Rapport – Architecture Logicielle

Matisse SENECHAL

23 avril 2025

Table des matières

1 Introduction

Ce mini-projet vise à concevoir et implémenter un outil de synchronisation de fichiers, s'inspirant de solutions existantes telles que Rsync ou Unison, tout en adoptant une approche simplifiée. Son fonctionnement repose essentiellement sur la comparaison des dates de dernière modification des fichiers situés dans deux répertoires distincts, afin de garantir, en sortie, un contenu strictement identique dans les deux emplacements.

Trois applications principales composent l'outil :

- **new-profile** (NewProfileApp) initialise un profil de synchronisation à partir d'un nom et de deux chemins,
- **sync** (SyncApp) exécute la synchronisation bidirectionnelle entre les dossiers,
- **sync-stat** (SyncStatApp) permet de visualiser les détails d'un profil ainsi que son registre de synchronisation.

Tout au long du développement, une importance particulière a été accordée à la modularité, à la lisibilité de l'architecture, à l'utilisation appropriée des patrons de conception, ainsi qu'au respect systématique des principes S.O.L.I.D, assurant ainsi extensibilité, maintenabilité et robustesse du code.

2 Architecture générale

L'application repose sur une architecture modulaire, conçue dès l'origine pour garantir l'extensibilité, la réutilisabilité et la testabilité. Le projet est divisé en onze paquetages aux responsabilités bien délimitées. Ce découpage repose à la fois sur les principes S.O.L.I.D, sur la séparation stricte des préoccupations, et sur une application maîtrisée des patrons de conception les plus adaptés à chaque problématique rencontrée.

Le diagramme de packages UML ci-dessous offre une vue d'ensemble des relations inter-paquetages. Chaque flèche représente une dépendance orientée, respectant le principe d'inversion des dépendances, dans le but d'éliminer les couplages cycliques et de rendre le système facilement évolutif.

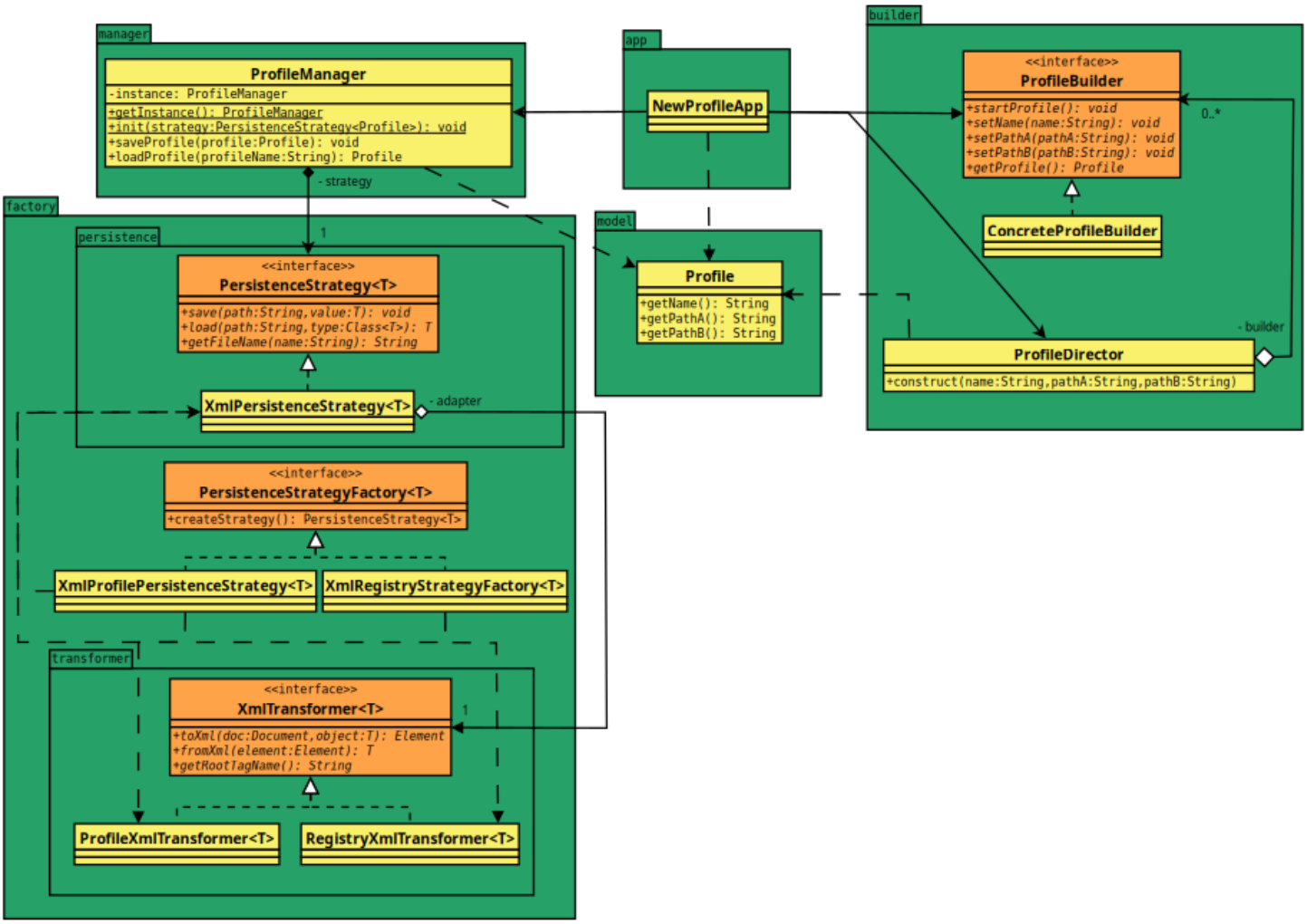


FIGURE 1 – Diagramme de classes du programme `new-profile`, illustrant l'usage des patrons **Builder** (construction d'un profil), **Singleton** (gestionnaire unique), **Strategy** (persistance interchangeable) et **Factory Method** (instanciation encapsulée).

