DVCS MODULE SPECIFICATION: DATA TYPES, ACCEPTANCE AND MODULE TESTS, AND DVCS PROTOTYPE (GROUP WORK)

Yuesong Huang
yhu116@u.rochester.edu

Alvin Jiang
yjiang54@ur.rochester.edu

Duy Pham
tuanduy601@gmail.com

Shervin Tursun-Zade
s.tursun-zade@rochester.edu

Group Name: DAYS
CSC 253/453
University of Rochester
October 27, 2024

1 PART A: ACCEPTANCE TESTS

Designer: Alvin Jiang, **Reviewer:** Yuesong Huang

The following functions are required for the DVCS system:

1. `init`
2. `clone`
3. `add`
4. `remove`
5. `status`
6. `heads`
7. `diff`
8. `cat`
9. `checkout`
10. `commit`
11. `log`
12. `merge`
13. `pull`
14. `push`

We have designed seven acceptance tests to evaluate every function.

1.1 ACCEPTANCE TEST 1: REPOSITORY INITIALIZATION

- **Functions tested**: `init` and `clone`
- This test verifies whether the initialization of a repository is successful or not. `init` and `clone` are the most suitable functions for this acceptance test since one initializes a global repository, and the other initializes a local repository from a global one.
- **Testing**: Run `init` to initialize a new global repository and run `clone` on it to create a local copy.

- **Expected Results**: The repository is successfully initialized with a new global repository structure created by `init`. When `clone` is run, a local copy of the global repository is created with all branches, files, and histories identical to the original.

1.2 ACCEPTANCE TEST 2: REPOSITORY STATUS

- **Functions tested**: `add`, `remove`, `diff`, and `status`
- This test verifies that the stage editing functions (`add` and `remove`) work as expected by allowing files to be staged or unstaged. Staging is a critical step in version control, as it allows users to prepare selected changes for a commit, ensuring that only relevant modifications are saved into the repository. Change tracking commands like `diff` and `status` visualize all the changes to assist the staging process.
- **Testing**: Write changes to several files in the repository. Run `add` to add these changes to a stage and run `remove` to remove some (but not all) of these changes. Run `diff` on a changed file to verify changes. Run `status` to verify that changes are correctly tracked.
- **Expected Results**: `add` successfully stages specified files, and `remove` unstages them without deleting the files. `diff` displays line-by-line differences for staged or modified files, indicating added, removed, or modified lines. `status` lists all staged, modified, and untracked files, accurately reflecting their current state.

1.3 ACCEPTANCE TEST 3: SYNCHRONIZATION 1

- **Functions tested**: `commit`, `push`, and `merge`
- This test ensures that the repository can synchronize changes across different remote machines using `commit`, `push`, and `merge`. `commit` and `push` facilitate data transfer between the repository and remote machines, while `merge` is important for combining changes made in different branches.
- **Testing**: On a side branch, run `commit` to commit changes and `push` to push this change to the repository. Then run `merge` to merge the side branch into the main branch.
- **Expected Results**: The `commit` command successfully records changes with a unique commit ID and user-provided message. `push` uploads this commit to a remote repository, and `merge` integrates the changes from a side branch into the main branch without conflicts or with conflict alerts if discrepancies exist.

1.4 ACCEPTANCE TEST 4: SYNCHRONIZATION 2

- **Function tested**: `pull`
- This test ensures that the repository can synchronize changes across different remote machines using `pull`. `pull` acquires the most recent push from the repository.
- **Testing**: Run **Acceptance Test 3** on MACHINE 1 and run `pull` on MACHINE 2 to retrieve the latest version.
- **Expected Results**: Running `pull` on MACHINE 2 retrieves the latest pushed changes from MACHINE 1, synchronizing the local repository on MACHINE 2 with the current state of the remote repository.

1.5 ACCEPTANCE TEST 5: REPOSITORY HISTORY

- **Function tested**: `log`
- This test provides a commit history with all relevant metadata such as commit messages, author names, and timestamps.
- **Testing**: After running **Acceptance Test 3** multiple times, run `log` to verify that all commits are recorded and metadata are displayed correctly.
- **Expected Results**: The `log` command displays a list of all commits in chronological order, showing details such as commit messages, author names, and timestamps. This confirms that the commit history is accurately recorded and that metadata is correctly preserved.

1.6 Acceptance Test 6: Branch Management

- **Functions tested**: `heads` and `checkout`
- This test ensures that the repository can effectively manage multiple branches. Viewing available branches (`heads`) and switching between them (`checkout`) are fundamental features for branching workflows, enabling users to manage different lines of development and experiment with new features without affecting the main branch.
- **Testing**: While on the main branch, run `heads` to list all available branches in the repository. Run `checkout` to switch to any available side branch. Switch back to the main branch when finished.
- **Expected Results**: The `heads` command lists all branches, with an indicator of the current branch. Running `checkout` switches to the specified branch, with changes reflected in the working directory according to the target branch's latest commit state. Switching back to the main branch restores its specific content and commit history.

1.7 Acceptance Test 7: File Inspection

- **Functions tested**: `cat`
- This test ensures that file content is displayed correctly.
- **Testing**: Run `cat` on a file to retrieve its content. Compare the content to the original file to ensure they are identical.
- **Expected Results**: Running `cat` on a file displays its complete content in the output. The displayed content should match the original file content exactly, verifying that file retrieval and display work accurately.

2 Part B: Updated Module Guide and Work Assignment

2.1 Introduction

This module guide is for the Distributed Version Control System (DVCS) designed by the DAYS group. The system is divided into three top-level modules, each responsible for different aspects of the system. This guide breaks down each module into smaller sub-modules, describing their responsibilities, secrets, facilities, interfaces, and how they interact with other modules.

2.2 Top-Level Modules Overview

- **File System Hiding:** Manages parts of the system that need to change if the underlying file system is altered (e.g., switching between Windows, Linux, or MacOS).
- **Behavior Hiding:** Manages user-facing commands and interfaces, adapting to changes in user requirements or commands.
- **Repository Hiding:** Manages core repository functionality, including operations like cloning, committing, pushing, and managing repository status.

