Chapitre 2 : Un paradigme : Conception orientée objet

« N'importe quel programmeur peut écrire du code que l'ordinateur comprend. Les bons programmeurs écrivent du code que les humains peuvent comprendre. » ***Martin Fowler***

1. Objectif du chapitre

Ce chapitre est une initiation au paradigme objet pour la conception logicielle. Il doit vous permettre de répondre à la question : face à une demande d'un client, comment aborder la construction du logiciel ?
À l'issue de ce chapitre :

- Vous serez capable de concevoir une modélisation qui tire profit des concepts introduits par le paradigme objet.
- Vous saurez relire une modélisation objet.
- Vous mesurerez l'intérêt du paradigme objet pour concevoir des logiciels complexes.

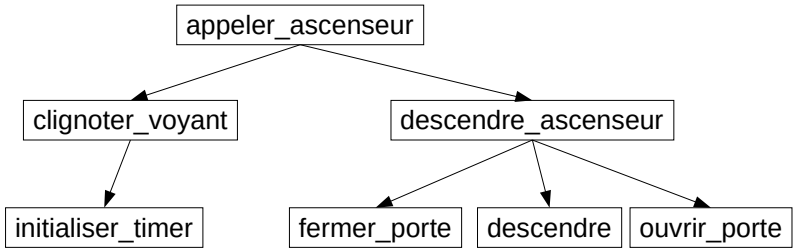
2. Le paradigme objet

Jusqu'à présent, vous avez essentiellement étudié que le paradigme procédural avec les langages C et Shell comme exemples pour programmer. Le paradigme objet est le dual de ce paradigme procédural.

2.1. Paradigme procédural

On s'intéresse aux traitements : actigramme.

- La question à résoudre est : « Que veut-on faire ? »
- La solution est un programme sous la forme d'un enchaînement de procédures s'échangeant des données (les fonctions ne sont que des procédures qui retournent une valeur).
- Les données sont **inertes** : elles sont échangées et modifiées par les procédures.



2.1.1. Conception procédurale

La méthode de conception est l'**Algorithmique**. Elle consiste à trouver la séquence d'appels de procédures.

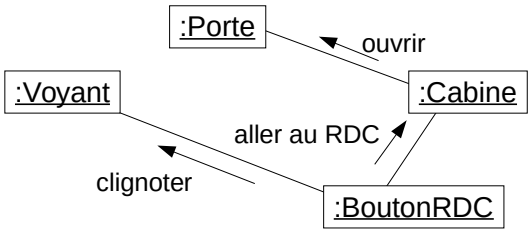
- Avantages :
  - Modélisation proche du fonctionnement de la machine.
  - Calculs de complexité pour estimer la durée des traitements.

- Limites :
  - Modélisation incompréhensible pour les clients.
  - Inadaptée aux gros logiciels.
    - Nécessite de savoir résoudre le problème globalement pour l'implémenter.
    - Mélange conception et implémentation par le fait que l'on s'intéresse aux procédures qui sont des implémentations.
  - Enchaînement figé dans le temps.
  - Maintenance et réutilisabilité difficiles dues à l'interdépendance des procédures entre elles et avec les données, comme un plat de spaghettis froid dont il est difficile d'extraire un spaghetti. Une modification locale aura donc des répercussions à plusieurs endroits.

2.2. Paradigme objet

On s'intéresse aux données : datagramme.

- La question à résoudre est : « de quoi parle-t-on ? »
- La solution est un programme sous la forme d'un ensemble d'objets s'échangeant des services.
- Les données sont **animées** : ce sont elles qui exécutent les procédures.



2.2.1. Conception orientée objet

La méthode de conception est la **Modélisation**. Elle consiste à trouver les objets du domaine d'application à utiliser. Le problème de la modélisation peut s'exprimer sous la forme : « *Si je disposais d'un chapeau magique, quels types de données voudrais-je voir sortir du chapeau pour m'aider à résoudre le problème ?* ».

