MBM UNIVERSITY

JODHPUR

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

WINSHELL

A Custom Shell Implementation with CLI & GUI in .NET C#

A Project Report

Submitted in partial fulfillment of the requirements

for the award of degree of

Bachelor of Engineering

in

Computer Science & Engineering

Submitted by:

AASHITA BHANDARI (23UCSE4055)

HARSH RAJANI (23UCSE4013)

AARYAN CHOUDHARY (23UCSE4002)

OCTOBER 2025

CERTIFICATE

This is to certify that the project report entitled **"WinShell - A Custom Shell Implementation with CLI & GUI in .NET C#"** submitted by **Aashita Bhandari (23UCSE4055)**, **Harsh Rajani (23UCSE4013)**, and **Aaryan Choudhary (23UCSE4002)** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science & Engineering** at MBM University, Jodhpur is a record of bonafide work carried out by them under my supervision and guidance.

The matter embodied in this project report has not been submitted by them for the award of any other degree or diploma.

Project Supervisor:

Name: _____

Designation: _____

Signature: _____

Date: _____

Head of Department:

Name: _____

Designation: HOD, CSE Department

Signature: _____

Date: _____

DECLARATION

We hereby declare that the project work entitled **"WinShell - A Custom Shell Implementation with CLI & GUI in .NET C#"** submitted to the Department of Computer Science & Engineering, MBM University, Jodhpur, is a record of an original work done by us under the guidance of our project supervisor and has not formed the basis for the award of any degree/diploma or similar title to any candidate of any university.

AASHITA BHANDARI
Roll No: 23UCSE4055

Signature: _____

Date: _____

HARSH RAJANI
Roll No: 23UCSE4013

Signature: _____

Date: _____

AARYAN CHOUDHARY
Roll No: 23UCSE4002

Signature: _____

Date: _____

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to all those who contributed to the successful completion of this project. First and foremost, we are deeply thankful to our project supervisor for their invaluable guidance, continuous support, and constructive feedback throughout the development of WinShell.

We extend our heartfelt thanks to the Head of the Department of Computer Science & Engineering, MBM University, Jodhpur, for providing us with the necessary facilities and resources to carry out this project work.

We are also grateful to all the faculty members of the Computer Science & Engineering department for their encouragement and academic support during our undergraduate program. Their teachings and insights have been instrumental in building the foundation of knowledge required for this project.

We acknowledge the open-source community, particularly the developers of .NET, WPF, and various libraries that made this project possible. The extensive documentation and community support were invaluable resources during the development process.

Special thanks to our families and friends for their unwavering support, patience, and encouragement throughout this journey. Their belief in our abilities motivated us to overcome challenges and complete this project successfully.

Finally, we are thankful to our peers and classmates who provided feedback and suggestions during various stages of the project development.

Aashita Bhandari
Harsh Rajani
Aaryan Choudhary

ABSTRACT

WinShell is a comprehensive shell implementation developed in .NET C# that provides both Command-Line Interface (CLI) and Graphical User Interface (GUI) capabilities. This project aims to create a modern, cross-platform shell environment that combines the power and flexibility of traditional command-line interfaces with the accessibility and user-friendliness of graphical interfaces.

The shell implements core functionality including command parsing, execution, pipeline support, I/O redirection, job control, and command history management. The architecture follows object-oriented design principles with a modular structure consisting of three main components: Winshell.Core (shared functionality), Winshell.CLI (terminal interface), and Winshell.GUI (graphical interface).

Key features include support for both built-in commands (cd, pwd, help, jobs, etc.) and external command execution, advanced pipeline processing using the pipe operator (|), background job management with fg/bg commands, persistent command history with SQLite database integration, and cross-platform compatibility through .NET runtime.

The GUI component, built using Windows Presentation Foundation (WPF), provides a modern terminal experience with multiple tab support, real-time command execution, visual job management panel, customizable themes and appearance, and integrated help system.

