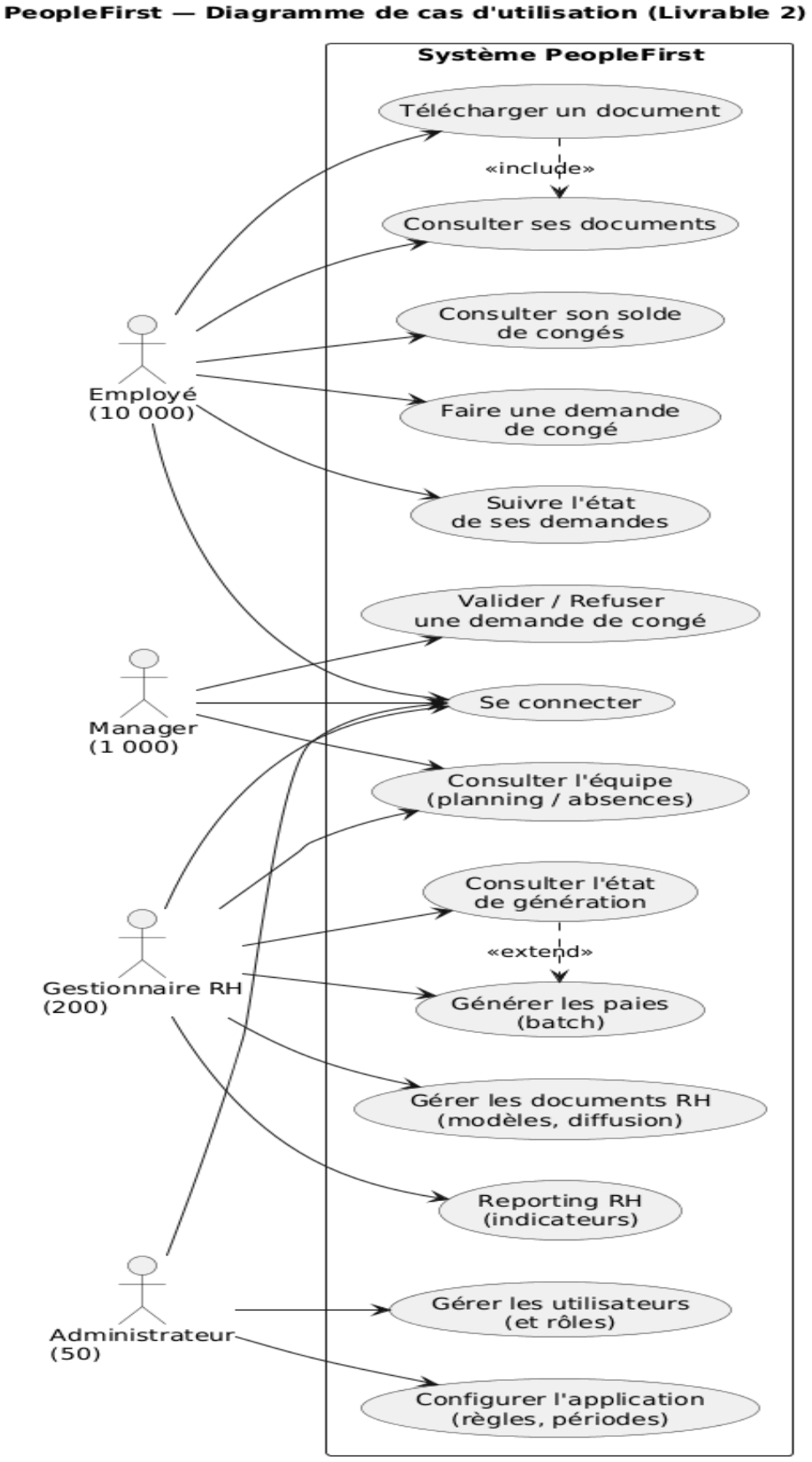
Livrable 1 — Conception détaillée

Modélisation UML

- **Diagramme de cas d'utilisation**



- **Diagrammes de classes**

Voir le fichier DiagrammeDeClasse.png dans le repo github.
https://github.com/herve-beziat/PeopleFirst/blob/main/conception-saas-rh/diagrammes/DiagrammeDeClasses.png

Explication du diagramme de classes — Entités métier principales

Le diagramme de classes présente les **entités métier centrales** du système PeopleFirst.
Il modélise les données nécessaires pour répondre aux principaux cas d'utilisation : **gestion des utilisateurs**, **congés**, **documents RH** et **paie**.