2.3 Updated Module Guide

A.1 File System Hiding Module

- **Role:** Manages storage and retrieval of repository data, ensuring platform compatibility and abstracting file system-specific operations. It enables reading, writing, creating directories, and managing metadata, allowing **Repository Hiding** and **Behavior Hiding** modules to focus on logical operations.
- **Primary Secrets:**
  - How files and directories are managed across different file systems (e.g., handling differences between Windows, Linux, and macOS).
  - The method of reading and writing data in a format compatible with DVCS needs, such as serializing and deserializing metadata.

- **Secondary Secrets:**
  - Optimizations for file access and directory management.
  - Management of metadata, including file timestamps, permissions, and size.
- **Facilities:**
  - `read_file(path: &str) -> Result<String, Error>`
  - `write_file(path: &str, content: &str) -> Result<(), Error>`
  - `create_directory(path: &str) -> Result<(), Error>`
  - `list_files(directory: &str) -> Result<Vec<String>, Error>`
  - `delete_file(path: &str) -> Result<(), Error>`
  - `get_file_metadata(path: &str) -> Result<FileMetadata, Error>`
- **Interfaces:**
  - **Repository Hiding Module**: Uses `read_file`, `write_file`, and `create_directory` for storing and accessing repository metadata, such as during `init`, `clone`, or `commit` operations.
  - **Behavior System Hiding Module**: Utilizes `list_files` and `get_file_metadata` to display file-related information in commands like `status`.
- **Notes on Decomposition**:
  - **Reasons**: This module isolates file system interactions to ensure that DVCS can support different operating systems and storage methods. It allows changes to storage mechanisms without affecting the repository logic, keeping data storage and management concerns separate.
  - **Rules**: This module is solely responsible for reading, writing, and managing data on disk, making no assumptions about the structure or semantics of the data.
  - **Fuzzy Areas**:
    * Overlap with **Repository Hiding** occurs when performing operations like `init` and `clone` that involve creating directories and writing initial data. This is managed by **Repository Hiding** initiating these processes while **File System Hiding** handles the actual data storage.
    * Interacts with **File System Hiding** during commands like `init` that involve creating directories or files, requiring clear interfaces to ensure proper division of responsibilities.

B.1.1 File Interaction Module

- **Role:** Manages the reading and writing of file content, ensuring consistency and compatibility with different storage formats.
- **Primary Secret**: How files are accessed and stored, including format-specific optimizations.
- **Secondary Secret**: Error handling strategies for scenarios like missing files or permission issues.
- **Facilities**:
  - `read_file(path: &str) -> Result<String, Error>`
  - `write_file(path: &str, content: &str) -> Result<(), Error>`

B.1.2 Directory Management Module

- **Role:** Manages the creation, deletion, and organization of directories, enabling repositories to maintain their structure.

- **Primary Secret**: The logic for handling directory structures and permissions across various file systems.
- **Secondary Secret**: Methods for efficiently listing files and handling recursive operations.
- **Facilities**:
  - `create_directory(path: &str) -> Result<(), Error>`
  - `list_files(directory: &str) -> Result<Vec<String>, Error>`
  - `delete_file(path: &str) -> Result<(), Error>`

B.1.3 Metadata Management

- **Role:** Manages access to file metadata, allowing the DVCS to retrieve information about files like size, modification time, and other attributes.
- **Primary Secret**: How metadata is stored and retrieved for different operating systems.
- **Secondary Secret**: Efficient retrieval of metadata without impacting file access performance.
- **Facilities**:
  - `get_file_metadata(path: &str) -> Result<FileMetadata, Error>`

A.2 Behavior Hiding

- **Role:** Manages user-facing interactions with the DVCS, interpreting user input, validating commands, coordinating execution, and displaying results. It abstracts the complexities of repository and file system operations, allowing users to interact with the DVCS intuitively.
- **Primary Secrets:**
  - The logic for command parsing and validation, ensuring that commands are interpreted correctly.
  - Methods for transforming user inputs into high-level commands that other modules can process.
- **Secondary Secrets:**
  - Strategies for error handling and user feedback mechanisms when commands are invalid or fail.
  - Formatting rules for presenting output to ensure it is user-friendly and adaptable to different display contexts.
- **Facilities:**
  - `parse_command(input: &str) -> Result<Command, Error>`
  - `validate_command(command: &str) -> Result<ValidCommand, Error>`
  - `execute_command(command: Command) -> Result<(), Error>`
  - `get_user_confirmation(prompt: &str) -> Result<bool, Error>`
  - `display_output(output: &str, style: OutputStyle)`
  - `format_output(data: &str, format_type: OutputFormat) -> String`
  - `log_error(error: &Error, context: &str)`
- **Interfaces:**
  - **Repository Hiding Module**: Communicates with this module to invoke repository-related actions such as `commit`, `merge`, `checkout`, and `log`, ensuring that user commands translate into repository operations.
  - **File System Hiding Module**: Uses this module's facilities when commands indirectly require file interactions, such as status checks or init commands that affect the file system.

  - **Display**: Use `display_output` to present status updates, logs, and other results derived from repository and file system data.
- **Notes on Decomposition**:
  - **Reasons**: By isolating user interaction logic, this module simplifies changes to user commands or input handling without needing to modify core repository or file system operations. It allows the user interface to evolve independently.
  - **Rules**: This module exclusively handles parsing, validation, and execution of user commands, ensuring that each command reaches the correct module for execution without needing to know the details of how those actions are implemented.
  - **Fuzzy Areas**:
    * Overlaps with **Repository Hiding** when displaying status and log information, as the behavior module needs to understand how to present data retrieved from the repository. This is managed by keeping presentation logic within **Behavior Hiding** and data retrieval within **Repository Hiding**.
    * Interacts with **File System Hiding** during commands like `init` that involve creating directories or files, requiring clear interfaces to ensure proper division of responsibilities.

