**Livrable 1 :**

**Projet EasySave**

**Génie Logiciel**

# INTRODUCTION

Dans le cadre de notre intégration chez l’éditeur ProSoft, nous avons été chargés de concevoir un outil de sauvegarde baptisé **EasySave**, pensé pour s’intégrer à la suite logicielle de l’entreprise. Ce projet avait pour but de proposer une solution simple et robuste pour automatiser les copies de fichiers, tout en garantissant une bonne lisibilité du code, une modularité efficace et un suivi rigoureux des opérations.

La version 1.0 du logiciel repose sur une interface en ligne de commande, avec la possibilité de gérer plusieurs tâches de sauvegarde et de suivre en temps réel leur avancement. Elle est conçue pour être utilisable aussi bien en français qu’en anglais, selon la préférence de l’utilisateur. Un accent particulier a été mis sur la traçabilité, grâce à un système de journalisation structuré au format JSON.

Ce premier livrable pose les bases d’un outil évolutif, prêt à accueillir des fonctionnalités plus avancées dans ses futures versions. Le développement s’est appuyé sur des pratiques professionnelles exigeantes, avec une attention portée à la clarté du code, à sa maintenabilité et à sa conformité avec les attentes de ProSoft.

# PRÉSENTATION DU PROJET ET CACHIER DES CHARGES

## CONTEXTE GÉNÉRAL DU PROJET

Le projet **EasySave** a été initié afin de proposer une solution simple mais complète de sauvegarde de fichiers, intégrée à la suite logicielle de ProSoft. Cette solution vise à répondre à un besoin fréquent dans les environnements professionnels : permettre à un utilisateur, sans expertise technique avancée, d’effectuer des sauvegardes fiables, traçables et personnalisées, depuis une interface en ligne de commande.

Le logiciel est conçu pour s’exécuter en console, de manière intuitive, tout en offrant des fonctionnalités avancées telles que la gestion multilingue, le suivi en temps réel des sauvegardes ou encore la création de fichiers de logs structurés. Ce premier livrable se concentre sur l’essentiel du besoin métier, tout en préparant le terrain pour des évolutions futures, notamment une version graphique et une meilleure automatisation.

## OBJECTIFS FONCTIONNELS

La version 1.0 du logiciel devait couvrir les fonctions clés suivantes :

* **Création et gestion de sauvegardes personnalisées** : l’utilisateur peut créer jusqu’à cinq travaux de sauvegarde, chacun identifié par un nom, une source, une destination et un type de copie.
* **Choix du type de sauvegarde** : deux types sont proposés. Une **sauvegarde complète**, qui recopie l’ensemble des fichiers, et une **sauvegarde différentielle**, qui ne recopie que les fichiers nouveaux ou modifiés.
* **Exécution ciblée** : l’utilisateur peut exécuter une ou plusieurs sauvegardes via une syntaxe simple (1-3 pour les sauvegardes 1 à 3 ; 1;3 pour la 1 et la 3 ; ou encore 1 pour une sauvegarde unique).
* **Multilingue** : l’interface doit être utilisable en français ou en anglais, au choix de l’utilisateur dès le lancement.
* **Compatibilité de stockage** : les chemins sources et cibles peuvent pointer vers un disque local, un périphérique USB ou un lecteur réseau.

## JOURNALISATION DES ACTIONS (FICHIER LOG JOURNALIER)

Une exigence centrale du projet était de tracer toutes les opérations réalisées pendant les sauvegardes. Pour cela, le logiciel génère en temps réel un fichier **log journalier**, au format **.json**, dans lequel chaque fichier traité est enregistré avec les informations suivantes :

* Date et heure de traitement (horodatage)
* Nom de la sauvegarde concernée
* Chemin complet du fichier source (format UNC)
* Chemin complet du fichier destination (format UNC)
* Taille du fichier en octets
* Temps de transfert du fichier, exprimé en millisecondes (ou valeur négative en cas d’erreur)

Ce fichier est conçu pour être exploité facilement par des utilisateurs ou des outils de supervision : son format JSON est structuré, lisible et indenté pour une ouverture directe dans un éditeur de texte simple comme Notepad.