- Avantages :
  - Permet d'appréhender la complexité (gros logiciels).
    - Conception modulaire avec une vision locale de la résolution de problème.
    - On peut repousser le plus tard possible l'implémentation.
    - La conception est basée sur un raffinement progressif.
  - Maintenance et réutilisabilité facilitées parce que les objets sont des blocs indépendants. Les modifications n'ont alors que des répercussions locales.
- Limites :
  - Vision fractionnée du fonctionnement du logiciel.
  - Calculs de complexité difficiles (p.ex. à cause du polymorphisme).

2.3. Langages de programmation objet

Il existe aujourd'hui un foisonnement de langages de programmation orientés objet. :

- Simula (Ole-Johan Dahl & Kristen Nygaard, 1963).
- SmallTalk (Alan Kay, 1972), Eiffel (Bertrand Meyer, 1986).

- **C++** (Bjarne Stroustrup, 1983), Objective C, D.
- **Java** (Sun MicroSystem, 1991), C# : *basés sur UML*.
- **PHP**, **Python**, Perl, Clos.
- Ruby, **Scala**, Dart.

La différence entre les langages procéduraux et orientés objet ne concerne que quelques mots clés. Par exemple, la différence minimale entre le langage C et le langage Java peut se réduire aux mots clés suivants : `Class`, `Interface`, `new`, `extends`, `implements`, et `String`. Pourtant, ce sont deux paradigmes totalement différents. Le paradigme objet ne s'apprend donc pas à partir du langage.

2.4. Concepts de la conception objet

La conception orientée objet s'appuie sur 5 concepts :

1. Objet
2. Classe
3. Association
4. Héritage
5. Polymorphisme

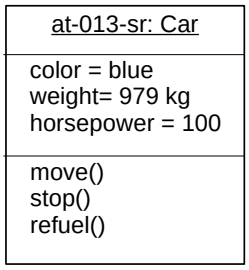
3. Les objets

3.1. Notion d'objet

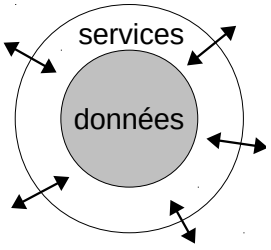
Techniquement, un objet peut se concevoir comme une structure en C incluant des données et de pointeurs de procédures. (note : à ses débuts, le C++ n'était qu'un préprocesseur du langage C qui générait du C) Mais, conceptuellement c'est plus que cela :

- Objet = attributs + méthodes
- Attribut : une donnée
  - possédant une valeur,
  - évoluant au cours du temps.
- Méthode : procédure attachée à l'objet :
  - utilisant potentiellement les attributs pour fonctionner,
  - déclenchée par appel explicite à partir de l'objet.

Par exemple, la voiture « at-013-sr » est définie par :



3.2. Le principe d'encapsulation



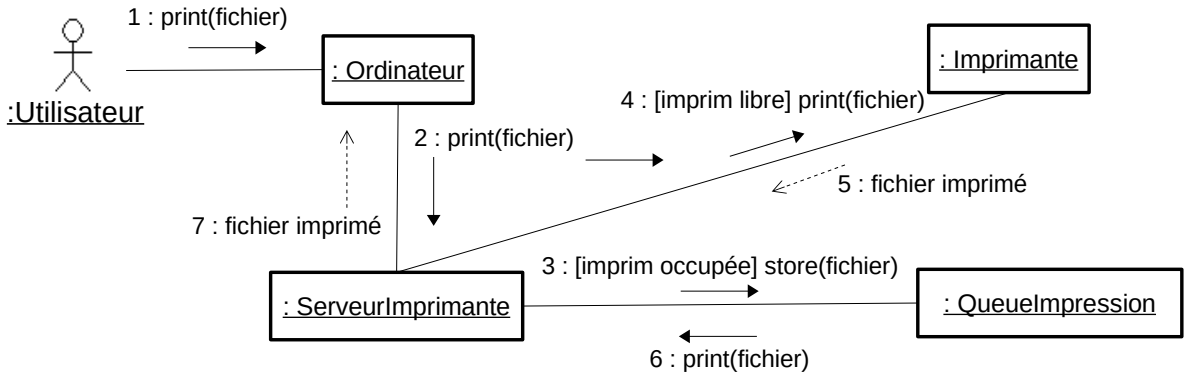
Il faut penser un objet comme un **fournisseur de services** et non comme une structure qui stocke des valeurs de données. Les données sont **cachées et protégées**, elles ne servent qu'à assurer les services.

