Group 2 - SynCode: Demin, Suumil, Helia, Pranay

DVCS Module Specifications

Part A: Module Guide

Module A: Machine Hiding

Designers: Demin, Suumil | **Reviewers:** Suumil, Demin

Role

Machine hiding is categorized to handle the necessary changes in commands used by the other two modules for different machine operating systems (more specifically file systems). Provide a consistent interface for interacting with the underlying operating system and hardware. Abstract file system operations, network communication, environment variable management.

Secret

- *Primary:* This module will cover the difference of file systems in different machines. The primary secret will involve methods that recognize file system differences and keep cross-platform data consistency.
- *Secondary:* Will be listing potential file system conflict and implementation to work around potential conflicts while keeping repository data consistency between targeted systems.

Some examples are listed as follows:

- *Line Endings:* The handling of line endings can vary between operating systems. Unix-based systems (Linux, macOS) use LF (Line Feed) characters, while Windows uses CRLF (Carriage Return and Line Feed) characters.
- *Filesystem Case Sensitivity:* Some operating systems, like Linux and macOS, have case-sensitive file systems, while Windows has a case-insensitive file system.
- *File Path Length:* Windows has a maximum path length limitation (260 characters) that may lead to issues with long file paths.
- *Path Separators:* The character used to separate directory paths in file paths can vary between operating systems. On Unix-based systems, it's "/", while on Windows, it's "".

Notes on Decomposition

Reason(s) The Machine Hiding module is designed to collect and handle cross-platform conflicts in different file system I/O settings. Most common differences are handled by Rust inner design.

If time permits, we are interested in distributed features above basic multi-repository single machine systems.

Rule(s)

- Unexpected file system I/O conflict collecting module will collect all special conflicts encountered in the other two modules that are not solved by Rust.
- File system specific I/O adaptor module will solve the conflicts collected.
- Cross-machine communication authenticator module will manage and verify remote repository communication request.
- Data encryption module will handle data encryption and key to be transferred once the communication is authenticated.
- Cross-machine data flow manager module will handle encrypted data upload and download.

Fuzzy Area(s) The Machine Hiding Module is expected to interact with both Behaviour Hiding and Repository Hiding modules. Changes in machines may make a facility design decision less appropriate or deviate from expectation. This module will evolve with unexpected demands from the other two modules.

Module A.1: File System Operations Module

Designer: Demin | **Reviewer:** Suumil

Role: The file system module will list and handle potential conflicts that do not have a genuine solution in Rust, for different file systems of operating systems.

Secret:

- *Primary:* Unexpected conflicts in different file systems of operating systems. Especially the conflicts that cannot be handled by Rust in the implementation, though Rust I/O module itself is a big module hiding that helps most of the cross-platform issues.
- *Secondary:* The implementation of necessary system-specific I/O commands.

Facilities:

1. **Cross-machine filesystem I/O conflicts collecting module**
    - *Description:* Collects and identifies unique filesystem conflicts that arise due to different operating systems, which are not inherently resolved by Rust.
    - *Method:*

```
fn identify_conflict(file_path: &str) -> Result<FileSystemConflict,
IoError>;
```

    - *Input:*
        - `file_path: &str`: The path of the file where a potential conflict might occur.
    - *Output:*
        - `Result<FileSystemConflict, IoError>`: The type of conflict identified or an error if the process fails.

2. **File system-specific I/O adaptor module**
    - *Description:* Adapts file system operations to handle identified conflicts in a way that ensures consistency across different operating systems.
    - *Method:*

```
fn adapt_io_operations(file_path: &str, conflict: FileSystemConflict)
-> Result<(), IoError>;
```

    - *Input:*
        - `file_path: &str`: The path of the file to be adapted.
        - `conflict: FileSystemConflict`: The specific conflict identified that needs resolution.
    - *Output:*
        - `Result<(), IoError>`: Success if adaptation is successful, or an error if it fails.

Tests:

1. **Conflict Identification Success:**

```
// Test ID: 1
let file_system_module = FileSystemOperationsModule::new();
let file_path = "/path/to/file";
let result = file_system_module.identify_conflict(file_path);
assert_eq!(result.is_ok(), true);
```

2. **Conflict Identification Failure:**

```
// Test ID: 2
let file_system_module = FileSystemOperationsModule::new();
let invalid_file_path = "/invalid/path";
let result = file_system_module.identify_conflict(invalid_file_path);
assert_eq!(result.is_err(), true);
```

3. **I/O Operation Adaptation Success:**

```
// Test ID: 3
let file_system_module = FileSystemOperationsModule::new();
let file_path = "/path/to/file";
let conflict = FileSystemConflict::LineEndings;
let result = file_system_module.adapt_io_operations(file_path,
conflict);
assert_eq!(result.is_ok(), true);
```

