Project 3 - Witcher Tracker

Systems Programming - Spring 2025

Musa Kaan Güney, 2022400300
Hasan Yiğit Akıncı, 2022400138

June 1, 2025

1 Introduction

In the world of The Witcher, Geralt of Rivia must relearn his skills after losing his memory. The goal of this project is to implement a C++-based command-line interpreter that simulates Geralt's adventure by managing alchemical ingredients, potions, encounters, and magical knowledge.

This report details the full design and implementation process, presents a high-level system overview, explains core components with pseudocode, and evaluates the program's structure, performance, and possible extensions.

1.1 Problem Description

The Witcher Tracker is a C++-based simulation system that models Geralt's alchemical and monster-hunting activities via structured text commands. It interprets user input, modifies internal data structures, and returns appropriate game-like responses in a terminal environment. The interpreter functions as a mini-command shell that handles actions and queries related to inventory, knowledge, and events.

Functional Scope

The system implements a finite-state model of Geralt's knowledge and resources. It supports both state-modifying commands (e.g., loot, brew, learn) and informational queries (e.g., inventory counts or bestiary lookups). These operations reflect the logical progression of a simplified role-playing game scenario. The supported actions are deterministic and operate on internally tracked maps and vectors.

Input Grammar

User commands must conform to a strict grammar that mimics natural language but is intentionally limited and well-defined to ensure consistent parsing.

Commands are case-sensitive and must be correctly formatted, or the system will return `INVALID`.

The input is parsed using custom logic in the `CommandParser` class, which tokenizes each line and validates syntax based on recognized sentence structures.

Types of Supported Commands

1. **Knowledge Acquisition**
   - Learn a new potion formula:
     `Geralt learns Black Blood potion consists of 2 Rebis, 3 Vitriol`
   - Learn monster countermeasures:
     `Geralt learns Igni sign is effective against Drowner`
2. **Inventory Management**
   - Loot ingredients:
     `Geralt loots 4 Quebrith, 2 Rebis`
   - Brew a potion (if formula is known and ingredients are sufficient):
     `Geralt brews Black Blood`
   - Trade trophies for ingredients:
     `Geralt trades 2 Drowner trophy for 5 Quebrith`

3. **Encounters**
    - Simulate combat:
      `Geralt encounters a Bruxa`
4. **Queries**
    - Inventory query:
      `Total potion?` → e.g., `2 Black Blood, 1 Swallow`
    - Specific item count:
      `Total ingredient Quebrith?` → e.g., `5`
    - Bestiary effectiveness:
      `What is effective against Bruxa?` → e.g., `Black Blood, Yrden`
    - Alchemy formula check:
      `What is in Black Blood?` → e.g., `3 Vitriol, 2 Rebis, 1 Quebrith`
5. **System Exit**
    - `Exit`

Expected Behavior

Upon receiving a valid command, the system:

- Tokenizes the input line.
- Validates it against grammar.
- Identifies the command type.
- Dispatches it to the relevant subsystem.
- Returns an output message that matches the specification exactly.

If any validation fails, the system prints:

```
INVALID
```

Constraints

- Input length:
  `textless= 1024` characters.
- All potion and monster names must consist of only alphabetic characters.
- Potion names may contain spaces; ingredient and sign names may not.

1.2 Objectives

- Develop a console-based interpreter that:
  - Processes structured natural language commands.
  - Tracks and updates Geralt's inventory for ingredients, potions, and trophies.
  - Manages the knowledge about monsters and effective signs and potions for them.
  - Handles alchemy recipes for brewing potions.
  - Simulates monster encounters, validating Geralt's preparedness based on signs he knows and potions he owns.
  - Responds accurately to queries about Geralt's inventory, knowledge about recipes, and effective potions and signs against beasts.
- Ensure all input follows a strict grammar format and reject invalid entries.
- Simulate inventory management, potion brewing, and monster encounters.
- Support both knowledge acquisition and interactive queries.
- Implement an extensible, object-oriented architecture using C++.
- Comply with project constraints.