Important : En modélisation, on ne parle ni de méthodes ni d'attributs mais

uniquement de services. Les services sont des méthodes que d'autres objets peuvent utiliser. Les attributs n'interviennent qu'au moment de l'implémentation et ils ne sont créés que parce qu'ils sont nécessaires à l'implémentation d'un service.

3.3. Communication entre objets

Pour pouvoir s'échanger des services, les objets doivent communiquer entre eux. L'unité de communication est le **message**. On ne peut accéder aux services que par l'intermédiaire du mécanisme de transmission de message. Cela correspond à l'appel d'une méthode d'un objet. La réalisation d'une fonctionnalité du logiciel se concoit comme une séquence de messages échangés entre des objets.



4. Les classes

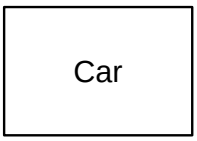
4.1. Classe

Représente un **concept** du problème, c'est-à-dire une entité ayant une représentation dans le monde de la modélisation. Une classe est une génératrice d'objets. Elle définit pour chaque objet :

- La liste de ses attributs.
- La liste de ses méthodes.
- La liste des objets avec qui elle peut communiquer.

Notation UML

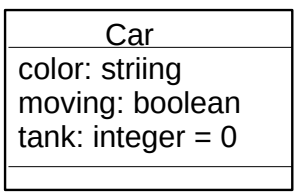
- *Rectangle.*
- *Nom un substantif au **singulier**.*
- *Casse : PascalCase.*



4.2. Attribut

C'est une donnée que possède tout objet de la classe (on parle aussi de propriété ou de donnée membre). Il stocke une valeur unique pour chaque objet.

- **Important : Les attributs ne peuvent être que :**
  - d'un type primitif : `float`, `double`, `boolean`, `int`, `char`…
  - d'un type primitif assimilé : `enumerate`, `string`, `date`, `time`…
  - d'un type tableau ou collection de types précédents : `int[]`, `string[]`…
- Notation UML
  - *Nom : un substantif.*



- *Placé dans un premier compartiment.*
- *Casse : camelCase.*
- *On peut ajouter :*
  - *le type.*
  - *une valeur par défaut.*
  - *la multiplicité pour les tableaux, listes et ensembles de types primitifs.*

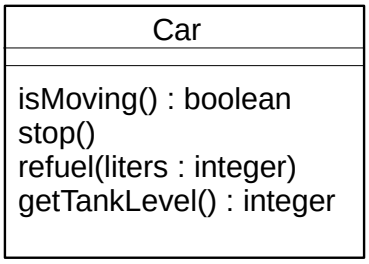
4.3. Méthode

C'est une procédure classique qui définit un service de l'objet. Elle possède des paramètres et une valeur de retour.

- La signature d'une méthode est la suivante :
  - `type_retour nom( paramètre_formel* ) { /* coups */ }`
  - L'appel se fait par l'intermédiaire d'un objet de la classe.

Notation UML

- *Nom : un verbe.*
- *Placé dans un deuxième compartiment.*
- *Casse : camelCase.*
- *On peut ajouter :*
  - *La signature + la valeur de retour.*



4.3.1. Note : cas particulier des constructeurs et destructeurs

Il n'y a pas de notation spécifique en UML pour les constructeurs et les destructeurs parce qu'ils relèvent de l'implémentation ; ce ne sont pas des services. Ainsi, le nom des constructeurs dépend des langages, par exemple :

- Nom de la classe en Java.
- `__init()` en Python.

