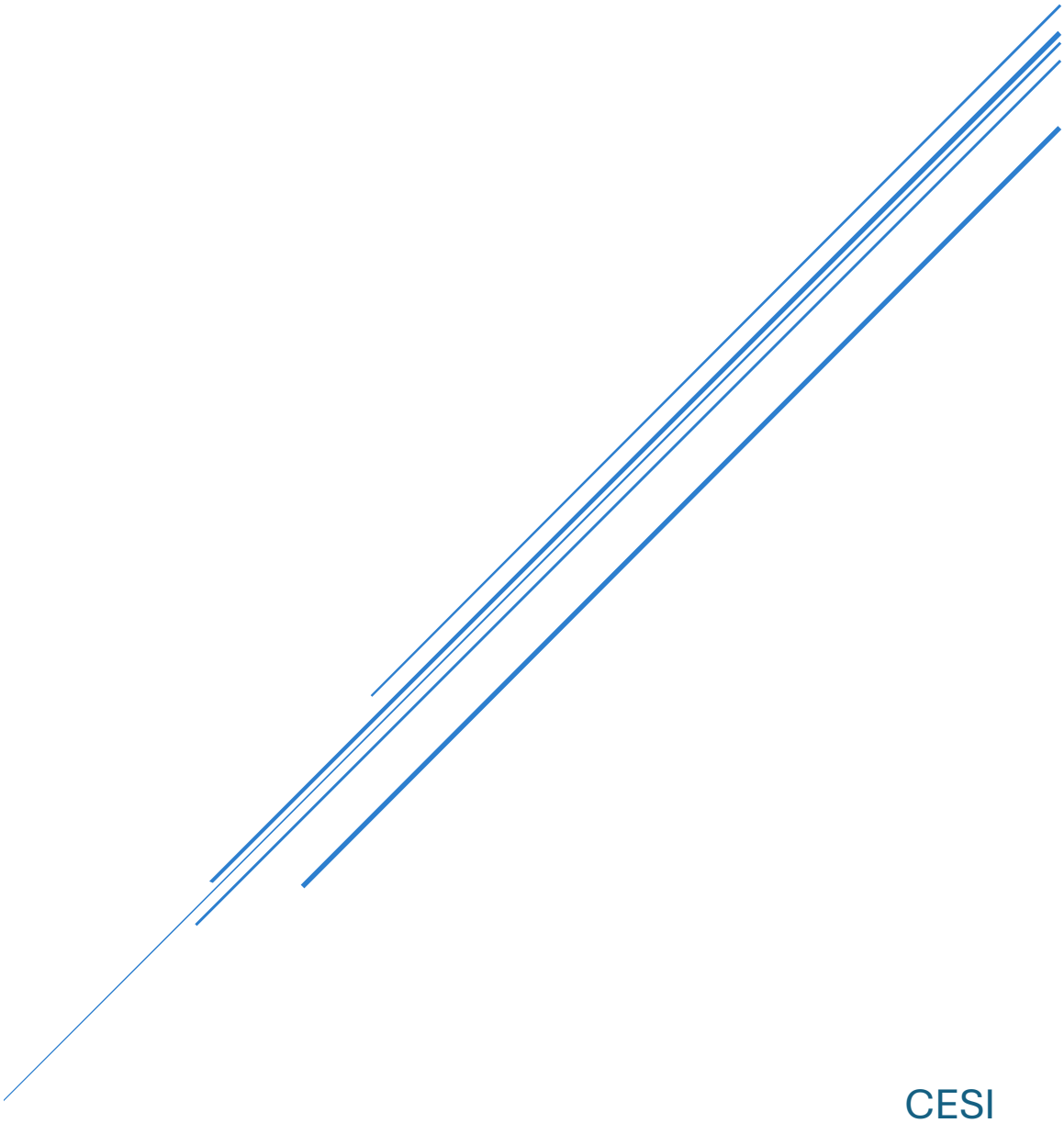
TECHNICAL ANALYSIS

Groupe 1 – EASY SAVE



CESI

Contents

Functional Analysis

The functional analysis of the EasySave ProSoft software focuses on translating end-user requirements into implementable features. The system offers the following core functionalities:

- **Language Selection at Startup**
  Upon launching the application, the user is prompted to select a display language: English (en) or French (fr). The choice dynamically affects all system messages and menu texts throughout the session.
- **Backup Job Creation**
  Users can create new backup jobs by specifying:
  - A unique job name
  - A valid source path (the directory to back up)
  - A valid target path (the backup destination)
  - A backup type: Full or Differential
    A maximum of **five jobs** can be created. If this limit is reached, the system notifies the user accordingly.
- **Execution of a Specific Backup Job**
  The user can launch a single backup job by providing its name. The backup proceeds according to the job's configuration. The system displays real-time progress, transfer statistics, and a completion message.
- **Execution of All Backup Jobs**
  Users may choose to execute all configured jobs sequentially. During execution, each job runs with its defined parameters, and status updates are printed to the console.
- **Job Listing**
  The application allows the user to display the current list of backup jobs. Each job is presented with its name, source and target paths, and backup type. If no jobs are defined, an appropriate message is shown.
- **Multilingual User Interface**
  All prompts, feedback, and menu options are fully translated and presented based on the selected language. The software supports dynamic language selection at startup.
- **Command-Line Based Interface (CLI)**
  All user interactions are handled through a structured and intuitive command-line menu. Navigation is achieved through numbered menu options, providing a simple yet effective interface for non-technical users.
- **Exit Application**
  An option is provided in the main menu to exit the application. Upon selection, the program terminates and returns control to the system console.

Technical Environment

The EasySave project was developed in compliance with the technical and organizational constraints defined by ProSoft's management. The work environment ensures compatibility with industrial practices, maintainability, and traceability.

- **Programming Language**:
  C#
- **Framework**:
  .NET 8.0
- **Development Environment**:
  Visual Studio 2022 or later
- **Version Control**:
  GitHub Repository
  (Includes structured commit history and project traceability)
- **UML Tool**:
  Draw.io and PlantUML
- **Coding Standards**:
  - All code, comments, and documentation are written in English
  - Function names and variables follow C# naming conventions (PascalCase and camelCase)
  - Functions are kept concise to maintain readability
  - Redundant code is avoided to promote maintainability and scalability
  - Modular structure to reduce development costs of future versions
- **Data Format and Storage**:
  - Configuration and job state are saved in JSON format
  - JSON files (log and error logs) include line breaks to support readability in Notepad
  - File paths and configurations are stored in a production-compatible format (avoiding temporary directories like C:\temp\)
- **Interface Type**:
  Command-line interface (CLI), designed for simplicity, usability, and future extensibility (e.g., WPF GUI)

V. Release Notes – EasySave v1.0

- **Software Name**: EasySave ProSoft
- **Version**: 1.0
- **Release Date**: 10-05-2025
- **Development Team**: Kouba Amine – Hachi Idris – Mohammed Bensalem – Ouanoughi Noufel
- **Git Repository**: https://github.com/8icey/SE-project-2025.git

Delivered Features

- Creation of up to five backup jobs
- Each job includes:
  - Name
  - Source directory
  - Target directory
  - Backup type (Full or Differential)
- Execution of:
  - A specific backup job by name
  - All configured jobs sequentially
- Multilingual support: English and French (selectable at startup)
- Backups supported on:
  - Local drives
  - External drives
  - Network locations
- Real-time daily log in JSON format (DailyLog.json) with:
  - Timestamps
  - Source and destination file paths

    - File size
    - Transfer time (ms)
    - Success/failure status
- Real-time error log in JSON format (ErrorLog.json)
- All JSON logs are formatted with line breaks for Notepad readability
- Modular design:
    - Logging and job persistence handled via dedicated services

Tested Functionalities

- Execution of full and differential backups
- Validation of log and error file writing in JSON format
- Dynamic language switching
- Job listing and job creation with input validation
- Execution flow via command-line interface

Known Limitations

- Only command-line interface (CLI) is available
- No advanced error recovery (e.g., inaccessible paths, permission errors)

Planned Enhancements (v1.1 / v2.0)

- Graphical User Interface using WPF (MVVM architecture)
- Improved robustness and error handling
- Scheduled backups and system notifications
- Enhanced logging and visual monitoring

Software Architecture

The architecture of the EasySave ProSoft software is based on a clear separation of responsibilities, following object-oriented design principles. The main components are:

- **BackupJob**
  Represents a configured backup task. Each instance contains a job name, a source path, a target path, and a backup type (Full or Differential). It encapsulates the execution logic, file processing, and transfer time measurement.
- **BackupManager**
  Manages the collection of backup jobs. It handles job creation, execution (by name or in sequence), and deletion. It also ensures persistence by saving and loading jobs through a JSON handler.
- **BackupManagerViewModel**
  Serves as a logical intermediary between the console UI and the job management logic. It simplifies user input handling and centralizes validation before delegating actions to BackupManager.
- **ConsoleUI**
  Manages user interaction through a command-line menu. It prompts for language selection, navigates user options, collects job parameters, and routes commands to the view model.
- **JsonHandler**
  Handles the serialization and deserialization of backup jobs to a persistent JSON file (backupWorks.json). It supports loading saved jobs and deleting executed jobs from storage.
- **Logger**
  Records structured logs of backup operations. It generates:
  - DailyLog.json for successful transfers
  - ErrorLog.json for exceptions encountered during backup
    Logs are formatted with line breaks for readability and include timestamps, file paths, sizes, transfer duration, and status.
- **LanguageService**
  Supports multilingual functionality. Based on the user's language selection (en or fr), it loads the corresponding dictionary of interface strings and applies them throughout the application.
- **FileItem**
  Represents metadata for each backed-up file, including source/destination paths, file size, transfer duration, and success status. Used by Logger for JSON log generation.

