Family members
    - General gaming enthusiasts

3. Selection Criteria

  - Availability for testing
  - Gaming experience level
  - Computer proficiency
  - Willingness to provide detailed feedback

5.8.2. Test Environment Setup

Physical Environment

1. Room Requirements

  - Quiet, closed space
  - Adequate desk space
  - Proper lighting
  - Minimal distractions

2. Equipment Setup

  - Standard computer model
  - Working mouse
  - Power supply
  - Desk and chair

Session Parameters

- Duration: 5-20 minutes
- One-on-one observation
- Controlled testing conditions
- Immediate feedback capability

5.8.3. Testing Protocol

Test Sequence (V1.0)

1. Login Process
    - Provided temporary credentials
    - Authentication verification
2. Basic Simulation Testing
    - Navigate to simulation
    - Complete default battle
    - Observe win/loss conditions
3. Rules Configuration
    - Access rules menu
    - Modify specific settings (e.g., Healing rule)
    - Verify rule changes
4. Modified Battle Testing
    - Execute battle with new rules
    - Observe gameplay differences
    - Verify feature functionality

Task Documentation

1. Observer Notes
    - User behavior tracking
    - Issue identification
    - Time tracking
    - User comments
2. User Instructions
    - Test default settings
    - Explore configuration options
    - Evaluate UI experience
    - Report bugs encountered
    - Suggest improvements

5.8.4. Data Collection and Analysis

Feedback Collection

1. Survey Implementation
    - Google Forms platform
    - Structured questions
    - Rating scales
    - Open-ended feedback
2. Data Categories
    - UI responsiveness
    - Feature functionality
    - User experience
    - Bug reports
    - Feature suggestions

Results Analysis

1. Satisfaction Metrics
    - Interface usability
    - Task completion success
    - Feature effectiveness

- Overall experience

2. Issue Prioritization
   - Bug frequency
   - Feature request patterns
   - Performance concerns
   - UI/UX improvements

5.8.5. Testing Management

Session Organization

1. Scheduling
   - Based on development progress
   - Tester availability
   - Version readiness
   - Resource availability
2. Technical Support
   - Issue documentation
   - Bug reproduction steps
   - Solution tracking
   - Version documentation

Data Management

1. Survey Results
   - Google Forms analytics
   - Response categorization
   - Trend identification
   - Priority assessment
2. Implementation Planning
   - Feature prioritization
   - Bug fix scheduling
   - Version planning
   - Update documentation

5.9. Problem Reporting

5.9.1. Bug Classification System

Severity Levels

Severity Level	Description	Response Time
Critical	- Immediate app functionality impact - Data loss risk - Core feature failure	Within 24 hours
High	- Major feature malfunction - Significant user experience impact - Important functionality affected	Within one week
Medium	- Non-critical feature issues - Minor functionality impact - Workaround available	Before next version release
Low	- Cosmetic issues - Minor inconveniences - No functionality impact	Can be deferred

Bug Categories

1. Technical Classification
    - Functional errors
    - Syntax errors
    - Logic errors
    - Calculation errors
2. Scope Classification
    - Unit-level bugs
    - System-level integration bugs
    - Out of bounds bugs

5.9.2. Documentation Process

Bug Report Template

	Bug Name	Description	Related Test Case ID	Date of Discovery	Severity Level	Current Status	Date of Resolution	Resolution Method
Details								

5.10. Risks and Assumptions

5.10.1. Testing Risk Matrix

Risk	Impact	Probability	Mitigation Strategy
Limited testing time	High	High	- Focus on battle system and navigation testing first - Prioritize critical test cases - Reduce scope if necessary
Performance issues	Medium	Medium	- Regular performance testing during development - Optimize code when issues detected - Document performance baselines
Simulation crashes	High	Medium	- Implement error logging - Regular stability testing - Document crash conditions
UI rendering issues	Medium	Low	- Regular UI testing across different screens - Simplify complex UI elements - Document visual requirements
Database integration	Medium	Medium	- Separate test database tables - Test data integrity regularly - Document database operations
External user confusion	Medium	High	- Provide clear testing instructions - Document common user errors - Simplify test procedures

5.10.2. Testing Assumptions

Environment Assumptions

1. Testing Tools
    - Qt Creator debug tools will be available
    - Database testing tools will be accessible
    - Screen recording software will be available when needed
2. Test Data
    - Separate database tables for testing
    - Test data can be easily reset

- Data integrity can be maintained

Testing Process Assumptions

1. Time Management
    - Core features can be tested within timeline
    - Critical bugs can be addressed quickly
    - External testers will be available
2. Test Coverage
    - Battle system testing is highest priority
    - Navigation testing is second priority
    - Other features tested as time permits

6. Test Cases

Test Cases Written : 106 Test Cases Completed : 45

6.1. Battle Simulation

6.1.1. Unit Tests

Entity Creation Tests

ID	Description	Priority	Pre-requisites	Procedure	Exp	Out	Status
UNT_1111	Verify that the `Entity` class can be instantiated and correctly initialized with values such as name, level, stats, and type. Ensure all properties are properly assigned and retrievable.	Critical	1. Class files (`Entity.cpp`, `Entity.h`) must be created and available in the project. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Initialize an `Entity` object with test values. 3. Use assertions to verify each member is correctly set.	The unit test should pass, confirming all properties are correctly initialized and retrievable.	✓	Tested without unit test
UNT_1112	Verify that the current HP of an `Entity` instance is correctly tracked and returned. Test proper initialization and retrieval of remaining HP.	Critical	1. Class files (`Entity.cpp`, `Entity.h`) must be created and available in the project. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Initialize an `Entity` object with a known HP value. 3. Use assertions to verify that the remaining HP is correctly returned.	The unit test should pass, confirming the correct initialization and access to remaining HP.	✓	Tested without unit test

ID	Description	Priority	Pre-requisites	Procedure	Exp	Out	Status
UNT_1113	Verify that an `Entity` instance can be assigned a moveset composed of 4 mock `Capacity` objects. Ensure each move is stored and accessible.	High	1. Class files (`Entity.cpp`, `Entity.h`) must be created and available in the project. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Initialize an `Entity` object. 3. Create 4 mock `Capacity` instances and store them in a list or array. 4. Assign the moveset to the entity and use assertions to verify each move.	The unit test should pass, confirming the moveset is correctly assigned and all 4 capacities are accessible.	N/A	To be tested
UNT_1135	Verify that the `Entity` class can randomly select one of four mock `Capacity` instances from its moveset, with equal probability (25% each). This simulates a basic AI decision-making process.	Medium	1. Class files (`Entity.cpp`, `Entity.h`) must be created and available in the project. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Create an `Entity` instance. 3. Create four mock `Capacity` instances and store them in a list or array. 4. Assign the moveset to the entity. 5. Call the selection function multiple times and verify that all four capacities are chosen over time with roughly equal distribution.	Unit test should pass, confirming that each `Capacity` has an equal chance of being selected.	✓	Tested without unit test

Capacity Creation Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
UNT_1121	Verify that the `Capacity` class can be instantiated and correctly initialized with values such as name, attack power, power points, category, and effect type. Ensure all properties are properly assigned and retrievable.	Critical	1. Class files (`capacity.cpp`, `capacity.h`) must be created and available in the project. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Initialize a `Capacity` object with test values. 3. Use assertions to verify each member is correctly set.	Unit test should pass, confirming all properties are correctly initialized and retrievable.	✓	Tested without unit test