B.2.1 Command Parsing Module

- **Role:** Converts user input into structured commands that the system can understand, ensuring consistency and correctness.
- **Primary Secret**: The grammar and syntax rules for interpreting commands, making the command language user-friendly and flexible.
- **Secondary Secret**: Handling syntax errors and unexpected input tokens, providing clear feedback to users.
- **Facilities**:
  - `parse_command(input: &str) -> Result<Command, Error>`
  - `validate_command(command: &str) -> Result<ValidCommand, Error>`

B.2.2 Command Handler Module

- **Role:** Delegates validated commands to the appropriate module, ensuring each action is carried out by the correct part of the system.
- **Primary Secret**: The logic for mapping commands to internal DVCS functions.
- **Secondary Secret**: Managing the sequence and dependencies of commands (e.g., ensuring that status is checked before performing a commit).
- **Facilities**:
  - `execute_command(command: Command) -> Result<(), Error>`
  - `get_user_confirmation(prompt: &str) -> Result<bool, Error>`

B.2.3 Output Display Module

- **Role:** Manages the presentation of command results and user feedback, supporting different output styles (e.g., concise vs. detailed) to cater to various user needs.
- **Primary Secret**: The formatting logic for different types of outputs, such as `status` reports, `diff` results, or `log`s.
- **Secondary Secret**: Methods for managing output styles, such as text formatting, verbosity levels, color-coding, and detailed logging for advanced users.
- **Facilities**:
  - `display_output(output: &str, style: OutputStyle)`

  - `format_output(data: &str, format_type: OutputFormat) -> String`
  - `log_error(error: &Error, context: &str)`

A.3 Repository Hiding Module

- **Role:** Manages the structure and organization of repositories in the DVCS, including revision history, branching, and synchronization between repositories, but delegates storage to the **File System Hiding Module**.
- **Primary Secrets:**
  - Internal data structures for repository states.
  - Mechanisms for managing branches and tracking heads.
- **Secondary Secrets:**
  - Algorithms for merging revisions and handling conflicts.
  - Metadata structures for tracking changes.

  Facilities:
  - `init(directory: &str) -> Result<(), Error>`
  - `clone(remote_url: &str, directory: &str) -> Result<(), Error>`
  - `commit(message: &str) -> Result<(), Error>`
  - `merge(branch: &str) -> Result<(), Error>`
  - `push(remote: &str) -> Result<(), Error>`
  - `pull(remote: &str) -> Result<(), Error>`
  - `status() -> Result<String, Error>`
  - `heads() -> Result<Vec<String>, Error>`
  - `log() -> Result<String, Error>`
  - `diff(revision1: &str, revision2: &str) -> Result<String, Error>`
  - `checkout(revision: &str) -> Result<(), Error>`
  - `add(file_path: &str) -> Result<(), Error>`
  - `remove(file_path: &str) -> Result<(), Error>`
  - `cat(file_path: &str, revision: &str) -> Result<String, Error>`
- **Interfaces:**
  - **File System Hiding Module:** Relies on facilities like `read_file`, `write_file`, and `create_directory` to perform actual storage and retrieval of repository data when modifying structures through commands like `commit`, `pull`, or `checkout`.
  - **Behavior Hiding Module:** Interacts with commands like `commit`, `merge`, and `status` through the `execute_command` function, providing user-facing access to repository operations.
- **Notes on Decomposition:**
  - **Reasons:** Separating repository data management from storage ensures that modifications to data structures, such as changes to how branches are organized or revisions tracked, do not impact how data is stored on disk. This allows the repository logic to evolve without requiring changes to file handling.
  - **Rules:** This module focuses on maintaining the integrity of the repository's logical structure, such as the organization of commits, branches, and history, while delegating storage to **File System Hiding**.
  - **Fuzzy Areas:**
    - Overlaps with **Behavior Hiding** may occur when determining what data to display through `status()` and `log()`. This is handled by **Behavior Hiding** determining how to present the data while **Repository Hiding** ensures the data's accuracy.

    * `Merge` and `Diff` interactions require clear coordination with **File System Hiding** to ensure that changes are applied correctly to the file system when revisions are compared or merged.
    * `init()` and `clone()` require a combination of setting up repository structures and creating directories, which is managed by clear delegation of directory creation to **File System Hiding**.

B.3.1 Repository Metadata Module

- **Role:** Manages the setup and maintenance of repository metadata, such as branch pointers and commit history.
- **Primary Secret**: The structure of the metadata files and how they interact with the commit tree.
- **Secondary Secret**: Optimizations for accessing and updating metadata.
- **Facilities**:
  - `init(directory: &str) -> Result<(), Error>`
  - `clone(remote_url: &str, directory: &str) -> Result<(), Error>`

B.3.2 Revision Management Module

- **Role:** Manages the creation and organization of revisions within the repository
- **Primary Secret**: The representation of commits and the history graph of revisions.
- **Secondary Secret**: How changes are serialized into commits.
- **Facilities**:
  - `commit(message: &str) -> Result<(), Error>`
  - `log() -> Result<String, Error>`

B.3.3 Branch Management Module

- **Role:** Handles the management of branches and their pointers.
- **Primary Secret**: The internal organization of branch pointers and references.
- **Secondary Secret**: Mechanisms for listing, adding, and removing branches.
- **Facilities**:
  - `heads() -> Result<Vec<String>, Error>`
  - `checkout(revision: &str) -> Result<(), Error>`

B.3.4 Synchronization Module

- **Role:** Manages interaction with remote repositories, including pulling and pushing changes.
- **Primary Secret**: How changes are synchronized between local and remote repositories.
- **Secondary Secret**: The strategies for handling synchronization conflicts.
- **Facilities**:
  - `push(remote: &str) -> Result<(), Error>`
  - `pull(remote: &str) -> Result<(), Error>`

B.3.5 Merge and Diff Module

- **Role:** Manages merging revisions and comparing differences between them.
- **Primary Secret**: The logic for conflict resolution during merges.
- **Secondary Secret**: Algorithms for computing diffs.