This project demonstrates practical implementation of operating system concepts including process management, inter-process communication, file system operations, and system call interfaces. The modular architecture allows for easy extension and customization, making WinShell suitable for both educational purposes and practical use as a development tool.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION TO THE PROBLEM

1.1 Project Overview

WinShell is a modern shell implementation designed to bridge the gap between traditional command-line interfaces and contemporary graphical user interfaces. Developed using .NET C#, this project represents a comprehensive approach to creating a flexible, extensible, and user-friendly shell environment that can serve both as a learning tool for understanding operating system concepts and as a practical utility for daily development tasks.

The shell environment has been a fundamental component of operating systems since the early days of computing. It serves as the primary interface through which users interact with the operating system, execute programs, manage files, and automate tasks. While traditional shells like Bash, PowerShell, and Zsh have proven their worth over decades of use, there remains room for innovation in terms of user experience, cross-platform compatibility, and modern development practices.

WinShell addresses several key challenges in modern shell development. First, it provides a unified architecture that supports both CLI and GUI modes, allowing users to choose their preferred interaction method based on their specific needs and preferences. Second, it implements core shell functionality using object-oriented design principles, making the codebase maintainable, testable, and extensible. Third, it leverages the .NET ecosystem to achieve cross-platform compatibility while maintaining high performance.

The project architecture consists of three main components working in harmony. The **Winshell.Core** library contains all the essential shell functionality, including command parsing, execution engine, job management, and I/O handling. The **Winshell.CLI** application provides a traditional terminal interface for users who prefer working in a console environment. The **Winshell.GUI** application offers a modern graphical interface built with WPF, featuring multiple tabs, visual job management, and an intuitive user experience.

1.1.1 Command Processing Architecture

The command processing pipeline in WinShell implements a sophisticated tokenization and parsing system. When a user enters a command, the input string passes through multiple stages of processing:

1. **Tokenization:** The input string is broken down into individual tokens
2. **Parsing:** Tokens are analyzed to construct a command tree
3. **Execution Planning:** The execution engine determines the optimal strategy
4. **Execution:** Commands are executed either as built-in functions or external processes
5. **Result Processing:** Output is captured, formatted, and displayed

1.2 Motivation

The motivation for developing WinShell stems from several observations about the current state of shell environments and the needs of modern developers:

Educational Value: Understanding how shells work internally is crucial for computer science students and developers. WinShell provides a clean, well-documented implementation of shell concepts that can serve as a learning resource.

Cross-Platform Development Needs: Modern development often requires working across multiple operating systems. WinShell, built on .NET, runs natively on Windows, Linux, and macOS without requiring virtual machines or compatibility layers.

GUI Integration: While command-line interfaces are powerful, they can be intimidating for new users. WinShell's GUI component makes shell functionality accessible to a broader audience.

Modern Development Practices: WinShell leverages contemporary development methodologies including test-driven development, dependency injection, and asynchronous programming.

1.3 Need for the Project

The need for WinShell arises from several gaps in existing shell solutions:

1.3.1 Learning Platform

Computer science curricula often include operating systems courses that cover process management and I/O. However, students rarely see a complete, modern implementation of these concepts. WinShell provides:

- Clear separation of concerns with well-defined interfaces
- Comprehensive documentation explaining design decisions
- Example implementations of core OS concepts
- Test suite demonstrating verification techniques

1.3.2 Development Tool

Developers need reliable tools that work consistently across platforms. WinShell addresses this by:

- Providing consistent behavior across Windows, Linux, and macOS
- Supporting modern development workflows
- Offering both CLI and GUI modes for different use cases
- Maintaining command history and session management

1.4 Scope of the Project

The scope of WinShell encompasses several key areas:

1.4.1 Core Functionality

- Command parsing and execution engine
- Built-in commands for file system navigation and process management
- External command execution with full argument passing
- Pipeline support with proper stream handling
- I/O redirection for input, output, and error streams
- Job control with foreground/background process management
- Command history with persistent storage