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
UNT_1122	Verify that the remaining power points (PP) of a `Capacity` object are correctly tracked and retrievable. Ensure accurate initialization and access.	High	1. Class files (`capacity.cpp`, `capacity.h`) must be created and available in the project. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Initialize a `Capacity` object with a known PP value. 3. Use assertions to verify the remaining PP is correctly returned.	Unit test should pass, confirming correct tracking and retrieval of PP.	✓	Tested without unit test
UNT_1123	Verify that a `Capacity` object can be assigned a stat modifier structure and that all modifier values are correctly stored and accessible.	Medium	1. Class files (`capacity.cpp`, `capacity.h`) must be created and available in the project. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Initialize a `Capacity` object. 3. Create a stat modifier structure. 4. Assign it and use assertions to verify each value.	Unit test should pass, confirming the stat modifier is correctly assigned and accessible.	N/A	To be tested

Battle Creation Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
UNT_1131	Verify that the `Battle` class can be instantiated and correctly initialized with mock entities, a `BattleState`, and a turn value. Ensure all properties are properly assigned and retrievable.	Critical	1. Class files (`Battle.cpp`, `Battle.h`) must be created and available in the project. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Initialize a `Battle` object with test values for entities, state, and turn. 3. Use assertions to verify each member is correctly set.	Unit test should pass, confirming that the `Battle` object is properly initialized.	✓	Tested without unit test

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
UNT_1132	Verify that the `BattleState` of a `Battle` object can be changed and correctly reflects the new state.	High	1. Class files (`Battle.cpp`, `Battle.h`) must be created and available in the project. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Initialize a `Battle` object. 3. Change its `BattleState` to another value. 4. Use assertions to verify the state change occurred.	Unit test should pass, confirming that the state change is handled correctly.	✓	Tested without unit test
UNT_1133	Verify that the turn order logic selects the faster entity based on their speed stat. If both speeds are equal, verify the system uses a 50% random selection between the two.	High	1. Class files (`Battle.cpp`, `Battle.h`) must be created and available in the project. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Create two mock entities with differing speed stats and assign them to a `Battle`. 3. Use assertions to verify the faster one acts first. 4. Repeat with equal speed values to test random turn handling.	Unit test should pass, confirming speed comparison and random resolution for ties.	✓	Tested without unit test
UNT_1134	Verify that the `Battle` class correctly increments the turn counter when a new turn begins.	Medium	1. Class files (`Battle.cpp`, `Battle.h`) must be created and available in the project. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Initialize a `Battle` object with a known turn value. 3. Call the method that increments the turn. 4. Use assertions to verify the turn was incremented.	Unit test should pass, confirming that the turn value increments as expected.	N/A	To be tested

Damage Calculation Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
UNT_1141	Verify that base damage is correctly calculated according to standard formula using inputs such as level, attack, power, and defense.	Critical	1. Class files responsible for damage calculation (e.g., `Battle.cpp`, `Battle.h`) must be created and available in the project. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Provide known input values: attacker level, attack stat, move power, and defender's defense. 3. Call the damage calculation function. 4. Use assertions to verify the result matches the expected value based on the formula.	Unit test should pass, confirming that the base damage is calculated accurately.	✓	Tested without unit test
UNT_1142	Verify that healing is correctly calculated based on the healing percentage, current HP, and max HP.	Medium	1. Class files responsible for healing logic must be created and available. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Initialize a mockup entity with known `currentHP` and `maxHP`. 3. Apply a healing percentage. 4. Use assertions to confirm the resulting HP is correct and does not exceed `maxHP`.	Unit test should pass, confirming that healing is accurately applied.	N/A	To be tested
UNT_1143	Verify that a stat increase (buff) is correctly calculated and applied to a mockup entity's stat.	Medium	1. Class files responsible for stat modification must be created and available. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Initialize a mockupentity with a known stat value. 3. Apply a buff of a certain amount. 4. Use assertions to verify the new stat value is correct.	Unit test should pass, confirming the buff is correctly applied.	N/A	To be tested

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
UNT_1144	Verify that a stat decrease (nerf) is correctly calculated and applied to a mockup entity's stat.	Medium	1. Class files responsible for stat modification must be created and available. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Initialize a mockup entity with a known stat value. 3. Apply a nerf of a certain amount. 4. Use assertions to verify the new stat value is correct.	Unit test should pass, confirming the nerf is correctly applied.	N/A	To be tested
UNT_1145	Verify that the STAB (Same-Type Attack Bonus) is correctly applied when a mockup entity uses a mockup capacity that matches its type.	Medium	1. Class files responsible for damage logic must be created and available. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Create a mockup entity and a mockup capacity with matching types. 3. Calculate STAB-adjusted damage. 4. Use assertions to verify the damage includes the STAB multiplier (e.g., 1.5x).	Unit test should pass, confirming STAB is applied correctly.	N/A	To be tested
UNT_1146	Verify that the critical hit bonus is correctly applied when a critical hit occurs in a damage calculation.	Medium	1. Class files responsible for damage logic must be created and available. 2. Qt Test framework must be properly set up and configured	1. Write a unit test in the Qt test framework. 2. Provide a scenario where a critical hit is guaranteed (e.g., force the flag). 3. Call the damage function and verify the bonus multiplier (e.g., 1.5x or 2x) is applied.	Unit test should pass, confirming critical hit bonus is applied properly.	N/A	To be tested

HP Management Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
UNT_1151	Verify that a mockup entity correctly handles damage and sets HP to zero when appropriate.	Critical	1. Mockup entity class must exist and be included in the test project. 2. Qt Test framework must be properly set up.	1. Write a unit test using a mockup entity. 2. Assign a known HP value and apply damage greater than or equal to it. 3. Use assertions to confirm resulting HP is 0 and any "faint" flag or equivalent is triggered.	Unit test should pass, confirming damage and zero HP logic.	✓	Tested without unit test

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
UNT_1152	Verify that a mockup entity heals properly and does not exceed its maximum HP.	Medium	1. Mockup entity class must exist and be included in the test project. 2. Qt Test framework must be properly set up.	1. Write a unit test using a mockup entity. 2. Assign known values for currentHP and maxHP. 3. Apply healing. 4. Use assertions to verify resulting HP is correct and does not exceed maxHP.	Unit test should pass, confirming healing and full HP handling.	✓	Tested without unit test

Stat Change Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
UNT_1161	Verify that stat buffs applied to a mockup entity do not exceed the allowed maximum limit.	Medium	1. Mockup entity class with stat management must be included. 2. Qt Test framework must be properly set up.	1. Write a unit test using a mockup entity. 2. Apply repeated buffs to a stat. 3. Use assertions to confirm the stat caps at the defined maximum.	Unit test should pass, confirming maximum buff limit is enforced.	N/A	To be tested
UNT_1162	Verify that stat nerfs applied to a mockup entity do not go below the allowed minimum limit.	Medium	1. Mockup entity class with stat management must be included. 2. Qt Test framework must be properly set up.	1. Write a unit test using a mockup entity. 2. Apply repeated nerfs to a stat. 3. Use assertions to confirm the stat caps at the defined minimum.	Unit test should pass, confirming minimum nerf limit is enforced.	N/A	To be tested

Win/Lose Condition Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
UNT_1171	Verify that when a mockup entity has 0 HP, the correct defeat message or outcome is triggered.	Critical	1. Mockup entity class must support message generation or defeat flag. 2. Qt Test framework must be properly set up.	1. Write a unit test using a mockup entity. 2. Reduce HP to 0. 3. Use assertions to check if appropriate message or lose condition is activated.	Unit test should pass, confirming 0 HP triggers correct defeat handling.	✓	Tested without unit test
UNT_1172	Verify that a mockup entity with no usable attacks (0 PP on all 4 mock moves) triggers a no-action condition.	Medium	1. Mockup entity with fake move list (4 moves) and PP values must be available. 2. Qt Test framework must be set up.	1. Write a unit test using a mockup entity. 2. Set all mock move PP to 0. 3. Assert that the system detects no available action.	Unit test should pass, confirming no-attack state is handled correctly.	N/A	To be tested