- **TimerUtil**
  A helper class used to measure execution time of backup operations, aiding accurate logging.
- **Program.cs**
  Acts as the entry point of the application. It initializes the console UI and starts the menu loop.

This modular architecture offers:

- Strong code maintainability
- Clear role separation aligned with SOLID principles
- A scalable foundation for transition to graphical interfaces (e.g., WPF with MVVM pattern)
- Simplified debugging and extension, due to loosely coupled components

UML Diagrams

The UML (Unified Modeling Language) diagrams are essential tools for visually representing the structure and behavior of a software system. They are used for business process modeling, system design, and technical documentation. For this deliverable, several UML diagrams were created to illustrate different aspects of the EasySave ProSoft project.

Activity Diagram

The following diagram represents the full user workflow within the EasySave application. It covers the main menu logic, user interactions, and the internal control flow for backup operations.

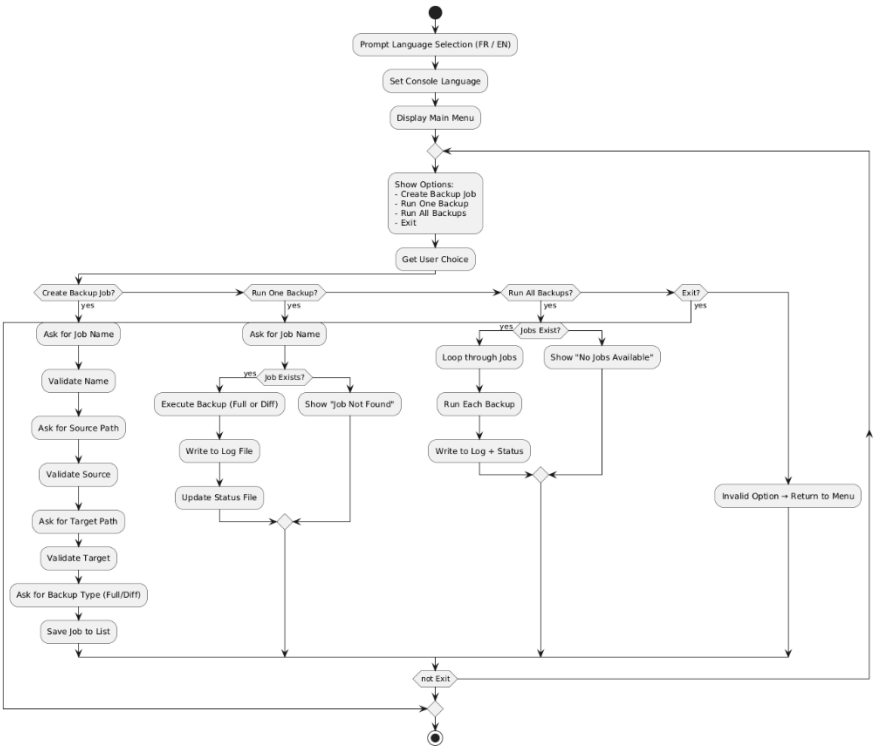


Figure 01: Activity Diagram – EasySave v1.0

Diagram Analysis:

The activity diagram is divided into distinct decision points and process blocks that reflect the structure of the CLI menu system. The flow begins with language selection, followed by a loop over the main menu options. The core logic branches into four main paths:

- **Create Backup Job**
  The user is prompted to enter job details (name, source path, target path, and type). Each input is validated before the job is saved to the internal list.
- **Run One Backup**
  The system asks for the job name. If the job exists, the corresponding backup is executed. Log and status files are updated. If the job is not found, an error message is displayed.
- **Run All Backups**
  If jobs exist, the application loops through all saved jobs, executes them in order, and logs the results. If no jobs are found, a message informs the user.
- **Exit**
  Terminates the application gracefully.

Additional safeguards include input validation and fallback handling for invalid menu choices, which redirect the user back to the main menu without interruption.

This diagram provides a clear and structured view of the application's decision flow and user interaction logic, ensuring predictable behavior and a guided experience.

Class Diagram

The following diagram illustrates the object-oriented structure of the EasySave ProSoft application. It defines the main classes, their attributes and methods, and the relationships between components in different layers (Model, ViewModel, View, and Helpers).

