Vendredi 3 Mars 2017 — 3I012 – Partiel –Durée 1h45

Aucun document autorisé sauf les deux feuilles précisant les règles de génération de code et reverse engineering.
Toute réponse doit impérativement être justifiée !!!
Barème donné à titre indicatif et susceptible d'être modifié.

Question de cours (4 points)

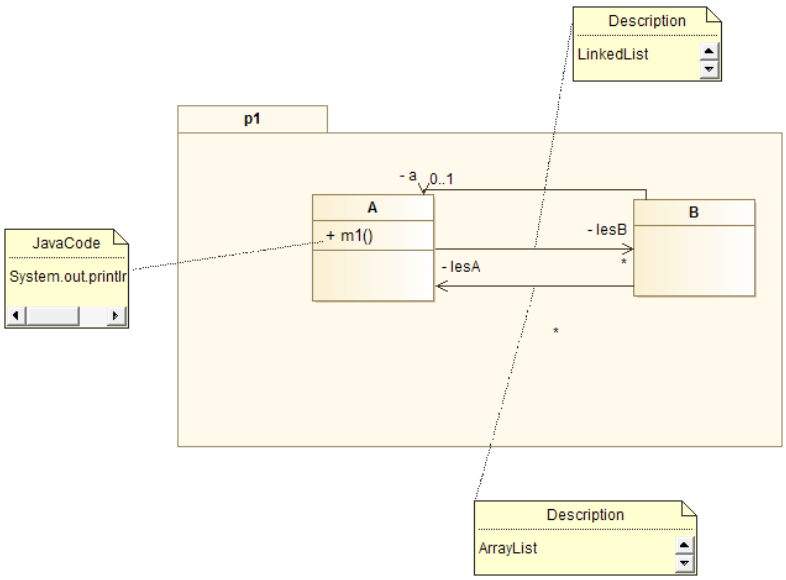
Question 1 (2 points) : En appliquant les règles de reverse-engineering vues en cours, donnez le diagramme de classe UML obtenu après reverse-engineering du code suivant :

```
package p1
public Class A {
 private LinkedList<B> lesB;
public void m1(){ System.out.println("Test");}

}
```

```
package p1
public Class B {
 private ArrayList<A> lesA;
 private A a;
}
```

CORRECTION :



NOTATION :

- **10% package p1/**
- **10% pour la classe B**
- **25% pour la classe A**
  - **10% pour la classe**
  - **5% pour la méthode**
  - **10% pour la note de code**

- **15% association navigable de B vers A avec multiplicité 0..1**
- **20% association navigable de A vers B**
  - **10% Association avec les bonnes multiplicités**
  - **10% la note avec LiknedList**
- **20% association navigable de A vers A**
  - **10% Association avec les bonnes multiplicités**
  - **10% la note avec ArrayList**
  -

Si la présence des propriétés typés dans les classes A et B pour modéliser els associations ➔ enlever toutes les notes dédiées aux associations.

BQuestion 2 (1 point) : Quelles sont les modifications nécessaires à appliquer à l'association entre les classes *Banque* et *Compte* du diagramme de classe UML ci-dessous pour ajouter l'opération :

+ajouterCompte(In c :Compte) à la classe *Banque*

CORRECTION :
Il faut ajouter la navigabilité à l'association. Cette navigabilité est dirigée de Banque vers Compte.

NOTATION :

- **100% si tout**
  - **-50% si on ajoute d'autre associations.**
- **0% sinon.**

Question 3 (1 point) : Donnez brièvement la définition des concepts suivants qui sont liés au test logiciel : *Faute* et *Cas de Test*.

CORRECTION

- **Faute : l'application retourne un résultat différent du résultat attendu**
- **Cas de test : une simulation d'un scénario d'exécution de l'application pour révéler l'existence d'une faute.**

NOTATION : 50% par réponse correcte. 0% sinon.