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
UNT_1173	Verify that exceeding a maximum number of turns in a mockup battle triggers a tie or timeout condition.	Medium	1. Mockup battle class with turn count tracking must exist. 2. Qt Test framework must be set up.	1. Write a unit test using a mockup battle. 2. Simulate turns until limit is exceeded. 3. Use assertions to confirm the expected outcome (e.g., tie/end condition).	Unit test should pass, confirming turn limit handling functions correctly.	N/A	To be tested

6.1.2. Integration Tests

1.2.1 Battle System & UI Integration

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
INT_1211	Verify that battle creation properly initializes and updates all UI elements, ensuring proper display of entity stats, HP bars, and move buttons.	Critical	1. All related unit tests must have passed (UNT_1111, UNT_1131, UNT_2171, UNT_2172, UNT_2173) 2. Entity and Battle classes must be implemented 3. UI components must be created and available	1. Create a battle instance with two entities 2. Initialize the UI components (HP bars, stat displays, move buttons) 3. Link battle state to UI elements 4. Verify initial UI state matches battle data 5. Trigger battle state changes 6. Verify UI updates reflect the changes	All UI elements correctly display and update according to battle state changes	✓	Tested without unit test
INT_1212	Verify that combat actions properly trigger visual feedback, including battle text updates, HP bar changes, and stat modification indicators.	Medium	1. All related unit tests must have passed (UNT_1141, UNT_1142, UNT_2171, UNT_2173, UNT_2174) 2. Battle system must be operational 3. UI components must be responsive	1. Initialize a battle with test entities 2. Set up UI elements for battle feedback 3. Execute a test attack sequence 4. Verify HP bar updates 5. Check battle text changes 6. Confirm visual indicators for stat changes	Combat actions correctly trigger all associated visual feedback elements	N/A	To be tested

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
INT_1213	Verify that battle templates properly load and configure both battle system and UI elements according to specified rules.	Low	1. All related unit tests must have passed (UNT_1131, UNT_3141, UNT_3144) 2. Template system must be operational 3. Battle system must be configurable	1. Create a test battle template 2. Load template into battle system 3. Initialize UI with template settings 4. Verify battle rules are applied 5. Check character stats loading 6. Confirm damage calculator configuration	Battle system and UI correctly reflect template configuration	N/A	To be tested

1.2.2 Battle Flow Integration

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
INT_1221	Verify that a complete attack sequence executes properly from selection to resolution, including damage calculation and HP updates.	Critical	1. All related unit tests must have passed (UNT_1112, UNT_1141, UNT_1151, UNT_2121, UNT_2171) 2. Battle system must be operational 3. Entity damage calculation system must be implemented	1. Initialize a battle with two test entities 2. Select an attack for the first entity 3. Execute the attack sequence 4. Calculate and apply damage 5. Update HP values 6. Verify battle state after attack	Complete attack sequence executes in correct order with proper damage calculation and HP updates	N/A	To be tested
INT_1222	Verify that multiple modifiers (buffs, STAB, critical hits) are properly applied and interact correctly during damage calculation.	Medium	1. All related unit tests must have passed (UNT_1141, UNT_1143, UNT_1145, UNT_1146, UNT_1161) 2. Modifier system must be implemented 3. Damage calculation system must support multiple modifiers	1. Set up a battle with test entities 2. Apply multiple stat modifiers to an entity 3. Execute an attack with STAB bonus 4. Force a critical hit 5. Calculate final damage 6. Verify all modifiers were properly applied	All modifiers are correctly applied and interact as expected in the damage calculation	N/A	To be tested

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
INT_1223	Verify that a healing move is properly processed and applied when a character uses it on themselves.	Medium	1. All related unit tests must have passed (UNT_1121, UNT_1122, UNT_1123, UNT_1142) 2. Healing system must be implemented 3. Battle system must support self-targeting moves	1. Initialize battle with an entity having a healing move 2. Reduce entity's HP below maximum 3. Execute the healing move 4. Verify PP reduction 5. Confirm HP increase within maximum limits	Healing move correctly restores HP and consumes PP	N/A	To be tested
INT_1224	Verify that turn order and resolution are properly handled based on speed stats and modifiers.	Medium	1. All related unit tests must have passed (UNT_1131, UNT_1133, UNT_1134, UNT_1135) 2. Turn system must be implemented 3. Speed comparison system must be operational	1. Set up battle with entities having different speeds 2. Apply speed modifiers to one entity 3. Initialize turn sequence 4. Verify turn order determination 5. Execute multiple turns 6. Check turn counter updates	Turn order is correctly determined by speed and modifiers, with proper turn progression	N/A	To be tested

6.1.3. System Tests

1.3.1 Complete Battle Flows

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
SYS_1311	Verify that a complete battle executes correctly using default configuration settings.	Critical	1. All related integration tests must have passed (INT_1211, INT_1221, INT_1224) 2. Battle system must be fully operational 3. Default configuration must be implemented	1. Initialize battle with default settings 2. Execute multiple turns with various moves 3. Verify damage calculations 4. Check turn order handling 5. Monitor battle state changes 6. Confirm proper battle conclusion	Complete battle executes successfully with all mechanics working together	✓	Finished
SYS_1312	Verify that battles execute correctly with custom rule combinations.	Low	1. All related integration tests must have passed (INT_1213, INT_3212) 2. Rule system must be fully operational 3. Custom configuration must be supported	1. Create battle with modified rules 2. Execute battle with rule variations 3. Verify rule interactions 4. Test edge case combinations 5. Confirm battle conclusion under rules	Battle executes correctly with custom rules properly affecting gameplay	N/A	To be tested
SYS_1313	Verify that battles execute correctly with different character configurations.	Low	1. All related integration tests must have passed (INT_1211, INT_3221) 2. Character system must support various configurations	1. Select different character combinations 2. Verify character stats and movesets 3. Execute battle with different characters 4. Check character-specific interactions	Battle executes correctly with different character configurations	N/A	To be tested

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
SYS_1314	Verify that battles execute correctly with modified damage calculator configurations.	Low	1. All related integration tests must have passed (INT_1212, INT_3213) 2. Custom damage calculator must be operational	1. Set up custom damage formula 2. Initialize battle with modified calculator 3. Execute multiple attacks 4. Verify damage calculations 5. Check battle balance	Battle executes correctly with modified damage calculations	N/A	To be tested

1.3.2 Edge Cases

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
SYS_1321	Verify that the battle system properly handles maximum stat values.	Medium	1. Related unit tests must have passed (UNT_1111, UNT_1141) 2. Battle system must handle maximum values	1. Create entities with maximum stats (999) 2. Execute battle with max-stat entities 3. Verify damage calculations 4. Check for overflow issues	Battle executes correctly with maximum stat values without overflow	N/A	To be tested
SYS_1322	Verify that battles execute properly with minimum move count.	Medium	1. Related unit tests must have passed (UNT_1113, UNT_1172) 2. Battle system must support limited moves	1. Create entities with single move 2. Execute battle with limited movesets 3. Verify move selection 4. Check battle progression	Battle executes correctly with minimum move configuration	N/A	To be tested
SYS_1323	Verify proper battle handling when all moves run out of PP.	Medium	1. Related unit tests must have passed (UNT_1122, UNT_1172) 2. PP depletion handling must be implemented	1. Start battle with low PP moves 2. Deplete all PP 3. Verify no-PP state handling 4. Check battle conclusion	Battle handles PP depletion correctly and concludes appropriately	N/A	To be tested