1.3 Motivation

This project requires systems programming skills allowing to:

- Gain experience with parsing and interpreting structured inputs.
- Handle state management and simulation in a user-driven environment.
- Appreciate the importance of syntax rules and command validation.

Moreover, it serves as a foundational exercise in building more complex interpreters or command-line tools, and reflects applications of language parsing, stateful systems, and modular programming in C++.

2 Design and Implementation

Class and Object Design

The Witcher Tracker project embraces an object-oriented architecture centered around well-defined classes, each encapsulating a distinct responsibility within the simulation. This modular design promotes maintainability, extensibility, and clear abstraction.

At the core of the system lies the `WitcherTracker` class, which serves as the main controller and orchestrator of command execution. It holds references to the three major subsystems: `Inventory`, `AlchemyKnowledge`, and `Bestiary`. These subsystems manage data and logic related to resource tracking, potion formulas, and monster effectiveness, respectively. The `WitcherTracker` also delegates command parsing and classification to the static utility class `CommandParser`, which tokenizes user input and determines the appropriate action to be executed.

The `Inventory` class is responsible for tracking Geralt's stock of ingredients, potions, and trophies. It internally uses `std::map<std::string, int>` containers to efficiently store quantities and support dynamic additions. This class exposes public methods for querying, incrementing, and decrementing item counts, thus serving as the state manager for all collectible resources.

The `AlchemyKnowledge` class manages Geralt's known potion recipes and magical signs. Potion data is stored in a map that associates potion names with `Potion` objects. Each `Potion` object encapsulates a recipe consisting of ingredient names and their corresponding quantities. The `AlchemyKnowledge` class also maintains a collection of `Sign` objects, allowing the system to recognize and validate magical signs when used as counters.

Complementing this, the `Bestiary` class records the effectiveness of various potions and signs against specific monsters. It maps monster names to `Beast` objects, where each `Beast` maintains a list of effective potions and signs. This design supports dynamic learning of new counters through input commands and enables reverse lookups when querying the bestiary.

The `CommandParser` class plays a foundational role in the system. It is designed as a purely static class, with no instance state, and is responsible for sanitizing input strings, tokenizing them into meaningful components, and validating command structures. It identifies the type of each command (e.g., loot, brew, trade, query) and maps it to a corresponding `CommandType` enumeration, which is then passed to the `WitcherTracker` dispatcher.

Individual supporting classes such as `Potion`, `Beast`, and `Sign` are lightweight containers that hold structured data. They provide basic methods for adding elements to their internal lists, such as ingredients in a potion or effective counters in a beast entry. Their simplicity ensures that they function effectively as value objects without introducing unnecessary behavior or complexity.

Inter-class communication is handled through method calls and shared data references. For instance, when Geralt attempts to brew a potion, the `WitcherTracker` queries the recipe from `AlchemyKnowledge`, checks availability in the `Inventory`, and updates both modules accordingly. Similarly, when learning a new counter for a monster, the system updates both the `Bestiary` and `AlchemyKnowledge` to maintain consistency.

This architecture adheres to key object-oriented principles:

- **Encapsulation**: Internal data such as inventory contents or potion formulas are protected behind public interfaces.
- **Separation of Concerns**: Parsing, command dispatching, data storage, and behavior logic are isolated in dedicated classes.
- **Single Responsibility**: Each class focuses on one domain—such as storing ingredients, mapping monsters, or parsing commands.
- **Open/Closed Principle**: New command types or game mechanics can be added with minimal impact on existing code, making the system easily extensible.

In conclusion, the class design of the Witcher Tracker leverages the strengths of object-oriented programming to deliver a modular, extensible, and intuitive framework that cleanly separates data concerns from behavior, enabling robust handling of simulated game actions.

Data Structures

The Witcher Tracker project relies on a combination of STL containers and custom classes to represent the core entities and relationships in the simulated game world. The use of standard library data structures allows the program to efficiently store, access, and manipulate data without imposing fixed limitations on size or capacity.