Exercice (16 points)

Considérons l'application **eLabo** qui permet la gestion des laboratoires de recherche de l'UPMC.
La **Figure 1** montre une première version d'un diagramme de classe modélisant l'aspect structurel de cette application. Chaque laboratoire de recherche (classe *Laboratoire*) est défini par un nom et peut contenir plusieurs départements. Un département (classe *Departement)*, défini par un nom, peut à son tour contenir plusieurs équipes de recherche. Une équipe de recherche (classe *Equipe)* est composée de plusieurs membres (classe *Membre)*. Nous distinguons trois types de membres : des enseignant-chercheur (classe *EnseignantChercheur)* des chercheurs (classe *Chercheur ;* i.e. ; des chercheurs associés à des organismes de recherche comme INRIA ou CNRS), des doctorants (la classe *Doctorant ; i.e. ;* des étudiants préparant une thèse de doctorat) ou des agent administratifs (classe *AgentAdminsitratif)* qui s'occupent de la gestion administrative du laboratoire. Chaque membre est affecté à un bureau particulier (classe *Bureau)*. Le diagramme de classe montre aussi les différentes opérations qui sont définies avec leurs signatures détaillées.
Par la suite, il est demandé d'étudier et comprendre attentivement ces opérations.

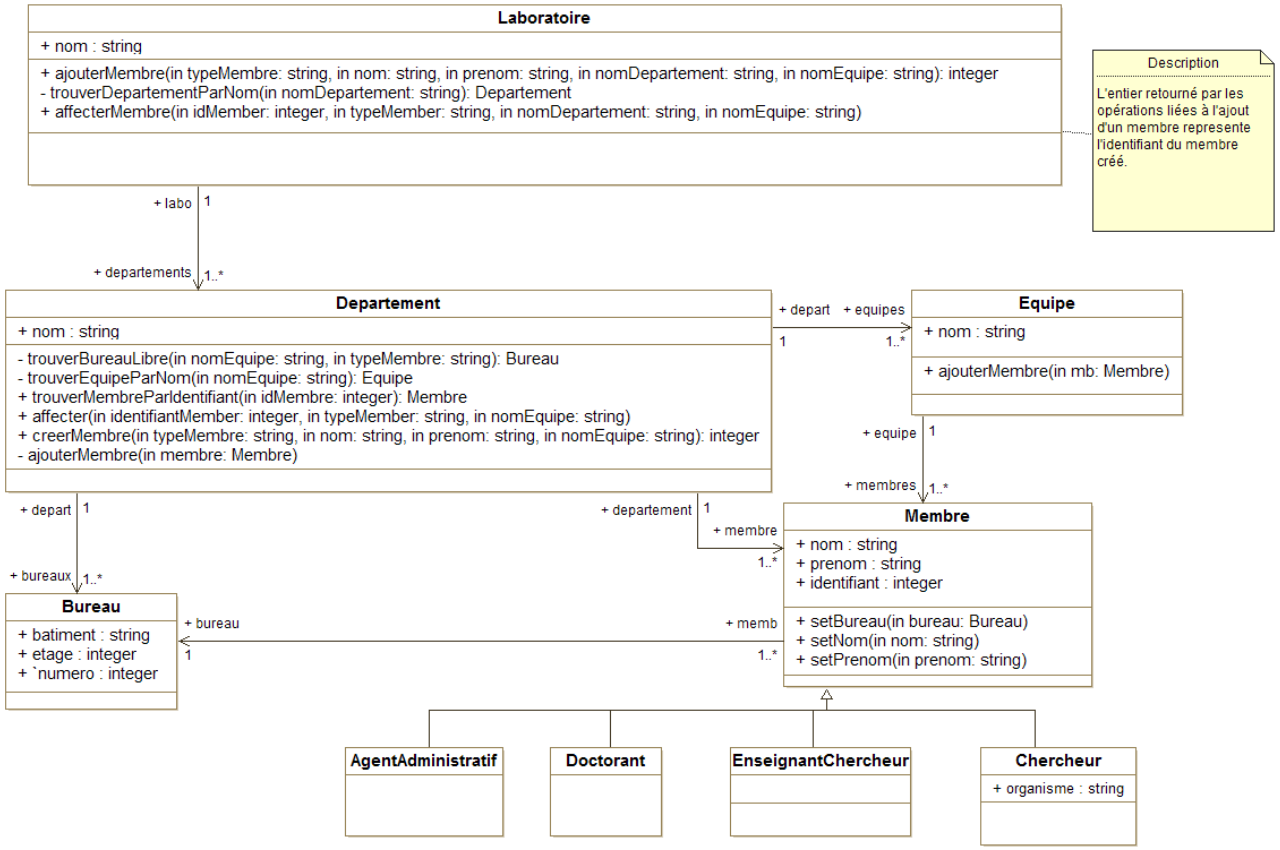


Figure 1: Diagramme de classe de eLabo

Pour les questions E1 et E2 outre les opérations présentées dans le diagramme de la Figure 1, les seules autres opérations pouvant (et devant) être utilisées sont les constructeurs

E1 (3 points). Nous souhaitons modéliser un diagramme de séquence montrant l'ajout d'un nouveau membre par un agent administratif :

Question 1 : Comment peut-on modéliser l'agent administratif dans ce diagramme de séquence ?

CORRECTION

- **objet non typé**

NOTATION : 100% si tout, 0% sinon

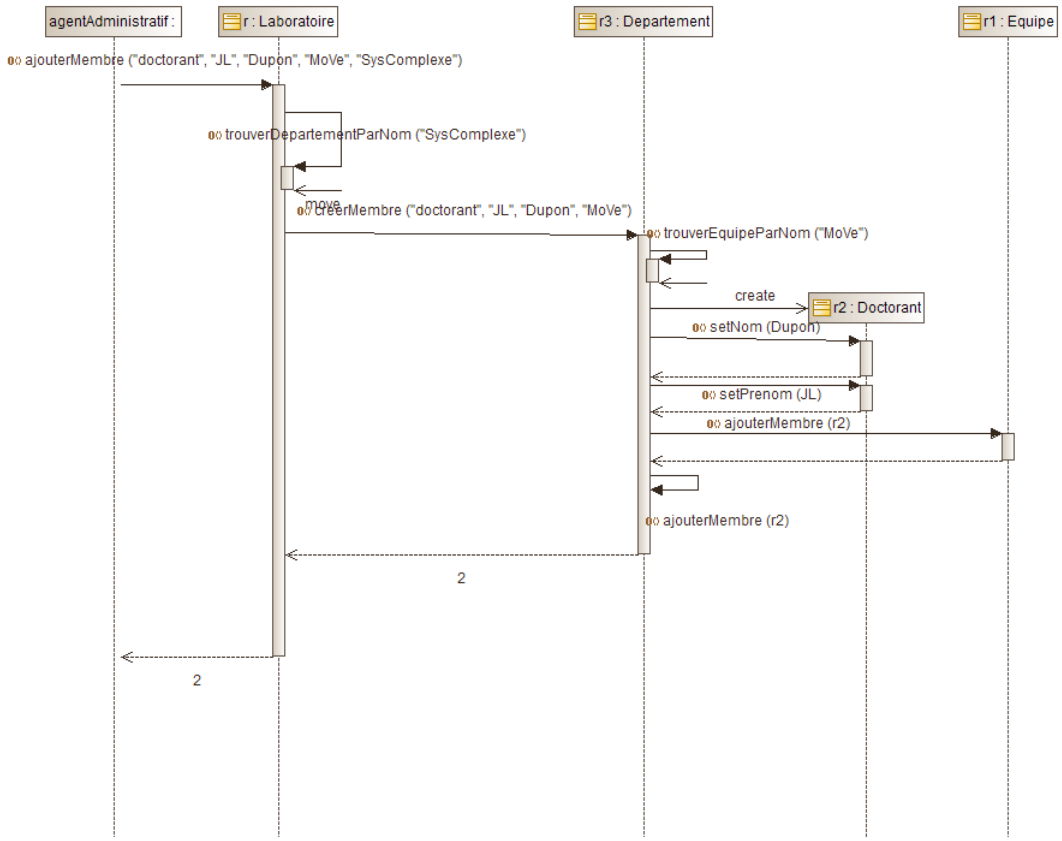
Question 2 : Donnez le diagramme de séquence correspondant au scénario d'ajout d'un nouveau membre de type *Doctorant*. Vous considérez l'exemple d'un agent administratif qui veut ajouter un doctorant avec les informations suivantes :

