**Suggested Rust Source Code Structure**

src/

├── cli/ # Command Line Interface

│ ├── mod.rs # CLI module entry point

│ ├── args.rs # Argument parsing and CLI options handling

│ └── commands.rs # CLI command definitions (traverse, collect, etc.)

│

├── gui/ # Future Graphical User Interface (can be optional for now)

│ ├── mod.rs # GUI module entry point (placeholder for now)

│ ├── app.rs # GUI application logic

│ └── windows.rs # Platform-specific GUI details for Windows (optional for now)

│

├── core/ # Core functionality (platform-independent)

│ ├── mod.rs # Core library entry point

│ ├── traversal.rs # Directory traversal logic (platform-agnostic)

│ ├── entry\_mapper.rs # Mapping DirEntry to custom DirEntry struct

│ ├── metadata.rs # File metadata extraction

│ ├── filters.rs # Path/file filters, exclusions, etc.

│ └── polars\_adapter.rs # Adapter for Polars integration, transforms data into a dataframe

│

├── fs/ # File system-specific handlers

│ ├── mod.rs # File system trait and dispatcher

│ ├── ntfs.rs # NTFS-specific logic and implementation

│ ├── ext.rs # EXT2/3/4-specific logic and implementation

│ ├── xfs.rs # XFS-specific logic and implementation

│ └── macfs.rs # Mac file systems logic (HFS/APFS)

│

├── platform/ # Platform-specific logic

│ ├── mod.rs # Platform module entry point, exposing OS traits and platform handling

│ ├── windows.rs # Windows-specific platform code (drive listing, etc.)

│ ├── unix.rs # Unix-specific platform code (handling EXT, symbolic links, etc.)

│ └── macos.rs # macOS-specific code (HFS+, APFS, etc.)

│

├── disk/ # Disk-level operations (raw disk reading, block devices, etc.)

│ ├── mod.rs # Disk operations module entry point

│ ├── ntfs\_disk.rs # NTFS raw disk reader (handling partitioning, MFT parsing)

│ ├── ext\_disk.rs # EXT raw disk reader

│ ├── macfs\_disk.rs # macOS raw disk reader (APFS, HFS+)

│ └── common.rs # Shared disk-level utilities (sector reading, caching, etc.)

│

├── config/ # Configuration management (global settings, paths, ignored files)

│ ├── mod.rs # Configuration management entry point

│ └── settings.rs # Parsing and managing config options (e.g., via YAML, TOML, JSON)

│

├── logging/ # Centralized logging

│ ├── mod.rs # Logging module entry point

│ └── logger.rs # Sets up loggers, formatting, log levels

│

├── utils/ # Utility functions, helper methods

│ ├── mod.rs # Utilities module entry point

│ ├── path\_utils.rs # Path-related utility functions

│ └── time\_utils.rs # Time-related utilities, timestamp conversion, etc.

│

└── main.rs # Main entry point (for CLI)

**1. CLI/GUI Separation**

* **cli/**: This directory is dedicated to handling command-line operations. This is where you'd define:
  + **args.rs**: Handle parsing CLI arguments using a crate like clap.
  + **commands.rs**: Define commands like traverse, collect, or any subcommands your CLI will offer. This keeps CLI logic separate from core logic, allowing easy expansion later (like supporting GUI).
* **gui/**: This folder is set up for future GUI work. It would contain your Rust code interacting with a library like egui or tauri for building cross-platform GUIs. Keeping this separate ensures no tight coupling between GUI and CLI, allowing them to share core logic but not interfere with each other.

**2. Core Module**

The core/ folder holds the heart of the platform-independent logic:

* **traversal.rs**: Responsible for walking directories and files across platforms, handling the hierarchical structure of directories.
* **metadata.rs**: Extracts metadata (size, timestamps, permissions, etc.) and creates a common interface, leaving file-system-specific differences to the fs/ module.
* **filters.rs**: Contains logic for filtering files/directories based on user input (e.g., ignoring hidden files, .git directories, etc.).
* **polars\_adapter.rs**: Converts the data collected during traversal into a Polars DataFrame, which can later be used for further analysis.
* **entry\_mapper.rs**: A module specifically designed to map std::fs::DirEntry objects into your custom DirEntry struct as files are discovered. This could happen concurrently within the traversal threads. The main function in this module takes in a std::fs::DirEntry and returns your custom DirEntry struct, transforming any necessary metadata. This modular approach allows you to change or extend how file data is mapped without modifying the traversal logic itself.

