Design Notes for Journal CLI

Manjunatha Sarma Majety

December 3, 2025

Abstract

This document serves as a personal log and design record for the `jrnl` CLI tool. It captures the evolution of the project, lessons learned, and insights into the thought process behind each design choice.

Contents

1 Motivation

The goal of this project was to create a simple, fast, and minimalistic command-line journaling tool that felt native to Unix-style workflows. I wanted something I could rely on daily—lightweight, intuitive, and with a design philosophy centered around clarity and control.

2 Design Evolution

2.1 Phase 1: Structure Before Clarity

nitially, the entire program was written in a purely procedural style. It consisted of two main functions—`display()` and `addEntry()`—which handled reading and writing entries respectively. A small helper function took care of timestamp generation and UNIX-to-human time conversion.

While this design gave the program a basic structure, it was fragile. Any edit to the `jrnl` file could easily break `getline()`. More importantly, the procedural nature of the code made adding new functionality difficult, often requiring partial or complete rewrites.

It worked and fulfilled my initial needs, but it felt far too brittle for long-term or even personal use. Still, this phase was a major milestone. It was my first fully coherent, working program—something I could actually use. I also learned a lot: about `CMake`, time libraries (at least superficially), ANSI color codes, and the importance of clean header/implementation separation. These lessons laid the groundwork for smoother development later on.

2.2 Phase 2: OOP Refactor

This was the biggest structural change—converting the program from a procedural to an object-oriented design. Classes were the main hurdle at first, but once I got them working together, the overall design started to feel cleaner and more natural.

During this phase, I also reworked the display functionality to support more flexible arguments and specifiers. One interesting challenge was parsing expressions like `"*3"`—I needed to extract the numeric part while detecting whether the `*` appeared before or after the number.

My first attempt was a naive ASCII-based conversion, simply doing `num - '48'`. That failed for multi-digit numbers. I eventually fixed it using `stoi()`, which taught me more about how string parsing and type conversion actually work in practice.

This phase gave me a much deeper understanding of C++—not just its syntax, but how its pieces interact beneath the surface. It felt like progress in both code and mindset.

2.3 Post-Refactor Milestones: Atomic Saves and Configuration

After the OOP refactor, the project crossed its most important stability milestone so far: **atomic file saves** and **robust configuration handling**. These two changes fundamentally altered how safe and trustworthy the tool became.

Initially, the program followed a dangerously naive save strategy. The journal file was cleared as soon as it was read into memory, and then rewritten from scratch on every save. If the program crashed, the system powered off, or any unexpected failure occurred during a write, the entire journal could be lost. At the time, I underestimated how catastrophic this flaw was.

Eventually, I realised that correctness is meaningless without *durability*. This led me to learn about atomic file replacement. Since modern C++ does not provide a direct equivalent of `fsync()`, I moved fully into a POSIX-based approach:

- Write all entries to a temporary `.tmp` file.
- Explicitly flush it to disk using `fsync()`.
- Close the file descriptor.
- Atomically replace the original journal using `rename()`.
- Finally, `fsync()` the parent directory for extra safety.

This ensured that at no point could the journal exist in a partially-written state. Either the old version survived, or the new one replaced it completely. There was no in-between corruption anymore. This alone transformed the tool from a toy into something I could genuinely rely on.

Around the same phase, I also implemented full configuration file parsing. The journal now respects XDG base directory specifications when available, falling back to traditional `$HOME` locations otherwise. If no configuration is found, one is generated automatically with sane defaults. Similarly, all target directories and journal files are created lazily and safely.

Alongside these changes, I also completed a long-overdue **parser refactor**. The earlier argument handling logic had grown organically and was tightly coupled to the execution flow. This made extending the interface increasingly fragile. The refactor separated *what* the user requests from *how* the program executes it, bringing the design closer to the subcommand-based structure used by tools like `git`. This change did not add visible features, but it significantly improved modularity, readability, and future extensibility.

This phase also forced me to confront a hard truth about software development: **it is practically impossible to write fully error-proof code from the start**. While I eventually implemented comprehensive edge-case handling, I realised that I should have introduced defensive checks much earlier in development. Doing so incrementally would have prevented the error-handling phase

from turning into such a large and intimidating task.

Nonetheless, this stage became one of the most valuable learning experiences of the entire project. I gained a real, working understanding of:

- POSIX file descriptors and low-level I/O
- Command dispatchers and the simplicity they bring
- The guarantees and limitations of atomic filesystem operations
- The difference between logical correctness and durability
- Why safety mechanisms must be designed, not patched on later

More than any feature addition, this milestone changed how I think about software reliability. The journal no longer just *works*—it now fails *responsibly*.

2.4 Work in Progress and Future Plans

The next major focus areas are as follows:

- **Backup System:**
  Implement an automatic backup feature with optional timed snapshots to protect against data loss.
- **Search and Filter:**
  Add the ability to search entries by dates—potentially merging with the display specifiers.
- **Export Features:**
  Allow exporting journal entries to different formats (for example, LaTeX) for archival or printing.

Long-term, I aim to refine configuration handling, improve modularity, and make the project robust enough for daily use. Eventually, I'd like to release it publicly—not as a polished product, but as something honest, useful, and genuinely mine.