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
SYS_1324	Verify proper handling of multiple interacting rules.	Medium	1. Related integration test must have passed (INT_3212) 2. Rule interaction system must be implemented	1. Enable multiple interacting rules 2. Execute battle with rule combinations 3. Verify rule priorities 4. Check for conflicts	Rules interact correctly without conflicts	N/A	To be tested

6.1.4. Performance Tests

1.4.1 Response Time Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
PERF_1411	Verify that battle actions respond within acceptable time limits.	High	1. Related integration tests must have passed (INT_1211, INT_1221) 2. Performance monitoring tools must be configured 3. Battle system must be operational	1. Initialize battle with performance monitoring 2. Execute various battle actions 3. Measure response times 4. Test under different system loads 5. Record and analyze timing data	All battle actions complete within 100ms response time threshold	N/A	To be tested
PERF_1412	Verify that battle state updates occur in real-time without delays.	Medium	1. Related integration tests must have passed (INT_1211, INT_2222) 2. State monitoring system must be implemented	1. Set up battle with state monitoring 2. Trigger rapid state changes 3. Measure update times 4. Test concurrent updates 5. Verify state consistency	State updates complete within 50ms and maintain consistency	N/A	To be tested
PERF_1413	Verify smooth animation of HP bar changes.	Low	1. Related unit test must have passed (UNT_2171) 2. Animation system must be implemented	1. Set up HP bar monitoring 2. Trigger various HP changes 3. Measure frame rates 4. Test different animation speeds	HP bar animations maintain 60 FPS with smooth transitions	N/A	To be tested

1.4.2 Resource Usage Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
PERF_1421	Verify that memory usage during battles remains within acceptable limits.	High	1. Memory monitoring tools must be configured 2. Battle system must be fully operational	1. Start memory monitoring 2. Execute complete battle sequence 3. Monitor memory allocation 4. Check for memory leaks 5. Verify cleanup	Memory usage remains stable and leaks are not detected	N/A	To be tested
PERF_1422	Verify CPU usage efficiency during battle operations.	Medium	1. CPU monitoring tools must be configured 2. Battle system must be operational	1. Start CPU monitoring 2. Execute various battle scenarios 3. Measure CPU utilization 4. Identify processing peaks 5. Analyze bottlenecks	CPU usage remains under 25% during normal operation	N/A	To be tested
PERF_1423	Verify system stability during extended battle sessions.	High	1. System monitoring tools must be configured 2. Battle system must support extended sessions	1. Initialize extended battle session 2. Monitor system resources 3. Execute battles for 15+ minutes 4. Track performance metrics 5. Check for degradation	System maintains stability with consistent performance	N/A	To be tested

1.4.3 Load Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
PERF_1431	Verify system performance with multiple battle templates stored in the database.	Medium	1. Related integration tests must have passed (INT_3211, INT_3212) 2. Database performance monitoring must be configured 3. Template system must support multiple templates	1. Create multiple battle templates (5+) 2. Store templates in database 3. Monitor database access times 4. Test template loading speeds 5. Verify search and filter operations 6. Measure overall system responsiveness	System maintains performance with 5+ templates in database with sub-500ms access times	N/A	To be tested
PERF_1432	Verify system handling of rapid successive user actions.	High	1. Related integration test must have passed (INT_1221) 2. Input handling system must be implemented	1. Set up action monitoring 2. Execute rapid action sequences 3. Measure response times 4. Check action queue 5. Verify action processing	All actions are properly queued and processed in order	N/A	To be tested
PERF_1433	Verify system resource recovery after heavy load periods.	Medium	1. Resource monitoring tools must be configured 2. System cleanup processes must be implemented	1. Generate heavy system load 2. Monitor resource usage 3. End load period 4. Measure recovery time 5. Verify resource normalization	System returns to baseline resource usage within 30 seconds	N/A	To be tested

6.2. Interface

6.2.1. Unit Tests

2.1.1 Window Creation Tests

ID	Description	Priority	Pre-requisites	Procedure	Exp	Out	Status

ID	Description	Priority	Pre-requisites	Procedure	Exp	Out	Status
UNT_2111	Verify that the `AuthenticationMenu` window (Login) can be instantiated and displayed from its corresponding `.ui` file. Ensure the widget loads and becomes visible.	Medium	1. The `.ui` file for the Login window must exist. 2. The corresponding class must be generated and properly set up.	1. Instantiate the Login window class. 2. Use Qt functions to check if the widget is valid and visible.	UI widget is successfully created and appears visible.	✅	Tested without unit test
UNT_2112	Verify that both `simulationMenu` (**Critical**) and `CharacterSelectionMenu` (**Medium**) windows are correctly created and visible upon instantiation.	Mixed	1. Both `.ui` files must exist. 2. Corresponding classes must be implemented.	1. Instantiate both windows. 2. Verify visibility and proper rendering of each.	Both widgets are successfully created and displayed.	✅ (for simulationMenu)	Tested without unit test
UNT_2113	Ensure that all Template Management windows are properly created and visible: – `MainTemplateMenu` (**Critical**) – `NewTemplateMenu` (**Medium**) – `TemplateGalleryMenu` (**Low**)	Mixed	1. `.ui` files for all three windows must exist. 2. Each corresponding class must be implemented and compilable.	1. Instantiate each of the three window classes. 2. Confirm that each UI widget appears correctly.	All three windows are correctly instantiated and visible on screen.	✅ (Main Template Menu)	Tested without unit test
UNT_2114	Confirm that Configuration windows are properly created and visible: – `rulesmenu` (**High**) – `DamageCalculatorMenu` (**Low**)	Mixed	1. Corresponding `.ui` files and their respective classes must be available and compilable.	1. Instantiate both `rulesmenu` and `DamageCalculatorMenu` classes. 2. Verify visibility of each.	Both configuration windows are successfully loaded and shown on screen.	✅ (Rules Menu)	Tested without unit test

2.1.2 Button Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_2121	Verify that a button can be created and its visual properties (text, size, style, color) adjusted.	Critical	1. Qt Creator must be open with a working project.	1. Create a `.ui` file in the project. 2. Use the UI editor to add a button widget. 3. Run the project to verify the button appears. 4. Modify the button's text, size, and style properties in the UI editor. 5. Re-run and observe changes.	The button appears on the interface with the specified text, size, and style formatting.	✅	Tested without unit test

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_2122	Verify that a button emits the `clicked()` signal when clicked.	Critical	1. The `.ui` file must exist and contain a QPushButton element.	1. Access the button using `findChild<QPushButton*>()`. 2. Use `QTest::mouseClick()` to simulate a click. 3. Use `QSignalSpy` to check signal emissions.	The `clicked()` signal is emitted exactly once upon simulated click.	☑	Tested without unit test
UNT_2123	Verify that the button's state (visible/hidden, enabled/disabled) can be changed programmatically.	Medium	1. A `.ui` file with a QPushButton must be created and available.	1. Locate the button using `findChild<QPushButton*>()`. 2. Programmatically change its visibility (`setVisible(false)`), then check state. 3. Change `setEnabled(false)` and check.	Button becomes hidden/disabled as expected, and changes are accurately reflected in the UI.	☑	Tested without unit test