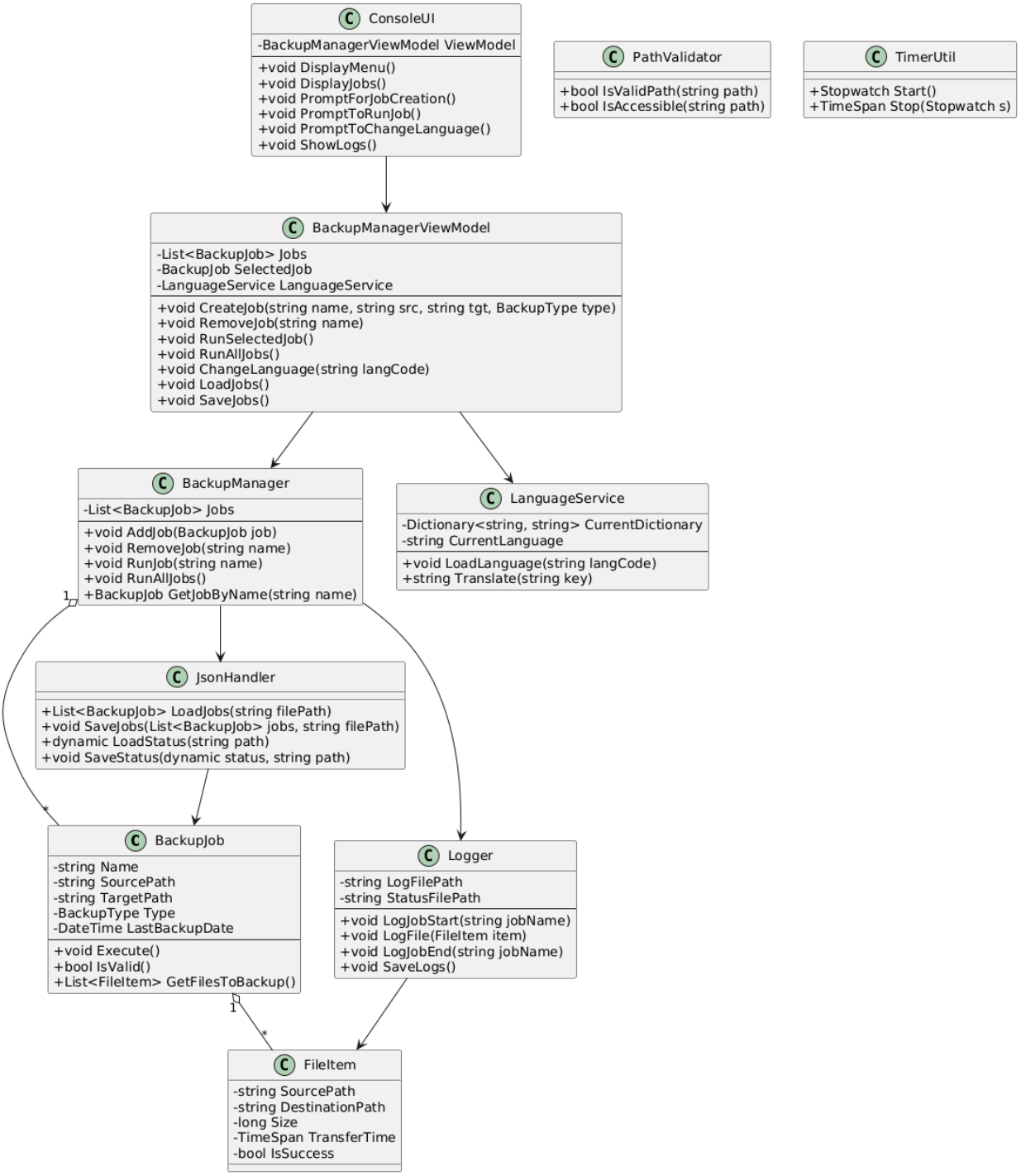


Figure 02: Class Diagram – EasySave v1.0

Diagram Analysis:

The class diagram reflects a modular and layered architecture, designed for clarity, testability, and future extensibility. The major components are organized as follows:

- **Model Layer**:
    - BackupJob represents a backup operation with attributes such as job name, source/target paths, backup type, and last execution time. It contains methods for execution logic and file filtering.
    - FileItem encapsulates metadata for each file involved in a backup: paths, size, transfer duration, and status.
    - BackupManager maintains a list of BackupJob instances and provides methods to add, remove, execute, or retrieve jobs.
    - Logger handles the generation of logs for job execution and individual file transfers. It supports structured JSON output for DailyLog.json and ErrorLog.json.
    - JsonHandler manages serialization and deserialization of job and status data to JSON files.
    - LanguageService supports multilingual message handling by loading and translating UI strings dynamically.
- **ViewModel Layer**:
    - BackupManagerViewModel serves as an intermediary between the user interface and the business logic. It manages backup job operations, language control, and the loading/saving of persisted job data.
- **View Layer**:
    - ConsoleUI handles the entire user interaction flow through a command-line interface. It offers menu rendering, prompts for job creation and execution, and interacts with the BackupManagerViewModel.
- **Helper Classes**:
    - PathValidator validates and checks accessibility of file system paths.
    - TimerUtil is used to measure and return the execution time of backup tasks.