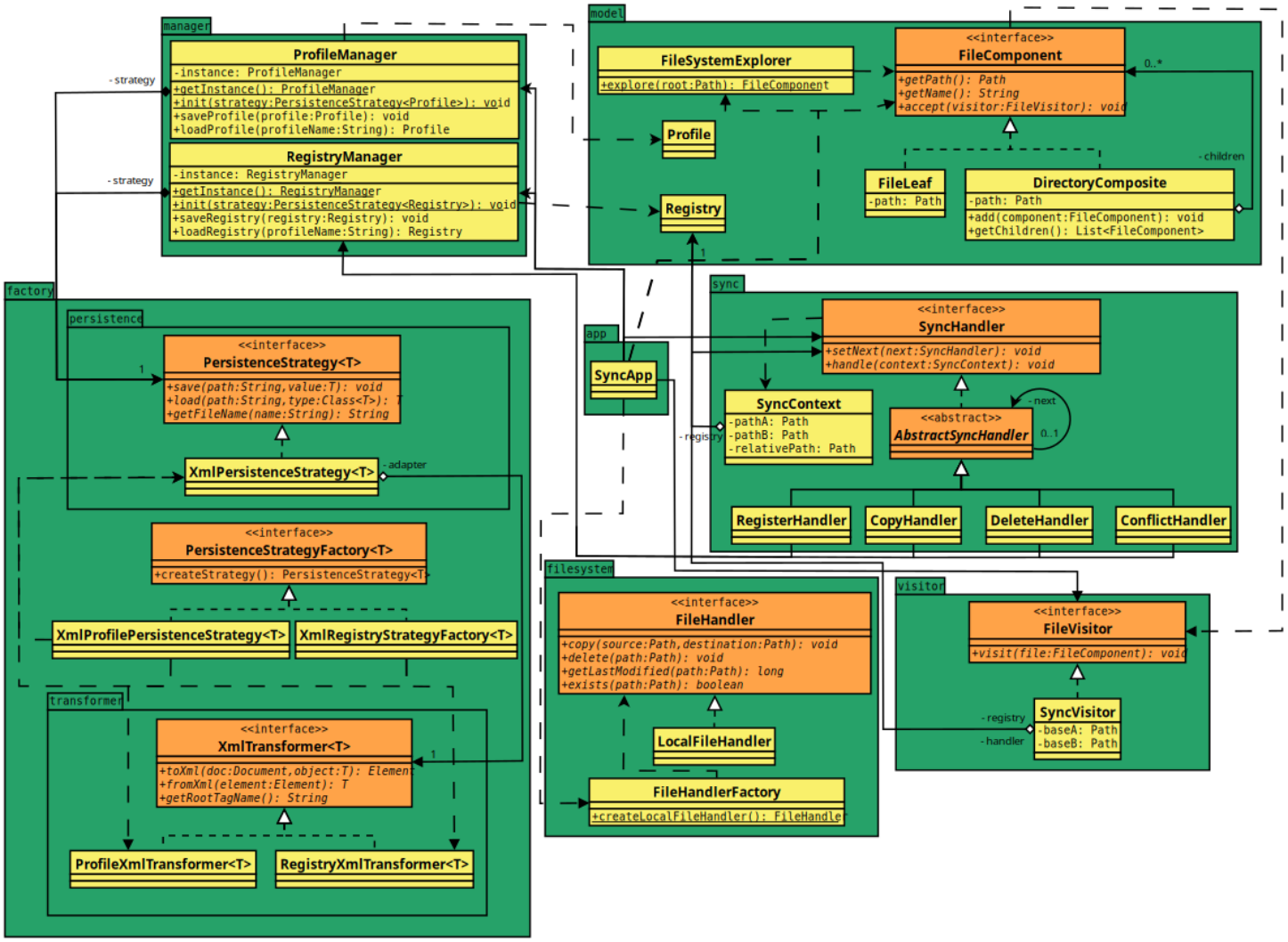


FIGURE 2 – Diagramme de classes du programme `sync`, illustrant les patrons de conception suivants : **Composite** (structure des fichiers), **Visitor** (parcours des arborescences), **Chain of Responsibility** (chaîne de synchronisation), **Singleton** (gestionnaires de profils et registres), **Strategy** (persistance interchangeable) et **Factory Method** (instanciation de `FileHandler` et de la stratégie de persistance).

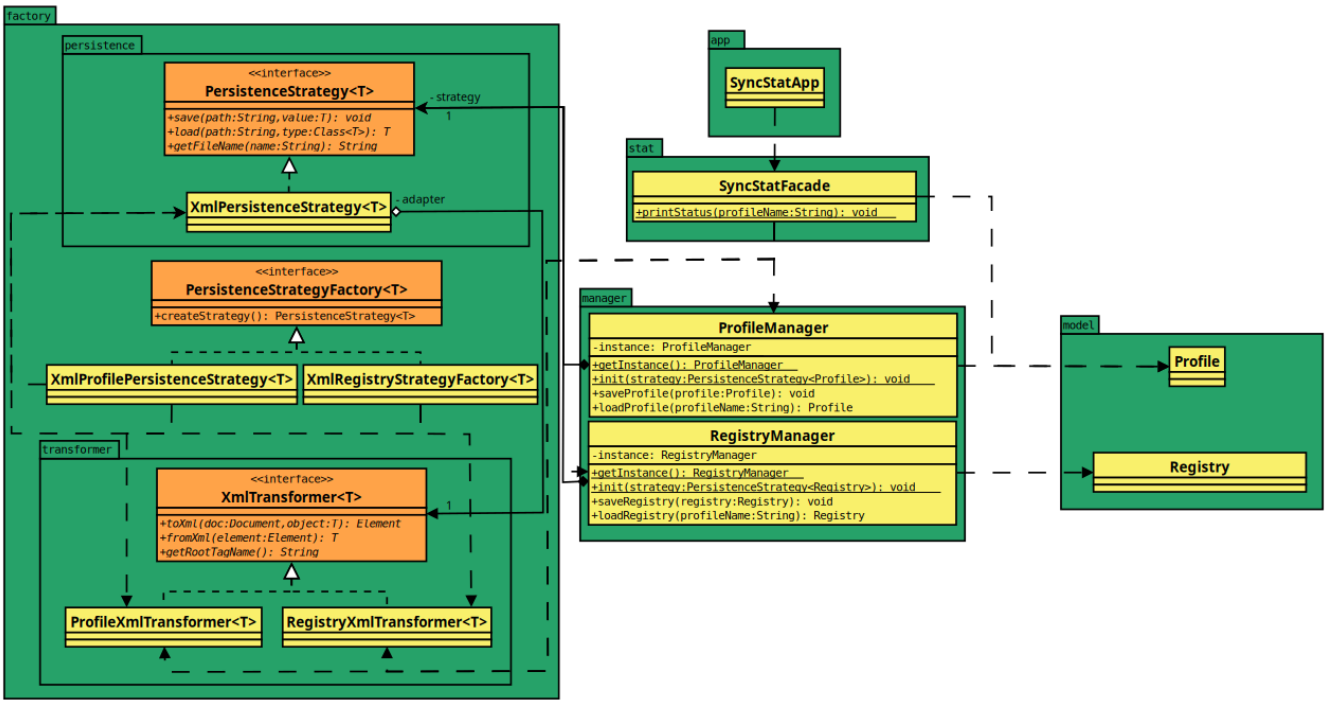


FIGURE 3 – Diagramme de classes du programme `sync-stat`, illustrant l'usage combiné de plusieurs patrons de conception : **Façade** (simplification de l'accès aux données), **Singleton** (gestionnaires de profils et registres), **Factory Method** (instanciation de stratégie de persistance), et **Strategy** (persistance interchangeable).