L'objectif n'est pas de représenter l'ensemble du modèle de données exhaustif, mais d'identifier les **concepts métier structurants** et leurs relations.

---

1. Gestion des utilisateurs et des rôles

User

La classe **User** représente le compte technique permettant l'accès à l'application.
Elle contient les informations nécessaires à l'authentification et à l'activation du compte.

Role

La classe **Role** permet de gérer les profils fonctionnels du système (Employé, Manager, Gestionnaire RH, Administrateur).
Un utilisateur peut posséder **plusieurs rôles**, ce qui permet de couvrir des cas comme un manager ayant également des responsabilités RH.

Cette séparation facilite la mise en place d'un **RBAC (Role-Based Access Control)**.

---

2. Profil employé et organisation

EmployeeProfile

La classe **EmployeeProfile** représente le profil RH d'un utilisateur.
Elle regroupe les informations métier propres à un salarié (identité, numéro employé, date d'embauche).

Tous les utilisateurs ne disposent pas nécessairement d'un profil employé (ex. comptes techniques ou administratifs).

Team

La classe **Team** permet de regrouper les employés par équipe ou service.
Elle est notamment utilisée pour la consultation des équipes par les managers et les RH.

Ce découplage entre **User** et **EmployeeProfile** permet de faire évoluer le système sans dépendre directement de l'authentification.

---

3. Gestion des congés et absences

LeaveType

La classe **LeaveType** définit les différents types de congés (CP, RTT, maladie, etc.).
Elle permet de centraliser les règles liées aux types de congés.

LeaveBalance

La classe **LeaveBalance** représente le solde de congés d'un employé pour un type donné et une période donnée.
Elle est liée à un **EmployeeProfile** et à un **LeaveType**.

LeaveRequest

La classe **LeaveRequest** modélise une demande de congé effectuée par un employé.
Elle contient les dates, le nombre de jours demandés et le statut de la demande (soumise, validée, refusée, annulée).

La relation avec **EmployeeProfile** permet d'identifier l'employé demandeur, et une relation optionnelle permet d'identifier le manager ayant validé ou refusé la demande.

Cette modélisation permet de tracer l'ensemble du cycle de vie d'une demande de congé.

---

4. Gestion documentaire

Document

La classe **Document** représente un document RH ou de paie (contrat, fiche de paie, attestation, etc.).
Elle contient les métadonnées du fichier, tandis que le contenu réel est stocké dans un système de stockage objet.

DocumentCategory

La classe **DocumentCategory** permet de classer les documents par type (Paie, RH, Administratif).
Elle facilite la consultation et la gestion documentaire.

Chaque document est rattaché à un **EmployeeProfile**, ce qui garantit que les documents sont bien associés à un salarié précis.

---

5. Gestion de la paie

PayrollRun

La classe **PayrollRun** représente une exécution de paie pour une période donnée (ex. mois).
Elle permet de suivre l'état global de la génération de la paie (en cours, terminée, en erreur).

Payslip

La classe **Payslip** représente une fiche de paie individuelle générée pour un employé lors d'un **PayrollRun**.
Elle contient les montants principaux et la date de génération.