Relationships:

- A BackupManager is composed of multiple BackupJob instances (1..*).
- A BackupJob contains multiple FileItem objects.
- BackupManagerViewModel aggregates BackupManager and LanguageService.
- ConsoleUI interacts directly with the BackupManagerViewModel to manage all user commands.

This class diagram emphasizes the modular design and separation of concerns within EasySave ProSoft. It provides a solid base for scalability (e.g., transition to GUI with WPF) and ensures that components can evolve independently while maintaining cohesive interaction

Sequence Diagrams

Sequence diagrams are used to illustrate the temporal flow of interactions between the components of a system. They show the order of messages exchanged and help clarify system behavior during specific use cases.

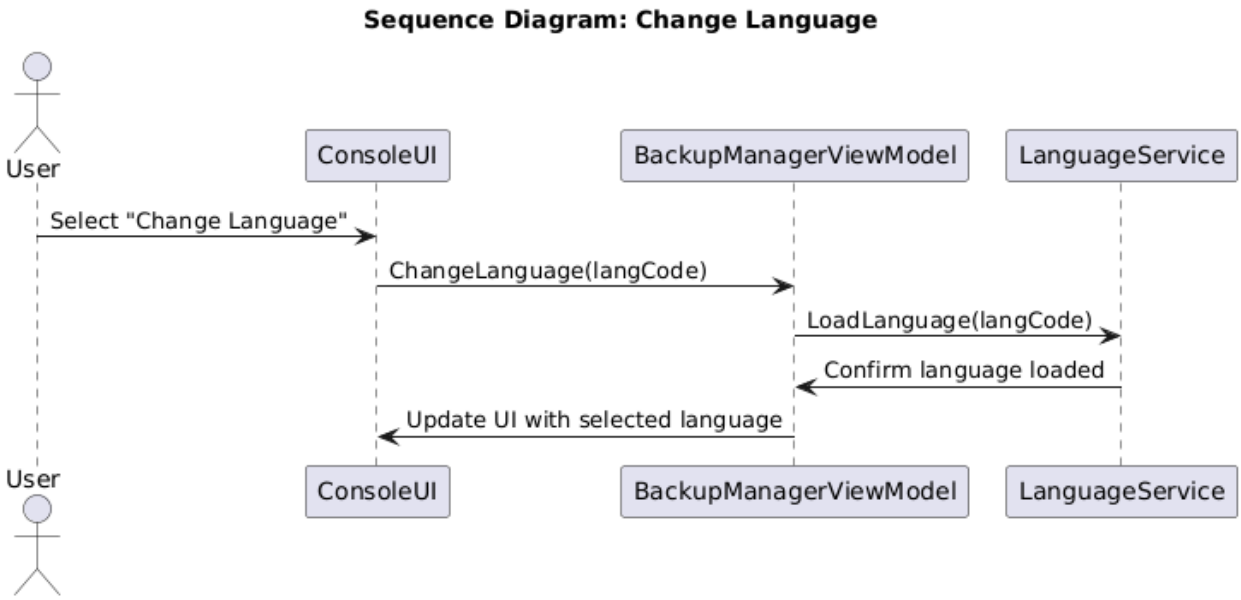


Figure 03: Sequence Diagram – Change Language

Diagram Analysis:

This diagram models the dynamic flow triggered when the user selects the option to change the interface language. The process is as follows:

- The User selects the language change option from the menu presented by ConsoleUI.
- ConsoleUI forwards the language code to BackupManagerViewModel, which acts as an intermediary.
- BackupManagerViewModel delegates the request to LanguageService, which loads the corresponding dictionary.
- Once the language is successfully loaded, the user interface is updated to reflect the selected language dynamically.

This sequence confirms that the multilingual feature is modular, with a clear separation of display logic (ConsoleUI) and language data management (LanguageService).

