Université Paris Diderot **Langages à objets avancés**
M1 **Héritage Multiple / Composition**

Exercice 1 Cet exercice explore les choix que l'on peut faire lors de la modélisation : composition ou héritage ?

On s'intéresse à des Objet (avec un nom), qui ont tous un Propriétaire (avec un id), et qui disposent de deux méthodes **void** marche(), **void** arret(). L'opérateur d'affichage est redéfini pour les Objet et pour les Propriétaire. Dans la famille des Objet, il y a ceux qui peuvent enregistrer du son (SoundRecord) et ceux qui peuvent enregistrer des images (PicRecord). Il existe aussi des Camera qui peuvent faire les deux en même temps.

Première modélisation

- La relation entre Objet et Proprietaire est faite par un héritage privé.
- SoundRecord, PicRecord et Camera sont descendant de Objet, avec un double héritage pour Camera

Seconde modélisation

- On encapsule dans les Objet un propriétaire (on veut une agrégation).
- Les Camera héritent directement d'Objet, et encapsulent (composition) un PicRecord et un SoudRecord. On se posera évidemment la question du propriétaire.

Avez vous une préférence ? Des précisions à apporter ? Une autre proposition ? Peut-on changer facilement des données ?

Ecrivez des prototypes de démonstration pour ces 2 modélisations.

Exercice 2 On s'interesse à une hiérarchie de classe : Animal, Mammifere, Carnivore et Lion dont voici une modélisation [1] :

```
class Animal {
  public:
  string const nom;
  Animal(string nom) : nom(nom) {}
  void manger() { cout << nom << " mange."; }
};

class Mammifere : public virtual Animal {
  public:
  Mammifere(string nom) : Animal(nom+"_mammifère") {}
  void boisDuLait() { cout << nom << " bois du lait."; }
};

class Carnivore : public virtual Animal {
  public:
  Carnivore(string nom) : Animal(nom+"_carnivore") {}
```

1. Désolé si Lion n'est pas une catégorie aussi générale que le sont Mammifere et Carnivore. L'important ici est l'exercice de style (acceptez svp le diamant, et les méthodes proposées).

```
  void chasse() { cout << nom << " chasse."; }
};

class Lion : public Mammifere, public Carnivore {
  public:
  Lion(string nom) : Animal(nom), Mammifere(nom), Carnivore(nom) {}
};

int main() {
  Mammifere baleine("moby");
  Carnivore loup("Lupa");
  Lion lion("Simba");
  baleine.manger();
  loup.manger();
  lion.manger();
}
```

Nous allons essayer d'en faire une autre qui n'utilise absolument pas l'héritage. Elle sera bien entendu un peu artificielle, mais devrait vous aider à vous représenter ce qui est propre à l'héritage en diamant. Il s'agit d'une sorte d'exercice de modélisation comparée.

Structure et construction :

Voici l'idée générale :

- conservez la classe Animal telle quelle
- conservez le main() tel quel aussi
- pour la classe Mammifere (sans héritage) encapsulez un Animal. Ajoutez une méthode mange() qui se contente de faire appel à mange() sur l'animal encapsulé, et allaite ()
- idem avec Carnivore, mange() et chasse()

On voudrait ensuite, pour Lion :

- encapsuler un mammifère et un carnivore
- redéfinir chasse() et allaite () en les redirigeant naturellement
- puis se posera le problème de mange() ce qui vous amènera probablement à faire quelques retouches. Indications :
  - Vos différentes encapsulations (il devrait y en avoir 4 ou 5) ont-elles la forme d'une agrégation ou d'une composition ? De laquelle avez vous besoin pour répondre aux difficultés rencontrées sur l'écriture de Lion ?
  - Pour assurer l'équivalent de l'héritage virtuel, vous aurez besoin d'un constructeur supplémentaire dans Mammifere et Carnivore qui prend en argument un pointeur ou une référence vers un Animal. Quelle visibilité (public/ private / protected) lui donner ? (Ils devront être utile dans Lion. Rappel : il n'y a pas d'héritage)

Pour les destructions :

Vous pouvez rencontrer deux cas de figures :

— soit l'objet à détruire est celui qui a construit l'animal de base, et alors c'est à lui de se charger de le détruire au moment où lui même disparaît.

— soit l'animal de base lui a été transmis, et alors il n'a pas à se préoccuper de sa destruction

Il faut trouver un moyen de distinguer ces deux cas. Vous pouvez le faire en ajoutant un booléen estProprietaire que vous positionnerez à vrai ou faux selon le constructeur qui aura été invoqué.

Pour les copies :

Vous pouvez rediriger le constructeur de copie vers le constructeur qui attend simplement un nom, ce qui vous évitera de partager la propriete

Pour les affectations :

Là aussi, il ne faut pas partager la propriete car c'est elle qui indique qui est l'unique responsable de la destruction. Si vous choisissez de l'autoriser il vous faut vérifier que vous vous sortirez des problèmes de gestion de mémoire selon les cas.

En conclusion prenez un moment pour contempler les différences dans la réalisation des mêmes objectifs entre la première modélisation avec l'héritage, et celle avec les encapsulations.