Chaque fiche de paie est liée :

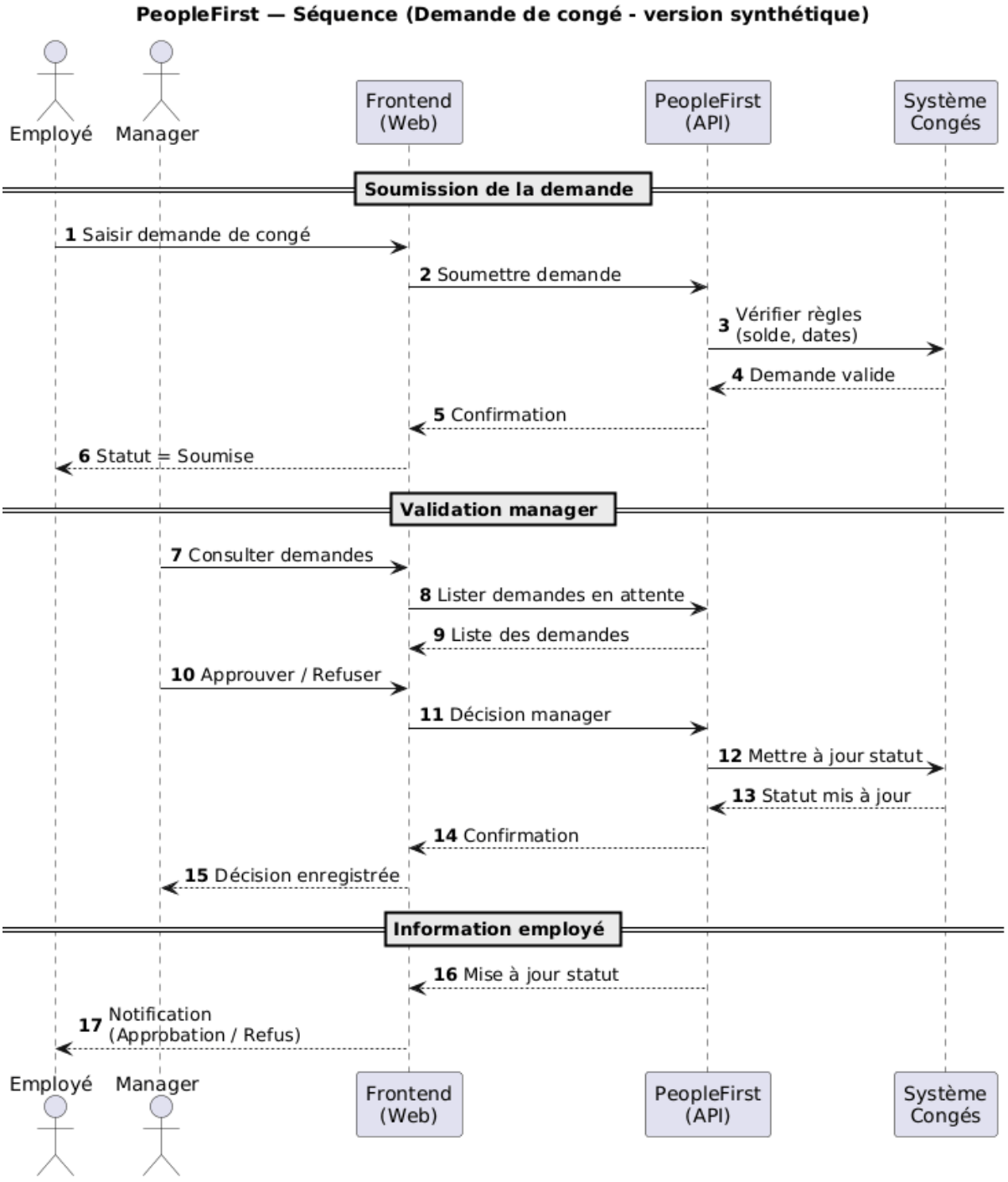
- à un **EmployeeProfile**,
- à un **Document** correspondant au PDF généré.

Cette séparation permet de traiter la paie comme un **processus batch**, tout en conservant une traçabilité individuelle par salarié.