2.1.3 Label Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_2131	Verify that a QLabel is created and displays the expected text with correct formatting.	Critical	1. Qt Creator must be open with a working project. 2. QLabel widget must be available in `.ui`.	1. Create a `.ui` file and add a QLabel using the UI editor. 2. Set sample text, font size, and style via the property panel. 3. Run the project and confirm the label appears as configured.	QLabel is visible and displays the correct text with defined size and style.	☑	Tested without unit test
UNT_2132	Verify that QLabel text can be updated dynamically via code.	High	1. The `.ui` file must exist and contain a QLabel element.	1. Locate the QLabel using `findChild<QLabel*>()`. 2. Use `setText("New Text")` to update its text. 3. Use an assertion to confirm the text has been updated (`label->text()`).	QLabel's text is updated in the interface to the new value programmatically.	☑	Tested without unit test

2.1.4 Login Input Field Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_2141	Verify creation of login and password input fields using `QLineEdit` and validate basic layout behavior.	Medium	1. Qt Creator must be open with a working project. 2. `QLineEdit` widgets must be placed in the `.ui` file.	1. Create a `.ui` file and add two `QLineEdit` widgets for login and password. 2. Add a `QPushButton` for login confirmation. 3. Run the UI and ensure fields are visible and editable.	Input fields for login and password are displayed correctly and accept user input.	☑	Tested without unit test

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_2142	Verify that the password input field hides characters as the user types.	Medium	1. `.ui` file must contain a `QLineEdit` configured for password entry.	1. Locate the password input field using `findChild<QLineEdit*>()`. 2. Set its echo mode using `setEchoMode(QLineEdit::Password)`. 3. Simulate user typing.	Characters typed into the password field are visually hidden (e.g., replaced by dots).	✓	Tested without unit test
UNT_2143	Verify input field error handling for incorrect login or password submission.	Medium	1. `.ui` file must contain login form fields and a submit button.	1. Access login and password `QLineEdit` widgets in code. 2. Set up logic with `if/else` to compare input against correct values. 3. Simulate incorrect input and press the button. 4. Display error label.	When incorrect data is submitted, an error message or visual indicator is shown to the user.	✓	Tested without unit test

2.1.5 Image Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_2151	Verify that an image can be inserted and displayed in the UI using a QLabel.	Medium	1. Qt Creator must be open with a working project. 2. An `images/` folder must exist next to the project.	1. Create a `.ui` file and add a `QLabel` to hold the image. 2. Place the image file in the `images/` folder. 3. In code, use `QPixmap` and `setPixmap()` to load and insert the image into the label. 4. Run the project to verify.	The image is displayed in the `QLabel` on the UI as expected.	✓	Tested without unit test
UNT_2152	Verify that visual properties of the inserted image (size, alignment, scaling) are applied correctly.	Medium	1. A `.ui` file must exist with a `QLabel` displaying an image. 2. The image source must still be present.	1. Locate the image label using `findChild<QLabel*>()`. 2. Apply visual changes using methods like `setFixedSize()`, `setAlignment()`, or `setScaledContents(true)`. 3. Run the project to observe the effects.	The image is displayed with the specified size, alignment, and scaling properties.	✓	Tested without unit test

2.1.6 Checkbox/Radio Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_2161	Verify that a checkbox can be created and displayed in the UI.	High	1. Qt Creator must be open with a working project.	1. Create a `.ui` file. 2. Add a `QCheckBox` widget to the form using the UI editor. 3. Run the project.	The `QCheckBox` is visible in the UI and can be checked or unchecked.	✓	Tested without unit test

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_2162	Verify that the checkbox emits signals correctly when toggled.	High	1. A `.ui` file must be created and must contain a `QCheckBox` widget.	1. Locate the checkbox in code using `findChild<QCheckBox*>()`. 2. Connect its `stateChanged(int)` or `toggled(bool)` signal to a slot or test handler. 3. Simulate check/uncheck and observe the signal behavior.	The checkbox emits appropriate signals when its state changes (checked/unchecked).	✓	Tested without unit test

2.1.7 Display Element Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_2171	Validate creation and dynamic update of the HP bar based on HP/max HP values.	Critical	1. Qt Creator must be open with a working project.	1. Create a `.ui` file. 2. Add a progress bar or custom gauge widget to represent HP. 3. In code, initialize it with test values for current and max HP. 4. Run the project. 5. Modify HP value and observe update.	The HP bar is displayed, accurately reflecting current HP in proportion to the max value, and updates dynamically.	✓	Tested without unit test
UNT_2172	Verify display of Pokémon details (name, sprite, and numeric HP).	High	1. Qt Creator must be open. 2. Pokémon sprite image must exist in the project's image folder.	1. Create a `.ui` file. 2. Add a `QLabel` for the name and HP, and an image display widget. 3. Load and assign a test Pokémon name, sprite, and HP. 4. Run the project. 5. Change HP in code and observe the update.	The UI correctly displays the Pokémon's name, sprite, and HP, and updates the HP value when changed in code.	✓	Tested without unit test
UNT_2173	Verify correct display and updating of capacity (attack) buttons with PP values.	Critical	1. Qt Creator must be open with a working project.	1. Create a `.ui` file. 2. Add a `QPushButton` to represent a move. 3. Set the button text to show the move's name and remaining PP. 4. Run the project. 5. Simulate PP decrease and update button text.	The button shows the move name and PP, and updates correctly after a simulated use (PP -1).	✓	Tested without unit test

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_2174	Validate the dynamic update of battle text in a label widget.	Medium	1. Qt Creator must be open with a working project.	1. Create a `.ui` file. 2. Add a `QLabel` to hold battle narration. 3. Run the project to confirm visibility. 4. Programmatically update the label's text. 5. Run and verify updated text appears as expected.	Battle narration is shown inside the label and updates successfully as the text changes.	☑	Tested without unit test
UNT_2175	Validate that pop-up message windows can be triggered and displayed properly.	Medium	1. Qt Creator must be open with a working project.	1. Create a `.ui` file. 2. Implement a message box or custom popup dialog in code. 3. Trigger the popup from a simulated event. 4. Run the project and confirm popup is displayed.	A message popup appears when triggered, showing the desired content or alert.	☑	Tested without unit test

2.1.8 Menu Component Tests

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_2181	Verify creation of a main menu with UI elements.	Critical	1. Qt Creator must be open with a working project.	1. Create a new `.ui` file. 2. Use the UI editor to add a QWidget window. 3. Insert various UI elements (e.g., buttons, labels) into the window. 4. Run the project and ensure the widget window appears as expected.	The QWidget window is displayed with all added UI elements properly visible and interactive.	☑	Tested without unit test

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_2182	Validate navigation between pages using a QStackedWidget and button signals.	High	1. A `.ui` file with a QStackedWidget has been created.	1. Add a `QStackedWidget` with two pages. 2. On Page 1, add unique UI elements including a navigation button. 3. On Page 2, add different UI elements and another button. 4. Implement signal-slot connections in code so each button switches between the two pages. 5. Run the project. 6. Click the button on Page 1 to go to Page 2. 7. On Page 2, click the button to return to Page 1.	Buttons successfully switch between Page 1 and Page 2 of the QStackedWidget, each displaying different content.	✓	Tested without unit test

6.2.2. Integration Tests

2.2.1 Window Navigation Flow

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
INT_2211	Verify proper navigation between authentication menu and simulation menu using menu buttons, ensuring correct window transitions and state preservation.	High	1. All related unit tests must have passed (UNT_2111, UNT_2112) 2. Authentication and Simulation menus must be implemented 3. Button navigation system must be operational	1. Launch authentication menu window 2. Enter valid login credentials 3. Click login button to navigate to simulation menu 4. Verify simulation menu state 5. Test logout button to return to authentication menu 6. Verify authentication menu resets properly	Navigation between authentication and simulation menus works correctly with proper state handling	✓	Tested without unit test

