

RAPPORT DU PROJET MINI SHELL

HATIM FATIMA-EZZAHRA
EL FERGOUCH YASSINE
EL WAQOUDI AMINA
GHANMOUNI YASSINE



TABLE DE MATIERES

Introduction	2-3
Conception	4-7
Implémentation	8-21
Tests et validation.....	22-25
Conclusion.....	26

Objectif du projet:

L'objectif principal de ce projet est de développer un mini shell, un interpréteur de commandes simplifié permettant aux utilisateurs d'interagir avec un système d'exploitation fictif ou limité. Ce mini shell vise à reproduire l'expérience d'utilisation d'un shell dans un environnement contrôlé, en s'inspirant des principes de conception des shells sous Linux. Il doit être capable de parser et d'exécuter un ensemble de commandes internes, gérer les erreurs et maintenir un historique des commandes pour permettre aux utilisateurs de naviguer et réexécuter facilement les commandes précédentes. En outre, des fonctionnalités avancées telles que la redirection, le pipelining et l'exécution de scripts de commandes seront également implémentées, offrant ainsi une simulation plus réaliste et complète des fonctionnalités des shells Unix.

Contexte:

En tant que participants à ce projet de développement d'un mini shell, nous explorons les bases des systèmes d'exploitation en créant une interface de commande simplifiée. Inspiré par des environnements comme bash, zsh et ksh sous Linux, ce mini shell nous offre l'opportunité unique de maîtriser les mécanismes essentiels de l'interprétation de commandes.

Ce projet nous permet d'approfondir notre expertise en programmation système en abordant la gestion des processus, la manipulation des fichiers et des répertoires, ainsi que des concepts avancés tels que la redirection et le pipelining. En résolvant des défis techniques, nous améliorons notre capacité à concevoir des solutions robustes et évolutives, tout en perfectionnant nos compétences en gestion d'erreurs.

La réalisation de ce mini shell constituera une étape clé dans notre apprentissage des environnements de développement Linux, nous dotant des compétences nécessaires pour nos futures carrières en informatique et développement logiciel. Ce projet est une occasion passionnante de consolider notre compréhension des systèmes d'exploitation et de cultiver notre intérêt pour les technologies fondamentales de l'informatique.

Description des fonctionnalités principales du projet

Le projet vise à développer un mini shell, un interpréteur de commandes simplifié inspiré des shells Linux, permettant aux utilisateurs d'interagir avec un environnement simulé à travers diverses commandes. Voici les fonctionnalités principales attendues :

1. Parsing des commandes : Le mini shell analysera la ligne de commande entrée par l'utilisateur pour identifier les commandes et leurs arguments.
2. Exécution des commandes internes : Il prendra en charge l'exécution des commandes internes telles que `cd`, `ls` et `mkdir`, permettant ainsi la gestion des répertoires et fichiers.
3. Gestion de l'historique des commandes : Le shell maintiendra un historique des commandes entrées par l'utilisateur, facilitant ainsi la navigation et la réexécution des commandes précédentes.

Ces fonctionnalités de base fourniront une expérience utilisateur semblable à celle d'un shell traditionnel, tout en permettant une immersion dans les principes fondamentaux de la programmation système et de l'interaction utilisateur-système.

Description des fonctionnalités avancées du projet

En plus des fonctionnalités de base, le mini shell inclura des capacités avancées pour enrichir l'expérience utilisateur et simuler des fonctionnalités cruciales des shells Unix :

1. Redirection et Pipelining : Permettra aux utilisateurs de rediriger la sortie d'une commande vers un fichier (`>`, `>>`) ou d'enchaîner plusieurs commandes (`|`), facilitant ainsi la manipulation et le traitement avancé des données.
2. Scripts de commandes : Supportera l'exécution de scripts de commandes simples, permettant aux utilisateurs de combiner plusieurs commandes en un seul script exécutable, augmentant ainsi la flexibilité et l'automatisation des tâches.

Ces fonctionnalités avancées étendront les capacités du mini shell au-delà des interactions de base pour offrir une expérience plus riche et fonctionnelle, tout en reflétant les principes et les normes de conception des shells Linux modernes.

Architecture du Mini Shell

Pour développer un mini shell complet et fonctionnel, voici une architecture détaillée qui couvre chaque composant essentiel nécessaire à son implémentation. Cette architecture prend en compte à la fois les fonctionnalités de base et avancées du mini shell inspiré des shells Unix.

1. ****Interface Utilisateur (UI)****

- ****SimpleShellSwing.java : **** Gère l'interface graphique du mini shell en utilisant Swing.

Elle inclut :

- ****JTextArea outputArea : **** Pour afficher la sortie des commandes.
- ****JTextField commandField : **** Pour saisir les commandes.
- ****JScrollPane scrollPane : **** Pour faire défiler la sortie.
- ****JButton executeButton : **** Pour exécuter les commandes.

2. ****Gestion de l'Histoire des Commandes****

- ****CommandHistory.java : **** Gère l'historique des commandes entrées par l'utilisateur, permettant de naviguer dans l'historique et de réexécuter des commandes précédentes.

3. ****Parsing des Commandes****

- ****CommandParser.java : **** Classe dédiée au parsing des commandes entrées par l'utilisateur. Elle analyse la ligne de commande et extrait les différentes parties (commandes et arguments) pour une exécution correcte.

4. ****Exécution des Commandes****

- ****CommandExecutor.java : **** Classe responsable de l'exécution des commandes. Elle gère :
 - Les commandes internes comme ``cd``, ``ls``, ``mkdir``, ``history``, etc.
 - Les commandes système et externes en utilisant ``ProcessBuilder``.

5. ****Gestion du Répertoire Actuel****

- ****CurrentDirectory.java : **** Gère le répertoire courant et fournit des méthodes pour changer de répertoire, vérifier les permissions, etc.

6. ****Gestion des Fichiers et Répertoires****

- ****FileSystemManager.java : **** Classe utilitaire pour la gestion des fichiers et répertoires, incluant :
 - Méthodes pour lister les fichiers (`ls`), créer des répertoires (`mkdir`), supprimer des fichiers (`rm`), etc.

7. ****Redirection et Pipelining****

- ****CommandRedirection.java : **** Gère la redirection de la sortie d'une commande vers un fichier (`>`, `>>`) ou l'entrée à partir d'un fichier (`<`).
- ****CommandPipelining.java : **** Gère l'enchaînement de plusieurs commandes en utilisant des pipes (`|`).

8. ****Scripts de Commandes****

- ****ScriptExecutor.java : **** Permet l'exécution de scripts de commandes simples (`./script.sh`), en traitant chaque ligne du script comme une commande distincte.

