Modélisation UML : Système de gestion des ressources humaines

Service RH – Étude de cas UML

Contexte

Le service des ressources humaines d'une entreprise informatique souhaite réorganiser la gestion de ses collaborateurs. L'objectif est de modéliser le système à l'aide d'un **diagramme de classes UML**, en prenant en compte trois types de profils :

— les **employés** permanents,
— les **consultants** ou indépendants,
— les **managers**, responsables d'équipes internes.

Analyse objet

Une analyse orientée objet permet de dégager une abstraction commune à ces profils : tous sont des **travailleurs** ("Workers") qui possèdent un identifiant unique et sont capables de fournir certaines informations RH.

Nous proposons donc la création d'une **classe abstraite** `Worker`, qui sera spécialisée par les classes concrètes suivantes :

— `Employee` : un salarié avec une fonction fixe, un salaire et une date d'embauche.
— `Consultant` : un indépendant travaillant par mission, rémunéré à l'heure ou au forfait.
— `Manager` : un salarié avec une responsabilité d'encadrement, gérant une liste d'employés.

Structure des classes

Classe abstraite : Worker

— **Attributs** :
  — `id : int` ..... identifiant unique
  — `name : String` ..... nom complet
— **Méthodes** :
  — `getId() : int`
  — `getName() : String`
  — `getMonthlyCompensation() : float` ..... méthode abstraite

Classe : Employee (hérite de Worker)

— **Attributs** :

- `position : String`
- `salary : float`
- `hireDate : Date`

— **Méthodes** :

- `getMonthlyCompensation() : float`
- `getSeniority() : int`

Classe : Consultant (hérite de Worker)

— **Attributs** :

- `dailyRate : float`
- `contractDuration : int` ..... en jours

— **Méthodes** :

- `getMonthlyCompensation() : float`
- `isLongTermContract() : bool`

Classe : Manager (hérite de Employee)

— **Attributs** :

- `team : List<Employee>`
- `bonus : float`

— **Méthodes** :

- `addTeamMember(e : Employee)`
- `getTeamSize() : int`
- `getMonthlyCompensation() : float`

Diagramme de classes UML (textuel)

Worker (abstract)
– id : int
– name : String
– getId() : int
– getName() : String
– getMonthlyCompensation() : float (abstract)

Employee ⟶ Worker
– position : String
– salary : float
– hireDate : Date
– getMonthlyCompensation()
– getSeniority()

Consultant ⟶ Worker
– dailyRate : float
– contractDuration : int
– getMonthlyCompensation()
– isLongTermContract()

Manager ⟶ Employee
- team : List<Employee>
- bonus : float
- addTeamMember()
- getTeamSize()
- getMonthlyCompensation()

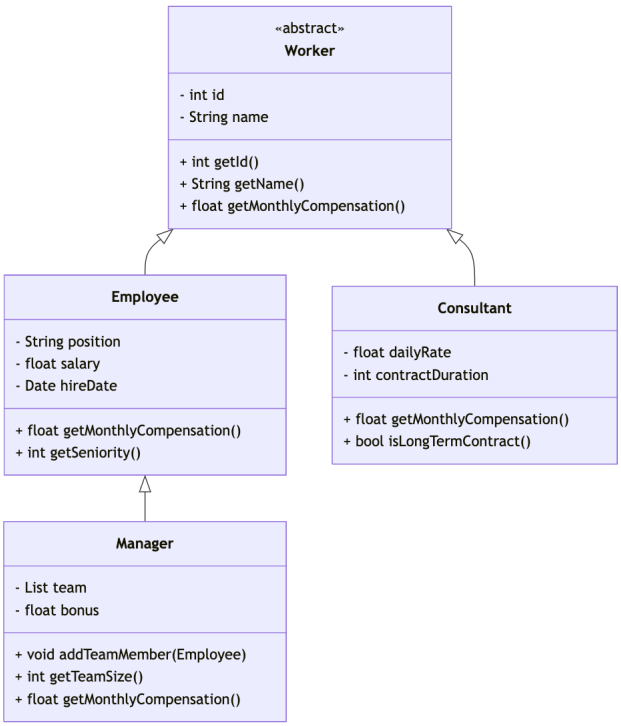
Diagramme UML en notation Mermaid

Voici la représentation du diagramme de classes en utilisant la syntaxe **Mermaid** :

```
classDiagram

class Worker {
  <<abstract>>
  - int id
  - String name
  + int getId()
  + String getName()
  + float getMonthlyCompensation()
}

class Employee {
  - String position
  - float salary
  - Date hireDate
  + float getMonthlyCompensation()
  + int getSeniority()
}

class Consultant {
  - float dailyRate
  - int contractDuration
  + float getMonthlyCompensation()
  + bool isLongTermContract()
}

class Manager {
  - List<Employee> team
  - float bonus
  + void addTeamMember(Employee)
  + int getTeamSize()
  + float getMonthlyCompensation()
}

Worker <|-- Employee
Worker <|-- Consultant
Employee <|-- Manager
```

Résultat graphique

Voici le résultat graphique attendu :



Conclusion

Cette modélisation permet de poser une architecture claire pour la gestion RH, en tenant compte des différences entre collaborateurs internes et externes. L'approche orientée objet permet une évolutivité future (ajout de stagiaires, intérimaires, etc.) tout en assurant une cohérence dans le traitement des profils.