4. **I/O Operation Adaptation Failure:**

```
// Test ID: 4
let file_system_module = FileSystemOperationsModule::new();
let file_path = "/path/to/file";
let invalid_conflict = FileSystemConflict::Unknown;
let result = file_system_module.adapt_io_operations(file_path,
invalid_conflict);
assert_eq!(result.is_err(), true);
```

Module A.2: Network Operations Module

Designer: Suumil | **Reviewer:** Demin

Role: Network operations module is a potential bonus feature in our project. It is responsible for cross-machine communication functioning, safety, and authentication.

Secret:

- *Primary:* Cross-machine repository data exchange and the authentication and safety features necessary for data integrity and safety.
- *Secondary:* The function modules that control the data flow among machines and the algorithms that manage authentications, data encryption, and key management.

Facilities:

1. **Cross-machine data flow manager module**
    - *Description:* This functionality manages and controls the flow of data between different machines, ensuring efficient and correct data transfer.
    - *Method:*

```
fn transfer_data(data: &str, destination: &str) -> Result<(),
DataFlowError>;
```

    - *Input:*
        - `data: &str`: The data to be sent to another machine.
        - `destination: &str`: The identifier or address of the destination machine.
    - *Output:*
        - `Result<(), DataFlowError>`: Success or an error if the data flow process encounters issues.

2. **Cross-machine communication authenticator module**
    - *Description:* Handles the authentication of machines involved in the commu- nication, ensuring that the data is exchanged between authenticated sources only.
    - *Method:*

```
fn authenticate(source: &str, auth_data: &str) -> Result<(),
AuthenticationError>;
```

- *Input:*
  - `source`: The identifier of the source machine.
  - `auth_data`: The authentication data or credentials.
- *Output:*
  - `Result<(), AuthenticationError>`: Success or an error related to authentication.

3. **Data encryption module for permitted data exchange**
   - *Description:* Encrypts data using a specified key before it is sent over the network, ensuring data confidentiality and integrity.
   - *Method:*

```
fn encrypt_data(data: &str, encryption_key: &str) -> Result<String,
EncryptionError>;
```

- *Input:*
  - `data`: The unencrypted data that needs to be encrypted.
  - `encryption_key`: The key used for encryption.
- *Output:*
  - `Result<String, EncryptionError>`: The encrypted data or an error if the encryption process fails.

Tests:

1. **Transfer Data Success:**

```
// Test ID: 1
let network_module = NetworkOperationsModule::new();
let data = "example data";
let destination = "10.101.202.21";
let result = network_module.transfer_data(data, destination);
assert_eq!(result.is_ok(), true);
```

2. **Transfer Data Failure:**

```
// Test ID: 2
let network_module = NetworkOperationsModule::new();
let data = "example data";
let unreachable_host = "10.12.12.12";
let result = network_module.transfer_data(data, unreachable_host);
assert_eq!(result.is_err(), true);
```

3. **Authentication Success:**

```
// Test ID: 3
let network_module = NetworkOperationsModule::new();
let source = "10.12.12.12";
let auth_data = "valid_credentials";
let result = network_module.authenticate(source, auth_data);
assert_eq!(result.is_ok(), true);
```

4. **Authentication Failure:**

```
// Test ID: 4
let network_module = NetworkOperationsModule::new();
let source = "10.12.12.12";
let invalid_auth_data = "invalid_credentials";
let result = network_module.authenticate(source, invalid_auth_data);
assert_eq!(result.is_err(), true);
```

5. **Data Encryption Success:**

```
let network_module = NetworkOperationsModule::new();
let data = "sensitive data";
let encryption_key = "sha256";
let result = network_module.encrypt_data(data, encryption_key);
assert!(result.is_ok());
assert_ne!(result.unwrap(), data);
```

6. **Data Encryption Failure:**

```
let network_module = NetworkOperationsModule::new();
let data = "sensitive data";
let invalid_key = "";
let result = network_module.encrypt_data(data, invalid_key);
assert_eq!(result.is_err(), true);
```

Module B: User Hiding

Designers: Helia | **Reviewers:** Demin, Pranay, Suumil

Role

This module is responsible for hiding the DVCS behaviors from users, including error handling, input/output, and user interface management.

Within the architecture of User Hiding, we have decomposed it into four leaf-modules, namely the Command Parser, Syntax Checker & Error Handling, User Interface, and Authentication Module. Each of these sub-modules specializes in distinct modes of user interactions with the DVCS system.