9. ****Utilitaires****

- ****FileUtilities.java : **** Contient des méthodes utilitaires pour manipuler les fichiers, comme lire un fichier, écrire dans un fichier, vérifier l'existence d'un fichier, etc.
- ****StringUtilities.java : **** Méthodes utilitaires pour manipuler les chaînes de caractères, comme découper une chaîne en morceaux, supprimer les espaces blancs, etc.

10. ****Gestion des Erreurs****

- ****ErrorHandler.java : **** Gère les exceptions et les erreurs, affiche des messages d'erreur détaillés à l'utilisateur en cas d'échec de l'exécution d'une commande.

11. ****Documentation et Explications****

- ****DocumentTechnique.pdf : **** Document détaillant l'architecture du programme, les choix d'implémentation, les algorithmes utilisés, les décisions de conception, etc.
- ****GuideUtilisation.pdf : **** Guide d'utilisation expliquant comment utiliser le mini shell, ses fonctionnalités, exemples d'utilisation, etc.

12. ****Compilation et Exécution****

- ****Makefile : **** Fichier Makefile pour automatiser la compilation du programme et gérer les dépendances.

Cette architecture détaillée divise le mini shell en plusieurs composants modulaires, chacun responsable d'une fonctionnalité spécifique. Elle permet une séparation claire des préoccupations, facilite la gestion et la maintenance du code, et offre une expérience utilisateur riche et interactive. Chaque composant est conçu pour répondre aux spécifications du projet, en fournissant une implémentation robuste et flexible des fonctionnalités attendues d'un shell moderne.

Choix de langage Java :

Le choix de Java comme langage de programmation pour ce projet de développement d'un mini shell s'appuie sur plusieurs raisons clés. Java est un langage orienté objet, ce qui permet une organisation claire et modulaire du code, facilitant ainsi la gestion et l'extension des fonctionnalités du mini shell. De plus, Java dispose de bibliothèques robustes pour la gestion des processus, la manipulation des fichiers et la création d'interfaces graphiques, comme Swing, qui est utilisé dans notre projet pour offrir une interface utilisateur intuitive. La portabilité de Java, grâce à la machine virtuelle Java (JVM), garantit que notre mini shell peut fonctionner sur divers systèmes d'exploitation sans modification majeure du code. Enfin, Java bénéficie d'une vaste communauté de développeurs et d'une abondante documentation, ce qui est un atout précieux pour résoudre les problèmes et implémenter des fonctionnalités avancées telles que la redirection et le pipelining. Ces caractéristiques font de Java un choix idéal pour développer un mini shell éducatif et fonctionnel, tout en assurant une expérience d'apprentissage riche pour les participants.

1. Parsing des commandes :

Techniques utilisées pour l'analyse syntaxique :

L'analyse syntaxique (ou parsing) des commandes est une étape cruciale dans le fonctionnement de notre mini shell. Pour décomposer les commandes saisies par l'utilisateur en composants compréhensibles par le programme, nous utilisons une combinaison de techniques simples mais efficaces.

1. Séparation par espaces :

La méthode de base consiste à diviser la ligne de commande en utilisant les espaces comme délimiteurs. Cela nous permet d'extraire le nom de la commande et ses arguments.

```
String[] str_arr = command_line.split(" "); // parsing
```

2. Gestion des guillemets :

Pour permettre aux utilisateurs de saisir des arguments contenant des espaces, nous devons gérer les guillemets. Nous utilisons des expressions régulières pour détecter et préserver les arguments entourés de guillemets.

```
Pattern pattern = Pattern.compile(Pattern.quote(word), Pattern.CASE_INSENSITIVE);  
Matcher matcher = pattern.matcher(line);  
StringBuffer sb = new StringBuffer();  
while (matcher.find()) {  
    matcher.appendReplacement(sb, "<" + matcher.group() + ">");  
}  
matcher.appendTail(sb);  
return sb.toString();
```

3. Traitement des caractères spéciaux :

Les caractères spéciaux tels que les symboles de redirection (>, >>, <) et le symbole de pipeline (|) nécessitent un traitement particulier pour découper correctement la commande et identifier les opérations spéciales.

```
} else if (line.contains("|")) {  
    handlePipelining(line);  
} else if (line.contains(">") || line.contains(">>") || line.contains("<")) {  
    handleRedirection(line);
```

Structure des commandes et arguments :

Pour chaque commande, nous définissons une structure qui comprend :

- Le nom de la commande : C'est le premier élément de la ligne de commande après avoir été découpée par espaces.
- Les arguments : Ce sont les éléments suivants dans la ligne de commande. Ils peuvent inclure des chemins de fichiers, des options et des paramètres supplémentaires.
- Les opérations spéciales : Comme la redirection ou le pipelining, détectées par la présence des symboles `>`, `>>`, `<`, ou `|`.

Exemple de commande et sa structure :

- Commande : `ls -l /home/user`
 - Nom de la commande : `ls`
 - Arguments : `-l`, `/home/user`
- Commande avec redirection : `cat file.txt > output.txt`
 - Nom de la commande : `cat`
 - Arguments : `file.txt`
 - Redirection : `>` vers `output.txt`
- Commande avec pipelining : `cat file.txt | grep "search term"`
 - Première commande : `cat file.txt`
 - Nom de la commande : `cat`
 - Arguments : `file.txt`
 - Deuxième commande : `grep "search term"`
 - Nom de la commande : `grep`
 - Arguments : `"search term"`

Implémentation de la méthode `prepare_command`

Voici comment nous implémentons la méthode `prepare_command` dans le code :

```
private String prepare_command(String command_line, ArrayList<String> str_list) {  
    String[] str_arr = command_line.split(" "); // parsing  
    str_list.add("cmd");  
    str_list.add("/c");  
    boolean quotation = false;  
    for (String str : str_arr) {  
        if (quotation) {  
            int last_index = str_list.size() - 1;  
            str_list.set(last_index, str_list.get(last_index) + " " + str);  
        } else {  
            str_list.add(str);  
        }  
        if (str.startsWith("\""))  
            quotation = true;  
        if (str.endsWith("\""))  
            quotation = false;  
    }  
    return str_list.get(2);  
}
```

Cette méthode permet de décomposer la ligne de commande en parties utilisables tout en gérant correctement les guillemets.

Gestion des commandes complexes :

Pour les commandes contenant des redirections ou des pipelines, nous utilisons des techniques similaires mais plus sophistiquées pour séparer les différentes parties et traiter chaque segment individuellement avant de les réassembler dans le flux d'exécution.

En résumé, le parsing des commandes dans notre mini shell est conçu pour être robuste et flexible, capable de gérer une variété de scénarios courants rencontrés dans l'utilisation des shells Unix.