2.1 Justification du découpage

2.1.1 Séparation des couches fonctionnelles

L'architecture du projet repose sur une stratification claire en couches fonctionnelles, inspirée des modèles d'architecture logicielle classiques (notamment l'architecture hexagonale et l'architecture en couches). Ce découpage vise à faciliter la compréhension, réduire les couplages, et favoriser l'évolution indépendante des composants. Il respecte pleinement le principe de responsabilité unique (SRP), en affectant à chaque couche une préoccupation bien définie.

L'organisation générale s'inspire d'une architecture en couches fonctionnelles :

- **Couche présentation** : app,
- **Couche métier** : sync, visitor, stat, manager,
- **Couche modèle** : model, builder,
- **Couche infrastructure** : persistence, transformer, factory, filesystem.

Ce découpage fonctionnel constitue le socle de la robustesse et de l'évolutivité de l'architecture retenue.

2.1.2 Couplage faible et cohésion forte

L'un des principes fondamentaux respectés dans cette architecture est l'articulation entre couplage faible et cohésion forte, deux notions centrales en conception logicielle.

Le couplage faible signifie que les modules dépendent au minimum les uns des autres, ce qui facilite leur test, réutilisation et évolution indépendante. À l'inverse, la cohésion forte traduit le fait que toutes les classes d'un même paquetage poursuivent un objectif commun bien délimité, renforçant la lisibilité et la maintenabilité. Exemple :

Le paquetage builder illustre parfaitement cette combinaison. Il ne contient que trois classes qui ne sont articulées qu'autour de la construction de Profile :

- ProfileBuilder (interface),
- ConcreteProfileBuilder (implémentation concrète).
- ProfileDirector

Ces trois éléments coopèrent exclusivement à la construction contrôlée et incrémentale d'un objet Profile, en garantissant sa complétude et sa validité. Le paquetage ne dépend d'aucune couche technique (pas de persistance, ni de logique métier), et n'est utilisé que dans un unique contexte : l'initialisation d'un nouveau profil (via NewProfileApp). Il est donc faiblement couplé et fortement cohésif.

2.2 Détail des responsabilités par paquetage

app — Points d'entrée (présentation) :

- **Fonction** : Contient les classesexécutables pour la création de profil (NewProfileApp), la synchronisation (`SyncApp`) et l'affichage de l'état (`SyncStatApp`).
- **Intérêt architectural** : Ce paquetage incarne la couche "interface utilisateur" de l'application, entièrement découplée de la logique métier. Il sert de point d'assemblage, orchestrant les autres sous-systèmes sans logique métier propre.
- **Principes SOLID** :
  - **SRP** : chaque classe correspond à un cas d'usage spécifique.
  - **OCP** : permet d'ajouter d'autres interfaces (GUI, REST...) sans modification.
  - **DIP** : dépend uniquement d'abstractions (interfaces, façades, factories).

model – Domaine métier :

- **Fonction** : Définit les entités métier fondamentales (`Profile`, `Registry`) et la structure arborescente des fichiers (`FileComponent`, `DirectoryComposite`, `FileLeaf`).
- **Intérêt architectural** : Représentant le noyau fonctionnel de l'application, ce paquetage est totalement indépendant des couches techniques. Il applique le patron Composite pour unifier le traitement des fichiers et dossiers, et garantit une immutabilité partielle sur les objets critiques (ex. `Profile`), assurant leur stabilité.
- **Principes SOLID** :
  - **SRP** : chaque classe est centrée sur une responsabilité métier bien définie.
  - **OCP** : le modèle est ouvert à l'extension (ajout d'un autre type de `FileComponent`, par ex.), sans nécessiter de modification.
  - **DIP** : il ne dépend d'aucune technologie concrète (sérialisation, I/O), ce qui garantit son stabilité et sa testabilité.

builder – Construction sécurisée des objets :

- **Fonction** : Encadre la création contrôlée des objets `Profile` en séparant les étapes de construction de leur représentation finale. Ce paquetage s'articule autour du patron Builder, avec (`ProfileBuilder`, `ConcreteProfileBuilder`, et `ProfileDirector`).
- **Intérêt architectural** : Il centralise la logique d'instanciation de profils tout en assurant leur validité (nom, chemins non nuls) avant persistance. Ce découplage garantit que les profils sont consistants et prêts à l'emploi, sans exposer au client les détails de construction. Le directeur joue un rôle crucial d'abstraction, masquant les étapes internes et facilitant les extensions futures (ex. profils enrichis).
- **Principes SOLID** :
  - **SRP** : chaque classe joue un rôle distinct dans le processus de construction (interface, implémentation, orchestration).
  - **OCP** : le processus peut être enrichi par d'autres types de constructeurs (ex. builder interactif ou REST) sans toucher au code existant.
  - **LSP** : toute classe implémentant `ProfileBuilder` peut être utilisée par le `ProfileDirector` sans rupture fonctionnelle.

manager – Gestion centralisée des entités persistées :

- **Fonction** : Fournit des services de lecture/écriture pour Profile et Registry, via `ProfileManager` et `RegistryManager`.

– **Intérêt architectural** : Implémente des Singletons configurables, initialisés avec une stratégie de persistance injectée, permettant une centralisation du stockage tout en conservant la flexibilité du format.

– **Principes SOLID** :

  - **SRP** : chaque classe gère une seule entité persistante.
  - **OCP** : support d'autres formats sans modifier les gestionnaires.
  - **DIP** : dépend d'une abstraction `PersistenceStrategy<T>`.

persistence – Abstraction de la sérialisation :

– **Fonction** : Définit une stratégie générique de persistance (`PersistenceStrategy<T>`) et son implémentation XML (`XmlPersistenceStrategy<T>`).