Conclusion

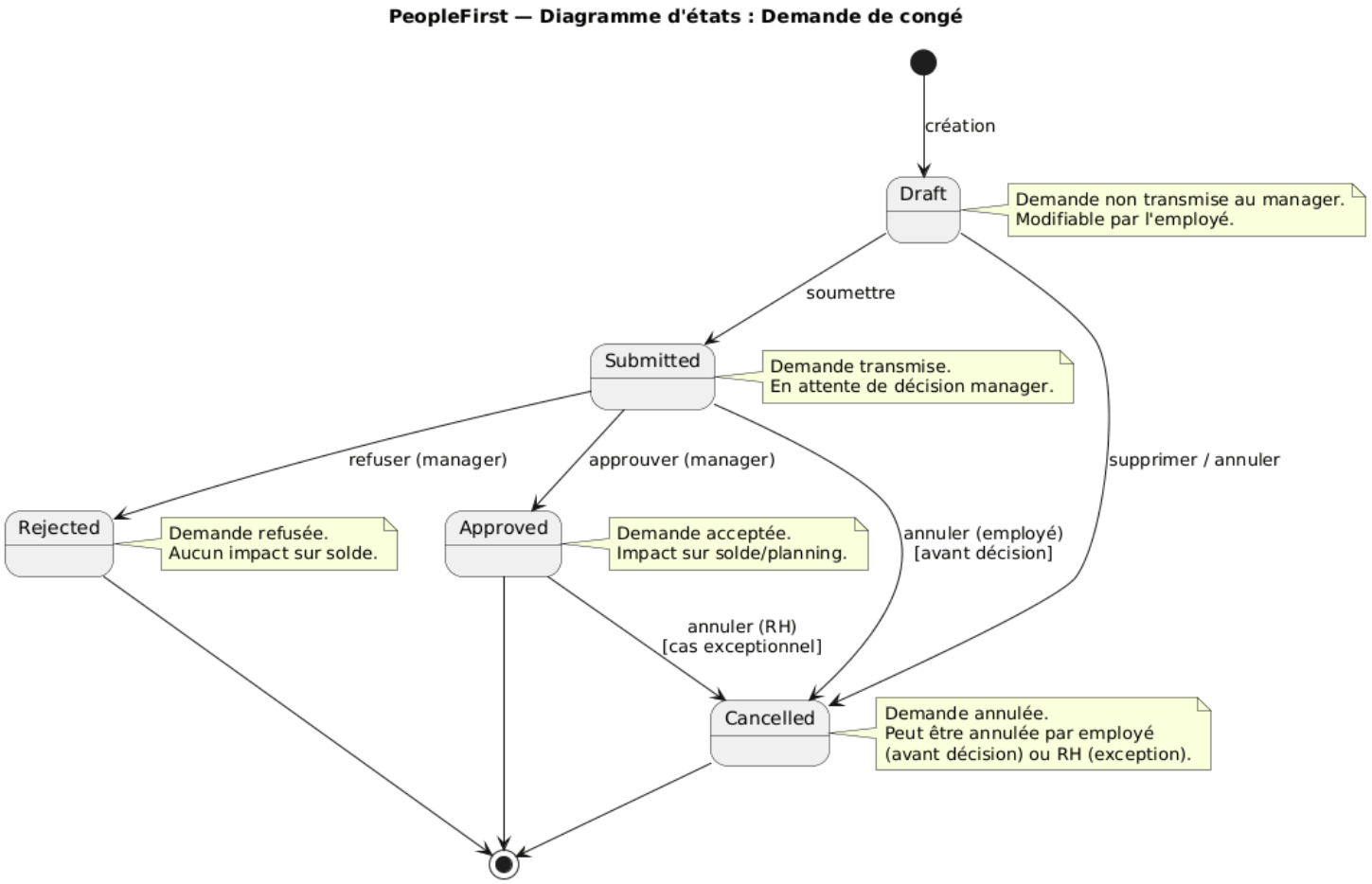
Ce diagramme de classes met en évidence les **concepts métier fondamentaux** de PeopleFirst et leurs relations.
Il constitue une base solide pour la conception de la base de données et pour l'implémentation des cas d'utilisation, tout en restant volontairement synthétique et évolutif.