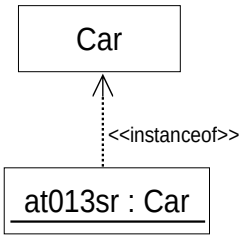
4.4. Instance d'une classe

C'est un objet créé à partir d'une classe. C'est la manifestation concrète d'une abstraction :

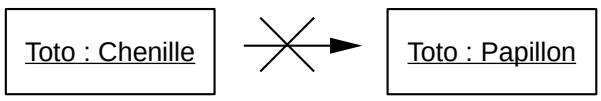
- qui occupe un emplacement mémoire.
- qui dispose des méthodes.
- qui possède des attributs capables de mémoriser les effets des méthodes.

Notation UML

- *Un rectangle avec un nom souligné.*
- *Nom : un substantif.*
- *Casse : camelCase.*



Caractère statique de l'instance : une instance ne peut pas changer de classe. Par exemple, une chenille ne devra jamais un papillon. La seule façon de réaliser ce changement est de récréer un autre objet.



- Accès à un attribut ou à une méthode se fait par la notation pointée :

- `instance.attribut`
- `instance.operation()`
- p. ex. : `at013sr.refuel()`

- À l'intérieur d'une méthode, la référence à l'objet appelant se nomme '`this`' en Java.

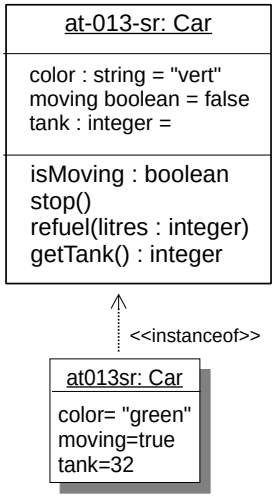
4.4.1. Note

Une classe peut générer autant d'objets avec les mêmes attributs et les mêmes méthodes que nécessaire. Une classe est une usine à objets de même type et de même comportement.

4.5. Implémentation en Java

- Déclaration de la classe :

```
class Car {
    // Attributs
    String color = "green";
    boolean moving;
    int tank = 0;
    // Méthodes
    boolean isMoving() { return moving; }
    void stop() { moving = false; }
    void refuel(int liters) { tank = liters; }
    int getTankLevel() { return tank; }
}
```



- Création d'un objet :

```
Car at013sr = new Car();
```

- Appel de méthodes :

```
at013sr.stop();
at013sr.refuel(32);
```

- Destruction d'un objet : Pas de destructeur en Java, reste :

```
at013sr = null;
```

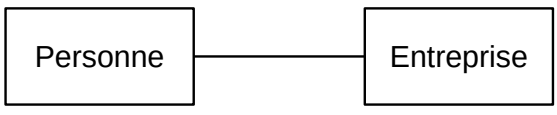
5. Associations entre classes

5.1. Objectif

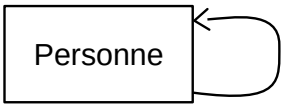
La réalisation d'une fonctionnalité du logiciel nécessite la collaboration de plusieurs objets qui s'échangent des services. Pour qu'un objet puisse utiliser les services d'un autre objet, il faut qu'il connaisse son emplacement mémoire. Cette adresse mémoire peut être stockée directement dans l'objet ou donnée dans la liste des paramètres d'une méthode. Quand le lien est stocké dans l'objet, il est nommé association. Une association est définie au niveau de la classe en précisant la classe avec laquelle les objets de la classe pourront dialoguer, mais elle est instanciée avec l'adresse réelle d'un objet au niveau de l'objet.

Notation UML :

- Un trait continu entre deux classes.



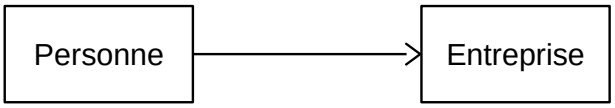
- Cas particulier : association réflexive. Les instances (objets) de la classe connaissent ainsi une autre instance de la même classe.