– **Intérêt architectural** : Sépare la logique métier des détails de sérialisation. Peut facilement être étendue à d'autres formats (JSON, YAML...).

– **Principes SOLID** :

  - **SRP** : responsabilité unique de sérialiser/désérialiser.
  - **OCP** : extensible à de nouveaux formats.
  - **LSP** : toute stratégie peut être utilisée sans altérer le fonctionnement global.

transformer – Transformation XML :

– **Fonction** : Contient les composants de conversion entre objets métier (`Profile`, `Registry`) et DOM XML (`ProfileXmlTransformer`, `RegistryXmlTransformer`).

– **Intérêt architectural** : Sert uniquement à isoler la logique de transformation.

– **Principes SOLID** :

  - **SRP** : chaque classe est responsable d'un seul type de transformation.
  - **OCP** : chaque nouveau type peut définir sa propre logique.
  - **DIP** : utilisé par des abstractions (ex. `XmlPersistenceStrategy`).

factory – Génération des stratégies :

– **Fonction** : Crée dynamiquement les stratégies de persistance adaptées via le patron Factory Method (`XmlProfileStrategyFactory`, `XmlRegistryStrategyFactory`).

– **Intérêt architectural** : Permet l'instanciation différée et encapsulée de composants techniques, favorisant la flexibilité et la maintenance.

– **Principes SOLID** :

  - **SRP** : chaque factory produit une seule famille de stratégies.
  - **OCP** : ajout d'un format = ajout d'une factory, sans modification ailleurs.

- **DIP** : le client utilise une abstraction, sans connaître la classe concrète.

filesystem – Abstraction du système de fichiers :

– **Fonction** : Définit une interface d'accès au système de fichiers (`FileHandler`) avec une implémentation locale (`LocalFileHandler`).

– **Intérêt architectural** : Prépare l'extension vers d'autres protocoles (WebDAV, FTP...), sans modification de la logique métier.

– **Principes SOLID** :

- **SRP** : chaque classe gère des opérations bas niveau sur les fichiers.
- **OCP** : support facile de systèmes distants via de nouvelles implémentations.
- **DIP** : les traitements métier s'appuient sur FileHandler, pas sur NIO directement.

sync – Logique de synchronisation :

– **Fonction** : Implémente la chaîne de traitement des fichiers via le patron Chain of Responsibility (`RegisterHandler`, `CopyHandler`, `DeleteHandler`, `ConflictHandler`).

– **Intérêt architectural** : Permet d'isoler et de combiner différentes responsabilités de synchronisation de manière modulaire, tout en facilitant les extensions futures.

– **Principes SOLID** :

- **SRP** : un handler = une règle de synchronisation.
- **OCP** : ajout d'un traitement = ajout d'un handler.
- **DIP** : ne dépend que du contexte de synchronisation (`SyncContext`).

visitor – Parcours de l'arborescence :

– **Fonction** : Contient les visiteurs appliquant des traitements sur les composants arborescents (`FileVisitor`, `SyncVisitor`).

– **Intérêt architectural** : Découple la structure (model) des comportements applicables, permettant d'ajouter des opérations sans modifier les classes du modèle.

– **Principes SOLID** :

- **SRP** : chaque visiteur implémente une opération distincte.
- **OCP** : nouvelles opérations = nouveaux visiteurs.
- **LSP** : tout visiteur peut être substitué sans rupture.

stat – Consultation d'état (façade) :

– **Fonction** : Fournit une interface simplifiée d'accès à l'état d'un profil (`SyncStatFacade`).

- **Intérêt architectural** : Masque la complexité interne (chargement de profil, lecture du registre) derrière une API minimale.
- **Principes SOLID** :
  - **SRP** : fournit une seule fonctionnalité : afficher un état.
  - **OCP** : le comportement interne peut évoluer sans impact côté client.
  - **DIP** : encapsule les appels à `ProfileManager` et `RegistryManager`.

2.3 Une architecture extensible

L'un des objectifs essentiels de cette architecture est de préparer l'évolution du projet :

- Le système de fichiers peut évoluer vers une implémentation distante (WebDAVFileHandler).
- Les formats de fichiers (.json, .bin, .yaml) peuvent être ajoutés sans toucher aux gestionnaires.
- De nouveaux traitements (LoggingVisitor, StatisticsHandler) peuvent être ajoutés sans casser la chaîne ou la hiérarchie Composite.

3 Patrons de conception utilisés

Afin de garantir une architecture claire, extensible et modulaire, plusieurs patrons de conception ont été appliqués. Chacun répond à une problématique précise de conception, comme la séparation des responsabilités, l'encapsulation du comportement ou encore l'inversion de dépendance.
Les sections suivantes détaillent, pour chaque patron : sa motivation, son intégration dans l'architecture, et sa représentation UML partielle.

3.1 Builder – Construction contrôlée d'un objet complexe

Motivation :
Le profil de synchronisation (Profile) est un objet simple en apparence, mais dont la construction nécessite la présence obligatoire de trois attributs (nom, chemin A, chemin B). Pour éviter toute instanciation partielle ou incohérente, le patron Builder a été employé.
Cela permet également de :

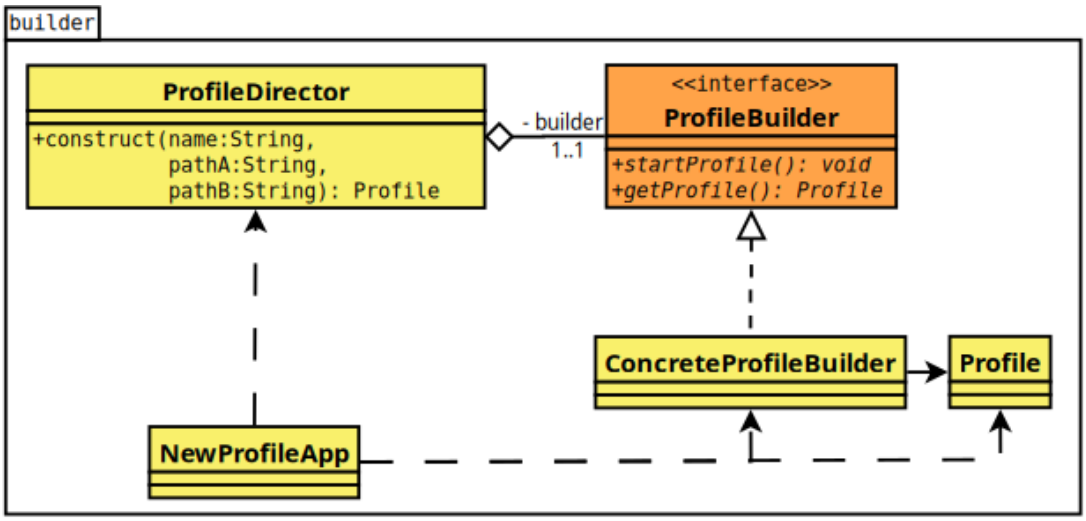
- séparer la logique de construction de l'objet lui-même,
- faciliter la validation des champs,
- préparer l'extension future (ajout de stratégie, options, etc.).