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
INT_2212	Verify data persistence and state management across different views during navigation.	High	1. All related unit tests must have passed (UNT_2181, UNT_2182, UNT_3111, UNT_3112) 2. State management system must be implemented 3. Data storage system must be operational	1. Initialize application with test data 2. Navigate through multiple views 3. Modify data in different views 4. Navigate back and forth 5. Verify data consistency 6. Test window closure and reopening	Data remains consistent across all view transitions and window states	N/A	To be tested

2.2.2 UI Component Interaction

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
INT_2221	Verify form submission process including data collection, validation, and backend transmission.	Medium	1. All related unit tests must have passed (UNT_2141, UNT_2142, UNT_2143) 2. Form handling system must be implemented 3. Data validation system must be operational	1. Initialize form with test fields 2. Input valid and invalid data 3. Submit form with invalid data 4. Verify error handling 5. Submit form with valid data 6. Confirm data transmission	Forms properly validate input and handle submission correctly	N/A	To be tested

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
INT_2222	Verify real-time UI updates in response to user actions and state changes.	Medium	1. All related unit tests must have passed (UNT_2171, UNT_2172, UNT_2173, UNT_2174) 2. Event handling system must be implemented 3. UI update system must be operational	1. Initialize UI components 2. Trigger various user actions 3. Verify immediate UI feedback 4. Test multiple rapid updates 5. Check state consistency 6. Verify all visual indicators	UI components update correctly and immediately in response to actions	Error: the battle text message doesn't update correctly when using Sword Dance at maximum Strength boost (+6)	To be fixed

6.2.3. System Tests

2.3.1 Complete UI Workflows

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
SYS_2311	Verify complete template creation workflow through UI.	Low	1. Related integration tests must have passed (INT_2211, INT_2212, INT_3211) 2. Template creation interface must be operational 3. Data validation system must be implemented	1. Launch template creation interface 2. Input all template parameters 3. Test validation feedback 4. Save template 5. Verify data persistence 6. Check template accessibility	Complete template creation process works with proper validation and storage	N/A	To be tested

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
SYS_2312	Verify battle configuration workflow through UI.	High	1. Related integration tests must have passed (INT_1213, INT_2222) 2. Battle configuration interface must be operational 3. Configuration system must be implemented	1. Access battle configuration menu 2. Set various battle parameters 3. Apply configuration changes 4. Initialize battle 5. Verify applied settings 6. Test configuration persistence	Battle configuration process works with all options properly applied	N/A	To be tested

2.3.2 Error Handling

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
SYS_2321	Verify comprehensive input validation system across UI.	Low	1. Related integration tests must have passed (INT_2221, INT_3213) 2. Input validation system must be implemented 3. Error display system must be operational	1. Test all input fields with invalid data 2. Verify error messages 3. Test boundary conditions 4. Check validation timing 5. Verify recovery options	All invalid inputs are caught with appropriate error messages	N/A	To be tested
SYS_2322	Verify UI state recovery after various error conditions.	High	1. Related integration test must have passed (INT_2212) 2. State recovery system must be implemented 3. Error handling system must be operational	1. Trigger various error conditions 2. Monitor state preservation 3. Test recovery procedures 4. Verify data consistency 5. Check error logging	UI recovers gracefully from errors with state preserved	N/A	To be tested

6.2.4. Performance Tests

2.4.1 UI Response Times

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
PERF_2411	Verify window transition speed meets performance requirements.	High	1. Related integration test must have passed (INT_2211) 2. Performance monitoring must be configured	1. Measure window transition times 2. Test under various loads 3. Record transition metrics 4. Test multiple navigation paths 5. Verify consistency	All window transitions complete within 200ms threshold	N/A	To be tested
PERF_2412	Verify UI input response time meets performance requirements.	Medium	1. Related integration test must have passed (INT_2222) 2. Input monitoring system must be configured	1. Test various input operations 2. Measure response times 3. Test rapid input sequences 4. Verify feedback timing 5. Check input queue handling	All UI inputs processed within 100ms threshold	N/A	To be tested

2.4.2 Resource Management

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
PERF_2421	Verify UI memory usage remains within acceptable limits.	Medium	1. Memory monitoring tools must be configured 2. UI system must be fully operational	1. Monitor UI memory usage 2. Test extended UI operations 3. Check for memory leaks 4. Test window cycling 5. Verify cleanup processes	UI maintains stable memory usage without leaks	N/A	To be tested

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
PERF_2422	Verify UI scaling performance across different resolutions.	Low	1. Display testing tools must be configured 2. UI scaling system must be implemented	1. Test multiple resolutions 2. Measure scaling times 3. Check rendering quality 4. Verify layout consistency 5. Test dynamic resizing	UI scales smoothly across all supported resolutions	N/A	To be tested

6.3. Template Configuration

6.3.1. Unit Tests

3.1.1 Database Management

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_3111	Database creation, connection, and verification	High	1. Qt Creator is open with a working project. 2. A database management tool (e.g., DB Browser) is installed.	1. Use the database tool to create a new database file and define a table with multiple fields (e.g., ID, name, score). 2. In Qt, create or open a `.ui` file. 3. Write code to connect to the newly created database. 4. Run the project and verify successful connection by retrieving the database name or table structure.	The application successfully connects to the correct database, which contains the expected table structure.	✓	Tested without unit test
UNT_3112	Read values from a database table	High	1. Qt project is open with a working database connection. 2. The database table contains sample data.	1. Create or open a `.ui` file if needed. 2. Write and run code to connect to the database. 3. Execute SQL queries to fetch values from each field in the table. 4. Print or display the fetched data in the application.	The retrieved values exactly match the contents of the database table.	✓	Tested without unit test
UNT_3113	Update values in a database table	High	1. Qt project is open and connected to a populated database. 2. DB Browser is installed and running.	1. Open or create a `.ui` file. 2. Write code to connect to the database. 3. Add functionality to update one or more field values. 4. Run the project to perform the update. 5. Verify the changes using DB Browser.	The database values are updated correctly, and changes are visible in DB Browser.	✓	Tested without unit test

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_3114	Delete values from a database table	High	1. Qt project is open and connected to a populated database. 2. DB Browser is installed and running.	1. Open or create a `.ui` file. 2. Write code to connect to the database. 3. Implement functionality to delete records from the table. 4. Run the project and execute the deletion. 5. Confirm deletion in DB Browser.	Records are successfully deleted from the database and are no longer visible in DB Browser.	N/A	To be tested

3.1.2 Rules Management

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_3121	Retrieve rule state	High	1. Qt unit testing framework is set up	1. Define a mock boolean variable representing a rule (e.g., `bool ruleEnabled = true`). 2. Define another variable intended to retrieve that state. 3. Use assertions to verify that the retrieved value matches the mock value.	Unit test passes, confirming the rule state is correctly retrieved.	✓	Tested without unit test
UNT_3122	Detect rule activation and deactivation	High	1. Qt unit testing framework is set up	1. Define a mock boolean variable representing a rule state (e.g., `ruleActive`). 2. Write a function to toggle or modify the rule state. 3. Use assertions to confirm the state changes as expected (true → false, false → true).	Unit test passes, confirming rule state changes are detected accurately.	✓	Tested without unit test
UNT_3123	Retrieve multiple rule states as a group	High	1. Qt unit testing framework is set up	1. Define multiple mock boolean variables (e.g., `rule1`, `rule2`, `rule3`). 2. Assign known true/false values to each. 3. Store the values in a container (e.g., array or vector). 4. Use assertions to verify the correct values are retrieved in each index. 5. Repeat with different combinations.	Unit test passes, confirming that sets of rule states are correctly retrieved and interpreted.	✓	Tested without unit test