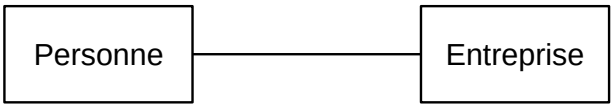
5.2. Navigabilité des associations

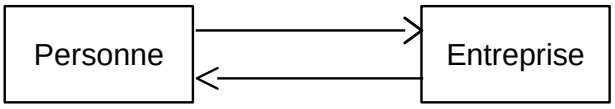
Restreindre l'accessibilité à un sens unique.
Notation UML

- *Une flèche ouverte.*



Une association non orientée correspond en fait à une association orientée de chaque côté.

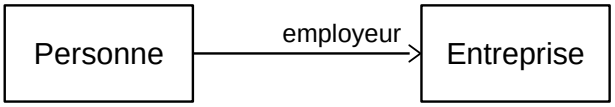




5.3. Rôle des associations

Un nom qui spécifie le rôle joué par la classe associée pour la classe qui détient l'association.
Notation UML

- *Placé sur le bout de flèche de l'association.*
- *Casse : en minuscule.*
- *En pratique : un substantif.*



5.4. Multiplicité des associations

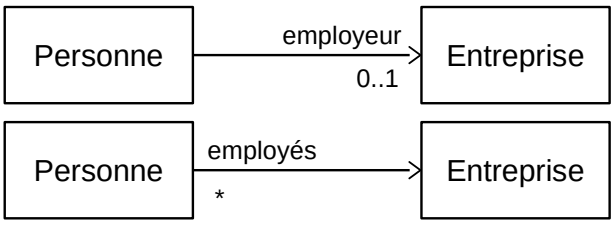
Elle permet de préciser le nombre d'objet liés par l'association. C'est un nombre entier ou un intervalle de valeurs.

1	Un et un seul objet dans l'association (par défaut)
0..1	Zéro ou un objet
M..N	De M à N objets
*	De zéro à plusieurs objets
1..*	De 1 à plusieurs objets
N	Exactement N objets

Notation UML :

- *Notée avec le rôle.*
- *Par défaut 1 (peut être omis).*

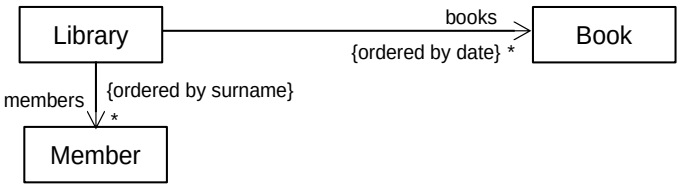
Attention : les multiplicités sont inversées par rapport à la notation Merise.



5.5. Contraintes sur les associations

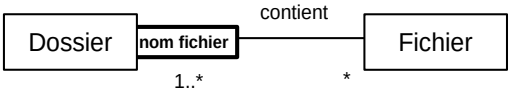
Association **ordonnée**.

- *unordered* (c'est la valeur par défaut).
- *ordered* : les associés sont stockés dans un ordre précis. Dans l'exemple ci-dessous, la bibliothèque garde la liste de ses membres et les livres de façon ordonnée par date.



Association **qualifiée** : tableau associatif, table de hachage, dictionnaire.

- Les objets liés par l'association sont accessibles par une clé. Dans l'exemple ci-dessous, un dossier accède directement à l'un de ses fichiers par son nom.



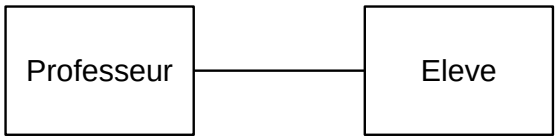
5.6. Types d'association

Le typage vise avant tout à donner une sémantique à l'association. Les classes, entre autres de par leur nom, donne une valeur métier à la modélisation en se référant aux concepts du domaine métier. Le typage des associations complète cette sémantique en explicitant la relation entre les classes. Cela peut aussi avoir des implications dans le code. Il existe quatre types d'association entre les classes : standard, agrégation, composition et relation de dépendance.