- typeMembre = « Doctorant »
- nom = « Dupont»
- prenom = « JL»
- nomDepartement = « SysComplexe »
- nomEquipe = « MoVe »

Le scénario d'ajout d'un membre est décrit comme suit :

« … pour chaque demande d'ajout d'un membre (opération *ajouterMembre()* de la classe *Laboratoire*, le laboratoire demande au département concerné de créer le membre (opération *creerMembre()* de la classe *Departement*). Une fois, le membre créé, le département ajoute ce membre à la fois à l'équipe concernée (opération *ajouterMembre()* de la classe *Equipe*) et au département (opération *ajouterMembre()* de la classe *Departement*).. »

CORRECTION



NOTATION :

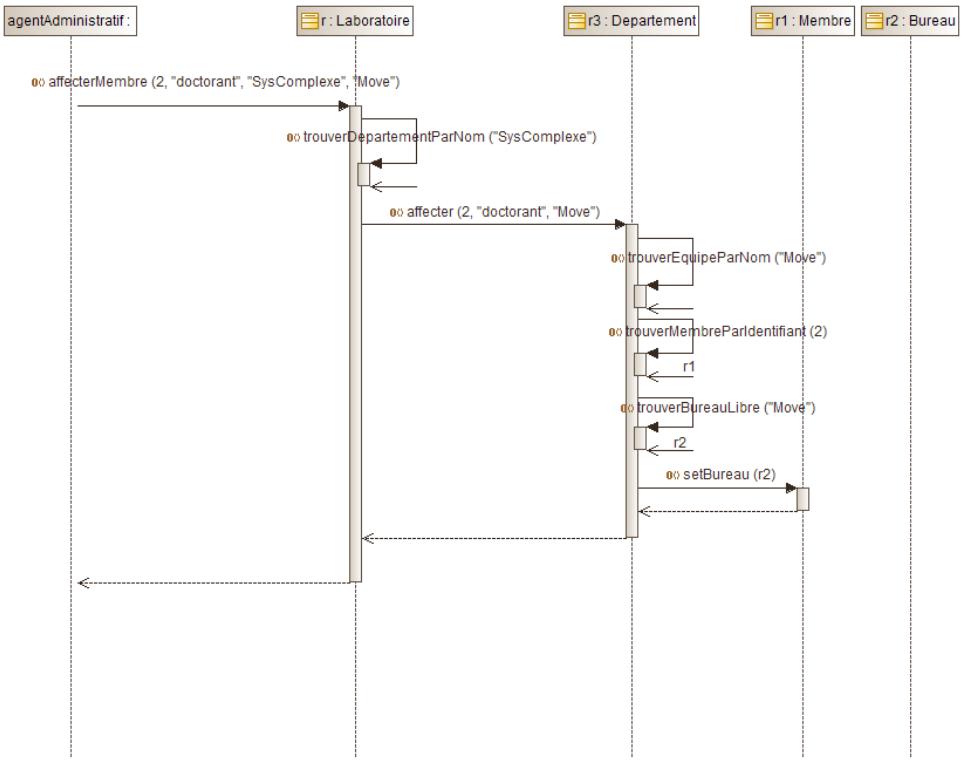
- **10% les objets avec les bons types.**
- **10% appel à l'opération ajouterMembre() avec les bons paramètres**
- **10% trouverDepartementParNom()**
- **10% creerMembre() avec les bons paramètres**
- **10% trouverEquipeParNom()**
- **15% Message de création de doctorant (0% si type Membre)**
- **10%% setNom et setPrenom (on donne 10% si on montre ca via le constructeur).**
- **10% ajouterMembre sur equipe**
- **10% ajouterMembre sur département**