3.1.3 Characters Management

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_3131	Retrieve characters and their attributes	Medium	1. Qt unit testing framework is set up	1. Define 6 mock `Entity` objects with attributes (name, level, type, stats, and 4 mock moves). 2. Store all entities in a container (e.g., QVector or QList). 3. Select one character and use assertions to verify its data.	Unit test passes, confirming data retrieval for each character is accurate.	N/A	To be tested
UNT_3132	Detect character selection and deselection	Medium	1. Qt unit testing framework is set up	1. Define a mock `Entity` object with a `selected` boolean. 2. Write a method to toggle this flag. 3. Use assertions to confirm state transitions (selected → unselected and vice versa).	Unit test passes, confirming toggle functionality works as intended.	N/A	To be tested
UNT_3133	Retrieve list of exactly two selected characters	Medium	1. Qt unit testing framework is set up	1. Define 6 mock `Entity` objects. 2. Set the `selected` state to `true` for 2 of them. 3. Store all in a container. 4. Filter and use assertions to confirm exactly 2 are selected.	Unit test passes, confirming that only the selected characters are retrieved.	N/A	To be tested

3.1.4 Template Management

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_3141	Create battle template	Low	1. Qt test framework is set up 2. Template class is implemented	1. Initialize a `BattleTemplate` object with attributes: • name (string) • array of 6 mock `Entity` objects • array of booleans (rules) • mock damage formula function. 2. Use assertions to verify all attributes are correctly set.	Unit test passes, confirming the template initializes correctly with given data.	N/A	To be tested
UNT_3142	Modify battle template settings	Low	1. Qt test framework is set up 2. Existing template instance available	1. Assign a new name to the template. 2. Replace the rule set with a different one. 3. Update the mock characters and damage formula. 4. Use assertions to confirm updated values.	Unit test passes, confirming the template data can be modified and updated successfully.	N/A	To be tested
UNT_3143	Delete battle template	Low	1. Qt test framework is set up	1. Initialize and store a `BattleTemplate` object in memory or container. 2. Remove or deallocate the object. 3. Use assertions to ensure the template is no longer accessible.	Unit test passes, confirming the template can be successfully deleted.	N/A	To be tested

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_3144	Load existing battle template and retrieve its data	Low	1. Qt test framework is set up 2. Template is saved in memory or file	1. Retrieve a previously stored or serialized template. 2. Use assertions to check that all attributes (name, characters, rules, formula) are restored correctly.	Unit test passes, confirming that templates load correctly with all relevant data.	N/A	To be tested

3.1.5 Damage Calculator Management

ID	Description	Priority	Pre-requisites	Procedure	Expected Result	Output	Status
UNT_3151	Get Damage Calculator Settings	Low	Qt test framework set up; classes implemented	1. Write unit test 2. Retrieve formula inputs from damage calculator 3. Use assertions to verify values	Unit test passes; retrieved inputs match expected values	N/A	To be tested
UNT_3152	Modify Calculator Input Values	Low	Qt test framework set up; classes implemented	1. Write unit test 2. Retrieve calculator inputs 3. Copy calculator and change input values 4. Use assertions to check new damage	Unit test passes; modified inputs produce updated output as expected	N/A	To be tested
UNT_3153	Modify Calculator Formula	Very Low	Qt test framework set up; classes implemented	1. Write unit test 2. Retrieve original formula 3. Apply new formula in a test instance 4. Use assertions to check new damage	Unit test passes; new formula correctly modifies the damage output	N/A	To be tested
UNT_3154	Validate formulas and handle input errors gracefully	Low	Qt test framework; calculator class available	1. Write unit test 2. Create test cases for invalid syntax (`a + * b`), bad variables (`1abc`), divide by zero, unmatched parentheses, empty input 3. Test valid edge inputs 4. Fix invalid inputs and re-test	Invalid inputs show clear error messages; valid inputs are accepted; system recovers after correction; unit test passes	N/A	To be tested

6.3.2. Integration Tests

3.2.1 Template & Database Integration

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
INT_3211	Verify template saving and loading operations with database, ensuring data integrity.	Low	1. All related unit tests must have passed (UNT_3111, UNT_3112, UNT_3113, UNT_3141, UNT_3144) 2. Database connection must be established 3. Template system must be operational	1. Create a new battle template 2. Save template to database 3. Verify database entry creation 4. Load template from database 5. Compare loaded data with original 6. Update template and verify changes in database	Templates are correctly saved to and loaded from database with data integrity maintained	N/A	To be tested
INT_3212	Verify rule configuration system properly integrates with template management.	Low	1. All related unit tests must have passed (UNT_3121, UNT_3122, UNT_3123, UNT_3141) 2. Rule system must be implemented 3. Template management system must be operational	1. Create template with specific rule configuration 2. Save rule configuration 3. Modify multiple rules 4. Update template with new rules 5. Verify rule persistence	Rule configurations are properly integrated and preserved in templates	N/A	To be tested
INT_3213	Verify calculator error handling and UI error display integration.	Low	1. All related unit tests must have passed (UNT_2174, UNT_3154) 2. Calculator error handling must be implemented 3. UI error display system must be operational	1. Input invalid formula in calculator 2. Trigger various error conditions 3. Verify error message display 4. Test error state recovery 5. Confirm UI updates	Calculator errors are properly caught and displayed in UI	N/A	To be tested

3.2.2 Character & Template Integration

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
INT_3221	Verify character selection system properly integrates with template management.	Low	1. All related unit tests must have passed (UNT_3131, UNT_3132, UNT_3133, UNT_3141) 2. Character selection system must be implemented 3. Template system must be operational	1. Display character selection interface 2. Select two characters 3. Create template with selections 4. Save template 5. Load template and verify characters 6. Test character deselection	Character selections are properly integrated and preserved in templates	N/A	To be tested

6.3.3. System Tests

3.3.1 Complete Template Workflows

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
SYS_3311	Verify end-to-end template creation process including all components.	Low	1. Related integration tests must have passed (INT_3211, INT_3212, INT_3221) 2. Template system must be fully operational 3. Database system must be available	1. Create new template 2. Configure all template parameters 3. Select characters 4. Set battle rules 5. Configure damage calculator 6. Save and verify template	Complete template creation process works with all components properly integrated	N/A	To be tested
SYS_3312	Verify complete template modification workflow.	Low	1. Related integration tests must have passed (INT_3211, INT_3212) 2. Template editing system must be operational 3. Database update system must be functional	1. Load existing template 2. Modify multiple parameters 3. Update rule configurations 4. Save modifications 5. Verify changes persistence 6. Test template versioning	Template modification process works with proper update handling	N/A	To be tested

3.3.2 Edge Case

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
SYS_3321	Verify system handling of templates with maximum allowed settings.	Low	1. Related integration tests must have passed (INT_3211, INT_3212) 2. Maximum value handling must be implemented	1. Create template with maximum values 2. Test all maximum settings 3. Verify data handling 4. Check storage limits 5. Test template loading	System properly handles templates with maximum allowed values	N/A	To be tested