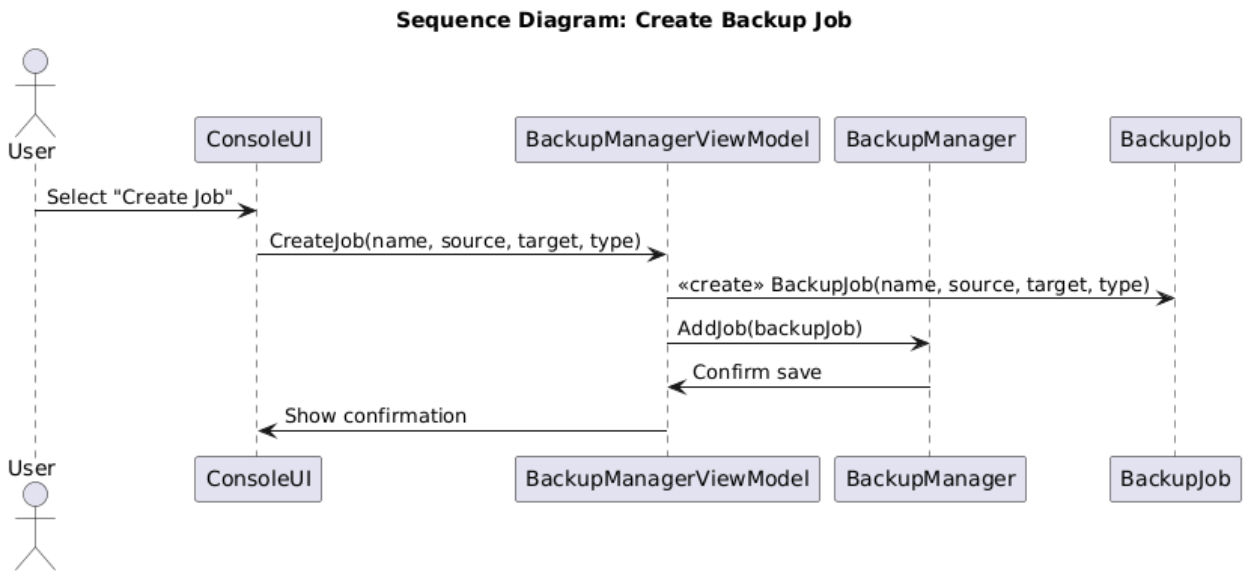


Figure 04: Sequence Diagram – Create Backup Job

Diagram Analysis:

This diagram shows how a backup job is created through user interaction:

- The User initiates the action from ConsoleUI.
- ConsoleUI prompts the BackupManagerViewModel to create a job using the specified parameters.
- A BackupJob instance is instantiated and passed to the BackupManager.
- BackupManager adds the job to the list and confirms the addition back to the view model.
- The confirmation message is then displayed to the user by ConsoleUI.

The diagram highlights the clean delegation of responsibilities: input and interaction are handled by ConsoleUI, job management is centralized in BackupManager, and the orchestration is managed via the view model.

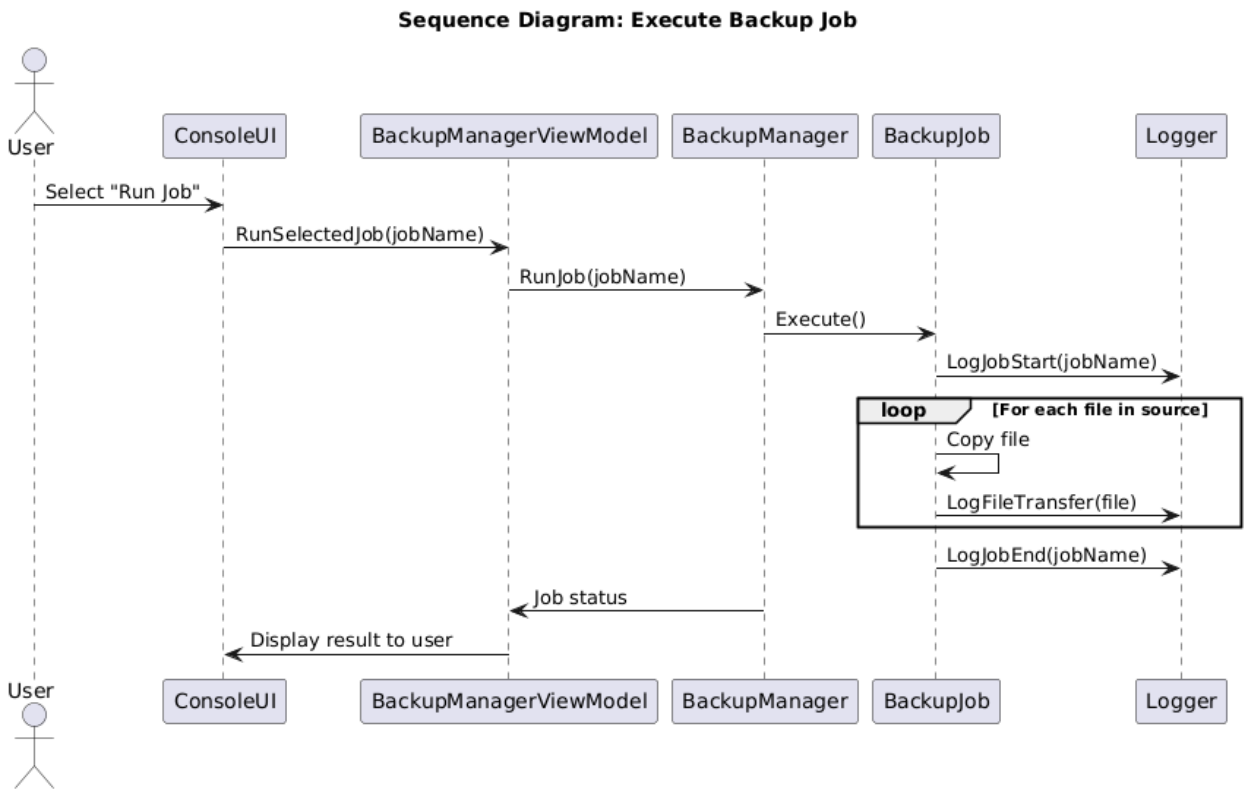


Figure 05: Sequence Diagram – Execute Backup Job

Diagram Analysis:

This diagram describes the workflow for executing a single backup job:

- The User triggers a job execution from the UI.
- The ConsoleUI delegates the task to BackupManagerViewModel, which forwards it to BackupManager.
- BackupManager locates the job and instructs the BackupJob instance to execute.
- During execution:
    - A log entry is created (LogJobStart) to mark the beginning.
    - For each file in the source directory, a copy operation is attempted.
    - Each transfer is logged individually using LogFileTransfer.
    - A final log entry (LogJobEnd) marks completion.
- Execution status is sent back through the chain and displayed to the user.

This diagram clearly outlines the operational path for file backups and demonstrates the layered interaction between the UI, application logic, and logging subsystem.

Use Case Diagram

The use case diagram provides a high-level view of the user interactions supported by the EasySave v1.0 system. It identifies the core functionalities available to the end user and their logical grouping.

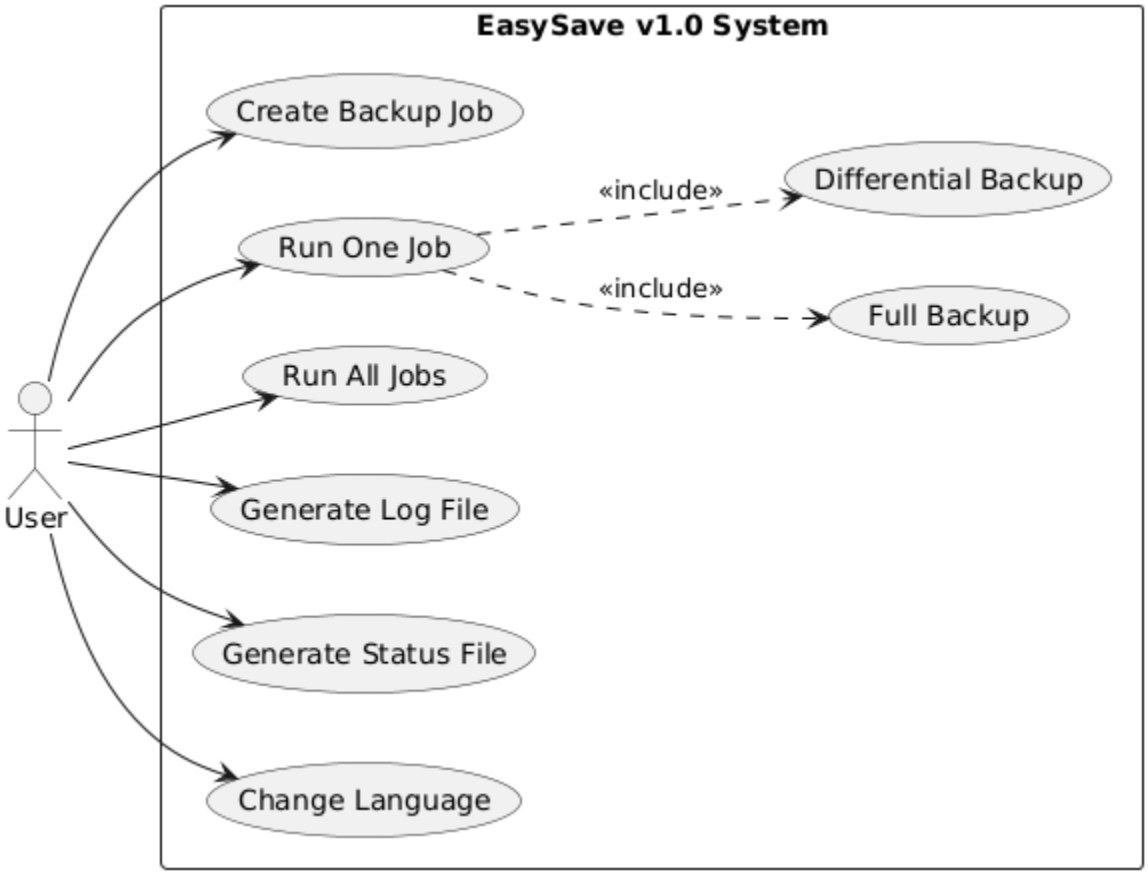


Figure 06: Use Case Diagram – EasySave v1.0

Diagram Analysis:

In this diagram, the primary actor is the **User**, who interacts with the EasySave system through a command-line interface. The available use cases are as follows:

- **Create Backup Job**
  Allows the user to define a backup task by specifying the job name, source path, target path, and backup type.
- **Run One Job**
  Enables the user to execute a specific backup task. This use case includes:
  - **Full Backup**: Copies all files from the source to the target.
  - **Differential Backup**: Copies only files modified since the last backup.