Notes on Decomposition

Reason(s) The User Hiding Module is decomposed into four specialized sub-modules to ensure that it effectively hides DVCS behaviors from users, providing a clean and modular interface. This decomposition allows for a clear focus on different aspects of user interaction, including parsing user commands, handling errors, managing the user interface, and implementing authentication. Specialization in each of these areas helps maintain a separation of concerns, making it easier to manage and extend the user interaction components while ensuring that the DVCS remains user-friendly and secure.

Rule(s)

- The Command Parser Module specializes in parsing user commands and validating command correctness. It is responsible for interpreting user input and translating it into actionable commands for the DVCS.
- The Syntax Checker & Error Handling Module is dedicated to handling errors, including syntax errors and semantic errors, as well as managing exceptions. This module ensures that users receive meaningful feedback when they make mistakes and that errors are properly logged and reported.
- The User Interface Module focuses on managing the interaction between users and the DVCS through a user-friendly interface. It handles user prompts, provides help messages, and ensures a smooth user experience.
- The Authentication Module is responsible for managing user authentication and access control. It ensures that only authorized users can perform certain DVCS operations and provides a secure access mechanism.

Fuzzy Area(s) The Behavior Hiding Module is expected to interact with both Machine Hiding and Repository Hiding Modules in order to handle machine-specific and DVCS-specific operations, which can lead to various fuzzy areas.

- *Machine Hiding:* For example, the Command Parser Module and the Syntax Checking & Error Handling Module might have to work closely with Machine Hiding in order to make sure differences related to different OS or devices get processed correctly.
- *Repository Hiding:* For instance, the User Interface Module will have to interact with the Repository Hiding module constantly in order to provide data/feedback/status back and forth between the user and the DVCS system, as well as between different modules in the DVCS system.

Module B.1: Command Parser Module

Designer: Pranay | **Reviewer:** Demin

Role: The Command Parser Module specializes in parsing user commands and validating command correctness. It is responsible for interpreting user input and translating it into actionable commands for the

DVCS.

Secrets:

- *Primary:* The module primarily hides the internal data structures for the input command, command processing mechanism, and logic behind checking command's correctness.
- *Secondary:* The module also hides the implementation details and low-level algorithms regarding command parsing and correctness checking.

Facilities:

1. **Parsing**
    - *Description:* This functionality reads the user input commands.
    - *Method:*

```
fn parse_command(user_input: &str) -> Result<Command,
ParseError>;
```

    - *Input:*
        - `user_input`: User-provided command as a string.
    - *Output:*
        - `Result<Command, ParseError>`: Parsed command or an error if the parsing process encounters issues.
2. **Validating the Command's Correctness**
    - *Description:* This functionality validates the user command's correctness.
    - *Method:*

```
fn validate_command(command: &Command) -> Result<(),
ValidationError>;
```

    - *Input:*
        - `command`: Parsed command to be validated.
    - *Output:*
        - `Result<(), ValidationError>`: Success or an error if the command validation process encounters issues.

Tests:

1. **Command Parsing Test - Valid Command**

```
// Test ID: 1
let user_input = "git commit -m 'Fix issue #123'";
```

```
let result = command_parser_module.parse_command(user_input);
assert_eq!(result.is_ok(), true);
```

2. **Command Parsing Test - Invalid Command**

```
// Test ID: 2
let user_input = "git invalid-command";
let result = command_parser_module.parse_command(user_input);
assert_eq!(result.is_err(), true);
```

3. **Command Validation Test - Valid Command**

```
// Test ID: 3
let valid_command = create_valid_command();
let result = command_parser_module.validate_command(&valid_command);
assert_eq!(result.is_ok(), true);
```

4. **Command Validation Test - Invalid Command**

```
// Test ID: 4
let invalid_command = create_invalid_command();
let result = command_parser_module.validate_command(&invalid_command);
assert_eq!(result.is_err(), true);
```

5. **Command Parsing and Validation Test - Complex Command**

```
// Test ID: 5
let user_input = "git merge -b feature-branch";
let parsed_command =
command_parser_module.parse_command(user_input).unwrap();
let result = command_parser_module.validate_command(&parsed_command);
assert_eq!(result.is_ok(), true);
```

6. **Command Parsing and Validation Test - Empty Command**

```
// Test ID: 6
let user_input = "";
let result = command_parser_module.parse_command(user_input);
assert_eq!(result.is_err(), true);
```

These tests cover various scenarios, including parsing valid and invalid commands, validating commands, and handling complex or empty commands. The reviewer, Demin, should focus on ensuring that the parsing

and validation methods behave correctly and handle different input scenarios appropriately.