1.4.2 User Interfaces

- CLI mode with ANSI color support
- GUI mode with tabbed interface
- Customizable themes and appearance options
- Keyboard shortcuts for common operations
- Context-sensitive help system

1.5 Current Status

As of October 2025, WinShell has reached version 1.0 with the following implementation status:

Table 1.1: Implementation Status of Core Components

Component	Status	Notes
Core Engine	100%	Fully implemented and tested
Command Parser	100%	Supports complex syntax
Built-in Commands	100%	12 commands implemented
Job Manager	100%	Full job control support
History System	100%	SQLite-based persistence
CLI Interface	100%	Cross-platform support
GUI Interface	95%	Minor enhancements pending
Pipeline Support	100%	Efficient stream processing
I/O Redirection	100%	File and stream redirection

1.6 User Characteristics

WinShell is designed to serve multiple user groups with different needs:

- **Beginner Users:** Those with limited command-line experience who prefer visual interfaces
- **Intermediate Developers:** Familiar with basic shell concepts, use terminals daily
- **Advanced Users:** Deep understanding of OS concepts, automation requirements
- **Students:** Learning OS concepts, need practical examples
- **Educators:** Teaching shell design and system programming

1.7 Report Organization

The remainder of this report is organized as follows:

Chapter 2 explores the evolution of shell environments and technical foundations of WinShell.

Chapter 3 provides a comprehensive comparison with existing shell implementations.

Chapter 4 examines practical applications and presents an honest assessment of advantages and limitations.

Chapter 5 details the architectural design with UML diagrams and formal requirements.

Chapter 6 describes the implementation, technology stack, and build instructions.

Chapter 7 summarizes achievements and outlines future development directions.

Chapter 2

HISTORY, EVOLUTION & TECHNICAL DETAILS

2.1 Historical Context of Shell Development

The concept of a command shell has evolved significantly since the earliest days of computing. Understanding this evolution provides context for the design decisions made in WinShell.

2.1.1 Early Command Interpreters (1960s-1970s)

The first command interpreters appeared in the 1960s as simple programs that allowed users to interact with mainframe computers. The Compatible Time-Sharing System (CTSS), developed at MIT in 1961, featured one of the earliest command-line interfaces.

The Multics operating system introduced several innovations that would influence shell design for decades, including command abbreviation, command history, and primitive scripting capabilities.

2.1.2 The Unix Shell Revolution (1970s)

Ken Thompson developed the first Unix shell in 1971. While simple, it established fundamental concepts:

- The pipe operator (|) for connecting commands
- I/O redirection using < and > operators
- Pattern matching for file operations
- Background process execution with &

Stephen Bourne's Bourne shell (1977) added programming language constructs including loops, conditionals, and functions.

2.1.3 Modern Shells (1980s-Present)

The 1980s and 1990s saw significant innovation:

Bash (1989): Brian Fox created Bash as a free software replacement for the Bourne shell. It became the default for Linux distributions.

Zsh (1990): Designed with extensive customization options and powerful completion systems.

PowerShell (2006): Microsoft's PowerShell represented a paradigm shift by treating commands as objects rather than text streams.

2.2 Technical Foundations

2.2.1 Process Management

At its core, a shell is a process manager. When a user enters a command, the shell must:

1. Parse the command line to identify the program and arguments
2. Create a new process using system calls
3. Set up the process environment
4. Configure I/O streams
5. Wait for completion or manage background execution
6. Collect exit status and cleanup resources

WinShell implements this through the ProcessExecutor class, which abstracts platform-specific mechanisms.