## SUIVI D’AVANCEMENT EN TEMPS RÉEL (FICHIER D’ÉTAT)

En complément du log, un second fichier unique (**state.json**) est généré pour suivre en direct l’état de chaque sauvegarde en cours. Ce fichier est mis à jour dynamiquement à chaque progression du traitement et contient, pour chaque travail actif :

* Le nom de la sauvegarde
* Le dernier horodatage d’action
* L’état du travail (actif, inactif, terminé…)
* Le nombre total de fichiers à traiter
* Le volume global à transférer
* Le nombre de fichiers restants
* La taille restante à copier
* Le chemin du fichier source en cours
* Le chemin de destination correspondant

Cette fonctionnalité permet d’interfacer facilement EasySave avec d’éventuels outils de supervision ou de monitoring métier.

## CONTRAINTES TECHNIQUES ET BONNES PRATIQUES

Le développement du logiciel devait respecter des exigences spécifiques imposées par la direction technique de ProSoft. Parmi les plus importantes :

* **Multilinguisme intégré dès la conception**, sans duplication de code.
* **Modularité** : chaque fonctionnalité (sauvegarde, journalisation, interaction utilisateur) devait être isolée en composants clairs.
* **Lisibilité et maintenabilité du code** : noms de fonctions explicites, commentaires en anglais, pas de copier-coller.
* **Architecture pensée pour l’évolutivité** : en prévision d’une version 2.0 graphique basée sur le pattern **MVVM**.
* **Localisation des fichiers sensibles** : les fichiers log et state devaient être stockés dans un emplacement sûr, compatible avec une utilisation en entreprise. Les chemins comme C:\\temp ont donc été proscrits, au profit de dossiers standards comme %APPDATA%\\EasySave.
* **Format des fichiers** : JSON lisible humainement, avec indentation et retours à la ligne. Une pagination JSON serait un plus pour les grandes sauvegardes.

## CADRE DE TRAVAIL IMPOSÉ

Afin d’assurer la qualité du code et la possibilité de reprise future par une autre équipe, la direction de ProSoft nous a imposé l’utilisation des outils suivants :

* **Visual Studio 2022 ou supérieur** pour le développement
* **GitHub** pour le versioning, avec invitation du tuteur
* **ArgoUML** pour la modélisation initiale (diagrammes de classes, cas d’usage)
* **Langage** : C# uniquement
* **Framework** : .NET 8.0

Tous les fichiers, code source et livrables documentaires devaient être maintenus sur le dépôt Git, avec une structuration propre et traçable.

## LIVRABLE ATTENDU POUR LA VERSION 1.0

Le projet devait produire à minima les éléments suivants :

* Le code complet, organisé et commenté
* Une **documentation utilisateur** tenant sur une page, en français et en anglais
* Une **fiche support technique** indiquant les répertoires d’installation, fichiers de configuration et prérequis techniques
* Une **note de version (release note)** précisant les fonctionnalités livrées
* Une **bibliothèque externe (DLL)** pour la gestion des logs, réutilisable et rétro compatible
* Les **diagrammes UML** de conception
* Ce **rapport de soutenance.**