5.6.1. Association standard

Elle décrit une relation **dynamique** de type « ***connaît*** ». C'est la relation par défaut. Une instance d'une classe connaît une instance d'une autre classe (ou de la même classe pour une association réflexive). Elle est dynamique dans le sens où chaque instance de la classe ne peut connaître les objets avec lesquels elle est en association qu'au moment de l'exécution (ie., elle doit connaître leur adresse mémoire). La classe définit l'association mais chaque objet doit lui donner une valeur au cours de l'exécution.

Notation UML :



5.6.2. Association de type agrégation

Elle décrit une relation dynamique **ensembliste non symétrique** de type « ***possède*** ». Une classe joue le rôle d'ensemble et une autre classe le rôle d'élément.

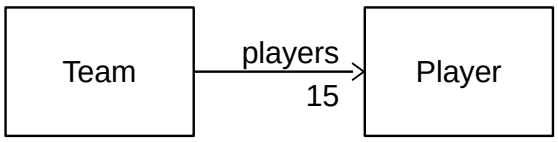
Relation	Exemple
Composé / Composant	*Voiture / Roues*
Collection / Élément	*Forêt / Arbres*
Espace / Position	*Désert / Oasis*
Événement / Étape	*Document / Chapitre*

Conséquences visibles sur le code :

- On crée dans la classe agrégat des méthodes pour ajouter ou supprimer des objets agrégés (p. ex. les méthodes `add()`, `remove()`).

Notation UML :

- *Un losange vide du côté de l'agrégat.*



Notation Java : une donnée membre de type tableau, liste ou vecteur, par exemple :

```
class Team {
  Player[] _players = new Player[15];
  void add( Player p, int i ) {
    _players[i] = p;
  }
}
class Player { }
```

5.6.3. Association de type composition

Elle décrit une relation dynamique de **subordination** non symétrique de type « ***est constitué de*** ». Elle s'interprète comme si la classe composant était une partie intrinsèque et privée de la classe composite.

Relation	Exemple
Corps / Portion	*Corps / Tête*
Matière / Substance	*Eau / Hydrogène*
Activité / Phase	*Achat / Paiement*

Durée de vie :

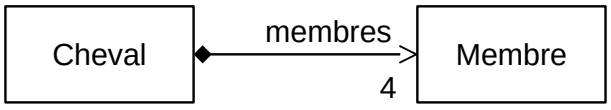
- L'objet composite a la responsabilité de la création, de la destruction et du stockage de l'objet composé.
- Conséquence : un objet ne peut être le composant que d'un objet composite.

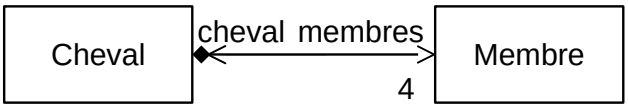
Conséquences visibles sur le code :

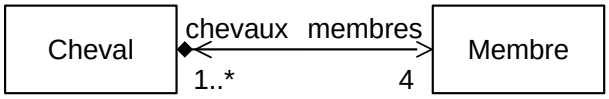
- Le composé contient du code pour créer et détruire les instances du composant dans ses méthodes généralement dans le constructeur et le destructeur.
- Il n'y a pas de méthode `add()` ou `remove()`.
- Typiquement, le composé est une classe interne du composite.

Notation UML

- *Un losange plein du côté du composite.*
- L'objet ne peut pas être partagé.