2. Exécution des commandes :

Notre mini shell implémente plusieurs commandes internes pour la gestion des répertoires et des fichiers. Voici une description des principales commandes :

1.cd (Change Directory) : Cette commande permet de changer le répertoire courant. Si le chemin spécifié est "~" ou "\$HOME", le répertoire est changé vers le répertoire home de l'utilisateur.

```

} else if (main_operation.equals("cd")) {
    if (command_parts.size() < 4) {
        outputArea.append("invalid cd command! hint: you need to provide new directory\n");
        return;
    }

    String new_path = command_parts.get(3).replaceAll("^\\|\\$", "");
    if (new_path.equals("~") || new_path.equals("$HOME")) {
        new_path = home_path;
    }

    Path p = Paths.get(new_path);
    File new_dir;
    if (p.isAbsolute())
        new_dir = new File(new_path);
    else
        new_dir = new File(dir, new_path);

    if (!new_dir.exists() || !new_dir.isDirectory()) {
        outputArea.append("Error: invalid directory requested\n");
    } else {
        dir = new_dir;
        current_path = dir.getAbsolutePath();
    }
}

```

2.ls (List Directory) : Cette commande liste les fichiers et les répertoires du répertoire courant.

```

} else if (main_operation.equals("ls")) {
    listDirectory(dir);

    private void listDirectory(File dir) {
        File[] files = dir.listFiles();
        if (files != null) {
            for (File file : files) {
                outputArea.append(file.getName() + "\n");
            }
        }
    }
}

```

3.cat (Concatenate and Display Files) : Cette commande affiche le contenu d'un fichier spécifié.

```

} else if (main_operation.equals("cat")) {
    if (command_parts.size() < 4) {
        outputArea.append("invalid cat command! hint: you need to provide a file name\n");
        return;
    }

    String file_name = command_parts.get(3);
    File file = new File(dir, file_name);

    if (!file.exists() || !file.isFile()) {
        outputArea.append("Error: file does not exist\n");
    } else {
        BufferedReader reader = new BufferedReader(new FileReader(file));
        String line_content;
        while ((line_content = reader.readLine()) != null) {
            outputArea.append(line_content + "\n");
        }
        reader.close();
    }
}

```

4.touch (Create Empty File) : Cette commande crée un nouveau fichier vide dans le répertoire courant.

```

} else if (main_operation.equals("touch")) {
    if (command_parts.size() < 4) {
        outputArea.append("invalid touch command! hint: you need to provide a file name\n");
        return;
    }

    String file_name = command_parts.get(3);
    File file = new File(dir, file_name);
    if (file.exists()) {
        outputArea.append("Error: file already exists\n");
    } else {
        file.createNewFile();
        outputArea.append("File " + file_name + " created successfully\n");
    }
}

```

5.rm (Remove File) : Cette commande supprime un fichier spécifié.

```

} else if (main_operation.equals("rm")) {
    if (command_parts.size() < 4) {
        outputArea.append("invalid rm command! hint: you need to provide a file name\n");
        return;
    }

    String file_name = command_parts.get(3);
    File file = new File(dir, file_name);

    if (!file.exists()) {
        outputArea.append("Error: file does not exist\n");
    } else {
        if (file.delete()) {
            outputArea.append("File " + file_name + " deleted successfully\n");
        } else {
            outputArea.append("Error: failed to delete file\n");
        }
    }
}

```

6.grep (Search for Patterns) : Cette commande recherche un mot spécifié dans un fichier donné et affiche les lignes contenant ce mot.

```

} else if (main_operation.equals("grep")) {
    if (command_parts.size() < 5) {
        outputArea.append("invalid grep command! hint: you need to provide a word and a file name\n");
        return;
    }

    String word = command_parts.get(3);
    String file_name = command_parts.get(4);
    File file = new File(dir, file_name);

    if (!file.exists() || !file.isFile()) {
        outputArea.append("Error: file does not exist\n");
    } else {
        BufferedReader reader = new BufferedReader(new FileReader(file));
        String line_content;
        while ((line_content = reader.readLine()) != null) {
            outputArea.append(highlightWord(line_content, word) + "\n");
        }
        reader.close();
    }
}

```

3. Historique des commandes :

Stockage et récupération des commandes précédentes :

L'historique des commandes est géré à l'aide d'une classe CommandHistory. Chaque commande entrée par l'utilisateur est ajoutée à une liste, qui conserve l'ordre chronologique des commandes. Lorsque l'utilisateur entre une nouvelle commande, celle-ci est automatiquement stockée dans cette liste.

Voici un extrait de code montrant comment les commandes sont stockées :

```
private class CommandHistory {  
    private List<String> history = new ArrayList<>();  
    private int history_index = -1;  
  
    public void add_command(String command) {  
        history.add(command);  
        history_index = history.size();  
    }  
  
    public String get_history() {  
        StringBuilder sb = new StringBuilder();  
        for (int i = 0; i < history.size(); i++) {  
            sb.append(i + 1).append(": ").append(history.get(i)).append("\n");  
        }  
        return sb.toString();  
    }  
}
```


Navigation dans l'historique :

La navigation dans l'historique des commandes permet à l'utilisateur de rappeler facilement les commandes précédentes à l'aide des flèches "haut" et "bas" du clavier. Lorsqu'une flèche est pressée, l'index de l'historique est ajusté pour récupérer la commande précédente ou suivante.

Voici un extrait de code montrant comment cette navigation est implémentée :

```
commandField.addKeyListener(new KeyAdapter() {
    @Override
    public void keyPressed(KeyEvent e) {
        if (e.getKeyCode() == KeyEvent.VK_UP) {
            String previousCommand = ch.get_previous_command();
            if (previousCommand != null) {
                commandField.setText(previousCommand);
            }
        } else if (e.getKeyCode() == KeyEvent.VK_DOWN) {
            String nextCommand = ch.get_next_command();
            if (nextCommand != null) {
                commandField.setText(nextCommand);
            }
        }
    }
});
```

```
public String get_previous_command() {
    if (history_index > 0) {
        history_index--;
        return history.get(history_index);
    }
    return null;
}

public String get_next_command() {
    if (history_index < history.size() - 1) {
        history_index++;
        return history.get(history_index);
    }
    return null;
}
```

4. Fonctionnalités avancées :

Redirection :

La redirection permet de rediriger la sortie d'une commande vers un fichier plutôt que vers l'écran.