Implémentation :

Rôle dans le patron	Classe correspondante
Builder	`ProfileBuilder` (interface)
ConcreteBuilder	`ConcreteProfileBuilder`
Product	`Profile`
Client	`NewProfileApp`

Table 1 – Implémentation du patron Builder dans l'application

Diagramme UML :



3.2 Composite – Représentation arborescente des fichiers

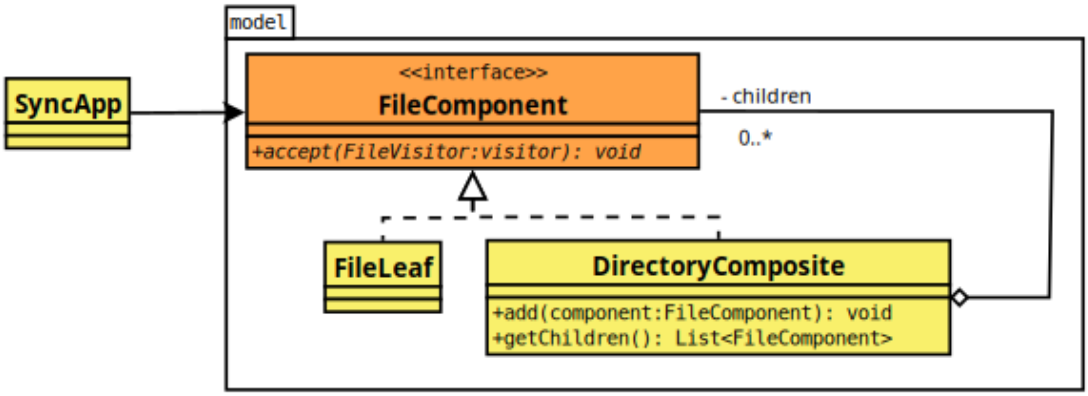
Motivation :
La structure d'un répertoire est par nature récursive : un dossier peut contenir d'autres dossiers ou des fichiers. Pour modéliser cela proprement, le patron Composite permet de manipuler uniformément les éléments de type FileComponent, qu'ils soient feuilles (FileLeaf) ou composites (DirectoryComposite).

Implémentation :

Rôle dans le patron	Classe correspondante
Component	`FileComponent` (interface)
Leaf	`FileLeaf`
Composite	`DirectoryComposite`
Client	`FileSystemExplorer, SyncVisitor`

Table 2 – Implémentation du patron Composite dans l'application

Diagramme UML :



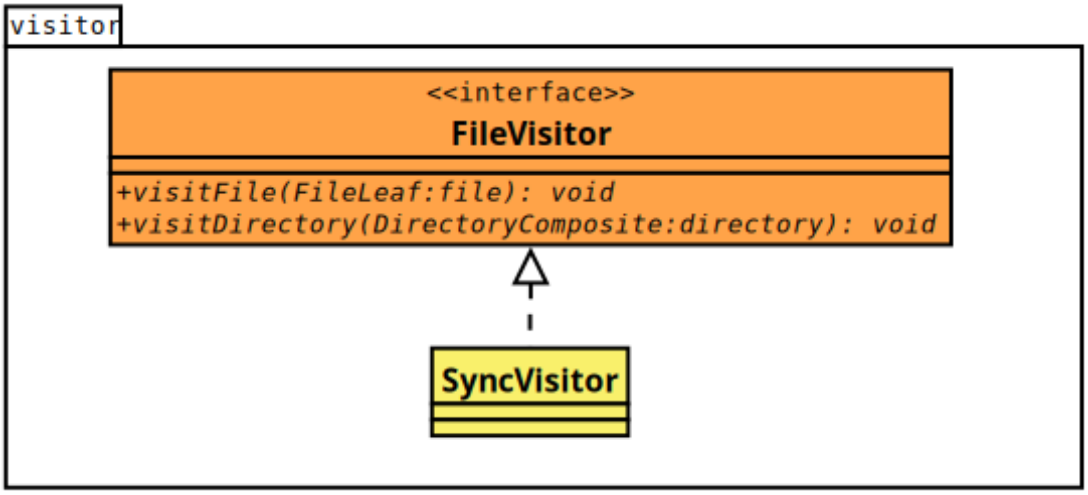
3.3 Visitor – Application d'un traitement aux composants d'une arborescence

Motivation :
Grâce à Visitor, il est possible d'ajouter des traitements sur la structure Composite sans modifier les classes de modèle. Cela permet une parfaite séparation entre structure et comportement : chaque traitement devient un visiteur indépendant.

Rôle dans le patron	Classe correspondante
Visitor	`FileVisitor` (interface)
ConcreteVisitor	`SyncVisitor`
Element	`FileComponent`
ConcreteElements	`FileLeaf, DirectoryComposite`

TABLE 3 – Implémentation du patron Composite dans l'application

Diagramme UML :



3.4 Chain of Responsibility – Enchaînement modulaire des traitements

Motivation :

La synchronisation de fichiers implique plusieurs étapes successives :