2.2.2 Command Parsing

The first stage of command processing is tokenization. The Tokenizer class breaks input into:

- **Words:** Command names and arguments
- **Operators:** Pipes, redirections, background markers
- **Quotes:** Single, double, and backticks
- **Special characters:** Variables, wildcards, escapes

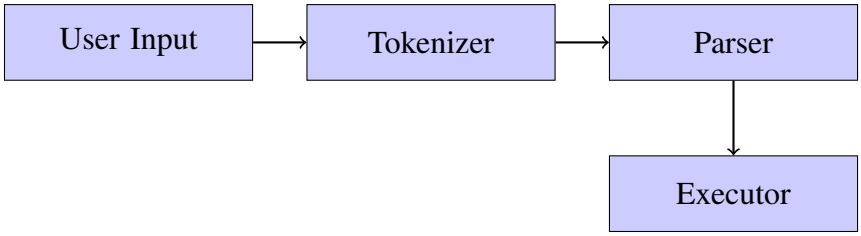


Figure 2.1: Command Processing Pipeline

2.3 Core Architectural Patterns

2.3.1 Command Pattern

WinShell uses the Command pattern to encapsulate command execution:

```
public interface ICommand
{
    string Name { get; }
    string Description { get; }
    Task<bool> ExecuteAsync(string[] args,
                            IShellEnvironment env);
}
```

Listing 2.1: ICommand Interface

This pattern provides:

- Uniform interface for built-in and external commands
- Easy addition of new commands
- Testability through interface mocking
- Support for command composition

2.3.2 Asynchronous Execution

WinShell leverages .NET's async/await pattern:

```
public async Task<bool> ExecuteCommandAsync(
    string commandLine)
{
    var command = await ParseCommandAsync(commandLine);
    var result = await command.ExecuteAsync();
    await ProcessResultAsync(result);
```

```
7       return result.Success;
8   }
```

Listing 2.2: Async Command Execution

Chapter 3

SIMILAR TECHNOLOGIES

3.1 Overview of Existing Shell Implementations

The shell ecosystem is rich with diverse implementations, each designed with specific philosophies and use cases in mind.

3.2 Bash (Bourne Again Shell)

Bash, developed by Brian Fox in 1989, has become the de facto standard shell for Linux and macOS systems.

Table 3.1: Bash vs WinShell Feature Comparison

Feature	**Bash**	**WinShell**
Platform Support	Unix/Linux, WSL	Windows, Linux, macOS
Scripting	Full-featured	Basic, object-oriented
GUI Support	Terminal only	Native WPF GUI
Object Pipelines	No (text-based)	Yes
.NET Integration	No	Full integration

3.3 PowerShell

Microsoft PowerShell introduced in 2006 represents the primary inspiration for WinShell.

PowerShell's strengths include:

- Object-oriented pipeline with .NET object passing
- Strong Windows system administration capabilities
- Extensive cmdlet library
- PowerShell Gallery for module sharing

WinShell builds upon PowerShell's foundation while adding:

- Native GUI interface
- Simplified syntax for common operations
- Visual job management
- Integrated documentation

3.4 Zsh (Z Shell)

Zsh, created by Paul Falstad in 1990, is known for its customization options:

- Advanced tab completion
- Theming via frameworks like Oh My Zsh
- Shared command history
- Powerful globbing patterns

While Zsh excels in customization, WinShell focuses on accessibility and cross-platform consistency.

3.5 Fish (Friendly Interactive Shell)

Fish prioritizes user-friendliness with:

- Syntax highlighting as you type
- Auto-suggestions based on history
- Web-based configuration
- Sensible defaults

Fish's focus on user experience has influenced WinShell's design.

3.6 Comparative Analysis

Table 3.2: Comprehensive Shell Comparison

Feature	**Bash**	**PowerShell**	**Fish**	**WinShell**
GUI Mode	No	No	No	Yes
Object Pipelines	No	Yes	No	Yes
Cross-Platform	Limited	Yes	Yes	Yes
Learning Curve	Medium	High	Low	Low
.NET Integration	No	Full	No	Full

Chapter 4

APPLICATIONS, ADVANTAGES & DISADVANTAGES

4.1 Applications of WinShell

4.1.1 Software Development

WinShell serves as a valuable tool in modern development workflows:

- **Build Automation:** Execute build scripts with real-time visualization
- **Version Control:** Streamlined Git operations
- **Testing:** Run and monitor test suites
- **Deployment:** Automate deployment processes

