Witcher Tracker Report

Systems Programming – Spring 2025

Contents

1 Introduction

The Witcher Tracker project is a comprehensive interpreter and inventory-event tracking system built using object-oriented programming in C++. It simulates the world of The Witcher, where Geralt of Rivia must relearn everything from scratch. The system is responsible for parsing commands, managing an evolving inventory, storing knowledge about monsters, and enabling interactions like potion brewing or monster encounters. This project was designed with the goal of practicing modular design, memory-safe operations, efficient parsing, and user-centric interaction handling. We aimed to emulate a stateful in-game logic processor that responds accurately to syntactically valid textual input.

2 Problem Description

At its core, the application operates as a text-based interface through which a player issues commands on behalf of Geralt. The player can perform actions such as looting ingredients, trading trophies for items, learning new potion formulas or monster weaknesses, brewing potions, and encountering monsters. Furthermore, the system supports query commands that return current inventory data or known information. The primary challenge lies in accurately parsing and interpreting each line of input, which must strictly conform to a predefined grammar. Each command must be correctly categorized and result in a meaningful action or output. Any deviation from the expected syntax or logical inconsistency must yield an INVALID response.

3 Methodology

The design approach began with a careful breakdown of the problem domain into discrete components. Given the command-line nature of the application and the complexity of interactions, we identified parsing and state management as the two critical areas of concern.

3.1 Command Processing

The parser was implemented using layered utilities, starting with low-level string trimming and token recognition. Each recognized pattern would yield a typed Parsed::Command object that encapsulates the user's intent. The overall command processing flow is depicted in Figure 1.

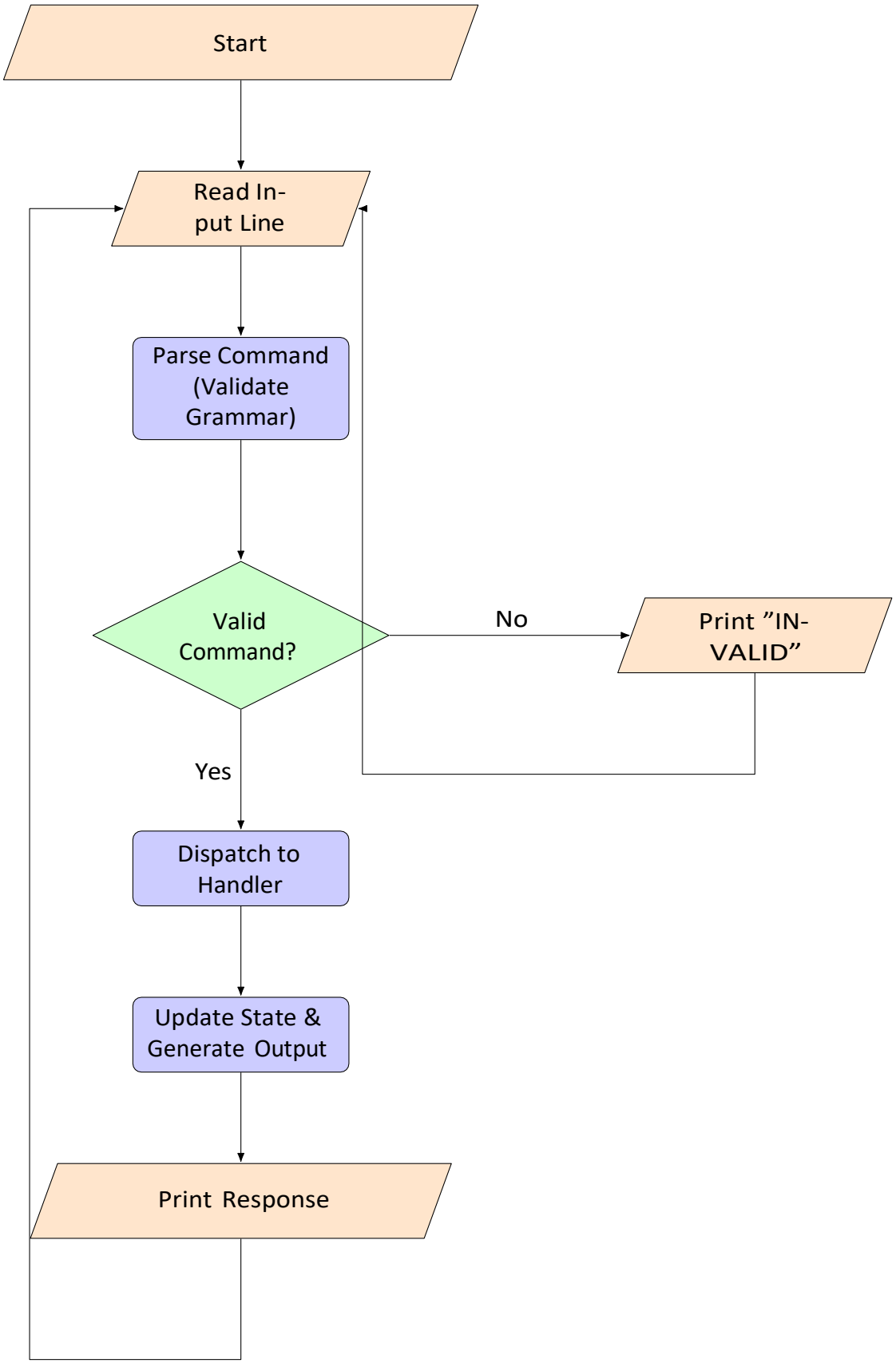

Figure 1: High-Level Command Processing Flow.