- **Facilities**:
  - `merge(branch: &str) -> Result<(), Error>`
  - `diff(revision1: &str, revision2: &str) -> Result<String, Error>`

2.4 Work Assignment

- **File System Hiding Module (A.1)** - Assigned to *Shervin Tursun-Zade*
  - **File Interaction Module (B.1.1)** - Responsible for reading and writing file contents.
  - **Directory Management Module (B.1.2)** - Manages creation, deletion, and listing of directories.
  - **Metadata Management Module (B.1.3)** - Retrieves file metadata such as timestamps and permissions.
- **Behavior Hiding Module (A.2)** - Assigned to *Alvin Jiang*
  - **Command Parsing Module (B.2.1)** - Converts user input into structured commands.
  - **Command Handler Module (B.2.2)** - Directs validated commands to appropriate modules.
  - **Output Presentation Module (B.2.3)** - Formats and displays command results and error messages.
- **Repository Hiding Module (A.3)** - Shared between *Duy Pham* and *Yuesong Huang*
  - **Repository Metadata Module (B.3.1)** - Initializes repository metadata.
  - **Revision Management Module (B.3.2)** - Manages commit history and revisions.
  - **Branch Management Module (B.3.3)** - Handles branch creation, deletion, and switching.
  - **Synchronization Module (B.3.4)** - Manages remote push/pull operations.
  - **Merge and Diff Module (B.3.5)** - Manages merging of branches and displaying differences between revisions.

3 Part C: Module Specification and Module Tests

3.1 Incremental Testing Strategy

To ensure that functionality increments do not disrupt existing features, we employ an incremental testing strategy. This approach verifies each increment's functionality individually before integrating it into the main system.

3.1.1 Testing Approach for Each Increment

- **Individual Increment Testing**: Each increment is tested in isolation with unit tests covering edge cases, exceptions, and expected outputs. By isolating each increment, we ensure that it behaves as expected without interference from other features.
- **Integration Testing**: After each increment passes individual testing, we conduct integration testing to ensure compatibility with existing system functionalities. This includes verifying that the new functionality interacts correctly with all other modules.
- **Regression Testing**: To prevent unintended side effects, we perform regression testing across the entire system after each functionality increment. This ensures that no existing functionality is disrupted.
- **Continuous Testing Cycle**: After adding each increment, we continuously run the incremental, integration, and regression tests throughout the development process. This approach guarantees that newly added functionalities coexist harmoniously with the core features.

3.2 Consistent Error Handling and Logging

To maintain a unified approach to error handling and logging across modules, we establish the following guidelines:

3.2.1 Error Handling Guidelines

Each module adheres to consistent error handling conventions, ensuring predictable behavior for both users and developers:

- **Standardized Error Types**: Modules use custom error enums (e.g., `FileError`, `CommandError`) to provide meaningful error messages tailored to each operation.
- **Error Propagation**: Errors are propagated up the call stack in a consistent manner, allowing the Behavior Hiding Module to manage user-facing errors while internal modules handle internal errors.
- **Descriptive Messages**: All errors include descriptive messages and, where possible, suggested resolutions to aid in debugging and user guidance.

3.2.2 Logging Guidelines

Logging practices are standardized across all modules to facilitate easier debugging and ensure logs are meaningful:

- **Logging Levels**: We use consistent logging levels (e.g., `info`, `warning`, `error`) to classify messages based on importance, helping developers quickly locate critical issues.
- **Format Consistency**: Log entries follow a uniform format, including timestamps, module name, function name, and a detailed description of the event.
- **Centralized Log Management**: All modules log to a central location, enabling comprehensive review and cross-module traceability for error diagnostics.

This consistent approach to error handling and logging ensures a uniform user experience and simplifies maintenance by making issues easily traceable and interpretable.

A.1 File System Hiding

Designer: Shervin Tursun-Zade, **Reviewers:** Duy Pham and Yuesong Huang

The File System Hiding Module is responsible for all file and directory operations, providing platform compatibility and abstracting file system-specific operations. The module includes submodules for handling file interactions, directory management, and metadata management.

B.1.1 File Interaction

- **Structs and Traits**
  1. `FileInteraction` (struct): Encapsulates file handling operations, providing a unified interface for file read and write functions.
  2. `FileOperations` (trait): Defines essential file interaction methods, including `write_file` and `read_file`.
- **Visible Methods**
  1. `write_file(path: &str, content: &str) -> Result<(), Error>`: Writes the provided content to the specified file path, overwriting the file if it already exists.
  2. `read_file(path: &str) -> Result<String, Error>`: Reads the content of the specified file and returns it as a `String`.
- **Module Tests**
  1. `test_write_file_new()`: Verifies that a new file is created and written successfully.

2. `test_overwrite_file()`: Ensures that an existing file's content is correctly overwritten.
3. `test_write_empty_file()`: Verifies handling of writing an empty string to a file.
4. `test_write_no_permission()`: Tests error handling when write permissions are lacking.
5. `test_write_invalid_path()`: Checks for correct error response with an invalid file path.
6. `test_large_content_write()`: Ensures performance and stability with large file sizes.
7. `test_read_existing_file()`: Verifies content retrieval from an existing file.
8. `test_read_empty_file()`: Confirms correct handling of reading an empty file.
9. `test_read_nonexistent_file()`: Tests for error response when reading a non-existent file.
10. `test_read_no_permission()`: Ensures error handling for files lacking read permissions.
11. `test_read_special_characters()`: Confirms correct reading of files with special characters.
12. `test_large_content_read()`: Tests stability when reading large files.

- **External Module**
  - **`std::fs`**: Rust's standard library module used for file operations. The **File Interaction** module uses `std::fs` for functions like reading and writing files.