- Enregistrement
- Copie
- Suppression
- Résolution de conflit

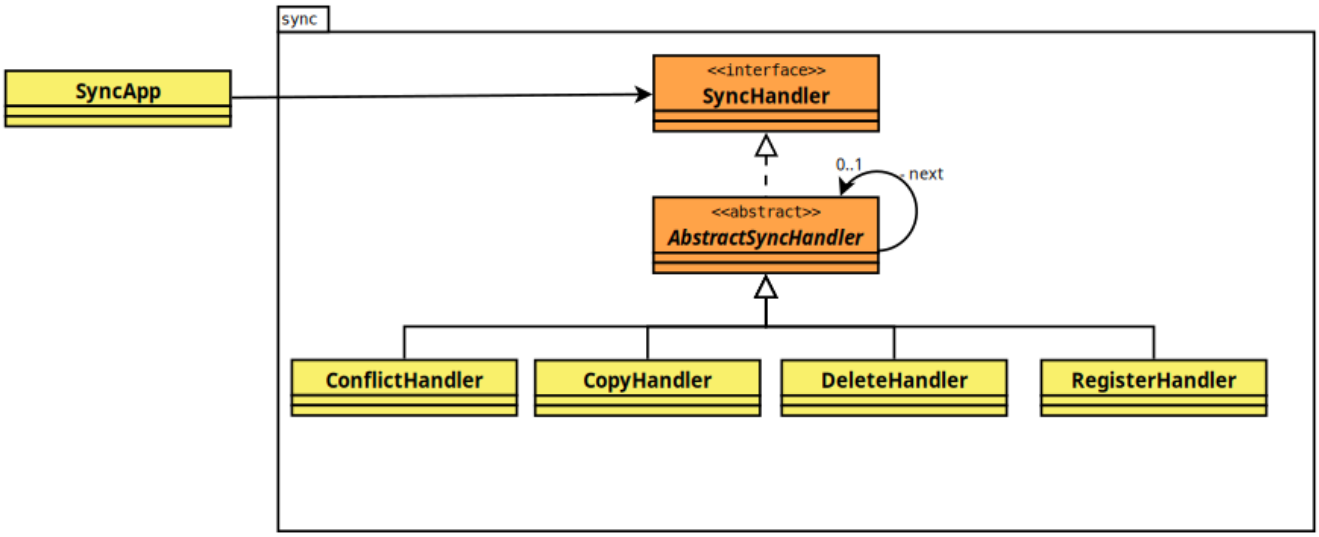
Ces étapes doivent pouvoir s'enchaîner dynamiquement. Le patron Chain of Responsibility offre cette souplesse tout en garantissant l'encapsulation de chaque responsabilité.

Implémentation :

Rôle dans le patron	Classe correspondante
Handler	`SyncHandler` (interface)
AbstractHandler	`AbstractSyncHandler`
ConcreteHandlers	`RegisterHandler, CopyHandler, DeleteHandler, ConflictHandler`
Client	`SyncApp, SyncVisitor`

TABLE 4 – Implémentation du patron Chain of Responsibility dans l'application

Diagramme UML :



3.5 Façade – Simplification d'un sous-système complexe

Motivation :
L'affichage de l'état d'un profil (syncstat) nécessite l'accès coordonné à plusieurs composants :

- Chargement du profil
- Chargement et lecture du registre
- Affichage formaté

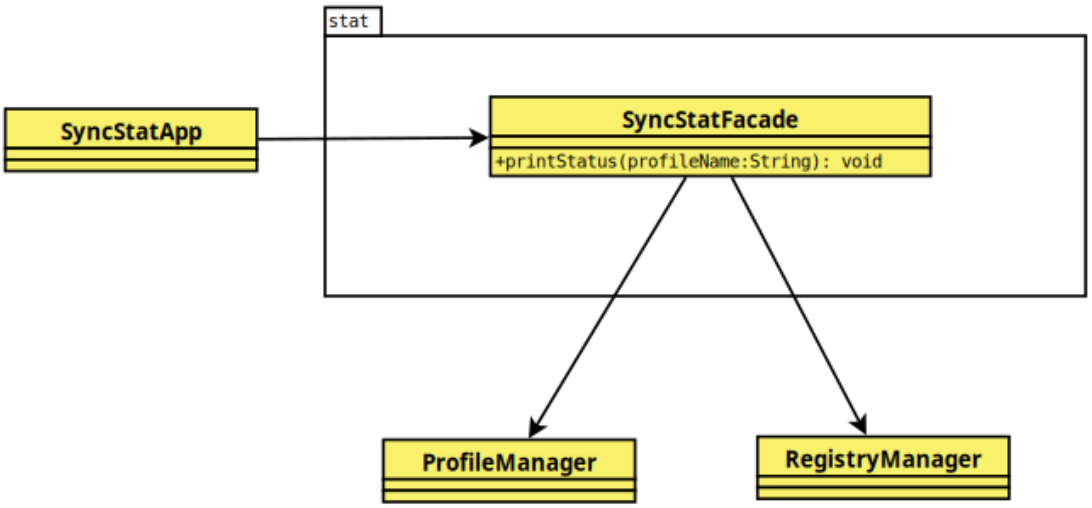
Le patron Façade permet de masquer cette complexité derrière une seule méthode publique.

Implémentation :

Rôle dans le patron	Classe correspondante
Facade	`SyncStatFacade`
Subsystems	`ProfileManager, RegistryManager`
Client	`SyncStatApp`

TABLE 5 – Implémentation du patron Composite dans l'application

Diagramme UML :



3.6 Singleton initialisable – Point d'accès global avec injection tardive

Motivation :
Les gestionnaires ProfileManager et RegistryManager doivent être accessibles globalement, mais tout en permettant l'injection d'une stratégie de persistance personnalisée. Cela a motivé l'usage d'un singleton non instancié d'emblée, mais initialisable par init(...).

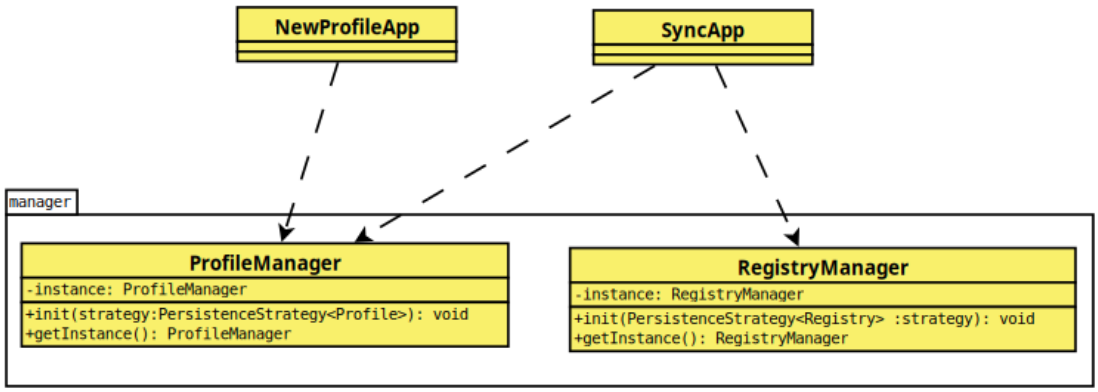
Implémentation :

Rôle dans le patron	Classe correspondante
Singleton (paresseux)	`ProfileManager, RegistryManager`
Stratégie injectée	`XmlPersistenceStrategy via Factory`

TABLE 6 – Implémentation du patron Singleton dans l'application

Cette approche respecte le principe d'inversion de dépendances (DIP) tout en offrant la souplesse nécessaire aux tests ou à l'extension.