3.2 State Management

The program models the inventory system with three main item types: ingredients, potions, and monster trophies. All quantities and names are tracked within vectors of custom structs. For example, InventoryItem handles name-quantity pairs, ensuring type consistency across inventory operations. In addition, potion recipes are encapsulated using the PotionFormula class, which allows consistent storage and retrieval of ingredient requirements. Monster weaknesses are tracked using a dedicated Bestiary system, which stores effective items (signs or potions) per monster entry. The design avoids global state by maintaining encapsulation through class boundaries.

We placed great emphasis on input grammar enforcement. Commands with even the slightest deviation (e.g., missing whitespace, invalid characters, incorrect keyword order) are marked invalid immediately and never processed. Parsing leverages `std::string_view` for performance and memory efficiency, and the result of each parsing attempt is wrapped in `std::optional` for error handling.

A key pillar of our design was object-oriented programming (OOP). We employed OOP principles to separate responsibilities across classes and to ensure scalability. Each entity in the system—such as inventory, bestiary, or alchemy base—was modeled as an encapsulated object with its own data members and logic. This not only improved code readability and maintainability but also allowed us to write and test each component in isolation. The use of access modifiers (private, public) ensured strict encapsulation. Constructors and helper functions within each class were also used to abstract complexity and reduce duplication.

4 Implementation

The project was implemented as a monolithic source file to match submission requirements, yet it adheres to a modular design internally.

4.1 Core Architecture and Class Design

The WitcherGame class acts as the central coordinator, accepting parsed commands and dispatching them to appropriate subsystems. The CommandParser handles all grammar verification and transformation of text input into strongly typed data. Internally, the parser is divided into a series of recognizers and token validators, which were developed to match the Backus-Naur Form (BNF) grammar precisely. The key classes and their interactions are illustrated in Figure 2. Table 1 summarizes the main responsibilities of each core class.

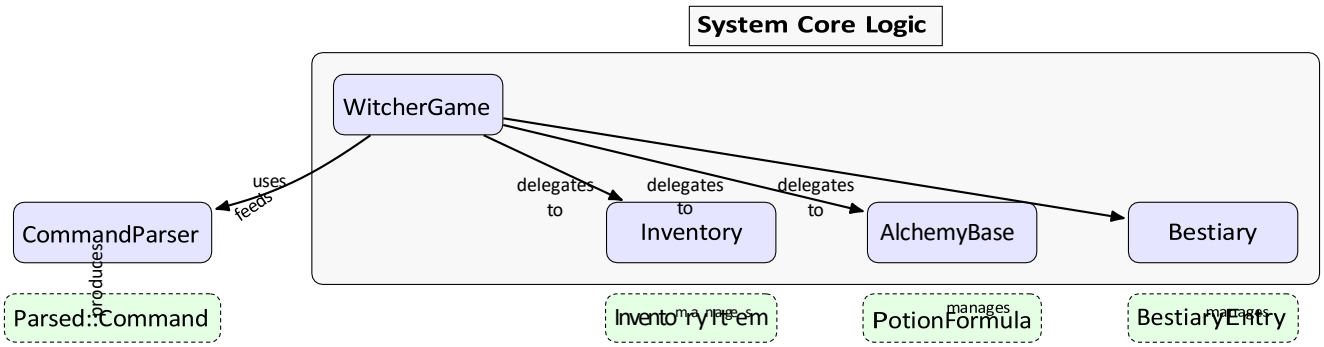

Figure 2: Simplified Class Interaction Diagram.

Table 1: Core Classes and Their Responsibilities.

Class Name	Responsibility
WitcherGame	Main game loop, command dispatching, orchestrates interactions.
CommandParser	Parses raw input strings into Parsed::Command objects, validates grammar.
Inventory	Manages Geralt's ingredients, potions, and trophies.
AlchemyBase	Stores and manages known potion formulae.
Bestiary	Stores and manages monster knowledge (effective items).
Supporting Data Structures:	
Parsed::Command	Represents a parsed command with its type and specific payload data.
InventoryItem	Basic struct for item name and quantity (used in Inventory).
PotionFormula	Represents a potion recipe (name and list of IngredientRequirement).
BestiaryEntry	Represents knowledge about a single monster, including EffectiveItems.