Voici comment la gestion de la redirection est implémentée dans le code :

```
private void handleRedirection(String line) throws IOException {
    String[] parts;
    boolean append = false;
    if (line.contains(">>")) {
        parts = line.split(">>");
        append = true;
    } else if (line.contains(">")) {
        parts = line.split(">");
    } else if (line.contains("<")) {
        parts = line.split("<");
    } else {
        return;
    }

    String commandPart = parts[0].trim();
    String filePart = parts[1].trim();

    if (line.contains(">") || line.contains(">>")) {
        executeCommandWithOutputRedirection(commandPart, filePart, append);
    } else if (line.contains("<")) {
        executeCommandWithInputRedirection(commandPart, filePart);
    }
}
```

```

private void executeCommandWithOutputRedirection(String commandPart, String filePart, boolean append) throws IOException {
    File outputFile = new File(dir, filePart);
    ArrayList<String> command_parts = new ArrayList<>();
    String main_operation = prepare_command(commandPart, command_parts);

    if (main_operation.equals("cat") && command_parts.size() > 3) {
        String inputFileName = command_parts.get(3);
        File inputFile = new File(dir, inputFileName);

        if (!inputFile.exists() || !inputFile.isFile()) {
            outputArea.append("Error: input file does not exist\n");
            return;
        }

        BufferedReader reader = new BufferedReader(new FileReader(inputFile));
        BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile, append));
        String line;
        while ((line = reader.readLine()) != null) {
            writer.write(line);
            writer.newLine();
        }
        reader.close();
        writer.close();
    } else {
        ProcessBuilder pb = new ProcessBuilder(command_parts);
        pb.directory(dir);
        Process process = pb.start();

        BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));
        BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile, append));
        String line;
        while ((line = reader.readLine()) != null) {
            writer.write(line);
            writer.newLine();
        }
        reader.close();
        writer.close();
    }
}

private void executeCommandWithInputRedirection(String commandPart, String filePart) throws IOException {
    File inputFile = new File(dir, filePart);
    if (!inputFile.exists() || !inputFile.isFile()) {
        outputArea.append("Error: input file does not exist\n");
        return;
    }

    BufferedReader reader = new BufferedReader(new FileReader(inputFile));
    ArrayList<String> command_parts = new ArrayList<>();
    String main_operation = prepare_command(commandPart, command_parts);

    ProcessBuilder pb = new ProcessBuilder(command_parts);
    pb.directory(dir);
    Process process = pb.start();

    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(process.getOutputStream()));
    String line;
    while ((line = reader.readLine()) != null) {
        writer.write(line);
        writer.newLine();
    }
    reader.close();
    writer.close();

    InputStream is = process.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);
    while ((line = br.readLine()) != null) {
        outputArea.append(line + "\n");
    }
    br.close();
}

```

Pipelining :

Le pipelining permet de chaîner plusieurs commandes ensemble, où la sortie d'une commande est utilisée comme entrée pour la suivante.

Voici comment la gestion du pipelining est implémentée :

```
private void handlePipelining(String line) throws IOException {
    String[] commands = line.split("\\|");
    if (commands.length != 2) {
        outputArea.append("Error: Only simple pipelining with one pipe is supported.\n");
        return;
    }

    ArrayList<String> command1_parts = new ArrayList<>();
    prepare_command(commands[0].trim(), command1_parts);
    ArrayList<String> command2_parts = new ArrayList<>();
    prepare_command(commands[1].trim(), command2_parts);

    if (!command2_parts.get(2).equals("grep")) {
        outputArea.append("Error: Only 'grep' is supported as the second command in a pipeline.\n");
        return;
    }

    String word = command2_parts.get(3);

    if (command1_parts.get(2).equals("cat") && command1_parts.size() > 3) {
        String file_name = command1_parts.get(3);
        File file = new File(dir, file_name);

        if (!file.exists() || !file.isFile()) {
            outputArea.append("Error: input file does not exist\n");
            return;
        }

        BufferedReader reader = new BufferedReader(new FileReader(file));
        String line_content;
        while ((line_content = reader.readLine()) != null) {
            if (line_content.contains(word)) {
                outputArea.append(highlightWord(line_content, word) + "\n");
            }
        }
        reader.close();
    } else {
        outputArea.append("Error: Only 'cat' command is supported as the first command in a pipeline.\n");
    }
}
```

Exécution de scripts de commandes :

L'exécution de scripts de commandes permet de traiter des fichiers de script contenant une série de commandes.

Voici comment cette fonctionnalité est implémentée :

```
private void executeScriptFile(String fileName) throws IOException {
    File scriptFile = new File(dir, fileName);
    if (!scriptFile.exists() || !scriptFile.isFile() || !fileName.endsWith(".sh")) {
        outputArea.append("Error: Script file does not exist or is not a valid .sh file\n");
        return;
    }

    BufferedReader reader = new BufferedReader(new FileReader(scriptFile));
    String line;
    while ((line = reader.readLine()) != null) {
        processCommand(line);
    }
    reader.close();
}
```

5. Interface Graphique :

L'interface graphique de notre mini shell est implémentée en utilisant Swing, une bibliothèque d'interface utilisateur pour Java. Elle comprend une fenêtre principale qui affiche la sortie des commandes et permet à l'utilisateur de saisir des commandes à exécuter.

Voici comment elle est configurée :

```
public class SimpleShellSwing extends JFrame {
    /**
     *
     */
    private static final long serialVersionUID = 1L;
    private CommandHistory ch = new CommandHistory();
    private String home_path = System.getProperty("user.home");
    private String current_path = System.getProperty("user.dir");
    private File dir = new File(current_path);
    private JTextArea outputArea;
    private JTextField commandField;

    public SimpleShellSwing() {
        setTitle("Simple Shell");
        setSize(600, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

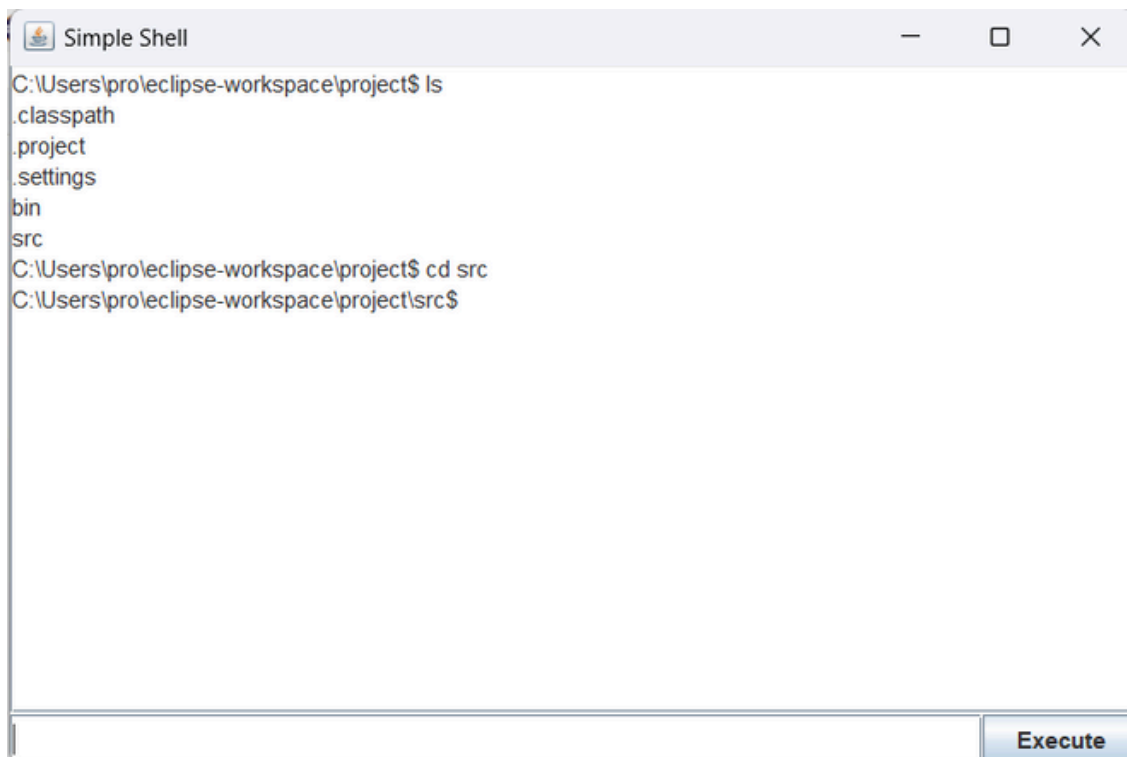
        outputArea = new JTextArea();
        outputArea.setEditable(false);
        JScrollPane scrollPane = new JScrollPane(outputArea);

        commandField = new JTextField();

        commandField.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                String line = commandField.getText();
                if (!line.equals("")) {
                    processCommand(line);
                    commandField.setText("");
                }
            }
        });

        commandField.addKeyListener(new KeyAdapter() {
            @Override
            public void keyPressed(KeyEvent e) {
                if (e.getKeyCode() == KeyEvent.VK_UP) {
```