B.1.2 Directory Management

- **Structs and Traits**
  1. `DirectoryTree` (struct): Represents the entire directory structure in a tree format, supporting file and directory nodes.
  2. `FileNode` (struct): Represents individual files as leaf nodes, each containing a directory of versioned files.
  3. `DirectoryNode` (struct): Represents directories as non-leaf nodes in the tree.
- **Visible Methods**
  1. `create_directory(path: &str) -> Result<(), Error>`: Creates a new directory at the specified path.
  2. `list_files(directory: &str) -> Result<Vec<String>, Error>`: Lists all files and directories within the specified directory.
  3. `delete_file(path: &str) -> Result<(), Error>`: Deletes a file at the specified path.
- **Module Tests**
  1. `test_create_directory()`: Verifies that a new directory is created successfully at the specified path.
  2. `test_list_files_in_directory()`: Ensures that the contents of a directory are listed correctly.
  3. `test_list_empty_directory()`: Checks that an empty directory is listed correctly as empty.
  4. `test_delete_file()`: Confirms that a file and its version history are deleted successfully.
  5. `test_create_nested_directory()`: Verifies that nested directories can be created and structured as expected.
  6. `test_permissions_error()`: Ensures that proper errors are returned when attempting operations without sufficient permissions.
- **External Module**
  - **`std::fs`**: Utilized for handling basic file system operations such as directory creation, file listing, and deletion.

B.1.3 METADATA MANAGEMENT

- **Structs and Traits**
  1. `FileMetadata` (struct): Stores metadata details of a file, such as size, modification time, and permissions.
  2. `MetadataAccess` (trait): Defines the method required for accessing file metadata across different operating systems.
- **Visible Methods**
  1. `get_file_metadata(path: &str) -> Result<FileMetadata, Error>`: Retrieves the metadata for the specified file path, including details such as file size, modification time, and permissions.
- **Module Tests**
  1. `test_get_file_metadata_existing()`: Verifies that metadata is correctly retrieved for an existing file.
  2. `test_get_file_metadata_nonexistent()`: Ensures that an appropriate error is returned for a non-existent file.
  3. `test_get_file_metadata_no_permission()`: Tests error handling when access permissions are insufficient.
  4. `test_get_file_metadata_special_chars()`: Confirms metadata retrieval for files with names containing special characters.
  5. `test_get_file_metadata_performance()`: Ensures that metadata retrieval is efficient and does not significantly impact file access performance.
  6. `test_get_file_metadata_large_file()`: Tests metadata retrieval for a large file, ensuring stability.
- **External Module**
  - **`std::fs::metadata`**: Part of Rust's standard library used to access file metadata information, such as size and modification time.

A.2 BEHAVIOR HIDING MODULE

Designer: Alvin Jiang, **Reviewer:** Shervin Tursun-Zade

The Behavior Hiding Module manages user-facing interactions with the DVCS, interpreting user input, validating commands, coordinating command execution, and displaying results. It abstracts complexities, allowing users to interact with the DVCS intuitively.

B.2.1 COMMAND PARSING

- **Structs and Traits**
  1. `Command` (enum): Represents the set of recognized commands (14 in total).
  2. `ValidCommand` (struct): Contains parsed and validated command details.
  3. `Valid` (trait): A trait indicating validness.
  4. `Error` (enum): Custom error handling for invalid commands, parsing errors, or validation failures.
- **Visible Methods**
  1. `parse_command(input: &str) -> Result<Command, Error>`: Parses a command string and returns a `Command` or an `Error`.
  2. `validate_command(command: &Command) -> Result<ValidCommand, Error>`: Validates the command structure, returning a `ValidCommand` or an `Error`.
  3. **Optional**: `suggest_command(input: &str) -> Option<String>`: Uses `clap` crate to suggest close matches if the command is invalid.
- **Module Tests**

1. `test_parse_valid_command()`: Ensures that a valid command, such as `cat`, is correctly parsed.
2. `test_parse_invalid_command()`: Confirms that an invalid command, like `unknown`, triggers an error response.
3. `test_validate_command()`: Verifies successful validation of a known command like `push`.
4. `test_invalid_validation()`: Ensures unrecognized commands fail validation, returning an error.
5. `test_suggest_command()`: Tests that slight typos (e.g., `puh` instead of `push`) result in a suggestion.
6. `test_no_suggestion()`: Ensures unrecognized commands (e.g., `xyz`) return no suggestion.

- **External Module**
  - **`clap`**: A Rust crate used to manage command-line interfaces and enhance command recognition.
    * `Regex::new` - Compiles regex.
    * `captures` - Matches components.

B.2.2 Command Handler

- **Structs and Traits**
  1. `CommandHandler` (struct): Manages command execution and user confirmations.
  2. `ConfirmationError` (enum): Handles errors related to confirmations.
- **Visible Methods**
  1. `execute_command(command: &Command, module: &Module) -> Result<(), Error>`: Routes command to another module.
  2. `get_user_confirmation(prompt: &str) -> Result<bool, Error>`: Prompts user for confirmation (e.g., "Are you sure?").
  3. `log_execution(command: &Command) -> Result<(), Error>`: Logs executed commands with metadata.
- **Module Tests**
  1. `test_execute_init()`: Ensures `init` command is executed successfully.
  2. `test_execute_clone()`: Verifies successful execution of the `clone` command.
  3. ...
  4. `test_execute_pull()`: Ensures `pull` command is executed correctly.
  5. `test_execute_push()`: Verifies successful execution of the `push` command.
  6. `test_log_execution()`: Confirms proper logging of executed commands.
  7. `test_system_no_execution()`: Checks handling when no command is provided, returning an error.
- **External Module**
  - **`std::io`**: Manages user input/output during confirmations.
    * `stdin` - Reads user input.
    * `stdout` - Displays output.

B.2.3 Output Presentation

- **Structs and Traits**
  1. `OutputStyle` (enum): Defines styling options, such as Plain, Error, and Success.
  2. `OutputFormat` (enum): Specifies formatting options like `JSON` or PlainText.
  3. `OutputFormatter` (struct): Handles formatting and logging.
- **Visible Methods**
  1. `format_output(data: &str, format_type: OutputFormat) -> String`: Formats the `data` string as specified.