Module B.2: Syntax Checker & Error Handling Module

Designer: Helia | **Reviewer:** Pranay

Role: This module is responsible for handling various types of errors that may occur in user input (besides command, which has been cleared by Module 2.1). These errors could include syntax errors, semantic errors, and exceptions in the file name(s), directory path(s), etc., that the user types next to their command.

Secret:

- *Primary:* The module primarily hides the internal error-checking and handling mechanisms for user input.
- *Secondary:* The module also hides the implementation details and low-level algorithms regarding error checking and handling of user input.

Facilities:

1. **Error Detection and Error Message:**
    - *Description:* This functionality detects all types of errors in the user's input.
    - *Method:*

```
   impl fmt::Display for DvcsError {
       fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
           match self {
               // These are errors related to the DVCS
initialization commands (init, clone)
               DvcsError::InvalidCommand => write!(f, "Invalid
command"),
               DvcsError::InvalidNumberOfArguments => write!(f,
"Invalid number of arguments"),
               DvcsError::InvalidArguments => write!(f, "Invalid
arguments"),
               DvcsError::InvalidPath => write!(f, "Invalid
path"),

               // TODO: add more error types and/or fix the error
messages as needed
               _ => write!(f, "To be implemented"),
           }
       }
   }
```

    - *Input:* `'&self: DvcsError'` - the errors returned by other modules.
    - *Output:*

- The appropriate error messages.

Tests:

1. **Error Detection Test - Invalid Command**

```
// Test ID: 1
let err = DvcsError::InvalidCommand;
assert_eq!(err.to_string(), "Invalid command");
```

2. **Error Detection Test - Invalid Number of Args**

```
// Test ID: 2
let err = DvcsError::InvalidNumberOfArguments;
assert_eq!(err.to_string(), "Invalid number of arguments");
```

3. **Error Detection Test - Invalid Arguments**

```
// Test ID: 3
let err = DvcsError::InvalidArguments;
assert_eq!(err.to_string(), "Invalid arguments");
```

Module B.3: User Interface Module

Designer: Pranay | **Reviewer:** Suumil

Role: The User Interface Module focuses on managing interactions between users and the DVCS through a user-friendly interface. It handles user prompts, provides help messages, and ensures a smooth user experience.

Secrets:

- *Primary:* This module primarily hides the internal data/info and task exchanging mechanism between the DVCS system and the user, as well as between different modules inside the DVCS.
- *Secondary:* This module also hides the implementation details and low-level algorithms regarding intermodule communication and user-DVCS communication.

Facilities:

1. **User Interface**
   - *Description:* This module provides a simple interface for the user and the DVCS system to interact with each other as per the user's (validated/correct) commands. We use clap crate to get a polished CLI user interface experience.
   - *Method:*

```
use clap::{Command, Arg};
```

Tests:

- No need tests for this as we can already see if interface works or not by running other modules' tests.

Module C: Repository Hiding

Designers: Pranay, Denim, Suumil | **Reviewers:** Helia, Shunzhi

Role

The Repository Hiding module is responsible for hiding the intricacies involved in the processes of managing, accessing, and manipulating repositories in the DVCS. This module ensures that the users and other modules interact with the repositories through well-defined interfaces, abstracting the underlying complexities.

Within the architecture of Repository Hiding, we have decomposed it into four leaf modules, namely the Initialization, Staging, Inspection Module and the Synchronization Module. Each of these leaf modules specializes in distinct modes of repository interaction.

Notes on Decomposition

Reason(s) The Repository module is decomposed into specialized modules to ensure a clear focus on different aspects of repository management, aligning with the responsibility hierarchy. This decomposition allows for specialization in specific functionalities, making it easier to manage and extend the system while keeping a clean separation of concerns. It promotes modularity and code reusability by organizing related operations into distinct modules.

Rule(s)

- The Initialization Module is designed to handle repository creation (init) and repository cloning (clone), which are fundamental steps in the DVCS setup process.
- The Inspection Module specializes in providing tools for revision inspection, such as listing heads, comparing revisions (diff), displaying file contents at specific points (cat), and checking out historical states.
- The Synchronization Module focuses on managing synchronization operations between repositories, including merging changes, pulling updates, and pushing commits to remote repositories.
- The Staging Module is dedicated to staging and managing changes, preparing them for commits, including facilities for committing, adding changes, removing staged files, and checking the status of changes in the working directory.

Fuzzy Area(s) It is expected to interact with both Machine Hiding and Behaviour hiding.

- *Machine Hiding:* The module might interact with "Machine Hiding" to ensure that the necessary environmental configurations and resources are available for the inspection facilities to function effectively.