4.1.2 System Administration

For system administrators, WinShell provides:

- Visual process monitoring
- Service management
- Log file processing
- Automated maintenance tasks

4.1.3 Education and Research

WinShell serves as an excellent teaching tool:

- Demonstrates process management concepts
- Illustrates I/O handling and redirection

- Shows job control implementation
- Provides working example of command interpretation

4.2 Advantages of WinShell

4.2.1 User Experience

- **Dual Interface:** Both CLI and GUI modes
- **Visual Process Management:** Real-time job visualization
- **Enhanced Discoverability:** Integrated help system
- **Syntax Highlighting:** Error prevention

4.2.2 Technical Advantages

- **Modern Architecture:** Leverages .NET ecosystem
- **Cross-Platform Consistency:** Identical behavior across OSes
- **Extensibility:** Plugin architecture
- **Asynchronous Execution:** Non-blocking operations

4.3 Disadvantages and Limitations

4.3.1 Current Limitations

- **Scripting Maturity:** Limited control structures
- **Performance:** Higher memory footprint than native shells
- **Platform-Specific Issues:** Some features unavailable on all platforms
- **Ecosystem:** Smaller community than established shells

4.3.2 Known Issues

- GUI requires .NET Desktop Runtime
- Pipeline depth limited to 10 stages
- Some terminal sequences not fully supported

Chapter 5

SYSTEM DESIGN & REQUIREMENT MODELING

5.1 System Architecture

WinShell follows a layered architecture with clear separation of concerns.

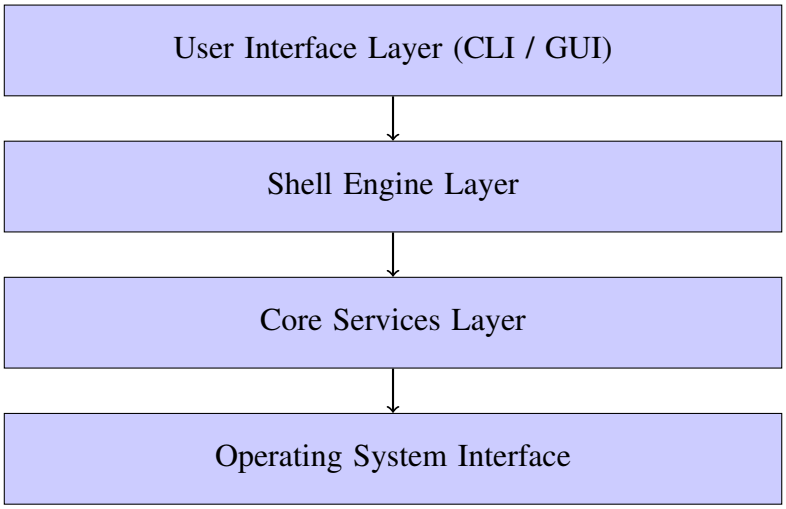


Figure 5.1: WinShell High-Level Architecture

5.1.1 Component Architecture

The system consists of three main components:

- **Winshell.Core:** Foundation library with command parsing, execution, job management
- **Winshell.CLI:** Terminal interaction handling
- **Winshell.GUI:** WPF-based graphical interface

5.2 Requirements Specification

Following IEEE 830 standards, we define functional and non-functional requirements.

Table 5.1: Core Functional Requirements

Req ID	Description
FR-1	Execute built-in commands (cd, pwd, exit, help, jobs, fg, bg, kill)
FR-2	Execute external system commands
FR-3	Support I/O redirection (>, », <, |)
FR-4	Maintain persistent command history
FR-5	Provide job control for background processes
FR-6	Support command-line completion
FR-7	Provide both CLI and GUI interfaces
FR-8	Support syntax highlighting