# PRÉSENTATION DES DIAGRAMMES UML RÉALISÉS

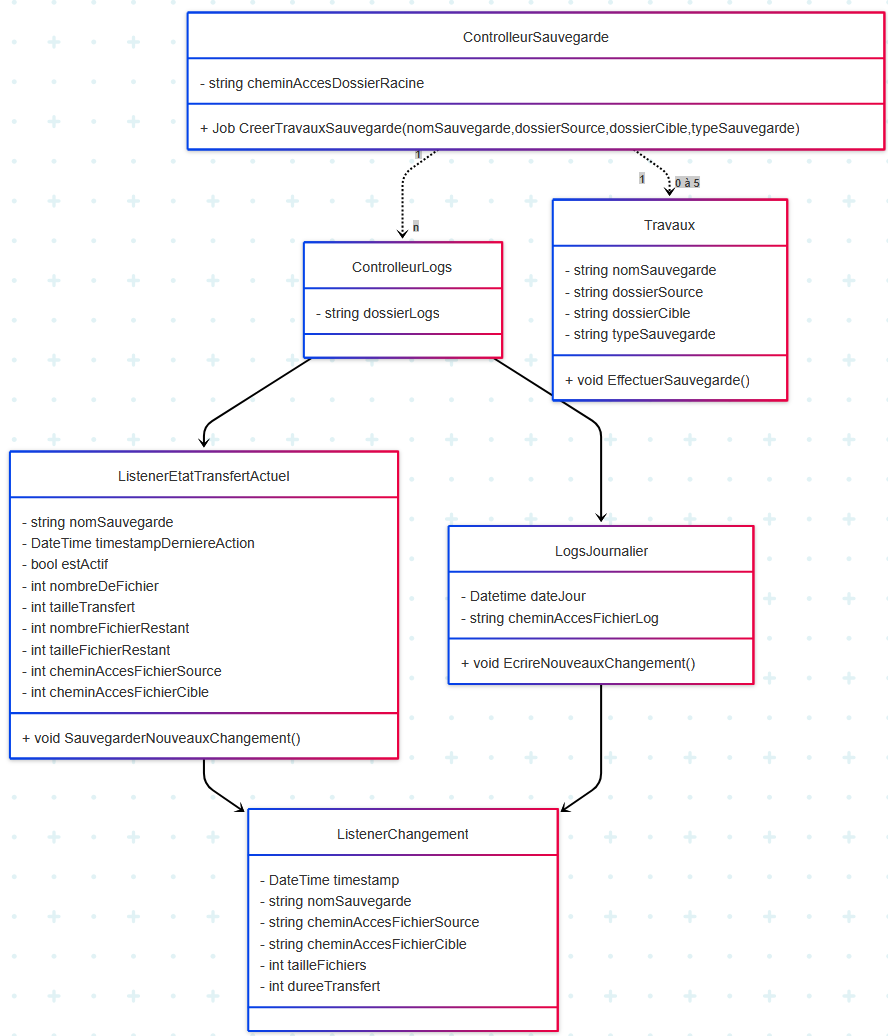
## LE DIAGRAMME DE CLASSES

Le diagramme de classes ci-dessous constitue le socle de la modélisation orientée objet du système de gestion de sauvegardes. Il permet de visualiser l’architecture statique du projet, en représentant les entités principales, leurs attributs, leurs méthodes, ainsi que les relations qui les lient.

Ce modèle repose sur une séparation claire des responsabilités, avec des contrôleurs spécialisés, des classes de données et des classes dédiées au suivi des changements :

* **ContrôleurSauvegarde** : Cette classe agit comme point d’entrée principal pour l’instanciation des travaux de sauvegarde. Elle contient un attribut représentant le chemin du dossier racine et une méthode CreerTravauxSauvegarde(...) qui retourne un objet de type Travaux.
* **Travaux** : Elle encapsule les informations nécessaires à l’exécution d’un travail de sauvegarde : nom, source, cible, et type. La méthode EffectuerSauvegarde() y est définie pour lancer l’opération de transfert.
* **ContrôleurLogs** : Responsable de la gestion centralisée des logs, elle agit comme médiateur entre les événements et les modules de journalisation.
* **Logs.Journalier** : Cette classe permet de tracer quotidiennement les événements de sauvegarde dans un fichier log, en enregistrant la date et le chemin d’accès aux fichiers. La méthode EcrireNouveauxChangement() est appelée à chaque mise à jour détectée.
* **ListenerEtatTransfertActuel** : Elle observe en temps réel l’état du processus de sauvegarde, avec des données comme le nombre de fichiers, la taille des transferts, et l’état d’activité. Elle propose une méthode SauvegarderNouveauxChangement() qui transmet les données aux modules de log ou de suivi.
* **ListenerChangement** : Elle enregistre ponctuellement les changements observés durant le transfert (source, cible, taille, durée). Ces informations sont précieuses pour les analyses ou la reprise en cas d’échec.

Ce diagramme illustre une architecture modulaire et évolutive, permettant d’ajouter facilement de nouveaux types de sauvegarde, des traitements de logs plus avancés ou des notifications externes sans impacter le noyau.