- **Diagramme de séquence (Workflow d'une demande de congé)**



Ce diagramme de séquence illustre le workflow complet d'une demande de congé.
L'employé soumet une demande via l'interface, qui est contrôlée par le système (cohérence des dates, solde disponible).
La demande est ensuite soumise au manager, qui peut l'approuver ou la refuser.
La décision est enregistrée par le système et communiquée à l'employé, garantissant un suivi clair et traçable.

- # Diagramme d’états d’une demande de congé



Ce diagramme d’états décrit le **cycle de vie d’une demande de congé** au sein du système PeopleFirst.
Il met en évidence les différents statuts possibles d’une demande ainsi que les transitions associées.

Une demande est d’abord créée à l’état **Draft**, ce qui correspond à une demande en cours de saisie par l’employé. Tant qu’elle n’est pas soumise, elle peut être modifiée ou annulée librement.

Lors de la soumission, la demande passe à l’état **Submitted**. Elle est alors transmise au manager et reste en attente de décision. À ce stade, l’employé peut encore annuler sa demande tant qu’aucune décision n’a été prise.

Le manager peut ensuite **approuver** ou **refuser** la demande.
En cas d’approbation, la demande passe à l’état **Approved**, ce qui entraîne un impact sur le solde de congés et le planning.
En cas de refus, la demande passe à l’état **Rejected**, sans modification du solde.

Enfin, l’état **Cancelled** permet de gérer les annulations, soit à l’initiative de l’employé avant décision, soit de manière exceptionnelle par un gestionnaire RH après approbation.

Ce diagramme garantit une **traçabilité claire** des demandes de congé et formalise les règles métier associées à chaque étape.

Principes de conception

Application de principes SOLID

Dans le cadre de la conception de PeopleFirst, plusieurs principes SOLID ont été pris en compte afin de garantir une architecture **maintenable, évolutive et testable**.
Cette section se concentre sur l'application de **deux principes clés**, directement visibles dans les diagrammes fournis.

---

1. Single Responsibility Principle (SRP)

Principe

Le principe de responsabilité unique stipule qu'une classe ou un composant ne doit avoir **qu'une seule raison de changer**.

Application dans PeopleFirst

Le monolithe applicatif est structuré selon une **architecture hexagonale**, clairement visible dans le **diagramme de composants**.
Chaque composant interne remplit une responsabilité précise :

- **API REST (Controllers)** : gestion des requêtes HTTP et des réponses.
- **Application (Use Cases)** : orchestration des cas d'usage (ex. création et validation d'une demande de congé).
- **Domaine** : implémentation des règles métier (solde de congés, cohérence des dates, statuts).
- **Adaptateurs sortants** : interaction avec les systèmes techniques (base de données, stockage de fichiers, file de messages).

Cette séparation permet d'éviter qu'une modification dans un aspect technique (ex. changement de base de données) n'impacte les règles métier, et inversement.

Référence aux diagrammes

- **Diagramme de composants** : séparation explicite entre API, Application, Domaine et Adaptateurs.
- **Diagramme de séquence (demande de congé)** : le frontend déclenche un cas d'usage, sans accéder directement aux règles métier ou à la persistance.

---

2. Dependency Inversion Principle (DIP)

Principe

Le principe d'inversion des dépendances stipule que les modules de haut niveau ne doivent pas dépendre des modules de bas niveau, mais **d'abstractions**.

Application dans PeopleFirst

Le cœur applicatif (Use Cases et Domaine) ne dépend pas directement des technologies utilisées pour la persistance ou la communication.
Les interactions avec la base de données, le stockage de documents ou les notifications sont réalisées via des **interfaces (ports)** définies dans la couche applicative ou domaine.