- **Run All Jobs**
  Executes all configured backup jobs sequentially, updating logs and statuses in real time.
- **Generate Log File**
  Involves the creation of a JSON-formatted log file (DailyLog.json) for each job, recording timestamped entries with transfer details.
- **Generate Status File**
  Refers to the creation and update of job progress tracking (ErrorLog.json), which captures success or failure states.
- **Change Language**
  Allows the user to switch between English and French. The selected language is applied dynamically across all menu prompts and messages.

This diagram provides a concise summary of the system's supported interactions and encapsulates the primary functionality delivered in version 1.0. The inclusion relationships clarify that different backup strategies are sub-behaviors of executing a job.

Tests, Limitations, and Future Enhancements

The testing phase was conducted to ensure that the EasySave ProSoft application functions correctly in accordance with the specified requirements. Tests were executed in a local environment using the provided command-line interface. Each core feature was evaluated individually, including job creation, execution of full and differential backups, logging, status file updates, and multilingual switching.

Test 1 – Language Selection

Objective:
Verify that the application correctly prompts the user to select a language and applies it dynamically.

Procedure:
At launch, the user is prompted to enter a language code (fr or en). Based on the input, the system loads the corresponding dictionary and updates all interface text.

Expected Result:
Menu and prompts appear in the selected language.

Observed Result:
Language is successfully switched; all displayed strings match the selected language.

Test 2 – Main Menu Navigation

Objective:
Ensure that the menu displays correctly and routes the user to appropriate functions.

Procedure:
After language selection, the main menu offers five options:

1. Create Backup Job
2. Run One Job
3. Run All Jobs
4. Display Backup Jobs
5. Exit

Expected Result:
Each selection triggers the corresponding process.

Observed Result:
Menu options are functional and responsive. Invalid inputs are handled with appropriate error messages.

Test 3 – Job Creation (Valid Paths)

Objective:
Validate that backup jobs can be created with valid source and target paths.

Procedure:
User inputs job name, source path, target path, and type (Full or Differential).

Expected Result:
The job is saved and added to the list.

Observed Result:
The job is successfully created and persisted in backupWorks.json. Input validation prevents invalid or inaccessible paths.

Test 4 – Job Listing

Objective:
Verify the accurate display of all configured backup jobs.

Procedure:
Select the “Display Backup Jobs” option.

Expected Result:
List includes job names, source paths, target paths, and types.

Observed Result:
All job details are shown correctly, even after restarts.

Test 5 – Execute One Backup Job

Objective:
Ensure that a single job can be executed and logged correctly.

Procedure:
User selects a job by name and confirms execution.

Expected Result:
The job runs, with progress updates shown in the console, and logs generated.

Observed Result:
Backup executes as expected. JSON logs include accurate timestamps, file sizes, and durations.

Test 6 – Execute All Jobs

Objective:
Test sequential execution of all configured jobs.

Procedure:
Trigger the "Run All Jobs" option.

Expected Result:
All jobs execute in order with full logging and status updates.

Observed Result:
Jobs execute sequentially. Results are logged for each job without skipping or duplication.

Known Limitations

- The application uses a command-line interface only. There is no graphical user interface.
- No advanced error handling is implemented for edge cases (e.g., unauthorized access, missing drives, network interruptions).

Planned Enhancements (v1.1 / v2.0)

- Introduction of a graphical user interface using WPF with MVVM architecture
- Robust error handling and exception reporting

- Scheduled backup support
- Desktop notifications for backup completion or failures
- Improved performance and logging metrics display

Conclusion

This first deliverable marks a foundational milestone in the development of the EasySave ProSoft project. The team has delivered a fully functional backup solution aligned with the technical and organizational constraints defined by ProSoft.

Version 1.0 includes the essential features required for operational use, including:

- Creation and execution of backup jobs (Full and Differential)
- Real-time JSON logging of operations
- Multilingual support (French and English)
- Persistent configuration management
- Command-line interface adapted for extensibility

The architecture of the application follows clear object-oriented principles and adopts a layered, modular design. Responsibilities are properly distributed across the system components, enabling maintainability, readability, and future evolution—particularly toward a graphical interface.

Testing has confirmed the correct operation of major use cases, and the known limitations have been clearly identified. The application provides a reliable foundation for future iterations that will introduce more robust functionality, such as scheduled backups, enhanced error handling, and GUI support using WPF and MVVM.

EasySave ProSoft v1.0 is therefore positioned as a stable and extensible software product, ready to be deployed in controlled environments and further developed to meet more advanced user needs.