E2 (3 points). Une fois un membre est ajouté et enregistré, l'agent administratif procède alors à son affectation à un bureau particulier. Ce scénario est décrit comme suit :

> « … pour chaque demande d'affectation (opération *affecterMembre()* de la classe *Laboratoire*, le laboratoire demande au département concerné d'affecter le membre (opération *affecter()* de la classe *Departement*). Une fois un bureau libre est trouvé (opération *trouverBureauLibre()),* le département affecte ce bureau au membre concerné (opération *setBureau()* de la classe *Membre).. »*

Question : Donnez le diagramme de séquence correspondant au scénario de l'affectation d'un membre à un bureau. Vous considérez l'exemple de l'affectation du membre doctorant ajouté en **E1**.

CORRECTION



NOTATION :

- **10% les objets avec les bons types.**
- **15% appel à l'opération affecterMembre() avec les bons paramètres**
- **10% trouverDepartementParNom()**
- **15% affecter() avec les bons paramètres**
- **10% trouverEquipeParNom()**
- **15% trouverMembreParIdentifiant()**
- **10% trouverBureauLibre() avec les bons parametres**
- **15% setBureau()**

E3 (2 points). Nous souhaitons organiser les classes de **eLabo** présentées en **Figure 1** en créant deux packages : *equipes* et *reste*. La package *equipes* contient les classes liées à la gestion des équipes de recherche et leurs membres : *Equipe*, *Membre, EnseignantChercheur, Chercheur, AgentAdministratif* et *Doctorant.* Le package *reste* contient le reste de classes.

Question 1 : Identifiez et justifiez le problème de conception engendré par ce découpage.

Question 2 : Proposez une solution pour résoudre ce problème. Votre solution doit assurer que le package *equipes* soit réutilisable sans le package *reste*.

CORRECTION :

- **Il s'agit d'un problème de cycle de dépendance**
  - **Le package** *equipes* **dépend du** *reste* **: une association navigable de** *Membre* **vers** *Bureau***.**
  - **Le package reste dépend de** *equipes* **: deux associations navigables de** *Departement* **vers** *Equipe* **et** *Membre***.**
- **Pour que le package** *equipe* **peut être réutilisé sans le package** *reste***, il existe une solution qui consiste à casser la dépendance du package equipe à reste.**

- **Introduire une nouvelle classe** *CopyBureau* **(ou** *SuperBureau***)**
- **Créer une association navigable de** *Membre* **à cette nouvelle classe.**
- **Supprimer l'association navigable de** *Membre* **à** *Bureau*
- **Créer un lien d'héritage entre** *Bureau* **et** *CopyeBureau***(ou** *SuperBureau***).**

NOTATION : une seule note pour les deux questions.

- **30% pour Question 1 : détecter le problème avec les justifications.**
- **70% pour Question 2 : La solution. Une seule solution possible**

E4 (2 points). Considérons l'opération *listerMembresDepartement() : Membre[*]* qui permet de lister les membres d'un département.

Question : Pour chacune des signatures suivantes, dites quelle(s) classe(s) pourrai(en)t définir cette opération :

1. *listerMembresDepartement (In nomDepartement : string) : Membre[*]*
2. *listerMembresDepartement (In departement : Departement) : Membre[*]*

CORRECTION

- **listerMembresDepartement (in nomDepartement : string) : Membre[*]➔ Seulement dans la classe Laboratoire.**
- **listerMembresDepartement (in departement : Departement) : Membre[*] ➔ Dans toutes les classes**