Les composants techniques (adaptateurs) implémentent ces interfaces, ce qui permet :

- de remplacer une technologie sans modifier le cœur métier (ex. stockage S3 vers une autre solution),
- de faciliter les tests unitaires grâce à des implémentations simulées (mocks).

Référence aux diagrammes

- **Diagramme de composants** : les adaptateurs sortants sont positionnés en périphérie du système.
- **Diagramme de déploiement** : les composants techniques (DB, S3, broker) sont clairement dissociés du cœur applicatif.

Conclusion

L'application conjointe des principes **SRP** et **DIP** permet de structurer PeopleFirst autour d'un **cœur métier stable**, indépendant des choix techniques.
Ces principes constituent un socle solide pour l'évolution future de l'application, notamment en facilitant la maintenance, les tests et l'extension du système.

2 design patterns identifiés et justifiés

1. Factory (Creational) — Génération de documents PDF

Contexte

PeopleFirst manipule des documents RH et paie, notamment la **fiche de paie au format PDF**. La génération de documents peut évoluer dans le temps (nouveaux types de documents, nouveaux templates, changement de moteur PDF, options spécifiques).

Problème

Sans pattern, la logique de création du bon générateur PDF risque d'être dispersée dans plusieurs endroits (services, use cases) avec des `if/else` selon le type de document, ce qui augmente le couplage et complique la maintenance.

Solution (Factory)

Le pattern **Factory** centralise la création des générateurs de documents via un point unique, par exemple :

- une `PdfGeneratorFactory` qui retourne le bon générateur selon le type de document,
- ou une `DocumentGeneratorFactory` capable de créer des générateurs PDF différents (Payslip, Attestation, etc.).

Le code métier (use case "Générer fiche de paie") ne connaît que l'interface `PdfGenerator` et demande à la factory "donne-moi le générateur adapté".

Bénéfices

- **Réduction du couplage** : le cœur applicatif ne dépend pas d'un moteur PDF spécifique.
- **Évolutivité** : ajout d'un nouveau document = ajout d'un nouveau générateur, sans modifier le flux principal.
- **Cohérence** : la création/configuration des générateurs est standardisée (templates, options, format).

Référence aux diagrammes

- **Diagramme de classes** : la classe `Payslip` est liée à `Document` (PDF stocké). La Factory intervient au moment de produire ce `Document`.

- **Diagramme de déploiement** : la génération s'exécute côté **Service Paie Python**, qui produit des PDFs stockés dans S3/MinIO.

Exemple

Interface commune

```
pseudo

interface PdfGenerator {
  generate(data): PdfFile
}
```

Implémentations concrètes

```
pseudo

class PayslipPdfGenerator implements PdfGenerator {
  generate(data):
    // charger template fiche de paie
    // injecter données (brut, net, cotisations)
    // produire PDF
    return pdf
}

class CertificatePdfGenerator implements PdfGenerator {
  generate(data):
    // charger template attestation
    return pdf
}
```

Factory

```
pseudo

class PdfGeneratorFactory {

  static create(documentType): PdfGenerator {
    if documentType == "PAYSLIP":
      return new PayslipPdfGenerator()
    if documentType == "CERTIFICATE":
      return new CertificatePdfGenerator()

    throw UnsupportedDocumentTypeException
  }
}
```

Utilisation dans un use case (service Paie)

```
pseudo

generator = PdfGeneratorFactory.create("PAYSLIP")
pdf = generator.generate(payrollData)
documentStorage.save(pdf)
```

---

2. Strategy (Behavioral) — Calcul et validation selon le type de congé

Contexte

Les demandes de congé ne se calculent pas toujours de la même manière : selon le type (`LeaveType`), les règles peuvent varier (jours ouvrés/ouvrables, justificatif, plafonds, délais, etc.).

Problème

Une implémentation naïve conduit souvent à de multiples conditions (`if/else` ou `switch`) basées sur le type de congé, rendant les règles difficiles à maintenir et à faire évoluer.