At the foundational level, the program uses `std::map<std::string, int>` to track quantities of items in the `Inventory` class. Three separate maps are maintained internally to represent ingredients, potions, and trophies respectively. Each map allows constant-time insertion and logarithmic-time lookups, enabling the system to respond quickly to inventory queries, additions, and removals. This choice is especially beneficial in a context where item names are string-based and can be dynamically added as new content is introduced.

The `AlchemyKnowledge` class manages potion formulas using a `std::map<std::string, Potion>`. Here, each potion name maps to a `Potion` object, which encapsulates a vector of ingredient names and a parallel vector of corresponding quantities. The parallel vector design ensures that each ingredient and its amount can be iterated or accessed in sync. This structure facilitates efficient parsing and construction of recipes during knowledge acquisition commands, as well as fast validation during brewing.

In addition, `AlchemyKnowledge` maintains a map of known magical signs using `std::map<std::string, Sign>`. The primary purpose of this structure is validation—ensuring that only known signs are used in commands related to monster counters. The signs themselves are lightweight objects holding only a name attribute.

The `Bestiary` subsystem uses a `std::map<std::string, Beast>` to manage monster knowledge. Each `Beast` object contains two `std::vector<std::string>` containers: one for effective signs and another for effective potions. These vectors are used to store lists of known counters for each monster. Their dynamic nature supports incremental updates when new knowledge is learned via user commands. The contents are also sorted for consistent output formatting during bestiary queries.

To represent individual items and entities, several custom classes are defined. These include `Ingredient`, `Potion`, `Trophy`, `Sign`, and `Beast`. Most of these classes are simple data containers, encapsulating only the fields necessary to represent each entity (e.g., name and quantity for ingredients and potions). However, some—such as `Potion` and `Beast`—include small helper methods for adding ingredients or counters, providing convenience for data manipulation.

The central class `WitcherTracker` integrates all these structures. It does not directly manipulate raw containers but instead delegates responsibilities to `Inventory`, `AlchemyKnowledge`, and `Bestiary`. This layered approach encourages separation of concerns and allows each subsystem to encapsulate its own data and operations.

Overall, the design avoids static arrays entirely, opting for flexible and type-safe STL containers that support dynamic expansion. This approach allows the Witcher Tracker to scale gracefully and ensures that the addition of new items, potions, or monsters does not require structural changes to the codebase. The interplay between maps and vectors, supported by custom objects, offers a clean and extensible data model well-suited to the simulation of a text-based RPG logic system.

Execution Flow

flowchart.drawio.png

Figure 1: Flowchart of command parsing and dispatch

Tokenizer Function Pseudocode

```
function tokenizeInput(input):
    tokens = []
    i = 0
    skip leading whitespace

    if input starts with "What":
        tokens.push("What")
        skip to "is"
        tokens.push("is")
        if next word is "in":
            tokens.push("in")
            extract potion name until '?'
            tokens.push(potion name)
            if '?' found: tokens.push('?')

        else if next words are "effective against":
            tokens.push("effective", "against")
            extract monster name until '?'
            tokens.push(monster name)
            if '?' found: tokens.push('?')

        return tokens

    else if input starts with "Total":
        tokens.push("Total")
        extract category (ingredient/potion/trophy)
        tokens.push(category)

        if next token is an item name:
            extract and push it

        if '?' found: tokens.push('?')
        return tokens

    else if input starts with "Geralt":
        tokens.push("Geralt")
        detect next keyword:
            if "brews":
                tokens.push("brews")
                extract and push potion name

            else if "learns":
                tokens.push("learns")
                locate "sign" or "potion"
                extract name before keyword
                tokens.push(name, keyword)

                if "is effective against":
                    tokens.push("is", "effective", "against")
                    extract and push monster name

                else if "consists of":
                    tokens.push("consists", "of")
                    loop over ingredients:
                    extract quantity and name
                    tokens.push(quantity, name)
                    handle commas

            else if "loots":
                tokens.push("loots")
                loop:
                    extract quantity and ingredient
                    tokens.push(quantity, name)
                    handle commas

            else if "trades":
                tokens.push("trades")
                loop:
                    extract trophy quantity and name
                    detect "for"
                    extract ingredient quantity and name

            else if "encounters":
                tokens.push("encounters", "a")
                extract and push monster name

        return tokens

    else if input == "Exit":
        return ["Exit"]

    // fallback parser
    while i < len(input):
        skip whitespace
        if input[i] == ',':
            tokens.push(',')
        else:
            extract next token and push it

    return tokens
```