Table 5.2: Non-Functional Requirements

Req ID	Description
NFR-1	Compatible with Windows 10/11, Ubuntu 20.04+, macOS 10.15+
NFR-2	Respond to commands within 100ms
NFR-3	Maintain consistent behavior across platforms
NFR-4	Handle multiple simultaneous processes
NFR-5	Consume less than 100MB memory in idle state
NFR-6	Be extensible through plugins

Chapter 6

PROJECT IMPLEMENTATION

6.1 Technology Stack

WinShell is built using:

- **.NET 8:** Core framework
- **C#:** Primary language
- **WPF:** GUI framework
- **SQLite:** History persistence
- **MSBuild:** Build system

6.2 Development Environment

- **Visual Studio 2022:** Primary IDE
- **VS Code:** Cross-platform editing
- **.NET SDK 8.0:** Development kit
- **Git:** Version control
- **GitHub:** Repository hosting

6.3 Code Repository Structure

6.3.1 Winshell.Core

Core library containing:

```
public class ShellEngine : IShellEngine
{
    private readonly CommandRegistry _registry;
```

```
    private readonly JobManager _jobManager;
    private readonly HistoryDatabase _history;

    public async Task<bool> ExecuteAsync(string input)
    {
        var tokens = _tokenizer.Tokenize(input);
        var command = _parser.Parse(tokens);

        _history.AddEntry(input);

        return await command.ExecuteAsync();
    }
}
```

Listing 6.1: ShellEngine Implementation

6.3.2 Job Management

```
public class JobManager
{
    private Dictionary<int, Job> _jobs = new();

    public Job StartJob(Process process)
    {
        var job = new Job(_nextId++, process);
        _jobs.Add(job.Id, job);

        process.EnableRaisingEvents = true;
        process.Exited += (s, e) => OnJobCompleted(job);

        return job;
    }

    public void BringToForeground(int jobId)
    {
        if (_jobs.TryGetValue(jobId, out var job))
        {
            job.Status = JobStatus.Running;
            job.Process.WaitForExit();
        }
    }
}
```