2. `display_output(output: &str, style: OutputStyle)`: Displays output in a specified style (e.g., plain or error).
3. `log_error(error: &Error, context: &str)`: Logs errors with context for easier diagnostics.

- **Module Tests**
  1. `test_display_output_plain()`: Checks plain output display.
  2. `test_display_output_error()`: Ensures error messages are displayed correctly.
  3. `test_format_output_json()`: Verifies data formatting in `JSON`.
  4. `test_format_output_plaintext()`: Checks data return as plain text.
  5. `test_success_display()`: Ensures success messages are displayed properly.
  6. `test_log_error()`: Confirms errors are logged with context.
- **External Module**
  - **`std::fmt`**: Rust's standard formatting library for structured output.
    * `format!` - Formats strings based on patterns.

A.3 Repository Hiding Module

Designers: Yuesong Huang and Duy Pham, **Reviewer:** Alvin Jiang

The Repository Hiding Module manages the DVCS repository structure and core operations, including initializing repositories, committing changes, managing branches, and synchronizing with remote repositories.

B.3.1 Repository Metadata

Description: This module manages the setup and maintenance of repository metadata, including branch pointers and commit history.

- **Structs and Traits**
  1. `Metadata` (struct): Represents repository metadata, including branch pointers and commit history.
  2. `MetadataError` (enum): Custom error type for handling metadata-related errors.
- **Visible Methods**
  1. `init(directory: &str) -> Result<(), MetadataError>`: Initializes repository metadata by creating a new directory structure for the repository.
  2. `clone(remote_url: &str, directory: &str) -> Result<(), MetadataError>`: Clones an existing repository from a remote URL to a local directory.
- **Module Tests**
  1. `test_init_creates_directory`: Verifies that `init()` successfully creates the specified repository directory.
  2. `test_init_existing_directory`: Tests that `init()` returns an error when the specified directory already exists.
  3. `test_clone_creates_local_repo`: Ensures that `clone()` creates a local repository identical to the specified remote.
  4. `test_clone_invalid_url`: Checks that `clone()` returns an error for an invalid remote URL.
  5. `test_clone_directory_conflict`: Verifies that `clone()` handles directory path conflicts correctly.
  6. `test_metadata_structure`: Confirms that metadata structures, such as branch pointers and commit history, are set up correctly after `init()` or `clone()`.
- **External Module**
  - **`serde_json`**: Used for serializing and deserializing metadata to/from JSON format.

B.3.2 Revision Management

Description: Manages the creation and organization of revisions in the repository.

- **Structs and Traits**
  1. `Revision` (struct): Represents a single revision within the repository.
  2. `RevisionError` (enum): Custom error type for handling errors related to revision management.
- **Visible Methods**
  1. `commit(message: &str) -> Result<(), RevisionError>`: Creates a new revision in the repository with the given commit message.
  2. `log() -> Result<String, RevisionError>`: Returns a string listing all revisions in the repository in chronological order.
- **Module Tests**
  1. `test_commit_creates_revision`: Verifies that `commit()` creates a new revision with the specified message.
  2. `test_commit_timestamp`: Checks that each commit includes a timestamp.
  3. `test_log_order`: Confirms that `log()` lists all revisions in the correct order.
  4. `test_log_content`: Ensures `log()` provides detailed information for each commit, including the timestamp.
  5. `test_commit_no_changes`: Verifies that `commit()` returns an error if no changes are staged.
  6. `test_log_empty`: Confirms that `log()` returns an empty string if there are no revisions.

B.3.3 Branch Management

Description: Handles the creation, listing, and management of branches in the repository.

- **Structs and Traits**
  1. `Branch` (struct): Represents a branch within the repository.
  2. `BranchError` (enum): Custom error type for handling errors related to branch management.
- **Visible Methods**
  1. `heads() -> Result<Vec<String>, BranchError>`: Returns a list of all branch names in the repository.
  2. `checkout(revision: &str) -> Result<(), BranchError>`: Switches the current working branch to the specified revision or branch.
- **Module Tests**
  1. `test_heads_lists_branches`: Ensures `heads()` returns a list of all branches in the repository.
  2. `test_checkout_switches_head`: Verifies that `checkout()` updates the HEAD to the specified branch or revision.
  3. `test_heads_new_branch`: Checks that `heads()` reflects newly created branches.
  4. `test_checkout_nonexistent`: Confirms that `checkout()` returns an error if the specified branch or revision does not exist.
  5. `test_checkout_uncommitted_changes`: Ensures `checkout()` does not disrupt uncommitted changes.
  6. `test_checkout_valid_revision`: Verifies that `checkout()` switches to a valid revision correctly.

B.3.4 SYNCHRONIZATION

Description: Manages synchronization of repository data with remote repositories, including pulling and pushing changes.

- **Structs and Traits**
  1. `SyncError` (enum): Custom error type for handling errors related to synchronization.
- **Visible Methods**
  1. `push(remote: &str) -> Result<(), SyncError>`: Pushes changes from the local repository to a specified remote repository.
  2. `pull(remote: &str) -> Result<(), SyncError>`: Pulls changes from a specified remote repository into the local repository.
- **Module Tests**
  1. `test_push_syncs_with_remote`: Ensures that `push()` successfully syncs changes with a remote repository.
  2. `test_push_invalid_remote`: Verifies that `push()` returns an error when an invalid remote URL is provided.
  3. `test_pull_fetches_new_commits`: Confirms that `pull()` retrieves new commits from the remote repository.
  4. `test_pull_conflict`: Checks that `pull()` returns an error if there is a conflict with local changes.
  5. `test_push_network_error`: Verifies that `push()` handles network interruptions gracefully.
  6. `test_pull_merge_success`: Ensures `pull()` merges changes without data loss when there are no conflicts.

B.3.5 MERGE AND DIFF

Description: Manages merging revisions and comparing differences between revisions.