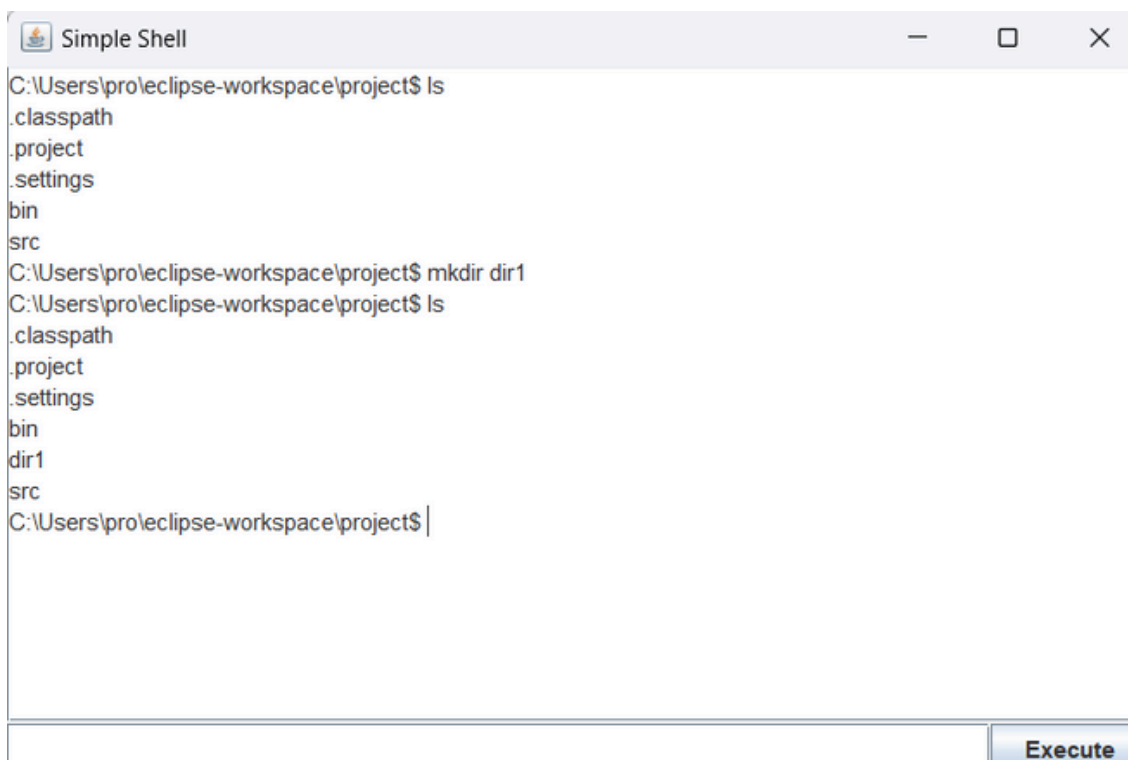
ls -cd :



```
Simple Shell
C:\Users\pro\clipse-workspace\project$ ls
.classpath
.project
.settings
bin
src
C:\Users\pro\clipse-workspace\project$ cd src
C:\Users\pro\clipse-workspace\project\src$
```

Execute

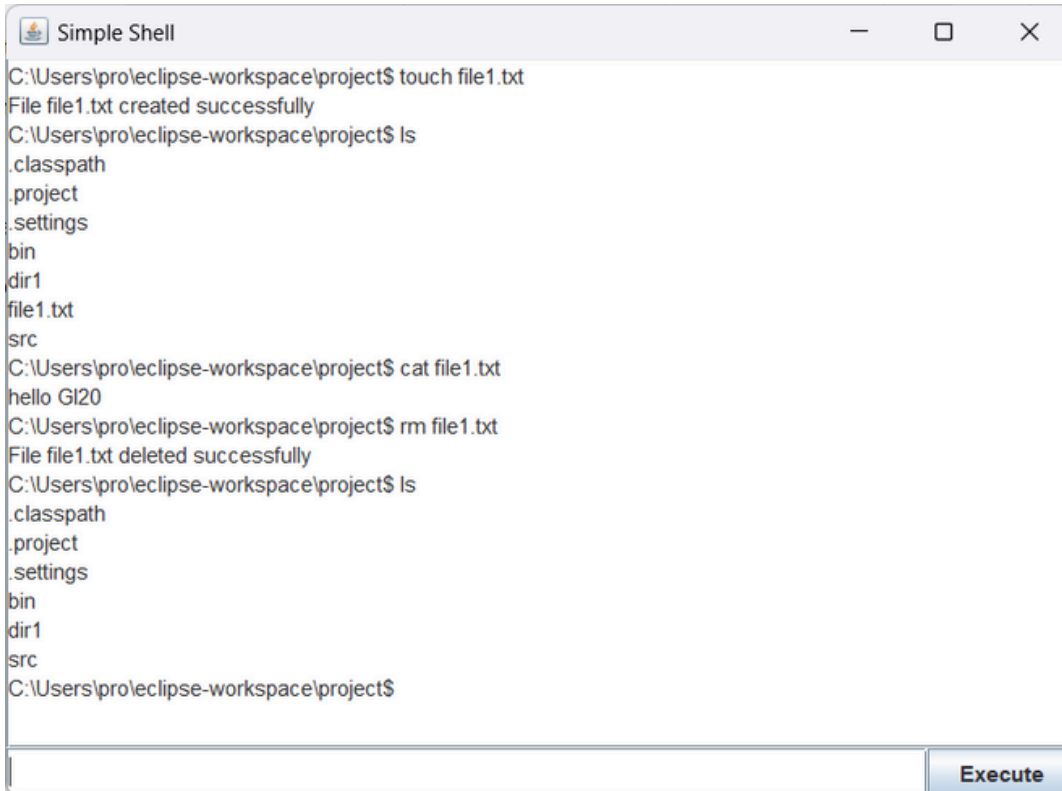
mkdir :



```
Simple Shell
C:\Users\pro\clipse-workspace\project$ ls
.classpath
.project
.settings
bin
src
C:\Users\pro\clipse-workspace\project$ mkdir dir1
C:\Users\pro\clipse-workspace\project$ ls
.classpath
.project
.settings
bin
dir1
src
C:\Users\pro\clipse-workspace\project$ |
```