Key Snippets

This section highlights two representative code excerpts that are crucial for understanding the internal mechanics of the Witcher Tracker application. These snippets exemplify the logic-heavy and decision-centric parts of the program that lie at the heart of user input handling and core gameplay logic.

1. Tokenization Logic — `tokenizeInput` Function

One of the most central components of the application is the tokenization mechanism implemented in the `CommandParser` class. The `tokenizeInput` function is responsible for parsing free-form user inputs into structured tokens that can be categorized and processed by the program. This function is particularly robust, as it identifies different command categories such as "What is in...", "Geralt brews...", and "Geralt learns...".

Below is a simplified excerpt from the part of the function that recognizes bestiary queries:

```
if (input.substr(i, 4) == "What" && ...) {
    tokens.push_back("What");
    i += 4;
    ...
    if (input.substr(i, 2) == "is") {
        tokens.push_back("is");
        i += 2;
        ...
        if (input.substr(i, 9) == "effective") {
            tokens.push_back("effective");
            i += 9;
            ...
            if (input.substr(i, 7) == "against") {
                tokens.push_back("against");
                i += 7;
                ...
                // Capture monster name
                ...
            }
        }
    }
}
```

This logic performs nested pattern matching, incrementally consuming the input string while identifying reserved keywords. It is robust against spacing inconsistencies and is capable of extracting variable-length monster or potion names. The result is a list of tokens that abstracts away the raw string into a format that can be validated and dispatched by the rest of the system.

The modular nature of this function also allows it to gracefully fall back on a generic parsing strategy if the command does not match known patterns, making it versatile yet tightly structured. This approach enables the program to interpret a variety of command types while maintaining strict grammar compliance.

2. Encounter Simulation — `executeEncounter` Function

Another core snippet resides in the `WitcherTracker` class, which coordinates the system's behavior. The `executeEncounter` function embodies the game's primary decision-making logic: determining whether Geralt can defeat a monster based on his inventory and known effectiveness counters.

```
Beast *beast = bestiary.getBeast(monsterName);
if (!beast) {
    cout << "Geralt is unprepared and barely escapes with his life\n";
    return 0;
}

bool hasEffectiveCounter = false;
for (const auto &potionName : beast->effectivePotions) {
    if (inventory.getPotionQuantity(potionName) > 0) {
        hasEffectiveCounter = true;
        break;
    }
}

if (!hasEffectiveCounter && !beast->effectiveSigns.empty()) {
    hasEffectiveCounter = true;
}

if (hasEffectiveCounter) {
    ...
    cout << "Geralt defeats " << monsterName << "\n";
} else {
    cout << "Geralt is unprepared and barely escapes with his life\n";
}
```

This snippet illustrates how the program bridges multiple components—`Bestiary`, `Inventory`, and `AlchemyKnowledge`—to evaluate combat outcomes. It first checks if the beast is known and whether any of its counters (potions or signs) are available. Effective potions must exist in the inventory, while signs are treated as passively available if known.

The conditional structure leads to one of two outcomes: either Geralt defeats the monster and gains a trophy, or he escapes unsuccessfully. This binary resolution mirrors traditional RPG mechanics and demonstrates how game logic is built on simple but well-structured rules tied to the internal state.

These two snippets, though partial in scope, reflect key architectural philosophies in the application: layered abstraction, state-driven behavior, and deterministic input evaluation. Together, they illustrate how the program translates textual user input into meaningful in-game consequences.