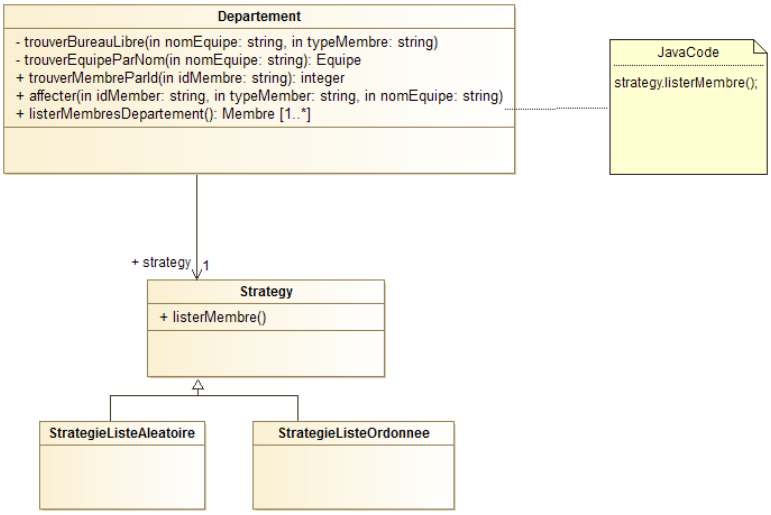
NOTATION : 50% par réponse.

E5 (4 points). Nous souhaitons maintenant utiliser le patron de conception *Strategy* (cf. annexe) pour modéliser les différentes stratégies pour implémenter l'opération *: listerMembresDepartement() : Membre[*]* discutée en **E4**. En effet, nous disposons de deux strategies (deux algorithmes) possibles pour lister les membres d'un départements : *StrategieListeAleatoire* et *StrategieListeOrdonnee* :

- *StrategieListeAleatoire* : il s'agit d'une stratégie simple ou les membres sont listés d'une manière aléatoire.
- *StrategieListeOrdonnee* : cette stratégie permet de lister les membres d'un appartement d'une manière ordonnée.

Question 1 : Nous considérons en premier temps la signature *listerMembresDepartement(): Membre[*]* qui sera définie dans la classe *Departement*. Appliquez le patron de conception Strategy pour implémenter les deux stratégies *StrategieListeAleatoire* et *StrategieListeOrdonnee.*

CORRECTION :



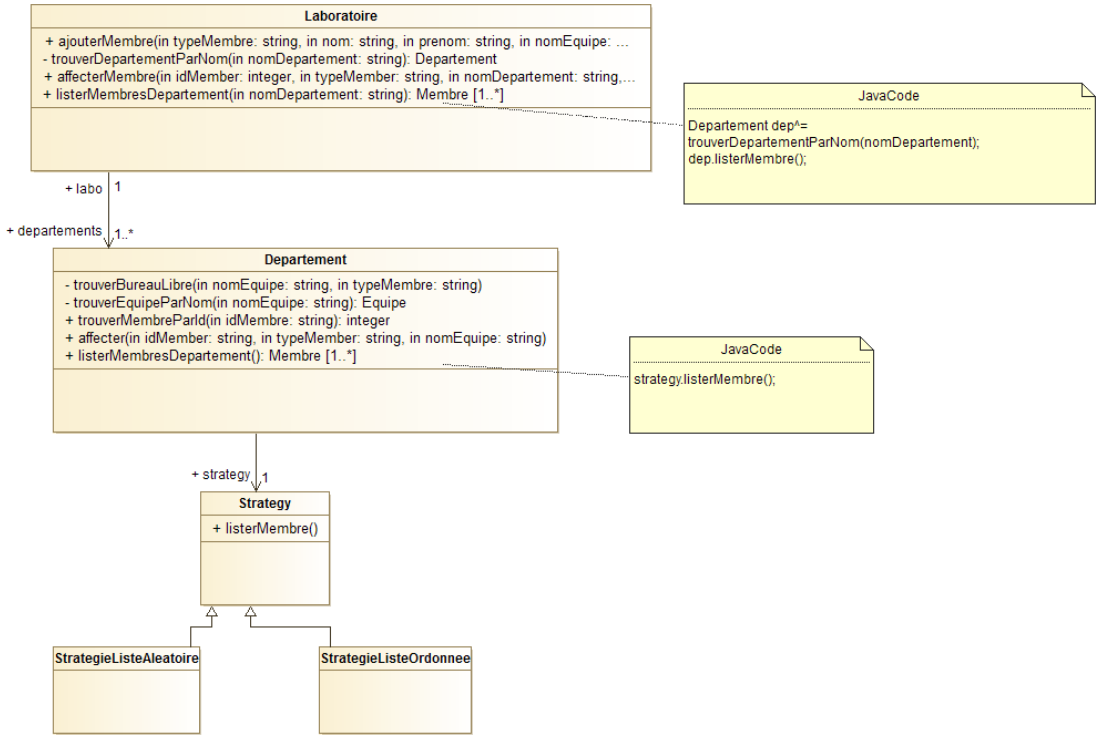
NOTATION :