Execute

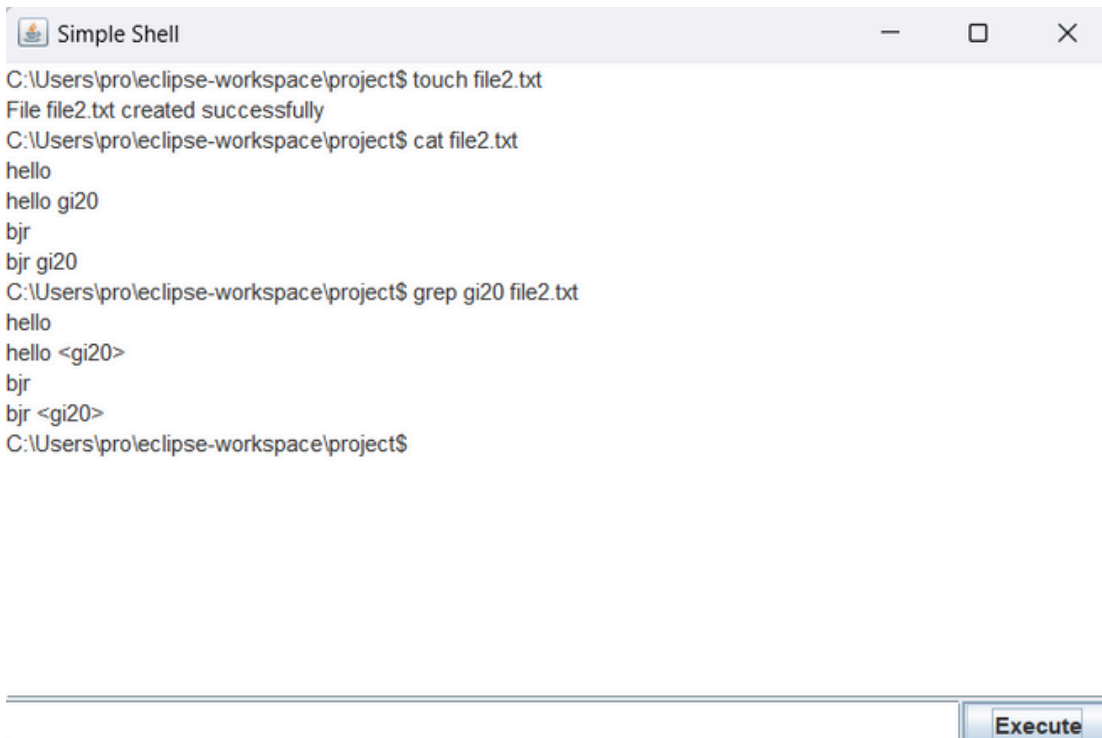
touch-cat-rm :



```
Simple Shell
C:\Users\proleclipse-workspace\project$ touch file1.txt
File file1.txt created successfully
C:\Users\proleclipse-workspace\project$ ls
.classpath
.project
.settings
bin
dir1
file1.txt
src
C:\Users\proleclipse-workspace\project$ cat file1.txt
hello GI20
C:\Users\proleclipse-workspace\project$ rm file1.txt
File file1.txt deleted successfully
C:\Users\proleclipse-workspace\project$ ls
.classpath
.project
.settings
bin
dir1
src
C:\Users\proleclipse-workspace\project$
```

Execute

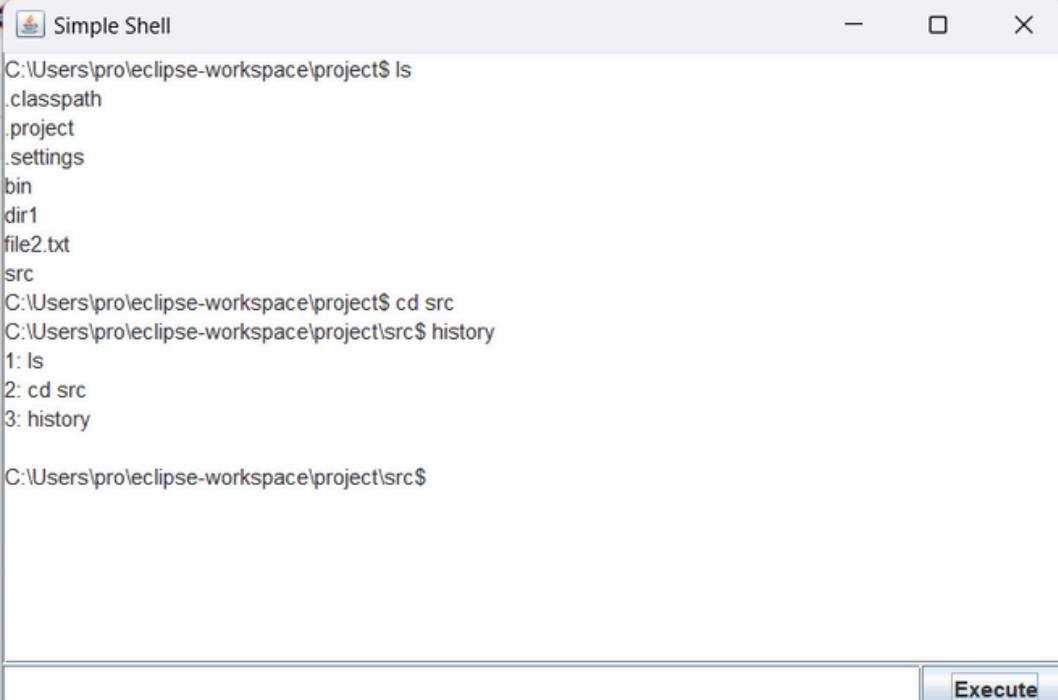
grep :



```
Simple Shell
C:\Users\proleclipse-workspace\project$ touch file2.txt
File file2.txt created successfully
C:\Users\proleclipse-workspace\project$ cat file2.txt
hello
hello gi20
bjr
bjr gi20
C:\Users\proleclipse-workspace\project$ grep gi20 file2.txt
hello
hello <gi20>
bjr
bjr <gi20>
C:\Users\proleclipse-workspace\project$
```