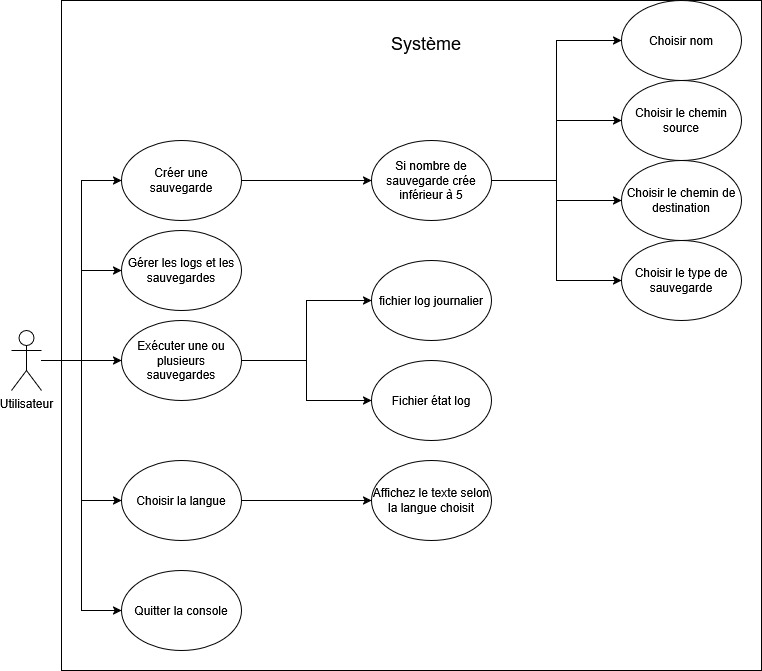


## DIAGRAMME DE CAS D’UTILISATION

Le diagramme de cas d’utilisation vise à représenter les interactions possibles entre les utilisateurs (ou *acteurs*) et le système de sauvegarde automatisée. Il permet d’avoir une vue d’ensemble fonctionnelle des différentes actions offertes par l’application, ainsi que du périmètre fonctionnel couvert.

Dans ce modèle, un seul acteur principal est identifié : **l’utilisateur final**, qui interagit avec l’interface pour déclencher ou surveiller les opérations. Les cas d’utilisation sont les suivants :

* **Créer un nouveau travail de sauvegarde** : L’utilisateur configure les paramètres d’une nouvelle tâche (nom, source, destination, type de sauvegarde), en appelant indirectement la méthode CreerTravauxSauvegarde.
* **Lancer une sauvegarde** : Permet de démarrer l’exécution d’un travail existant. Cela implique l’instanciation d’un objet Travaux et l’appel de la méthode EffectuerSauvegarde.
* **Consulter les logs** : Accès aux fichiers journaux générés par la classe Logs.Journalier. L’utilisateur peut y visualiser les opérations précédentes, les erreurs rencontrées ou les statistiques de transfert.
* **Suivre l’état en temps réel** : Cette fonctionnalité repose sur les classes observatrices comme ListenerEtatTransfertActuel, qui fournissent un retour dynamique sur la progression de la sauvegarde (nombre de fichiers, pourcentage, statut).
* **Être notifié en cas de changement** : Grâce au ListenerChangement, l’utilisateur peut recevoir des alertes ou visualiser les modifications enregistrées entre deux sauvegardes (nouveaux fichiers, modifications, suppressions).



Ce diagramme structure donc les besoins fonctionnels du projet, en identifiant clairement les services offerts à l'utilisateur tout en restant découplé de l’implémentation technique sous-jacente.