- Notation Java : tableau, liste ou vecteur, par exemple :

```
class Cheval {
  Membre[] _membres = new Membre[4];
  Cheval ( ) {
    for (int i = 0; i < _membres.length; i++) {
      _membres[i] = new Membre();
    }
  }
```

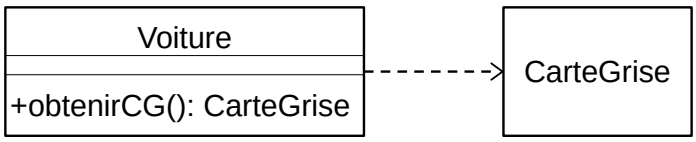
```
  class Membre { }
}
```

5.6.4. Relation de dépendance

Elle est utilisée pour indiquer une relation lâche entre objets qui n'est **pas** de nature **structurelle**, comme le fait qu'une méthode d'une classe utilise un paramètre ou définit une variable locale ou possède un type de retour correspondant à une autre classe. Cette relation n'a pas réellement d'intérêt autre que de préciser qu'une classe dépend d'une autre classe sans lien explicite. Elle est à noter sur le diagramme que si cela est réellement nécessaire. La seule incidence dans le code est que la première classe importe la seconde.

Notation UML

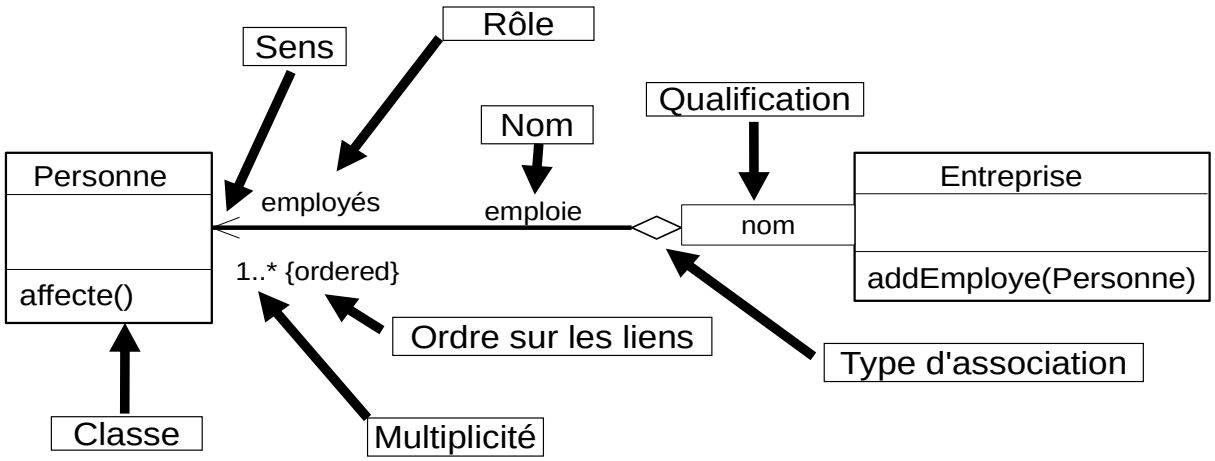
- *Une flèche pointillée.*



Exemple de code Java

```
class Voiture {
  CarteGrise obtenirCG() {
    cg = new CarteGrise();
    cg.setImmatriculation("AB-123-CD");
    return gc;
  }
}
```

5.7. Résumé de la notation



5.8. Les associations en Java

Une association est représentée en Java par une donnée membre. La représentation dépend, entre autres, de la multiplicité :

- Multiplicité 0 ou 1 : une référence.

```
class Personne {
  Entreprise employeur;
```

```
}
```

- Multiplicité de N : un tableau de N références.

```
class Entreprise {
  Personne[10] employes;
}
```

Cas d'une association normale avec une multiplicité de '*'.

- Vecteur ou liste : `ArrayList` ou `LinkedList`

```
class Entreprise {
  List<Personne> employes = new ArrayList<>();
}
```

Cas d'une association ordonnée.

- Ensemble ou tableau associatif ordonnés : `TreeSet` ou `TreeMap`

Cas d'une association qualifiée.