Listing 6.2: Job Control Implementation

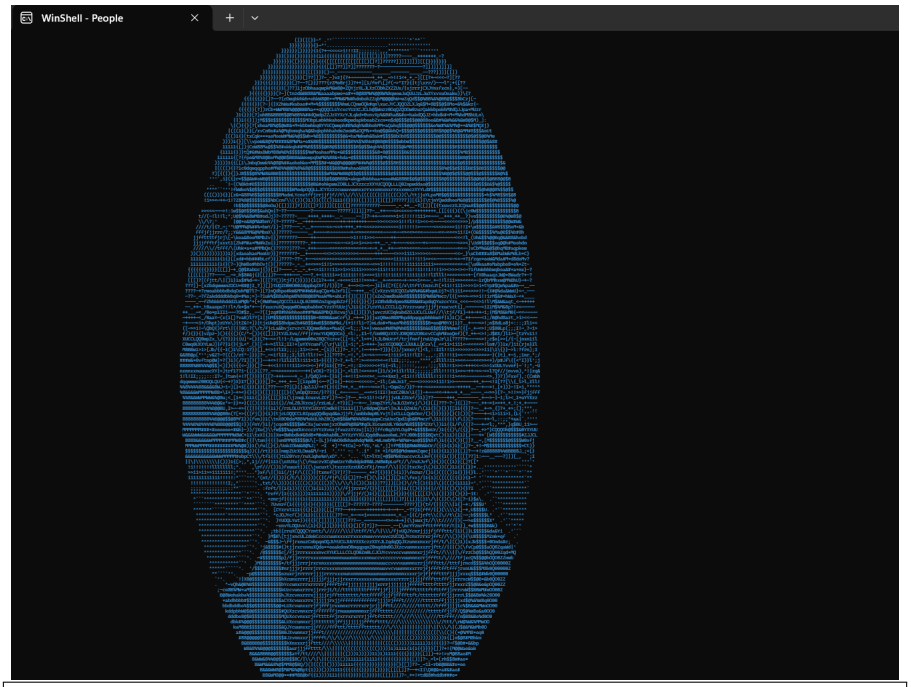
6.4 Building and Running

6.4.1 Build Instructions

Listing 6.3: Build Commands

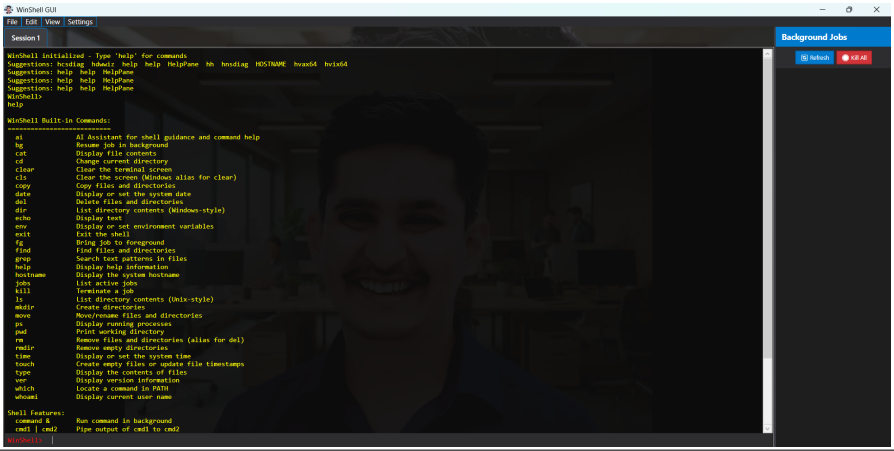
```
# Clone repository
git clone https://github.com/username/WinShell.git
cd WinShell

# Build solution
dotnet build

# Run CLI
dotnet run --project Winshell.CLI

# Run GUI
dotnet run --project Winshell.GUI
```

6.5 Screenshots



CLI Interface Screenshot

Figure 6.1: WinShell CLI Interface



GUI Interface Screenshot

Figure 6.2: WinShell GUI Interface

Chapter 7

CONCLUSION & FUTURE SCOPE

7.1 Project Achievements

WinShell has successfully achieved its core objectives:

- Implemented a fully functional shell with CLI and GUI
- Created cross-platform solution for Windows, Linux, and macOS
- Combined best elements of traditional shells and PowerShell
- Built extensible architecture for future enhancements
- Delivered user-friendly experience with modern features

7.2 Lessons Learned

During development, several valuable lessons were learned:

- Complexity of cross-platform compatibility
- Importance of well-designed architecture
- Challenges of balancing tradition with innovation
- Benefits of separating core functionality from UI
- Power of the .NET ecosystem

7.3 Future Scope

WinShell has significant potential for expansion:

7.3.1 Enhanced Scripting

- More comprehensive C#-based scripting language
- Script debugging and profiling tools
- Package management for scripts

7.3.2 Cloud Integration

- Built-in SSH capabilities
- Cloud provider integrations (AWS, Azure, GCP)
- Container management features

7.3.3 AI Features

- Command suggestions based on usage patterns
- Automated script generation
- Natural language command interpretation
- Intelligent error correction

7.4 Closing Thoughts

WinShell represents a significant step forward in shell design, combining traditional command-line efficiency with modern programming paradigms and graphical interfaces. As computing continues to evolve, shells like WinShell that bridge different paradigms and platforms will play an increasingly important role.