- *User Hiding:* This module might interact with "User Hiding" to get the current status of the repository to help with the heads, and the diff. Complicated workflows like conflict resolution require interfaces and requirements for the same might change to suit user requirements. Further it will also require us to ensure that we have the correct authentication rights and the files and directories for the repository. Thus there would be an intersection with Behavior Hiding.

Module C.1 : Initialization Module

Designer: Helia | **Reviewer:** Shunzhi

Role: The Initialization Module is responsible for initializing new repositories and cloning existing ones within the DVCS system. Its primary role is to provide users with essential tools for setting up and configuring repositories effectively.

Secret:

- *Primary:* The core algorithms and implementation details for repository initialization (init) and cloning (clone). This involves the low-level processes required to create and configure a new repository or clone an existing one, including handling the creation of repository data structures, managing user authentication, and access control.
- *Secondary:* Optimization techniques to enhance the performance of initialization and cloning processes, including strategies for efficient data caching and handling edge cases.

Facilities:

1. **Init:**
    - *Description:* This functionality creates a new repository with capabilities to handle cross-file-system differences. It can be used to convert an existing, unversioned project to a repository or initialize a new, empty repository.
    - *Method:*

      ```
      fn init(repository_name: &str) -> Result<(), InitError>;
      ```

    - *Input:* `'repository_name: &str'` - The user-defined name of the new repository
    - *Output:*
        - `Ok()`: The new repository is created and nothing has to be returned.
        - `Err(InitError)`: Initialization failed.
2. **Clone:**
    - *Description:* This functionality obtains a copy of a repository, covering cross-platform conflict handling. Internally, Clone would call Init to first initialize an empty repository and then clone the existing repository.
    - *Method:*

```
fn clone(url: &str) -> Result<(), CloneError>;
```

- *Input:* `'url: &str'` - the URL of the repository to be cloned.
- *Output:*
  - `Ok()`: The cloned repository is created and nothing has to be returned.
  - `Err(CloneError)`: Clone failed.

Tests:

1. **Successful initialization Test**

```
//Test ID: 1
let repository_name = "test_repo_1";
assert!(init_module::init(repository_name).is_ok());
```

2. **Failed Initialization - Duplicate repository name**

```
//Test ID: 2
let repository_name = "test_repo_1";
assert!(init_module::init(repository_name).is_ok());

let existing_repo_name = "test_repo_1";
assert!(init_module::init(existing_repo_name).is_err());
```

3. **Failed Initialization - Invalid repository name**

```
// Test ID: 3
let invalid_repo_name = "-392';.3d";
assert!(init_module::init(invalid_repo_name).is_err());
```

4. **Successful Cloning**

```
// Test ID: 4
let repo_url = "https://example.com/repo.dvcs";
assert!(init_module::clone(repo_url).is_ok());
```

5. **Failed Cloning - Invalid URL**

```
// Test ID: 5
let invalid_url = "invalid_url";
```

```
let result = init_module::clone(invalid_url);
assert_eq!(result, "Invalid URL Error found! Suggestion: XYZ.");
```

Module C.2 : Inspection Module

Designer: Demin | **Reviewer:** Helia

Role: It is responsible for providing users with tools to inspect and analyze revisions within the DVCS system. Its primary role is to facilitate the examination of various aspects of revision history.

Secret:

- *Primary:* The core algorithms, data structures, or implementation details that are essential for the proper functioning of the facilities provided by the "Revision Inspection Module." The primary secrets involve the algorithms used to calculate and display differences between revisions for the "Diff" facility, or the data structures used to manage and access revisions.
- *Secondary:* Involve optimization techniques for improving the performance of revision inspection, data caching strategies, or handling edge cases in the revision history.

Facilities:

1. **Heads:**
    - *Description:* This functionality allows users to view the current head, often representing different branches, and is essential for tracking the active development paths.
    - *Method:*

```
fn heads(repo_path: &str) -> enum HeadInfo {Branch(String),
Error(String)};
```

    - *Input:*
        - `repo_path`: target repository to check which branch or commit is currently active
    - *Output:*
        - `HeadInfo`: return the information about the current head, panic if not possible.
2. **Diff:**
    - *Description:* This functionality enables users to check the differences or changes between revisions. This helps understand the modifications that have been made to the codebase over time, aiding in the review process.
    - *Method:*

```
fn diff(file_path: &str, old_rev: &str, new_rev: &str) ->
Result<String, String> {};
```

- *Input:*
  - Target file_path to compare old_rev and new_rev
- *Output:*
  - Different content, otherwise error message.