3 Challenges and Solutions

The development of the Witcher Tracker project presented several notable challenges, both in terms of software architecture and specific implementation decisions. The following subsections outline key obstacles encountered during the project and the solutions applied to address them effectively.

1. Designing a Modular and Maintainable Architecture

Challenge: One of the early challenges was designing a scalable object-oriented architecture that could handle multiple systems—inventory, alchemy, bestiary, and command parsing—without making the code tightly coupled or overly complex.

Solution: This was addressed by applying principles of modular design and encapsulation. Each major responsibility was isolated in its own class: `Inventory` for item tracking, `AlchemyKnowledge` for potion recipes and signs, and `Bestiary` for monster knowledge. These subsystems are orchestrated by the `WitcherTracker` class, which serves as a controller without directly exposing internal data. This separation of concerns made it easier to reason about each component in isolation and extend or debug them without affecting unrelated parts of the codebase.

2. Implementing Flexible yet Strict Command Parsing

Challenge: Parsing natural-language-like input with rigid structure—such as `Geralt learns Swallow potion consists of 1 mandrake, 2 celandine`—required a tokenizer that could accurately identify commands while also detecting malformed input (e.g., extra commas, invalid numbers, or inconsistent spacing).

Solution: The `CommandParser` class was implemented with a custom tokenizer that uses a finite state approach and nested control structures to parse known command patterns. Helper methods such as `isPositiveInteger`, `isValidPotionNameToken`, and `hasCommaSpacingError` were created to enforce grammar rules and validate input at multiple stages. Additionally, fallback tokenization logic was added for unexpected or malformed input, allowing the program to handle errors gracefully and consistently output `INVALID` when needed.

3. Preventing Logic Duplication Across Subsystems

Challenge: Many operations—such as checking whether a potion exists, verifying sufficient ingredients, or updating an entry in the bestiary—required logic that could easily have been duplicated across multiple classes, leading to fragile code.

Solution: To avoid redundancy, specialized helper functions and centralized checks were implemented. For example, the `AlchemyKnowledge` class provides methods like `hasPotion()` and `getPotionIngredients()` to encapsulate lookup logic. Similarly, inventory checks and updates are encapsulated within the `Inventory` class. This not only improves code reuse but also ensures that rules (e.g., potion availability or removal) are consistently enforced across the program.

4. Managing Edge Cases in Brewing and Combat

Challenge: Commands such as brewing or encountering monsters involve multiple dependent systems and state changes. Handling edge cases—like brewing a potion without a known formula or fighting a monster without any effective counters—required careful control flow and validation.

Solution: Guard conditions were applied to prevent unsafe operations. For instance, `executeBrewAction` first checks whether the potion formula exists and whether Geralt has sufficient ingredients. Only if both checks pass does the function proceed to update the inventory. Similarly, `executeEncounter` ensures that monster knowledge exists in the bestiary and that Geralt has access to at least one effective potion or sign before declaring victory. These layered checks contribute to both correctness and user clarity.

5. Ensuring Deterministic Output and Sorting

Challenge: For queries such as `Total ingredient?` or `What is effective against Drowner?`, the output order needed to be deterministic (alphabetical or quantity-based) to facilitate grading and readability.

Solution: All relevant output-producing functions (e.g., `getAllIngredients()`, `getEffectiveCounters()`) internally sort the result vectors before formatting the output. Sorting criteria such as quantity-descending and name-ascending were carefully implemented to match the project requirements. This guarantees consistent and predictable output for the same inputs.

6. Balancing Strict Validation with User Flexibility

Challenge: It was important to enforce strict input formatting without frustrating the user experience with overly sensitive parsing logic.

Solution: The parser was carefully tuned to tolerate reasonable spacing variations and handle edge punctuation, while still rejecting commands with critical syntax violations such as duplicate commas, invalid tokens, or missing keywords. This balance ensures both robustness and usability, reducing false negatives while still catching malformed input reliably.