Diagramme UML :



3.7 Strategy – Encapsulation du mécanisme de persistance

Motivation :

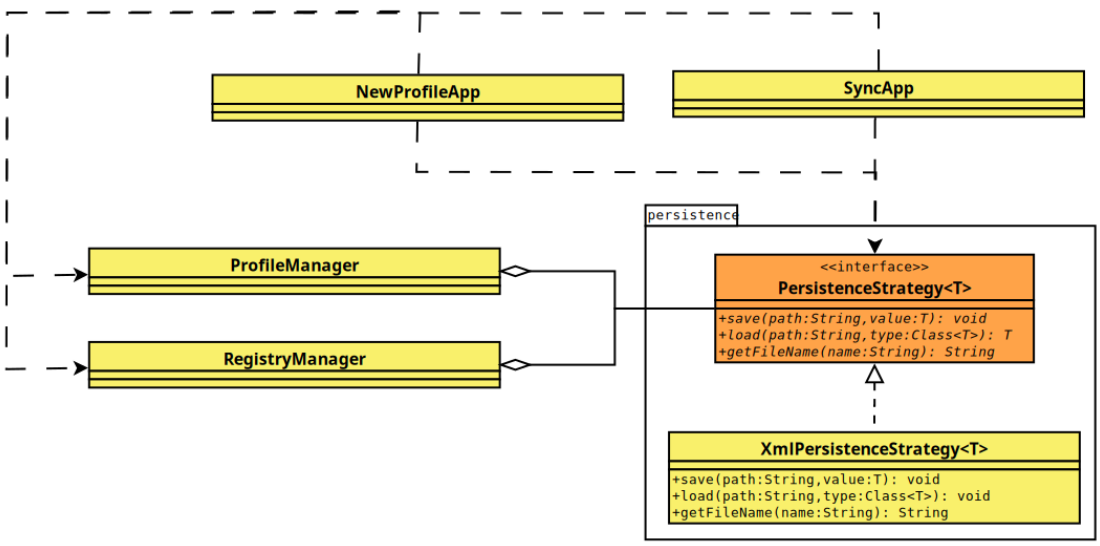
L'un des objectifs principaux du projet est de permettre à terme la flexibilité dans le format de persistance (XML, JSON, binaire, etc.) des profils et des registres. Afin de répondre à cette exigence d'extensibilité sans modifier les classes métiers ProfileManager ou RegistryManager, le patron Strategy a été mis en œuvre.
Ainsi, le comportement de sauvegarde/chargement est injecté dynamiquement, via une interface commune.

Implémentation :

Rôle dans le patron	Classe correspondante
Strategy	`PersistenceStrategy<T>`
ConcreteStrategy	`XmlPersistenceStrategy<T>`
Context	`ProfileManager, RegistryManager`

TABLE 7 – Implémentation du patron Strategy dans l'application

Diagramme UML :



3.8 Factory Method – Instanciation dynamique de stratégies de persistance

Motivation :

L'utilisation d'une stratégie de persistance différente selon le type de données manipulées (profils, registres...) nécessite une fabrique spécialisée par type, qui instancie la bonne implémentation sans dupliquer le code métier.

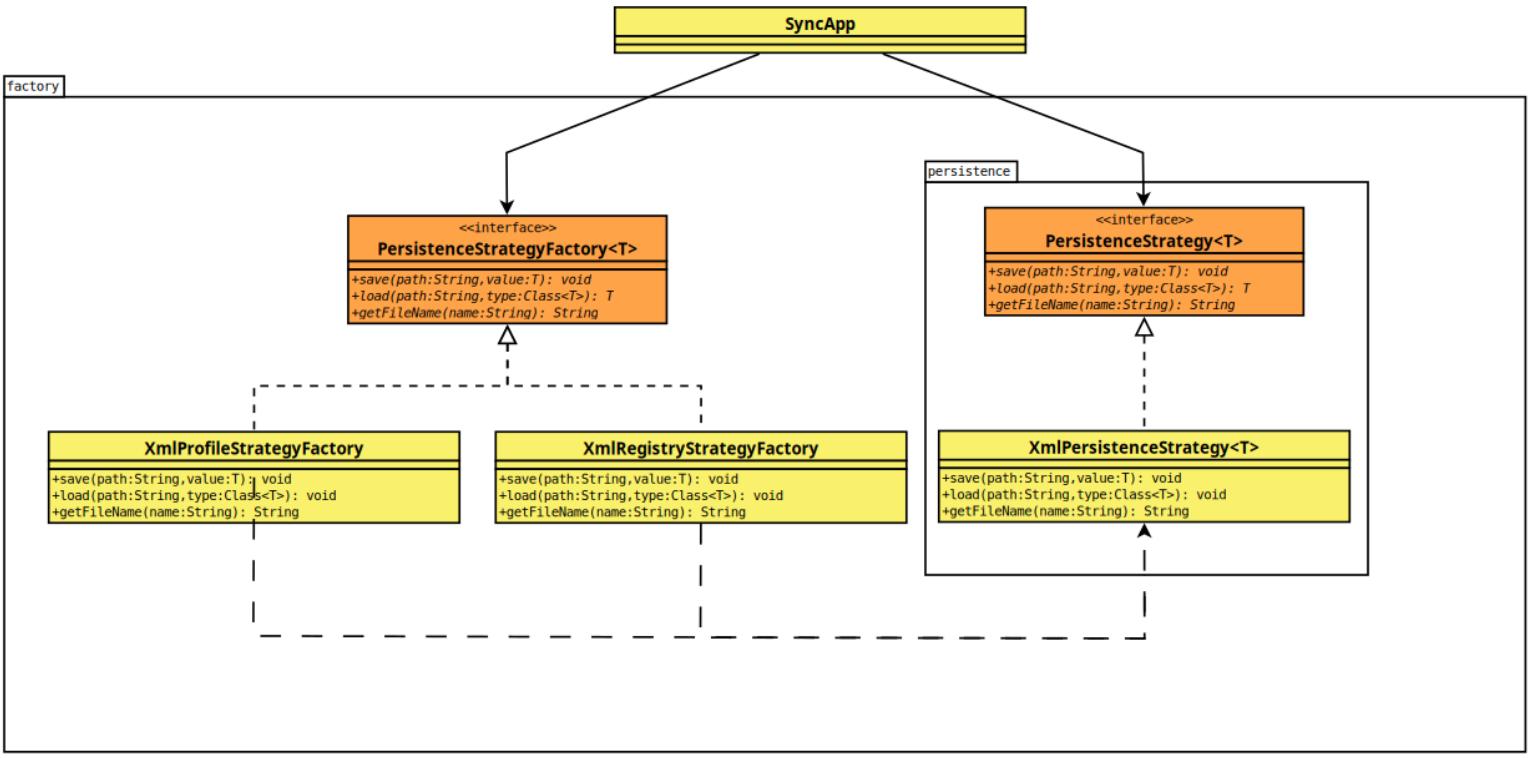
Le patron Factory Method permet ici de déléguer la création des objets PersistenceStrategy<T> aux fabriques concrètes, centralisant les règles de création.