3. **Inspect:**

- *Description:* This functionality provides the capability to inspect a specific file within a given revision. Helps users examine the contents of a file at a particular point in the code's history.
- *Method:*

```
fn cat(repo_path: &str, revision: &str, file_path: &str) ->
Result<String, String> {};
```

- *Input:*
  - file_path in repo_path to inspect with revision
- *Output:*
  - It opens the repository, resolves the commit ID from the revision string, and then retrieves the content of the specified file at that commit.

4. **Checkout**

- *Description:* This functionality allows users to check out a specific revision using its hash code. This is essential for reverting to a previous state of the codebase or for creating a new branch from a specific point in the revision history.
- *Method:*

```
fn checkout(repo_path: &str, revision_hash: &str) -> Result<(),
String> {};
```

- *Input:*
  - target `repo_path` to access file version with revision_hash
- *Output:*
  - specific file version, otherwise panic

Tests:

1. **Head Success**

```
// Test ID: 1
let goodrepo = SuccessRepository;
let result = heads(&goodrepo);
match result {
        HeadInfo::Error(error_message) => assert_eq!(branch,
"head_branch")=>panic("Expected HeadInfo::Branch, got something
else.")}
```

2. **Head Failure**

```
// Test ID: 2
let failingrepo = FailingRepository;
let result = heads(&failingrepo);
match result {
    HeadInfo::Error(error_message) => assert_eq!(
            error_message, "Failed to open repository"), _ => panic!
("Expected
            HeadInfo::Error, got something else."), }
```

3. **Diff Success**

```
// Test ID: 3
let result = diff("HEAD^", "HEAD", Some("sample.txt"));
match result { Ok(diff_content) => assert!
(diff_content.contains("Initial content")), Err(error) => panic!
("Unexpected error: {}", error), }
```

4. **Diff Success - Same**

```
// Test ID: 4
let result = diff("HEAD", "HEAD", Some("sample.txt"));
match result { Ok(diff_content) => assert!
(diff_content.contains("Initial content")), Err(error) => panic!
("Unexpected error: {}", error), }
```

5. **Inspect Existing File**

```
// Test ID: 5
let repo_path = goodRepo;
let result = cat(repo_path, "HEAD", "sample.txt");
match result { Ok(content) => assert_eq!(content, "Initial
content\n"), Err(error) => panic!("Unexpected error:{}", error), }
```

6. **Inspect Non-existing File**

```
// Test ID: 6
let repo_path = goodRepo;
let result = cat(repo_path, "HEAD", "non_exist.txt");
match result { Ok(content) => assert_eq!(content, "Initial
content\n"), Err(error) => panic!("Unexpected error:{}", error), }
```

7. **Checkout Valid Hash**

```
// Test ID: 7
let repo_path = goodRepo;
valid_hash = "0123456789abcdef0123456789abcdef01234567";
let result = checkout(repo_path, valid_hash);
assert!(result.is_ok());
```

8. **Checkout Invalid Hash**

```
// Test ID: 8
let repo_path = goodRepo;
invalid_hash = "0123456789";
let result = checkout(repo_path, invalid_hash);
assert!(result.is_ok());
```

Module C.3: Synchronization Module

Designer: Pranay | **Reviewer:** Helia

Role: The Synchronization Module is responsible for executing and managing updates within the DVCS. It focuses on applying commits, merging various versions, and connecting with remote repositories to maintain an updated and consistent code base across all platforms.

Secrets:

- *Primary:* The processes involved in committing changes, merging versions, and synchronizing with remote repositories are abstracted. This module encapsulates algorithms and logic that execute these tasks.
- *Secondary:* Conceals methodologies and interface implementations concerning complicated operations like conflict resolution, maintaining code integrity after a pull or push, etc.

Facilities:

1. **Merge**
    - *Description:* Combines two different code revisions, resolving any conflicts to establish a unified code base.
    - *Method:*

```
fn merge(revision_a: &Revision, revision_b: &Revision) ->
Result<UnifiedRevision, MergeError>;
```

    - *Input:*

        - `revision_a`: The first code revision.
        - `revision_b`: The second code revision.
    - *Output:*
        - `Result<UnifiedRevision, MergeError>`: A unified code revision or an error if the merge process encounters issues.

2. **Pull**

    - *Description:* Retrieves and integrates updates from a remote repository, ensuring local repositories are current.
    - *Method:*

```
fn pull(remote_repository: &RemoteRepository) -> Result<(),
PullError>;
```

    - *Input:*
        - `remote_repository`: The remote repository to pull updates from.
    - *Output:*
        - `Result<(), PullError>`: Success or an error if the pull process encounters issues.