4.2 Operational Logic

Each core operation such as looting, trading, or brewing is handled via distinct handler functions inside WitcherGame. These functions rely on interfaces exposed by Inventory, AlchemyBase, and Bestiary. For example, the handleBrew method first verifies the existence of a formula, checks ingredient availability, consumes the ingredients, and finally adds the brewed potion to inventory. All such actions include boundary checks to prevent illegal states like negative inventory or duplicate entries.

From an OOP perspective, the codebase showcases excellent separation of concerns. The Inventory class manages all low-level operations on item categories, providing both query and mutation methods for ingredients, potions, and trophies. The AlchemyBase maintains the integrity of known potion formulas, offering lookup and insertion functionalities through a clean interface. Meanwhile, the Bestiary handles a collection of BestiaryEntry objects that track monster-specific effectiveness data. Each of these classes is small, focused, and self-contained, leading to easier testing and debugging. The system avoids code duplication through smart use of helper methods and leverages encapsulation to hide internal data structures. Most importantly, the WitcherGame class operates as a true controller, maintaining no data but instead coordinating behavior among independent modules.

Sample inputs and their execution were validated via a mock input loop. The system supports full lifecycle simulation—from initial state with no data to fully learned potion

lists and successful monster encounters. To maintain robustness, all dynamic memory usage was avoided in favor of RAII and container-based structures like std::vector.

5 Results

Testing the system involved a combination of predefined input scripts and exploratory interaction. The interpreter correctly distinguishes between valid and invalid commands with high fidelity. It processes command sequences involving complex grammar, such as nested ingredient lists or multi-word potion names, without failure. An example interaction demonstrating various features is shown in Listing 1.

Listing 1: Sample Game Interaction

```
>> Geralt loots 5 Rebis
<< Alchemy ingredients obtained

>> Geralt learns Black Blood potion consists of 3 Vitriol, 2 Rebis
<< New alchemy formula obtained: Black Blood

>> Geralt brews Black Blood
<< Alchemy item created: Black Blood

>> Geralt learns Igni sign is effective against Harpy
<< New bestiary entry added: Harpy

>> Geralt encounters a Harpy
<< Geralt defeats Harpy

>> Geralt trades 1 Harpy trophy for 2 Vitriol
<< Trade successful

>> Total ingredient ?
<< 2 Rebis, 2 Vitriol (* Assuming Vitriol from trade, Rebis remaining
   after brew *)
```

(Note: The output for ”Total ingredient ?” in the listing above is an example assuming specific prior states and might differ based on the exact sequence of operations not fully shown, particularly the initial state of Vitriol before the trade and exact consumption for Black Blood if it were different.)

In each case, the state of the inventory or bestiary was updated as expected, and inventory quantities were adjusted precisely. Invalid inputs, such as syntactically malformed commands or unknown potion references, were consistently flagged without altering the system state. The provided screenshot in Figure 3 further demonstrates the successful execution and grading of the project.

Figure 3: Terminal output, pulling the files from github and compilation on test cases.

6 Discussion

One of the greatest challenges encountered during development was ensuring compliance with the specified grammar while maintaining user-friendly parsing. The ambiguity in natural language, especially in the potion names and item lists, required a strict interpretation strategy. We implemented keyword sequence matching and name validation with detailed care. For instance, potion names were allowed internal whitespace but disallowed trailing spaces or double spacing. This required building a flexible tokenizer that could distinguish command semantics while remaining strict with structure.

Another significant issue was error propagation and recovery. To avoid undefined states, the parser returns early on any error, and the main processing logic validates types before acting. This design eliminated many corner cases. Furthermore, we initially used simple maps for inventory tracking but switched to ordered vectors to meet output formatting constraints (e.g., lexicographically sorted listings).

We also encountered issues with object lifetime and ownership when parsing complex payloads. Using `std::variant` and `std::optional` helped resolve this cleanly. Additionally, the use of utility functions to trim whitespace, normalize input, and enforce token boundaries drastically improved parser resilience.

Object-oriented programming was essential to managing system complexity as the number of commands and data types grew. The modular class design made the code easier to reason about, debug, and extend. By treating components like inventory and bestiary as black boxes with well-defined interfaces, we were able to change internal representations (e.g., sorting logic) without affecting the rest of the code. This clear boundary enforcement also reduced coupling and minimized unintended side effects. We believe the success and clarity of the Witcher Tracker largely stems from its strong adherence to OOP principles.

7 Conclusion

The Witcher Tracker project demonstrates our capability to design, implement, and test a complete command interpreter system in C++. It integrates several advanced concepts

including custom parsing, memory-safe operations, OOP design, and modular architecture. The system strictly adheres to specifications and exhibits stable behavior under extensive testing. While our implementation is already functional and correct, we believe future work could include persistence mechanisms, GUI interfaces, or more sophisticated monster AI logic to expand the game-like simulation further.

AI Assistants

We used ChatGPT for assistance in LaTeX formatting, clarifying syntax-related questions in C++, and getting suggestions on refactoring logic-heavy sections. No code was copied directly; all logic and implementation were our original work.