—

Together, these challenges and the corresponding solutions shaped a robust and modular implementation. The structured object-oriented design, combined with defensive parsing strategies and strong separation of concerns, enabled the system to scale while remaining maintainable and testable.

4 Usage Guide

This section outlines how to compile, run, and interact with the Witcher Tracker program. It is designed as a command-line application and supports structured user input to simulate inventory management, potion brewing, bestiary updates, and monster encounters. Users are expected to enter commands following a predefined grammar that mimics natural language.

Compilation Instructions

To compile the program, ensure you have a C++ compiler installed (such as `g++`) that supports C++11 or later. Then, use the following command from the terminal:

```
g++ -std=c++11 -o WitcherTracker main.cpp WitcherTracker.cpp AlchemyKnowledge.cpp \
Beast.cpp Bestiary.cpp CommandParser.cpp Inventory.cpp Potion.cpp
```

This will generate an executable named `WitcherTracker` (or `WitcherTracker.exe` on Windows).

Execution

Once compiled, the program can be run using the following command:

```
./WitcherTracker
```

The program enters a prompt loop and waits for user input. Each line should be a complete command. The program terminates when the user types:
Exit

Example Session

Below is a sample session showing valid input commands and their corresponding output:

Test Case 1: Loot Action

Input: Geralt loots 4 Quebrith, 2 Rebis
Output: Alchemy ingredients obtained
Explanation: Geralt adds 4 Quebrith and 2 Rebis to his ingredient inventory.

Test Case 2: Effectiveness Knowledge

Input: Geralt learns Igni sign is effective against Drowner

Output: New bestiary entry added: Drowner

Explanation: The program updates the bestiary to show that Igni is effective against Drowners. If the beast already existed, it would print an update message instead.

Test Case 3: Potion Formula Knowledge

Input: Geralt learns Black Blood potion consists of 2 Rebis, 3 Vitriol

Output: New alchemy formula obtained: Black Blood

Explanation: A new potion recipe is stored with its required ingredients. The system now allows this potion to be brewed, provided the ingredients are present.

Test Case 4: Brew Action

Input: Geralt brews Black Blood

Output: Alchemy item created: Black Blood

Explanation: Geralt successfully brews Black Blood if the inventory contains at least 2 Rebis and 3 Vitriol. The required ingredients are consumed.

Test Case 5: Encounter

Input: Geralt encounters a Drowner

Output: Geralt defeats Drowner

Explanation: Geralt defeats the monster using either an effective sign (Igni) or a known effective potion (if available in inventory). A Drowner trophy is added to inventory.

Test Case 6: Trade Action

Input: Geralt trades 1 Drowner trophy for 5 Quebrith

Output: Trade successful

Explanation: Geralt trades a Drowner trophy for 5 Quebrith. Inventory is updated by removing the trophy and adding the ingredients.

Test Case 7: Specific Inventory Query

Input: Total potion Black Blood?

Output: 1

Explanation: The system returns how many of the specified potion Geralt currently has in his inventory.

Test Case 8: All Inventory Query

Input: Total ingredient?

Output: 5 Quebrith

Explanation: Lists all current alchemy ingredients and their quantities after a series of loot and trade actions.

Test Case 9: Bestiary Query

Input: What is effective against Drowner?

Output: Igni

Explanation: Lists all known effective signs and potions for the specified monster. Entries are sorted alphabetically if more than one is available.

Test Case 10: Alchemy Query

Input: What is in Black Blood?

Output: 3 Vitriol, 2 Rebis

Explanation: Returns the formula (ingredients and quantities) for the specified potion, sorted first by quantity descending, then alphabetically.

Error Handling

If the user inputs a malformed or unsupported command, the program will respond with:

INVALID

Examples of invalid input include missing keywords, extra commas, non-numeric quantities, or unrecognized command structures. The parser strictly enforces grammar to ensure consistency and prevent undefined behavior.