6.3.4. Performance Tests

3.4.1 Template Operation Speed

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
PERF_3411	Verify template loading performance meets requirements.	Low	1. Related integration test must have passed (INT_3211) 2. Performance monitoring must be configured	1. Create various template with large sizes 2. Measure load times 3. Test batch loading 4. Verify memory usage 5. Test under system load	All template loading operations complete within 500ms	N/A	To be tested
PERF_3412	Verify template save operation performance.	Low	1. Related integration test must have passed (INT_3211) 2. Database performance monitoring must be configured	1. Prepare multiple templates 2. Measure save times 3. Test rapid saves 4. Verify database integrity 5. Check transaction handling	Template save operations complete within 500ms with proper transaction handling	N/A	To be tested

3.4.2 Database Performance

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status

ID	Description	Priority	Pre-requisites	Procedure	Expected	Output	Status
PERF_3421	Verify database query performance meets requirements.	Low	1. Related integration test must have passed (INT_3211) 2. Database monitoring tools must be configured	1. Populate database with templates 2. Execute various queries 3. Measure response times 4. Test complex queries 5. Verify result accuracy	All database queries complete within 500ms threshold	N/A	To be tested

7 Bug Tracking

Number of Bugs encountered : 1

Number of Bugs fixed : 0

7.1. Bug Template

Bug Identifier	Description	Related Test Case	Discovery Conditions	Discovery Date	Current Status	Resolution Date	Fix Methodology

7.2. Unit Testing Bugs

Bug Identifier	Description	Related Test Case	Discovery Conditions	Discovery Date	Current Status	Resolution Date	Fix Methodology

7.3. Interface Testing Bugs

Bug Identifier	Description	Related Test Case	Discovery Conditions	Discovery Date	Current Status	Resolution Date	Fix Methodology

7.4. System Testing Bugs

Bug Identifier	Description	Related Test Case	Discovery Conditions	Discovery Date	Current Status	Resolution Date	Fix Methodology
BUG_INT_2222_1	Battle text displays "Entity's Strength rose sharply" instead of "Entity's Strength can't go beyond" when using Sword Dance at maximum Strength boost (+6)	INT_2222	Discovered during integration testing when verifying real-time UI updates. Bug occurs specifically when: 1. Entity's Strength stat is at maximum boost (+6) 2. Sword Dance move is used 3. Battle text updates incorrectly showing boost message instead of limit message	2025-04-20	To be fixed	TBD	TBD

7.5. Performance Testing Bugs

Bug Identifier	Description	Related Test Case	Discovery Conditions	Discovery Date	Current Status	Resolution Date	Fix Methodology

8. Glossary

8.1. RPG Lexical Field

Terms	Definitions
Ability	Also called "Trait", it's a specific characteristic of a character, generally allowing him to provide advantages during certain situations in battle.
Accuracy	A stat that determines how likely a move is to hit the target. Can be affected by buffs or debuffs.
Battle Template	A predefined configuration in the application that includes two characters, their movesets, and battle rules. Used to simulate a custom RPG battle scenario.
Buff	A skill, ability or other game mechanic that improves a character’s capabilities to increase his effectiveness.
Capacity	Another term for “Move” or “Skill” used in battle. Each character has up to four capacities they can use during a simulation.
Character	Persona created with characteristics embodied by a player within the context of the game.
Critical Hit	Successful attack dealing greater damage than a normal attack which occurs generally depending on the player's luck. Also nicknamed "Crit".
Damage	In game, any form of pain that decreases a character’s life due to an attack is expressed as damage.
Defend	Also called "Guard", action performed by a character to protect himself from an oncoming attack, generally to reduce the damage received.
Effectiveness	A mechanic that determines how effective a move is based on the types of the user and the opponent. For example, water > fire.
Element	An attribute given to a character or move that can create strengths and weaknesses (e.g., fire, water, grass types).
Evasion	A stat representing a character’s ability to avoid being hit. Higher evasion = lower chance of receiving an attack.
H.P.	Hit Points — how much damage a character can take before being knocked out.
Hazard	Passive effect on the battlefield that can hinder or benefit all characters.
HP Bar	A visual UI element representing remaining health.
K.O.	Knocked Out — a state when a character’s HP reaches zero.
Modifiers	Factors influencing battle outcomes (e.g., damage calculation, stat changes).
Move	Action used in battle — may cause damage, heal, or apply effects.
Move Set	The list of up to 4 capacities assigned to an Entity.
Nerf	Also called "debuff" — lowers effectiveness of a character.
Party	Group of characters teaming up in a role-playing game.
Pokemon	Refers to either the Pokémon franchise or the creatures themselves with abilities, types, and evolutions.
PP (Power Points)	Number of times a move can be used. Decreases with each use.
RPG (Role-Playing Game)	A game genre where players take on roles in a fictional setting, often with quests and character progression.
S.P.	Skill Points or mana — used to perform special actions.
Skill	Action used during battle that provides a specific effect, often costing SP.
Special Ability	Passive effect that gives a character an advantage (e.g., immunity to a type).
STAB (Same-Type Attack Bonus)	Bonus applied when move type matches character type.
Stat	Numeric value tied to character performance (Attack, Speed, etc.).
Status	Condition applied to a character (e.g., poison, paralysis).
Turn	One cycle of actions where each combatant makes a move.

8.2. Miscallenous

Term	Definition
API	Application Programming Interface — rules for software-to-software communication.
Backend	The server-side system that manages data and logic.
Bar	Graphical element in UI (e.g., health bar, progress bar).
Battle State	Enum representing the different stages of battle (e.g., Start, Finished).
Battle System	The rules and logic governing turn-based combat between entities.
Database	A structured data system (SQLite) used for storing rules, entities, and templates.
EffectResult	A structure capturing the outcome of a move, including damage or stat changes.
EffectType	The category of effect applied by a move (e.g., Attack, Heal, Buff, Debuff).
Framework	A foundation for software development providing reusable components and tools.
Frontend	The client-facing part of the app users directly interact with.
Game Design	Process of defining a game's structure, systems, and user experience.
Game Engine	Framework used to develop and run games (e.g., Unity, Unreal Engine).
GUI	Graphical User Interface — menus, buttons, and other interactive elements.
Layout	Arrangement of UI components for usability and design.
Log	Record of events or errors used for debugging.
Mechanics	Rules and systems that define how the game plays and reacts.
MoSCoW	Prioritization model: Must, Should, Could, Won't.
MoveCategory	Classification of a move: Physical, Special, or Status.
MoveResultState	A structure used to track the outcome of a turn and whether the battle continues.
Object Inspector	Qt Designer panel that shows all widgets in a UI hierarchy.
Plugin	Extension to an application that adds specific functionality.
Prototype	Early working version used to test ideas.
QStackedWidget	A Qt widget that displays one child widget at a time, used for menu navigation.
Qt Creator	The official IDE used to develop Qt-based applications.
Qt Designer	A visual tool within Qt Creator used to design `.ui` files.
Qt Framework	The C++ framework used to develop the application and UI.
QWidget	A base class for all UI components in Qt.
Responsive Design	Design approach that adapts to different screen sizes and devices.
RuleSet	A collection of gameplay rules (e.g., whether healing or buffing is allowed).
Setup Module	Initializes the simulation, loading rules and configuring entities.
Signal & Slot	Qt's event system allowing widgets to communicate.
Sprite	2D image used to represent a character or object in a game.
SQLite	Lightweight relational database used in the project.
Template	A predefined set of rules and entities used to quickly set up battles.
Turn-based	Game structure where players take alternating turns.
U.I. (User Interface)	Visual components for user interaction.
U.X. (User Experience)	Overall quality of interaction with the app, including ease of use and satisfaction.
Unit Testing	Testing individual parts of code for correctness.
Version Control	System for tracking changes to files and code over time (e.g., Git).