- **15% Association departement vers strategy pour montrer que departement est le contexte**
- **15% Note de code pour l'opération listerMembreDepartement() de Departement montrant la délégation**
- **25% classe abstraite Strategy**
  - **15% classe Strategy**
  - **10% pour l'opération listerMembre() – ou un autre nom**
- **20% strategie concrete StrategieListeAleatoire qui hérite de strategie**
- **20% strategie concrete StrategieListeOrdonnee qui herite de strategie**

Important : Si département n'est pas contexte mais il y a toutes les classes Strategy, Strategy concrete ➔ Ne donner que 25% sur toute la question.

Question 2 : Si nous considérons la signature *listerMembresDepartement(In nomDepartement : string) : Membre[*].* Que faut-il changer à votre application du patron de conception Strategy proposée à la question précédente pour gérer la nouvelle signature. Montrez cette nouvelle application.

CORRECTION :



NOTATION :

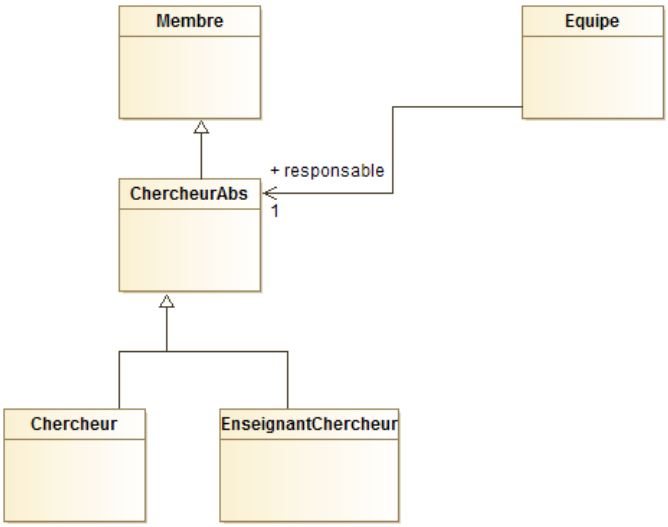
- **60% pour montrer la note de pseudo code pour l'opération listerMembresDepartement() du Laboratoire.**
  - **20% Trouver le bon département**
  - **40% Délégation vers listerMembre() du Departement**
- **40% pour le reste de la structure du design pattern**
  - **10% association departement vers strategy**
  - **10% Note de code montrant la délégation**
  - **10% classe abstraite Strategy**
  - **5% strategie concrete StrategieListeAleatoire qui hérite de strategie**
  - **5% strategie concrete StrategieListeOrdonnee qui herite de strategie**

Important : Si le pseudo code de l'opéation listerMembresDepartement() du Labo n'est pas définie (pas de 60%) mais il y a toutes les classes Strategy, Strategy concrete ➔ Ne donner que 20% sur toute la question.