Execute

history :

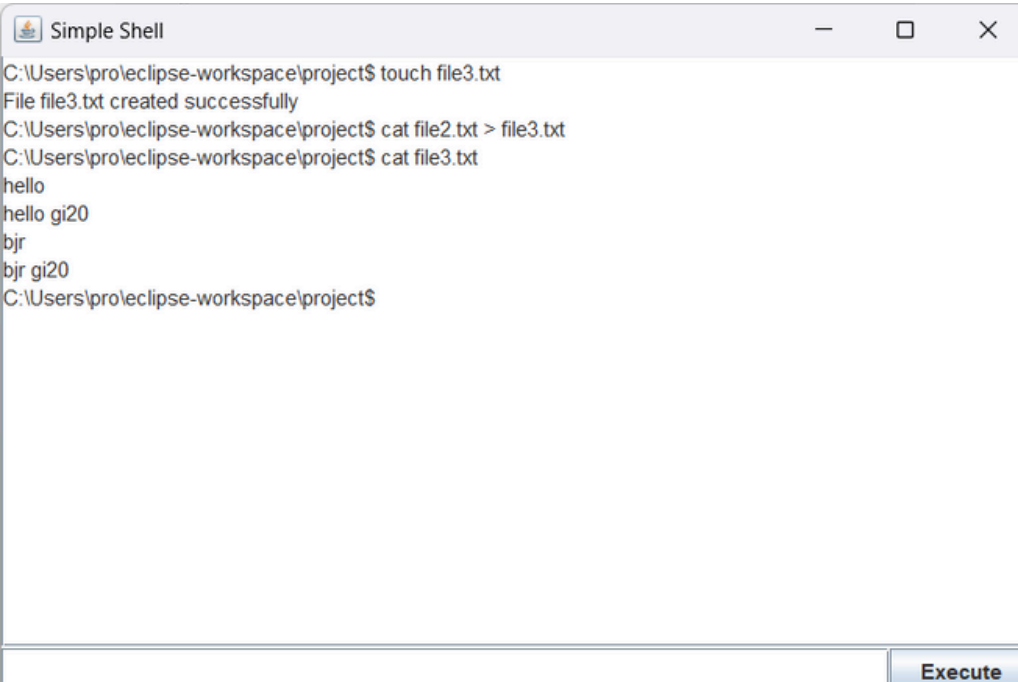


```
C:\Users\pro\workspace\project$ ls
.classpath
.project
.settings
bin
dir1
file2.txt
src
C:\Users\pro\workspace\project$ cd src
C:\Users\pro\workspace\project\src$ history
1: ls
2: cd src
3: history

C:\Users\pro\workspace\project\src$
```

Execute

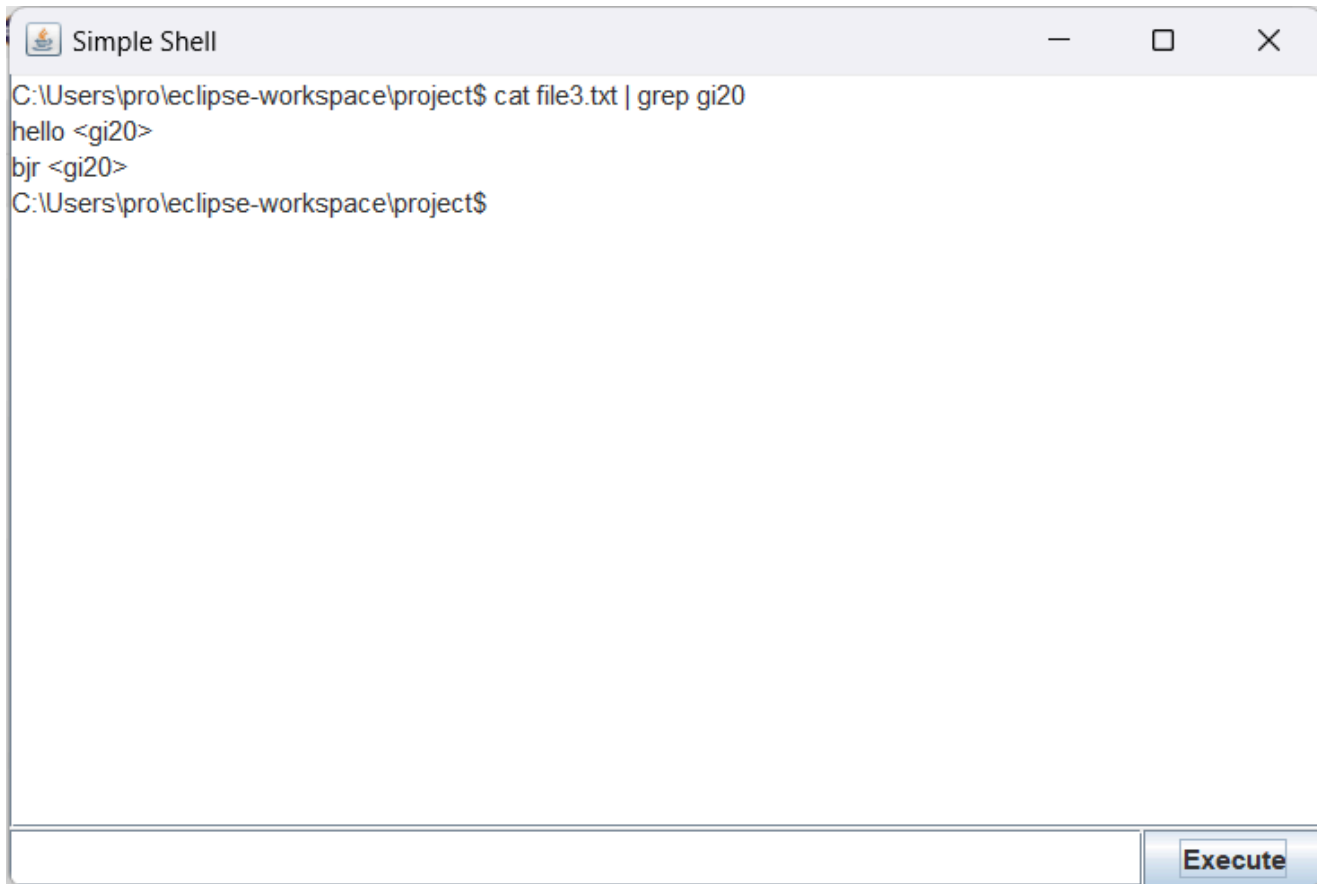
redirection :



```
C:\Users\pro\workspace\project$ touch file3.txt
File file3.txt created successfully
C:\Users\pro\workspace\project$ cat file2.txt > file3.txt
C:\Users\pro\workspace\project$ cat file3.txt
hello
hello gi20
bjr
bjr gi20
C:\Users\pro\workspace\project$
```