Implémentation :

Rôle dans le patron	Classe correspondante
Creator	`PersistenceStrategyFactory<T>`
ConcreteCreator	`XmlProfileStrategyFactory, XmlRegistryStrategyFactory`
Product	`PersistenceStrategy<T>`

TABLE 8 – Implémentation du patron Factory Method dans l'application

Diagramme UML :



4 Patrons envisagés mais non retenus

Lors de la phase de conception, plusieurs patrons de conception supplémentaires ont été étudiés afin de répondre aux problématiques de modularité, d'extensibilité et de cohésion logicielle. Cependant, certains d'entre eux ont été écartés après analyse, car leur implémentation ne répondait pas de manière optimale aux contraintes ou complexifiait inutilement l'architecture.

4.1 Observer

Contexte envisagé : permettre à certaines parties du système d'être notifiées lors de la mise à jour du registre (par exemple, pour déclencher une mise à jour d'interface graphique ou un log distant).

Raison du rejet : dans le contexte strictement en ligne de commande du mini-projet, aucune interface ou composant externe n'avait besoin d'être notifié. Le couplage ajouté par ce patron était donc superflu dans cette version simplifiée du logiciel.

5 DTD et structure XML

L'application sérialise les profils de synchronisation ainsi que les registres dans des fichiers XML et est validé à l'aide de sa DTD.

5.1 DTD

5.1.1 profile.dtd

```
<!ELEMENT profile (name, pathA, pathB)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT pathA (#PCDATA)>
<!ELEMENT pathB (#PCDATA)>
```

5.1.2 registry.dtd

```
<!ELEMENT registry (entry*)>
<!ATTLIST registry profile CDATA #REQUIRED>

<!ELEMENT entry (path, timestamp)>
<!ELEMENT path (#PCDATA)>
<!ELEMENT timestamp (#PCDATA)>
```

6 Conclusion

Ce projet a permis de mettre en pratique de manière approfondie les grands principes de l'architecture logicielle enseignés, notamment la modularité, la responsabilité unique, ainsi que l'application concrète de multiples patrons de conception. Bien que simplifié, l'outil de synchronisation obtenu est stable, extensible, et conçu dans une optique réutilisable et maintenable.

Note : Ce rapport est accompagné de la Javadoc complète et de tous les fichiers source dans l'archive finale.

A Annexe

Paquetage	Entité	Responsabilité
transformer	XmlTransformer<T>	Interface générique de conversion entre objet métier et élément DOM XML.
	ProfileXmlTransformer	Transformateur XML pour la classe `Profile`.
	RegistryXmlTransformer	Transformateur XML pour la classe `Registry`.
app	NewProfileApp	Application pour créer et sauvegarder un nouveau profil.
	SyncApp	Application principale exécutant la synchronisation bidirectionnelle.
	SyncStatApp	Application affichant le statut d'un profil et de son registre.
builder	ProfileBuilder	Interface définissant les étapes de création d'un `Profile`.
	ConcreteProfileBuilder	Implémentation standard avec validation incrémentale.
	ProfileDirector	Orchestrateur des étapes de construction d'un `Profile` via le builder.
sync	SyncHandler	Interface du patron Chain of Responsibility pour la synchronisation.
	AbstractSyncHandler	Classe abstraite pour orchestrer le chaînage des `SyncHandler`.
	RegisterHandler	Enregistre un nouveau fichier dans le registre et déclenche une copie.
	CopyHandler	Compare deux fichiers et copie celui le plus récent.
	DeleteHandler	Supprime les fichiers supprimés depuis la dernière synchronisation.
	ConflictHandler	Gère les conflits de modification en consultant l'utilisateur.
	SyncContext	Objet transportant les chemins et le registre pour la synchronisation.
factory	PersistenceStrategyFactory<T>	Interface de fabrique de stratégie de persistance.
	XmlProfileStrategyFactory	Fabrique une stratégie XML pour les profils.

Paquetage	Entité	Responsabilité
	XmlRegistryStrategyFactory	Fabrique une stratégie XML pour les registres.
filesystem	FileHandler	Interface d'abstraction pour manipuler les fichiers (copie, suppression...).
	LocalFileHandler	Implémentation locale de `FileHandler` utilisant NIO.
	FileHandlerFactory	Fabrique de gestionnaires de fichiers (actuellement local uniquement).
manager	ProfileManager	Singleton gérant la persistance des profils via une stratégie.
	RegistryManager	Singleton gérant les registres de synchronisation.
model	Profile	Représente un profil de synchronisation (nom, dossier A, dossier B).
	Registry	Représente le registre des fichiers synchronisés avec leur date.
	FileComponent	Interface Composite pour un fichier ou répertoire.
	FileLeaf	Représente un fichier (feuille dans le modèle Composite).
	DirectoryComposite	Représente un répertoire contenant des composants.
	FileSystemExplorer	Génère une arborescence Composite depuis un dossier réel.
persistence	PersistenceStrategy<T>	Interface abstraite de persistance (sauvegarde et chargement).
	XmlPersistenceStrategy	Implémentation de stratégie de persistance XML avec gestion DTD.
stat	SyncStatFacade	Façade permettant d'afficher rapidement les informations d'un profil.
visitor	FileVisitor	Interface du patron Visitor pour les composants de fichiers.
	SyncVisitor	Visiteur effectuant la synchronisation sur chaque composant.