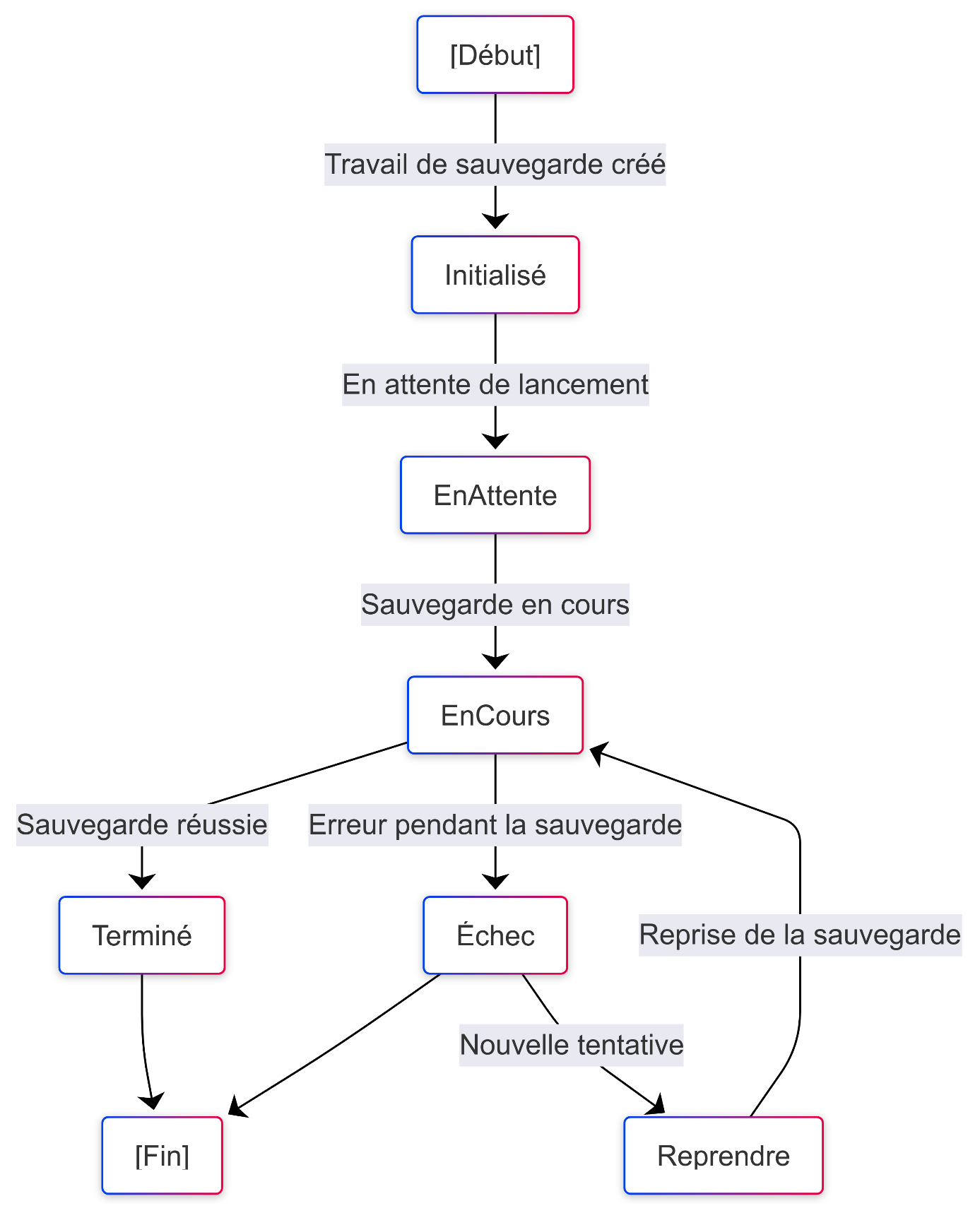
## DIAGRAMME DE CYCLE DE VIE (OU D’ACTIVITÉ)

Le diagramme de cycle de vie présente une vue dynamique du fonctionnement du système, en décrivant la séquence des activités lors du lancement d’un travail de sauvegarde. Il illustre le **flux de contrôle** entre les différentes étapes et les décisions prises à chaque étape clé.

Le processus s’articule de la manière suivante :

1. **Initialisation de la sauvegarde** : L’utilisateur déclenche une opération via l’interface, ce qui conduit à l’appel de la méthode CreerTravauxSauvegarde.
2. **Configuration des paramètres** : Le système initialise les objets nécessaires, configure les chemins source et cible, puis définit le type de sauvegarde (complète ou différentielle).
3. **Lancement de la sauvegarde** : La méthode EffectuerSauvegarde est invoquée. À ce stade, des observateurs sont activés (ListenerEtatTransfertActuel, ListenerChangement) pour suivre l’évolution du transfert.
4. **Transfert de fichiers** : Le système copie les fichiers de la source vers la cible, en tenant compte du type de sauvegarde choisi (copie complète ou uniquement les fichiers modifiés).
5. **Enregistrement des événements** : Les changements détectés sont capturés par les listeners et envoyés vers Logs.Journalier qui les consigne de manière persistante.
6. **Vérification de l’état final** : Une fois la sauvegarde terminée, un état de fin est généré et enregistré. Le système peut alors notifier l’utilisateur d’un éventuel succès ou échec.
7. **Fin du processus** : Le travail est marqué comme terminé. Le système est prêt à recevoir une nouvelle demande.

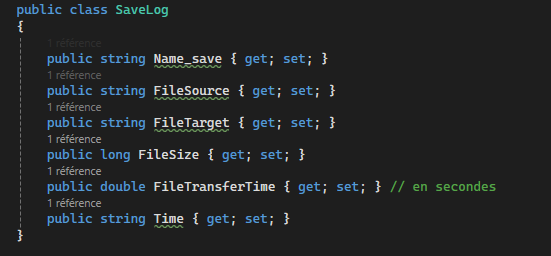
Ce diagramme met en évidence la logique interne du programme, notamment la synchronisation entre le transfert de données, la collecte d’état, et la journalisation. Il permet de mieux comprendre les transitions et les dépendances internes entre les différentes composantes du système.