**3. File System Handlers (fs/)**

This is where you manage file system-specific logic:

* **mod.rs**: Exposes the trait FileSystemHandler that defines methods like read\_metadata, list\_files, and other file system-specific operations.
* **ntfs.rs, ext.rs, xfs.rs, macfs.rs**: Each file system gets its own module. These implement the FileSystemHandler trait, ensuring a consistent interface across different file systems. This allows your core logic to call fs::FileSystemHandler::read\_metadata without worrying about specific file system internals.

This allows easy extension for new file systems in the future, following the Open/Closed Principle (open for extension, closed for modification).

**4. Platform-Specific Code (platform/)**

The platform/ module contains platform-specific logic, abstracted into traits and types. This layer abstracts away OS-specific operations:

* **windows.rs**: Handles Windows-specific behavior such as drive enumeration, path handling, and NTFS specifics.
* **unix.rs**: Handles Unix-like platforms (Linux, BSD) with EXT, XFS, and symbolic links.
* **macos.rs**: Handles macOS-specific operations such as reading from HFS+ or APFS, as well as drive-specific metadata.

Each platform module should expose traits or functions that are platform-specific, while the core functionality remains abstracted and platform-agnostic.

**5. Disk-Level Operations (disk/)**

Since you have separated disk reading from directory reading, this folder handles the low-level raw disk I/O:

* **ntfs\_disk.rs, ext\_disk.rs, macfs\_disk.rs**: Implement disk-specific logic such as raw sector reading, Master File Table (MFT) parsing for NTFS, and EXT superblock reading. These modules handle the low-level intricacies of each file system.
* **common.rs**: Contains shared utilities for disk-level operations (e.g., reading disk blocks, caching strategies).

This separation ensures that your high-level directory traversal is not concerned with how raw data is read, allowing you to optimize the disk reader for performance without affecting traversal logic.

**6. Configuration (config/)**

The config/ module manages application configuration, user preferences, and environment settings:

* **settings.rs**: Handles loading configuration options from TOML, YAML, or JSON files. This includes options like file path filters, file system preferences, ignored directories, logging verbosity, etc.

**7. Logging (logging/)**

Centralized logging ensures that the application has consistent logging across modules:

* **logger.rs**: Sets up the logging configuration, handling log levels, formatting, and file outputs. This allows your application to have structured logs for debugging or user output, with minimal logging code scattered across modules.

**8. Utilities (utils/)**

This folder holds various utility functions that don't belong to any specific module but are shared across the application:

* **path\_utils.rs**: Functions to handle path manipulation, normalization, and validation.
* **time\_utils.rs**: Functions for time formatting, timestamp conversion, and duration handling.

**9. Entry Point (main.rs)**

This is the entry point for your application:

* The main.rs file is minimal, responsible for calling the cli module to parse commands and arguments and dispatching them to the core logic.

**Key Advantages of This Structure:**

1. **Separation of Concerns**: Each component is responsible for a single part of the functionality (CLI vs core vs file system vs platform), which improves modularity.
2. **Extensibility**: New file systems, platforms, or interfaces (GUI) can be added with minimal changes to the core logic.
3. **Platform Independence**: The platform-specific details are isolated in their own modules (platform/, disk/), ensuring cross-platform support.
4. **Testability**: This structure allows for easy unit testing by mocking specific components (e.g., disk readers, file system handlers) while testing core logic independently.
5. **Performance Optimizations**: You can optimize low-level disk reading or platform-specific code without affecting higher-level logic, keeping the system efficient and maintainable.

**Conclusion**

This structure organizes the codebase in a clean, modular way while ensuring that future extensions like adding more file systems, integrating a GUI, or optimizing disk reads remain straightforward. By isolating platform-specific code and keeping file system logic separated, the structure ensures that the system is flexible and scalable.