Appendix A

SAMPLE CODE

A.1 Command Parser Implementation

```
using System;
using System.Collections.Generic;

namespace Winshell.Core.Parser
{
    public class Tokenizer
    {
        public IEnumerable<Token> Tokenize(string input)
        {
            var tokens = new List<Token>();
            var current = 0;

            while (current < input.Length)
            {
                char c = input[current];

                if (char.IsWhiteSpace(c))
                {
                    current++;
                    continue;
                }

                if (c == '|')
                {
                    tokens.Add(new Token(TokenType.Pipe, "|")
                        );
                    current++;
                }
                else if (c == '>')
                {
                    if (current + 1 < input.Length &&
```

```
                        input[current + 1] == '>')
                    {
                        tokens.Add(new Token(
                            TokenType.AppendRedirect, ">>"));
                        current += 2;
                    }
                    else
                    {
                        tokens.Add(new Token(
                            TokenType.OutputRedirect, ">"));
                        current++;
                    }
                }
                else
                {
                    var word = ReadWord(input, ref current);
                    tokens.Add(new Token(TokenType.Word, word
                        ));
                }
            }

            return tokens;
        }

        private string ReadWord(string input, ref int current
            )
        {
            var start = current;
            while (current < input.Length &&
                   !char.IsWhiteSpace(input[current]) &&
                   input[current] != '|' &&
                   input[current] != '>')
            {
                current++;
            }
            return input.Substring(start, current - start);
        }
    }

    public enum TokenType
    {
```

```
        Word,
        Pipe,
        OutputRedirect,
        AppendRedirect
    }

    public class Token
    {
        public TokenType Type { get; set; }
        public string Value { get; set; }

        public Token(TokenType type, string value)
        {
            Type = type;
            Value = value;
        }
    }
}
```

Listing A.1: Complete Tokenizer Implementation

A.2 Process Execution

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;

namespace Winshell.Core.Process
{
    public class ProcessExecutor
    {
        public async Task<ProcessResult> ExecuteAsync(
            string fileName,
            string arguments)
        {
            var startInfo = new ProcessStartInfo
            {
                FileName = fileName,
                Arguments = arguments,
                RedirectStandardOutput = true,
                RedirectStandardError = true,
```

```
                UseShellExecute = false,
                CreateNoWindow = true
            };

            using var process = new Process();
            process.StartInfo = startInfo;

            var output = new StringBuilder();
            var error = new StringBuilder();

            process.OutputDataReceived += (s, e) =>
            {
                if (e.Data != null)
                    output.AppendLine(e.Data);
            };

            process.ErrorDataReceived += (s, e) =>
            {
                if (e.Data != null)
                    error.AppendLine(e.Data);
            };

            process.Start();
            process.BeginOutputReadLine();
            process.BeginErrorReadLine();

            await process.WaitForExitAsync();

            return new ProcessResult
            {
                ExitCode = process.ExitCode,
                Output = output.ToString(),
                Error = error.ToString()
            };
        }
    }

    public class ProcessResult
    {
        public int ExitCode { get; set; }
        public string Output { get; set; }
```

```
60          public string Error { get; set; }
61          public bool Success => ExitCode == 0;
62      }
63  }
```

Listing A.2: Process Executor Implementation

Appendix B

UML DIAGRAMS

B.1 Class Diagram

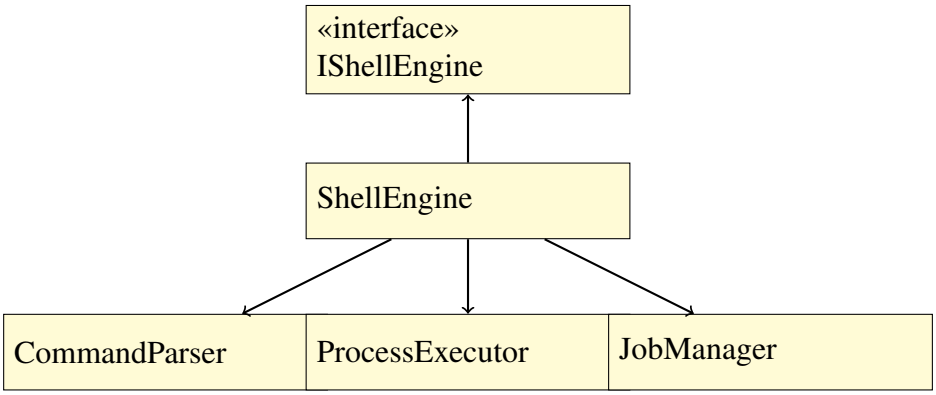


Figure B.1: WinShell Core Class Diagram