# Présentation du code

Le code est modifié et exécuté à l’aide de Visual Studio, comme expliqué plus haut, on utilise le .net 8.0.  
En premier lieu, on crée un fichier dll (dynamic link library) qui va contenir le code et les classes qui va s’occuper de générer les logs journaliers et d’état, divisé dans deux fichiers classe « Daily\_log » et « State\_log ».

En premier lieu pour Daily Log :

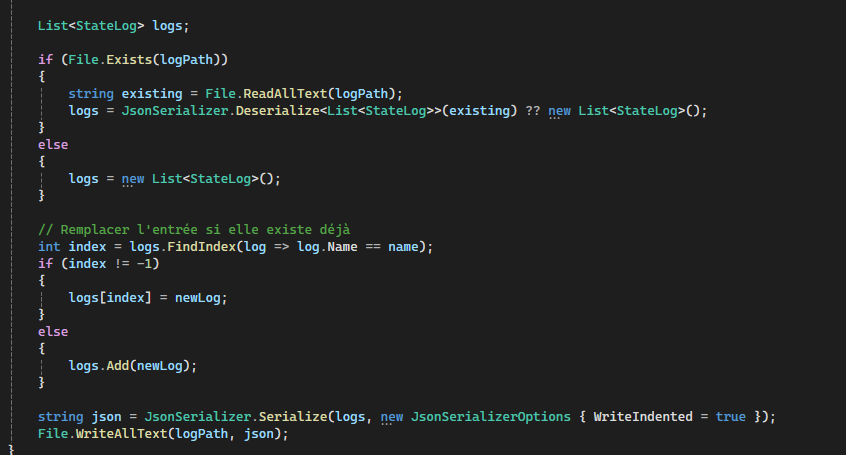


On crée la classe SaveLog, et on lui défini les différentes variables avec leur type string, long etc…

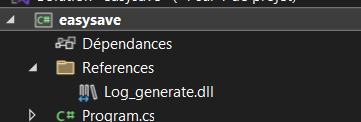
Une image contenant texte, capture d’écran, logiciel

Le contenu généré par l’IA peut être incorrect.

Ensuite on crée une fonction qui va générer dans appdata/EasySave/day\_Logs un fichier json nommé par rapport à la date du jour.  
Ensuite mettre les variables dans une nouvelle instance puis le mettre dans le fichier Log.

Dans State, il y’a peu prêt la même chose, cependant elle va check si le Nom\_Save existe et va l’override sinon le rajouter : 

Et ensuite dans l’arborescence, on le met dans un dossier référence, et on peut l’importer dans le programme.



Dans program.cs, on va gérer l’interface utilisateur avec la langue qui est géré à l’aide d’un dictionnaire :

Une image contenant texte, capture d’écran, Police, menu

Le contenu généré par l’IA peut être incorrect.

Et donc va afficher le texte par rapport à la langue choisit.



Ensuite pour l’affichage et le choix, avec le switch, tu vas pouvoir en tapant par exemple 3 appeler la fonction Path\_saver et permettra d’appeler d’autre fonction qui gère la création de fonction. Et permet à l’utilisateur d’accéder aux différentes parties.

Une image contenant texte, capture d’écran, logiciel, Police

Le contenu généré par l’IA peut être incorrect.

La fonction Display\_save, elle va afficher les backups déjà crée sinon afficher un message d’error. Manage\_save, va check si il y’a au minimum une sauvegarde et va ensuite proposer laquelle supprimer.

Une image contenant texte, capture d’écran, logiciel

Le contenu généré par l’IA peut être incorrect.

Show\_backup() lui va afficher le nom de toute les sauvegardes qui sont enregistré à travers une instance de list. Delete\_Save() va chercher en premier lieu la Save qui a ce nom et si elle existe va la supprimer et supprimer aussi le dossier s’il a été déjà copier.

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

Copy\_backup va gérer si dans la sauvegarde, l’élément à copier, c’est un fichier ou un dossier pour permettre la gestion correctement.

Create\_backup sert lors de la création de la sauvegarde et vérifie si le dossier de destination n’est pas un fichier car l’enregistrement ne se ferait pas correctement et va donc l’ajouter dans la list.