—

This section provides a comprehensive walkthrough for users and graders. If you're preparing a '.tex' report, this is ready to copy-paste directly. Would you also like a table summarizing valid command templates?

5 Code Structure Summary

The Witcher Tracker project is implemented in C++ using a modular, object-oriented design. The program is composed of multiple source files and headers, each encapsulating a distinct responsibility. This structure facilitates readability, reusability, and ease of debugging. Below is a brief overview of the key components and their respective roles in the system:

1. `main.cpp`

This is the entry point of the application. It initializes the `WitcherTracker` object and enters a user input loop, where it continuously reads commands from standard input. Each line is passed to the `WitcherTracker::executeLine` method for processing. This file contains no logic beyond launching and orchestrating the main loop.

2. `WitcherTracker.h` and `WitcherTracker.cpp`

These files define and implement the core controller class, `WitcherTracker`, which acts as the central coordinator of all subsystems. It delegates commands to appropriate handler functions based on parsed command types and maintains instances of the `Inventory`, `AlchemyKnowledge`, and `Bestiary` classes. It also includes command dispatching logic through the use of an enum class, `CommandType`.

3. `CommandParser.cpp`

This file contains the static utility class `CommandParser`, which is responsible for analyzing and validating user input. It includes methods to sanitize strings, tokenize commands, and classify input based on specific patterns. This module is essential for the interpreter's ability to distinguish between valid and invalid syntax and is central to robust user interaction.

4. `Inventory.cpp`

This module manages Geralt's inventory, including ingredients, potions, and trophies. It uses `std::map` containers to dynamically store and retrieve item quantities. It provides methods for adding, removing, querying, and listing items. The `Inventory` class ensures that inventory state remains consistent across actions such as looting, brewing, and trading.

5. `AlchemyKnowledge.cpp`

This file implements the `AlchemyKnowledge` class, which stores known potion formulas and magical signs. Potion data is stored as a mapping from potion names to `Potion` objects, which in turn hold vectors of ingredient names and required quantities. This module supports actions such as learning new recipes and querying potion contents.

6. `Bestiary.cpp`

The `Bestiary` class manages information about monsters (beasts) and the known effective potions or signs against them. Each monster is represented as a `Beast` object, and the bestiary stores them in a `std::map` for quick lookup. This module supports knowledge acquisition and in-combat decision-making.

7. `Potion.cpp`, `Beast.cpp`

These files define lightweight supporting classes used by the core systems. `Potion` represents a potion's recipe and name, while `Beast` tracks which signs or potions are effective against a given monster. These classes encapsulate logic specific to their entities and provide utility methods for updates.

8. `WitcherTracker.h`

This header file includes all class declarations and constants used across the system. It defines the interfaces for major classes and enumerations like `CommandType`. By including this header, all source files gain access to the complete type system of the project.

Integration Overview

Each component in the system communicates indirectly via the `WitcherTracker` class. For example, upon receiving a "Geralt brews Swallow" command, the program passes the input through `CommandParser`, which classifies the command. Then, the `WitcherTracker` dispatches execution to the corresponding method, which may involve querying `AlchemyKnowledge` for the recipe and updating the `Inventory` accordingly.

This layered and modular structure allows the program to cleanly separate concerns while enabling seamless integration between parsing, state management, and game logic.

6 AI Assistance

Throughout the development and documentation of this project, artificial intelligence tools were utilized to support various tasks, primarily focused on debugging, documentation structuring, and LaTeX formatting.

Tools Used

- **ChatGPT-4o (OpenAI)**: This tool was used extensively during the documentation phase. It assisted in structuring and generating formatted LaTeX sections such as the Methodology, Implementation Details, Results, and Discussion. It also helped convert logic and code structure into readable pseudocode and provided suggestions for improving report consistency and formatting aesthetics.

- **Claude Sonnet 4 (Anthropic)**: Used primarily for debugging support during the implementation of certain C++ modules. Claude provided code-level suggestions and helped analyze edge cases, assisting in resolving logic errors related to input parsing and command classification.