- **Structs and Traits**
  1. `MergeError` (enum): Custom error type for handling merge-related issues.
  2. `DiffError` (enum): Custom error type for handling diff-related issues.
- **Visible Methods**
  1. `merge(branch: &str) -> Result<(), MergeError>`: Merges changes from a specified branch into the current branch.
  2. `diff(revision1: &str, revision2: &str) -> Result<String, DiffError>`: Returns the differences between two specified revisions as a string.
- **Module Tests**
  1. `test_merge_incorporates_changes`: Verifies that `merge()` integrates changes from the specified branch.
  2. `test_merge_conflict_detection`: Ensures that `merge()` detects and flags conflicts when they occur.
  3. `test_diff_line_by_line`: Confirms that `diff()` returns a line-by-line comparison between two revisions.
  4. `test_merge_post_conflict_resolution`: Checks that repository state is correct after resolving conflicts during `merge()`.
  5. `test_diff_identical_revisions`: Ensures that `diff()` returns no differences when comparing identical revisions.
  6. `test_merge_no_conflicts`: Verifies that `merge()` completes successfully when there are no conflicting changes.

3.3 Incremental Testing Strategy

To ensure that functionality increments do not disrupt existing features, we employ an incremental testing strategy. This approach verifies each increment's functionality individually before integrating it into the main system.

3.3.1 Testing Approach for Each Increment

- **Individual Increment Testing**: Each increment is tested in isolation with unit tests covering edge cases, exceptions, and expected outputs. By isolating each increment, we ensure that it behaves as expected without interference from other features.
- **Integration Testing**: After each increment passes individual testing, we conduct integration testing to ensure compatibility with existing system functionalities. This includes verifying that the new functionality interacts correctly with all other modules.
- **Regression Testing**: To prevent unintended side effects, we perform regression testing across the entire system after each functionality increment. This ensures that no existing functionality is disrupted.
- **Continuous Testing Cycle**: After adding each increment, we continuously run the incremental, integration, and regression tests throughout the development process. This approach guarantees that newly added functionalities coexist harmoniously with the core features.

4 Part D: Prototype and Uses Structure

4.1 Rationale for Expansion Possibilities

Each module is designed with potential future expansions in mind, which allows the system to adapt to evolving requirements and performance needs. Below, we briefly outline the rationale for each expansion possibility.

- **Platform-Specific Optimization (File System Hiding Module)**: Different operating systems handle file operations differently. By incorporating platform-specific optimizations, the system can reduce latency in file read/write operations, especially for large repositories.
- **Enhanced Caching (File System Hiding Module)**: Adding caching strategies for frequently accessed files can drastically improve performance, particularly in scenarios where multiple files are repeatedly accessed, such as during branching and merging.
- **Command Aliases (Behavior Hiding Module)**: Allowing users to create custom command shortcuts improves efficiency and caters to user preferences, especially for power users who frequently use certain commands.
- **Localization and Internationalization (Behavior Hiding Module)**: Expanding support for multiple languages increases accessibility and usability, broadening the user base across various regions and language groups.
- **Revert and Reset Commands (Repository Hiding Module)**: By enabling the repository to revert to previous states or reset sections of the repository, users can more easily recover from errors or unintended changes.
- **History Filtering (Repository Hiding Module)**: Filtering commit logs based on criteria such as author, date, or message provides users with flexibility in reviewing revision history, supporting more complex version tracking and auditing.

4.2 File System Hiding Module Prototype

Module Overview: The File System Hiding Module (A.1) manages data storage and retrieval for the DVCS. It abstracts interactions with the file system, enabling compatibility across different platforms. This prototype design includes core functionalities needed to manage files, directories, and metadata, creating a stable basis for extension.

MINIMAL PROTOTYPE IMPLEMENTATION

The minimal prototype includes the following core facilities, each required for a foundational DVCS setup:

- `read_file(path: &str) -> Result<String, Error>`
- `write_file(path: &str, content: &str) -> Result<(), Error>`
- `create_directory(path: &str) -> Result<(), Error>`
- `delete_file(path: &str) -> Result<(), Error>`
- `get_file_metadata(path: &str) -> Result<FileMetadata, Error>`

These methods cover essential file operations (read/write), directory management, file deletion, and metadata retrieval, enabling file storage, access, and file system abstraction.

USES RELATION FOR FILE SYSTEM HIDING MODULE

The following diagram and list outline the dependencies between each function in the File System Hiding Module and how other modules interact with them:

- **`Repository Hiding Module`**
  - Uses `read_file`, `write_file`, and `create_directory` for repository metadata storage and retrieval during operations like `init` and `commit`.
- **`Behavior Hiding Module`**
  - Utilizes `list_files` and `get_file_metadata` to display status and file-related information during commands like `status`.

FUNCTIONALITY INCREMENTS

This prototype is designed to support two increments, each enhancing functionality while maintaining the stability of core methods.

Increment 1: Extended Metadata and Directory Operations

- `list_files(directory: &str) -> Result<Vec<String>, Error>`: Adds support for listing files within a directory, assisting in directory traversal and metadata inspection.
- `rename_file(old_path: &str, new_path: &str) -> Result<(), Error>`: Allows renaming files to support refactoring and restructuring within repositories.

Increment 2: Advanced Error Handling and Permissions Management

- `check_permissions(path: &str) -> Result<FilePermissions, Error>`: Provides detailed permission checks, enhancing error handling and ensuring secure file access.
- `set_file_permissions(path: &str, permissions: FilePermissions) -> Result<(), Error>`: Adds functionality for modifying file permissions, supporting customization for multi-user environments.

EXPANSION POSSIBILITIES

The File System Hiding Module is structured for further expansion to meet new requirements. Potential enhancements include:

- **Platform-Specific Optimization:** Additional facilities can be added to handle OS-specific file system nuances.
- **Enhanced Caching:** Future versions may incorporate caching strategies for frequently accessed files, improving performance.

USES STRUCTURE SUMMARY

The Uses Structure ensures that each module interacts with the File System Hiding Module in a controlled manner, maintaining abstraction for cross-platform compatibility. By keeping each increment independent and modular, the File System Hiding Module provides a foundation that supports easy addition of new features without requiring changes to core methods.