3. **Push**

    - *Description:* Uploads local repository changes to a remote repository, ensuring it is updated with the latest revisions.
    - *Method:*

```
fn push(local_repository: &LocalRepository, remote_repository:
&RemoteRepository) -> Result<(), PushError>;
```

    - *Input:*
        - `local_repository`: The local repository with changes to push.
        - `remote_repository`: The remote repository to push changes to.
    - *Output:*
        - `Result<(), PushError>`: Success or an error if the push process encounters issues.

Tests:

1. **Merge Test**

```
// Test ID: 1
let revision_a = create_revision("Code Revision A");
let revision_b = create_revision("Code Revision B");
let result = synchronization_module.merge(&revision_a, &revision_b);
assert_eq!(result.is_ok(), true);
```

2. **Pull Test**

```
// Test ID: 2
let remote_repo = create_remote_repo("https://example.com/repo.git");
let result = synchronization_module.pull(&remote_repo);
assert_eq!(result.is_ok(), true);
```

3. **Push Test**

```
// Test ID: 3
let local_repo = create_local_repo("path/to/local/repo");
let remote_repo = create_remote_repo("https://example.com/repo.git");
let result = synchronization_module.push(&local_repo, &remote_repo);
assert_eq!(result.is_ok(), true);
```

4. **Merge Conflict Test**

```
// Test ID: 4
let revision_a = create_revision("Code Revision A");
let conflicting_revision_b = create_revision_with_conflict("Code
Revision B");
let result = synchronization_module.merge(&revision_a,
&conflicting_revision_b);
assert_eq!(result.is_err(), true);
```

5. **Pull Error Test**

```
// Test ID: 5
let invalid_remote_repo =
create_remote_repo("https://invalid.example.com/repo.git");
let result = synchronization_module.pull(&invalid_remote_repo);
assert_eq!(result.is_err(), true);
```

6. **Push Error Test**

```
// Test ID: 6
let invalid_remote_repo =
create_remote_repo("https://invalid.example.com/repo.git");
let local_repo = create_local_repo("path/to/local/repo");
let result = synchronization_module.push(&local_repo,
&invalid_remote_repo);
assert_eq!(result.is_err(), true);
```

The Synchronization Module's methods and tests are designed to ensure efficient management of synchronization operations while handling various scenarios, including merges, pulls, pushes, conflicts, and error cases. The reviewer, Helia, should focus on validating the correctness and effectiveness of these functionalities.

Module C.4: Staging Module

Designer: Suumil | **Reviewer:** Shunzhi

Role: The Staging Module is responsible for the management and update of the user's current revision. It maintains a list of all tracked/untracked files in the user's current revision, allowing the user to add a specific file they want to store/track to the list and remove a particular file they no longer want to store/track from the list. Additionally, it enables the user to check the current status of the tracking list to see what has been tracked or untracked. The module is also involved in committing changes.

Secrets:

- *Primary:* The Staging Module conceals the processes involved in committing changes and hides the internal storage and data structures, updating mechanisms, and file tracking. This module encapsulates algorithms and logic that execute these tasks.
- *Secondary:* The module conceals methodologies and interface implementations concerning operations like opening an editor for commit and low-level algorithms regarding updating and maintaining the tracking list.

Facilities:

1. **Commit:**
    - *Description:* Records changes made to the code, creating an updated revision in the repository.
    - *Method:*

```
fn commit(message: &str) -> Result<(), CommitError>;
```

    - *Input:*
        - `message`: A message describing the changes made in the commit.
    - *Output:*
        - `Result<(), CommitError>`: Success or an error if the commit pro- cess encounters issues.

2. **Add/Remove**
    - *Description:* Allows a user to add/remove a specific local file to/from the tracking list.
    - *Method:*

```
fn add(file_path: &str) -> Result<(), TrackingError>;
fn remove(file_path: &str) -> Result<(), TrackingError>;
```

- *Input:*
  - `file_path`: The path of the file to add/remove to/from the tracking list
- *Output:*
  - `Result<(), TrackingError>`: Success or an error if the add/remove process encounters issues.

3. **Status**

- *Description:* Displays the content of the present tracking list of the current revision/repository to the user as requested, containing all the IDs of the tracked and untracked files.
- *Method:*

```
fn status() -> Vec<FileStatus>;
```

- *Input:*
  - None
- *Output:*
  - `Vec<FileStatus>`: A list containing the status of all tracked and untracked files in the current revision/repository.