Execute

pipelining :

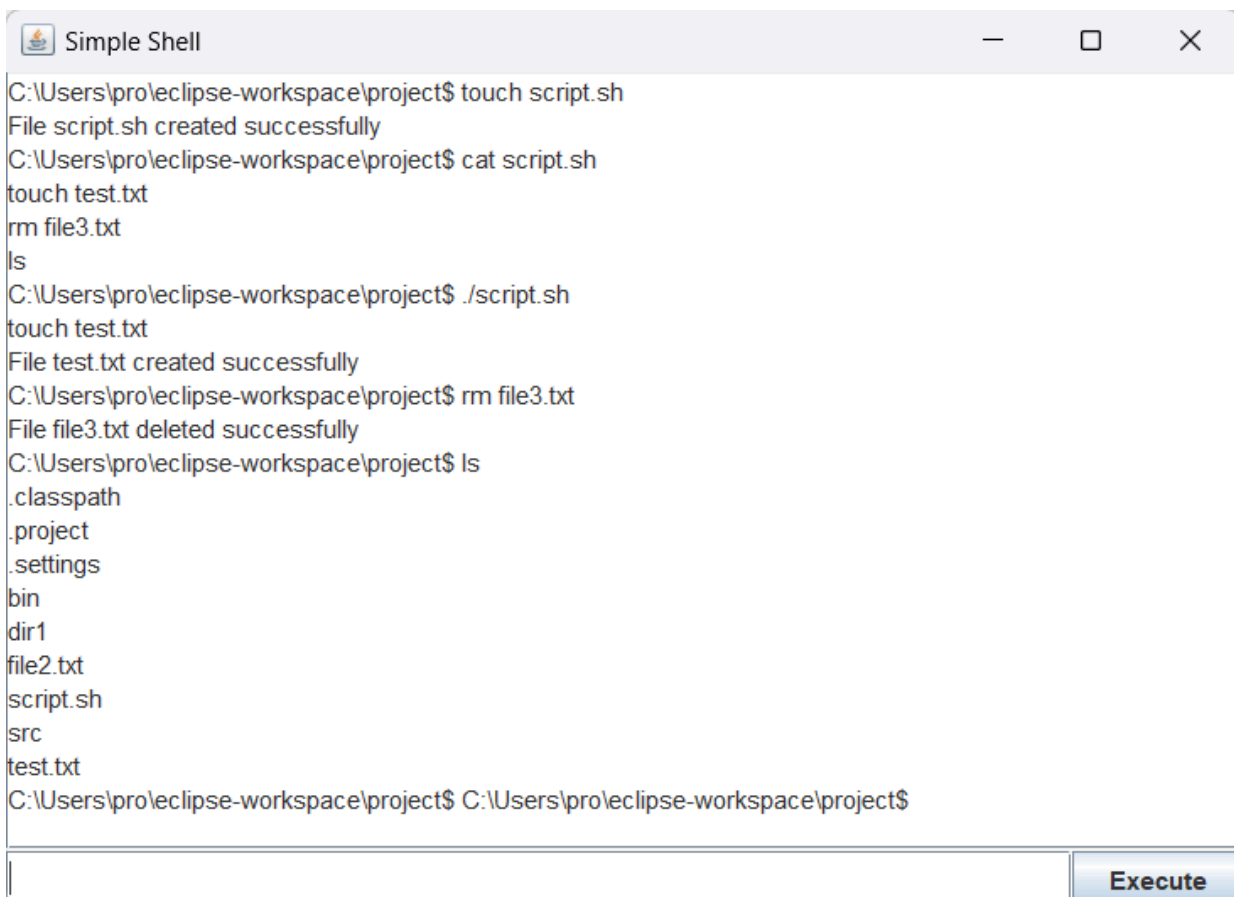


A screenshot of a 'Simple Shell' window. The title bar shows a small icon and the text 'Simple Shell'. The window contains a command prompt with the following text:

```
C:\Users\pro\workspace\project$ cat file3.txt | grep gi20
hello <gi20>
bjr <gi20>
C:\Users\pro\workspace\project$
```

At the bottom right of the window is a button labeled 'Execute'.

exécution de scripts :



A screenshot of a 'Simple Shell' window showing a series of commands and their outputs:

```
C:\Users\pro\workspace\project$ touch script.sh
File script.sh created successfully
C:\Users\pro\workspace\project$ cat script.sh
touch test.txt
rm file3.txt
ls
C:\Users\pro\workspace\project$ ./script.sh
touch test.txt
File test.txt created successfully
C:\Users\pro\workspace\project$ rm file3.txt
File file3.txt deleted successfully
C:\Users\pro\workspace\project$ ls
.classpath
.project
.settings
bin
dir1
file2.txt
script.sh
src
test.txt
C:\Users\pro\workspace\project$ C:\Users\pro\workspace\project$
```

At the bottom right of the window is a button labeled 'Execute'.

CONCLUSION

Ce projet de développement d'un mini shell a été une immersion enrichissante dans les fondements des systèmes d'exploitation et de la programmation système. Inspiré par des shells renommés comme bash et zsh sous Linux, notre mini shell non seulement émule les fonctionnalités de base telles que la gestion des processus et la manipulation des fichiers, mais va au-delà en implémentant des fonctionnalités avancées comme la redirection et le pipelining. À travers la mise en œuvre de commandes internes telles que ``cd``, ``ls``, ``mkdir``, ``rm``, ``cat``, ``grep``, ``touch``, ainsi que la prise en charge de l'exécution de scripts, nous avons consolidé nos compétences en programmation et en gestion des flux de données.

Nous avons également mis l'accent sur la robustesse et la fiabilité de notre mini shell, en menant des tests approfondis pour valider ses fonctionnalités et assurer une expérience utilisateur fluide. En intégrant les retours des utilisateurs et en itérant sur notre code, nous avons créé une interface de commande performante et intuitive. Ce projet a non seulement renforcé notre compréhension des environnements de développement Linux, mais il a également nourri notre passion pour l'innovation technologique et préparé le terrain pour des défis futurs dans le domaine de l'informatique et du développement logiciel.