4.3 BEHAVIOR HIDING MODULE PROTOTYPE

Module Overview: The Behavior Hiding Module (A.2) manages user interactions with the DVCS, including command parsing, validation, execution, and output formatting. It acts as an intermediary between the user and the DVCS, interpreting commands and coordinating with other modules to fulfill user requests.

MINIMAL PROTOTYPE IMPLEMENTATION

The minimal prototype for the Behavior Hiding Module includes essential methods to support a basic set of commands and feedback:

- `parse_command(input: &str) -> Result<Command, Error>`
- `validate_command(command: &Command) -> Result<ValidCommand, Error>`
- `execute_command(command: Command) -> Result<(), Error>`
- `display_output(output: &str, style: OutputStyle)`

These functions cover essential user interaction tasks: parsing input, validating commands, executing commands, and presenting results. This minimal set provides a base for supporting basic commands, error handling, and output display.

USES RELATION FOR BEHAVIOR HIDING MODULE

The Uses Structure below shows the relationships among public methods in the Behavior Hiding Module and the interactions with other modules.

- **`File System Hiding Module`**
  - Used indirectly through commands like `status` or `init`, where file operations are necessary to retrieve or create files and directories.
- **`Repository Hiding Module`**
  - Invoked via commands such as `commit`, `merge`, `checkout`, and `log`, requiring interaction with repository data.

FUNCTIONALITY INCREMENTS

The Behavior Hiding Module can be extended through a series of increments, each adding new capabilities to enhance user interaction and improve command handling.

Increment 1: Enhanced Command Parsing and Validation

- `suggest_command(input: &str) -> Option<String>`: Provides command suggestions for incorrect inputs, enhancing user experience by guiding them towards correct commands.
- `get_user_confirmation(prompt: &str) -> Result<bool, Error>`: Requests confirmation from the user for potentially destructive actions, improving security and preventing unintended actions.

Increment 2: Advanced Output Formatting and Logging

- `format_output(data: &str, format_type: OutputFormat) -> String`: Adds support for various output formats (e.g., JSON, plain text), allowing the DVCS to cater to different user preferences.
- `log_error(error: &Error, context: &str)`: Logs errors with contextual information, facilitating debugging and providing a clearer understanding of failures within the DVCS.

EXPANSION POSSIBILITIES

The Behavior Hiding Module's structure supports further expansion to incorporate additional user-facing features:

- **Command Aliases:** Allows users to create custom command shortcuts, providing flexibility in command usage.
- **Localization and Internationalization:** Future expansions could include multi-language support, enhancing usability for a broader audience.
- **Interactive Mode:** Provides an interactive shell for users to enter commands without reinitializing the DVCS for each command.

USES STRUCTURE SUMMARY

The Uses Structure is designed to provide a clear separation between user interactions and core DVCS functionality. By segmenting increments, each addition enhances specific aspects of user interaction without altering core module behavior, maintaining stability across the system.

4.4 REPOSITORY HIDING MODULE PROTOTYPE

Module Overview: The Repository Hiding Module (A.3) manages the structure and organization of repositories, including revision history, branching, and synchronization between repositories. This module focuses on repository-specific functions while delegating storage tasks to the File System Hiding Module.

MINIMAL PROTOTYPE IMPLEMENTATION

The minimal prototype for the Repository Hiding Module includes foundational methods to support basic repository functionalities:

- `init(directory: &str) -> Result<(), Error>`
- `clone(remote_url: &str, directory: &str) -> Result<(), Error>`
- `commit(message: &str) -> Result<(), Error>`
- `checkout(revision: &str) -> Result<(), Error>`
- `status() -> Result<String, Error>`

This minimal set enables the creation, cloning, committing, and checking out of repositories, along with a status check to display the current state of the repository.

USES RELATION FOR REPOSITORY HIDING MODULE

The Uses Structure illustrates the relationships among public methods within the Repository Hiding Module and its interactions with other modules.

- **`File System Hiding Module`**
  - Relies on methods such as `write_file`, `read_file`, and `create_directory` for performing storage operations when executing commands like `init`, `clone`, or `commit`.

- **`Behavior Hiding Module`**
  - Communicates through commands including `commit`, `status`, and `checkout` to provide user-facing actions within the DVCS.

Functionality Increments

The Repository Hiding Module can be extended through increments, progressively adding functionality to expand repository capabilities and user control.

Increment 1: Branch and Revision Management

- `heads() -> Result<Vec<String>, Error>`: Provides a list of all branches, facilitating branch navigation and management.
- `log() -> Result<String, Error>`: Returns a chronological list of commits in the repository, assisting users in tracking revision history.
- `merge(branch: &str) -> Result<(), Error>`: Enables merging of changes from one branch into another, supporting collaborative workflows.

Increment 2: Remote Synchronization and Diffing

- `push(remote: &str) -> Result<(), Error>`: Pushes local repository changes to a remote repository, supporting distributed version control.
- `pull(remote: &str) -> Result<(), Error>`: Pulls changes from a remote repository to the local repository, keeping repositories synchronized.
- `diff(revision1: &str, revision2: &str) -> Result<String, Error>`: Provides a line-by-line comparison between two revisions, allowing users to inspect changes.

Expansion Possibilities

The Repository Hiding Module can further expand to include additional features and utilities:

- **Revert and Reset Commands:** Functions to revert to specific revisions or reset parts of the repository, adding flexibility for error correction.
- **Stash Management:** Allows users to temporarily save work, supporting a more dynamic workflow.
- **History Filtering:** Enables users to filter commit logs based on criteria such as author, date, or message content, enhancing revision management.

Uses Structure Summary

The Uses Structure promotes a modular approach, with each functionality increment offering new capabilities while maintaining stability across interactions with the File System Hiding and Behavior Hiding Modules. This design allows the Repository Hiding Module to expand incrementally without compromising the existing functionality of the DVCS.