Solution (Strategy)

Le pattern **Strategy** permet d'encapsuler les règles de calcul/validation dans des stratégies distinctes, par exemple :

- `PaidLeaveStrategy` (CP),
- `RTTStrategy`,
- `SickLeaveStrategy`, etc.

Le système sélectionne dynamiquement la stratégie en fonction du `LeaveType` de la demande, puis exécute :

- `validate(request, balance)`
- `computeDays(startDate, endDate)`

Bénéfices

- **Lisibilité** : chaque type de congé a ses règles dans une classe dédiée.
- **Évolutivité** : ajouter un nouveau type = ajouter une stratégie, sans impacter le code existant.
- **Robustesse** : limite les régressions lors de l'évolution des règles.

Référence aux diagrammes

- **Diagramme de classes** : `LeaveRequest` est reliée à `LeaveType` ; Strategy s'appuie directement sur cette association.
- **Diagramme de séquence (demande de congé)** : les étapes "vérifier règles" et "calculer le nombre de jours" correspondent à l'application d'une stratégie.

Interface de stratégie

```pseudo
interface LeaveCalculationStrategy {
  validate(request, balance)
  computeDays(startDate, endDate): number
}
```

Stratégies concrètes

```pseudo
class PaidLeaveStrategy implements LeaveCalculationStrategy {

  validate(request, balance):
    if balance.remainingDays < requestedDays:
      throw InsufficientBalanceException

  computeDays(startDate, endDate):
    return countWorkingDays(startDate, endDate)
}
```

```pseudo
class SickLeaveStrategy implements LeaveCalculationStrategy {

  validate(request, balance):
    // pas de solde requis
    if request.medicalCertificateMissing:
      throw MissingDocumentException

  computeDays(startDate, endDate):
    return countCalendarDays(startDate, endDate)
}
```

Sélecteur de stratégie (simple)

```pseudo
class LeaveStrategyResolver {

  resolve(leaveType): LeaveCalculationStrategy {
    if leaveType == "CP":
      return new PaidLeaveStrategy()
    if leaveType == "SICK":
      return new SickLeaveStrategy()

    throw UnsupportedLeaveTypeException
  }
}
```

Utilisation dans le workflow de demande de congé

```pseudo
strategy = strategyResolver.resolve(leaveRequest.type)

strategy.validate(leaveRequest, leaveBalance)
days = strategy.computeDays(startDate, endDate)

leaveRequest.requestedDays = days
```

Les patterns **Factory** et **Strategy** répondent à deux besoins clés :

- la **production de documents** (PDF) avec une logique de création flexible,
- l'**application de règles métier variables** selon les types de congés.

Ces choix améliorent la maintenabilité, la testabilité et l'évolutivité du système, tout en restant cohérents avec l'architecture hexagonale retenue.

Conclusion — Livrable 2 : Modélisation fonctionnelle et conception

Le livrable 2 a permis de formaliser la conception fonctionnelle et logique du système PeopleFirst à travers plusieurs modèles complémentaires. Les diagrammes de cas d'utilisation ont identifié les **profils utilisateurs cibles** et leurs interactions principales avec le système, en cohérence avec les besoins métier exprimés.

Les diagrammes de classes ont mis en évidence les **entités métier structurantes** (utilisateurs, congés, documents, paie) et leurs relations, constituant une base cohérente pour la conception de la persistance et des règles métier. Les diagrammes de séquence et d'états ont ensuite permis de décrire de manière précise le **fonctionnement dynamique** du système, notamment le workflow complet d'une demande de congé et son cycle de vie.

Enfin, l'application de principes de conception (SOLID) et l'identification de design patterns adaptés (Factory et Strategy) démontrent une volonté de concevoir une architecture **maintenable, évolutive et robuste**, en adéquation avec l'architecture hexagonale retenue.

Ce travail de modélisation constitue un **socle solide pour la phase d'implémentation**, en réduisant les ambiguïtés fonctionnelles et en facilitant les choix techniques futurs, tout en garantissant l'alignement entre besoins métier et solutions logicielles.