Tests:

1. **Commit Success**

```
// Test ID: 1
let staging_module = StagingModule::new();
let message = "Commit message";
let result = staging_module.commit(message); assert_eq!
(result.is_ok(), true);
```

2. **Commit Failure**

```
// Test ID: 2
let staging_module = StagingModule::new();
let message = "Invalid message";
let result = staging_module.commit(message); assert_eq!
(result.is_err(), true);
```

3. **Add Success**

```
// Test ID: 3
let staging_module = StagingModule::new();
let file_path = "path/to/file.txt";
```

```
let result = staging_module.add(file_path); assert_eq!(result.is_ok(),
true);
```

4. **Add Failure**

```
// Test ID: 4
let staging_module = StagingModule::new();
let invalid_file_path = "non_existent_file.txt";
let result = staging_module.add(invalid_file_path);
assert_eq!(result.is_err(), true);
```

5. **Remove Success**

```
// Test ID: 5
let staging_module = StagingModule::new();
let file_path = "path/to/file.txt";
let result = staging_module.remove(file_path); assert_eq!
(result.is_ok(), true);
```

6. **Remove Failure**

```
// Test ID: 6
let staging_module = StagingModule::new();
let invalid_file_path = "non_existent_file.txt";
let result = staging_module.remove(invalid_file_path);
assert_eq!(result.is_err(), true);
```

7. **Status**

```
// Test ID: 7
let staging_module = StagingModule::new();
let status = staging_module.status();
assert_eq!(status.len(), 3);
```

Part B

---

1. Uses Relation

Module A: Machine Hiding

A.1: File System Operations Module

Uses:

- Possibly interacts with the Network Operations Module (A.2) for handling file operations that involve network communication.

A.2: Network Operations Module

Uses:

- File System Operations Module (A.1) for any file system-related operations during network communication.

Module B: User Hiding

B.1: Command Parser Module

Uses:

- May use error handling utilities from the Syntax Checker & Error Handling Module (B.2) for parsing errors.
- Interacts with User Interface Module (B.3) to receive input and display parsing results.

B.2: Syntax Checker & Error Handling Module

Uses:

- Command Parser Module (B.1) to check the syntax of parsed commands.
- User Interface Module (B.3) to display error messages.

B.3: User Interface Module

Uses:

- Command Parser Module (B.1) and Syntax Checker & Error Handling Module (B.2) to get command inputs and handle errors.
- Authentication Manager Module (B.4) for displaying authentication-related prompts or messages.
- Repository Hiding Module (B) to execute DVCS functionalities as requested by the user.

Module C: Repository Hiding

C.1: Initialization Module

Uses:

- Possibly interacts with Machine Hiding modules (Module A) for file system operations during initialization and cloning.
- Might use User Interface Module (B.3) for displaying initialization or cloning results.

C.2: Inspection Module

Uses:

- Depends on the Initialization Module (C.1) for accessing repository data.
- May use the User Interface Module (B.3) for outputting inspection results.

C.3: Synchronization Module

Uses:

- Network Operations Module (A.2) for pull and push operations involving remote repositories.
- Staging Module (C.4) for preparing and applying changes during merge operations.

C.4: Staging Module

Uses:

- Interacts with the User Interface Module (B.3) for user inputs and status updates.
- Might depend on the Initialization Module (C.1) for repository-specific operations.

2. Minimal Prototype System

The prototype focuses on basic functionalities that form the foundation of the DVCS. These include the following:

Module A (Machine Hiding):

- **File System Operations Module (A.1):** Basic operations like handling cross-platform file system conflicts.
- **Network Operations Module (A.2):** Essential network operations for communication between machines.

Module B (User Hiding):

- **Command Parser Module (B.1):** Basic command parsing for user inputs.
- **User Interface Module (B.3):** Minimal UI for user interaction.

Module C (Repository Hiding):

- **Initialization Module (C.1):** Basic repository initialization and cloning functionalities.

These core functionalities ensure that the system can initialize, manage basic user commands, and handle machine-specific operations.

3. Functionality Increment 1

The first increment adds more sophisticated features for handling errors, user authentication, and initial repository operations:

Module B (User Hiding):

- **Syntax Checker & Error Handling Module (B.2):** Enhanced error detection and handling.

Module C (Repository Hiding):

- **Inspection Module (C.2):** Basic functionalities for inspecting and analyzing revisions.

4. Functionality Increment 2

The second increment introduces advanced synchronization and staging features, along with enhanced machine-specific functionalities:

Module A (Machine Hiding):

- **Enhancements to the Network Operations Module (A.2):** Advanced features for cross-machine data flow management and encryption.

Module B (User Hiding):

- **User Interface Module (B.3):** Complex UI for merge conflicts.

Module C (Repository Hiding):

- **Synchronization Module (C.3):** Advanced synchronization operations like merge, pull, and push.
- **Staging Module (C.4):** Managing changes and updates in user's current revision.

5. Subsequent Increments

Further increments will focus on expanding the capabilities of the existing modules, enhancing user experience, security,providing more robust error handling and network operation features.