0duduE6 (2 points) : Une équipe de recherche est dirigé par un et un seul responsable représenté par un de ses membres. Ce responsable ne peut être qu'un *EnseignantChercheur* ou *Chercheur* (un responsable ne peut jamais être un *Doctorant* ou un *AgentAdministratif*).

Question : Proposez une modification du diagramme de classe de la **Figure 1** pour modéliser le responsable d'une équipe.

CORRECTION :



NOTATION :

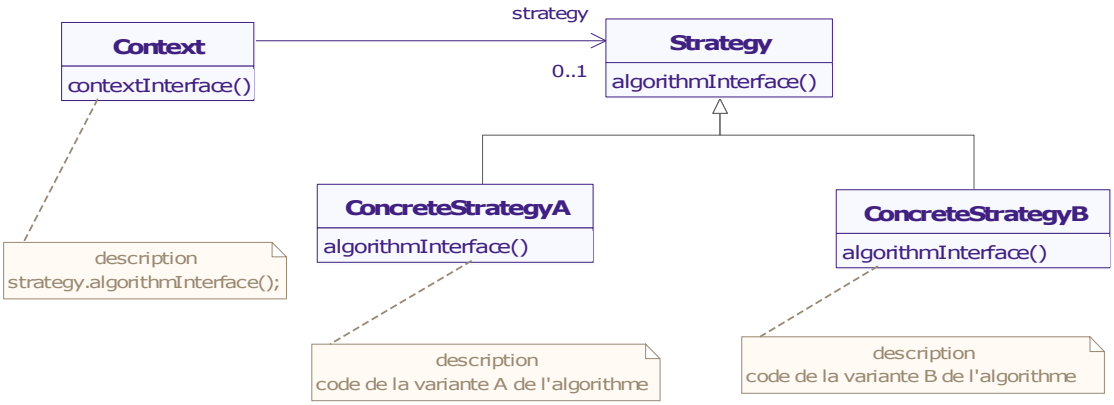
- **50% introduction de la super classe**
  - **20% classe super ChercheurAbs ou un autre nom**
  - **10% Lien héritage entre Chercheur et ChercheurAbs**
  - **10% Lien héritage entre EnseignantChercheur et ChercheurAbs**
  - **10% lien heritage ChercheurAbs et Membre**
-
- **50% association de Equipe vers ChercheurAbd**
  - **25% multplicité 1 coté ChercheurAbs**

ANNEXE : Design Pattern « Strategy »

Objectif : Permettre de définir une famille d'algorithmes, d'encapsuler chaque algorithme dans une classe afin de rendre chaque algorithme interchangeable.

Motivation : Il existe souvent plusieurs algorithmes permettant d'obtenir un même résultat final (il existe par exemple plusieurs algorithmes de tri). Cependant, chaque algorithme dispose de ses propres avantages et de ses propres inconvénients. Il est donc intéressant de pouvoir isoler chaque algorithme et de faciliter la sélection de l'algorithme particulier que l'on veut exécuter à un moment donné.

Solution : La solution consiste à définir une interface (ou une classe abstraite) (***Strategy***) contenant la définition de l'opération correspondant à l'algorithme (***algorithmInterface()***) puis à définir les classes concrètes représentant les variantes de l'algorithme (***ConcreteStrategyA***, ...). Enfin la classe ***Context***, qui représente l'utilisateur de l'algorithme, doit être associée à la classe ***Strategy*** afin de spécifier le fait qu'un contexte utilise à un moment donné une seule variante de l'algorithme. L'opération ***contextInterface()*** de la classe **Context** sera appelée par le contexte lui- même lorsqu'il voudra exécuter l'algorithme. Le code de cette opération ne fait qu'appeler l'opération ***algorithmInterface()***.


strategy
Context
contextInterface()
0..1
Strategy
algorithmInterface()
ConcreteStrategyA
algorithmInterface()
ConcreteStrategyB
algorithmInterface()
description
strategy.algorithmInterface();
description
code de la variante A de l'algorithme
description
code de la variante B de l'algorithme