- Tableau associatif (table de hachage) : `HashMap`

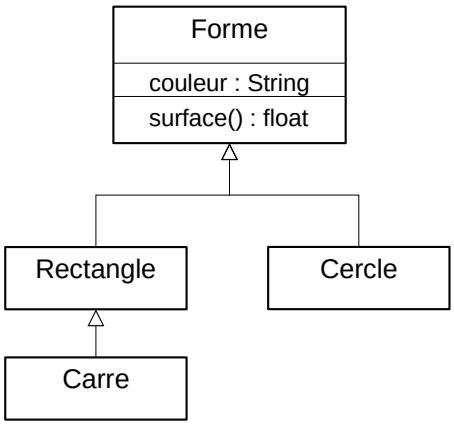
5.8.1. Note

Attention, en Java, une association s'implante par une donnée membre, mais en UML une association n'est pas un attribut (et donc n'apparaît pas dans le premier compartiment). En UML, il y a une réelle distinction entre attribut et association, ce qui n'est pas le cas en Java.

6. Héritage et polymorphisme

6.1. Héritage

C'est une relation **statique** très forte de **type « sorte-de »**. La relation est statique dans le sens où les classes sont associées au moment de la compilation. Elle permet de construire une nouvelle classe à partir d'une classe existante en lui ajoutant de nouveaux attributs et de nouvelles méthodes. La classe dérivée contient par héritage tous les attributs, toutes les méthodes et toutes les associations de la classe parent. Tout se passe comme si la classe créée contenait toute la classe de base.



Notation UML

- *Une flèche triangulaire anonyme.*

Vocabulaire

- Super-classe (*super-class*) ou classe de base.
- Sous-classe (*subclass*) ou classe dérivée (*derived class*).

6.2. Généralisation et spécialisation

L'héritage permet de répondre à deux objectifs :

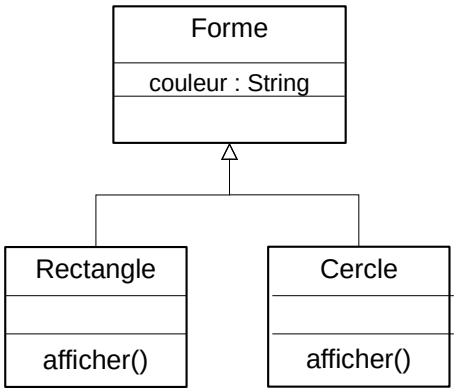
- Généralisation
  - On veut factoriser les éléments communs d'un ensemble de classes (méthodes et associations).
  - Une super-classe est une abstraction de ses sous-classes.
- Spécialisation

- On veut particulariser la liste des services d'une classe par rapport à une autre.
- Une sous-classe est un cas particulier de sa super-classe.

6.3. Notation Java

- L'héritage utilise le mot clé '*extends*'.

```
class Forme {
  int couleur;
}
class Cercle extends Forme {
  int rayon
  void afficher() { ... }
}
class Rectangle extends Forme {
  void afficher() { ... }
}
static void main(String args[]) {
  Cercle cercle = new Cercle();
  cercle.couleur = 1;
  cercle.afficher();
}
```



6.4. Transtypage : *Casting*

6.4.1. Sur-classement : *Upcasting*

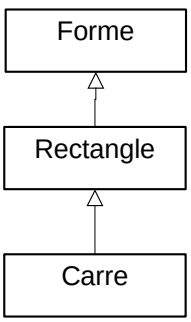
Le sur-classement consiste à référencer un objet d'une classe dérivée `B` héritant d'une classe `A` par une variable de type `A`.

```
class B extends A {}
A a = new B();
```

Un objet d'une classe dérivée peut se faire passer pour un objet de sa super-classe. La liste de ses services utilisables s'en trouve réduite à celle de la super-classe.

Dans l'exemple ci-dessous, un carré peut être vu comme un `Carre` à travers l'instance `c3`, comme un `Rectangle` à travers l'instance `c2`, ou tout simplement comme une `Forme` à travers l'instance `c1`. C'est le même objet qui est pointé mais à travers l'instance `c1`, on ne voit du carré que les méthodes de `Forme` bien que toutes les méthodes existent encore, mais elles sont invisibles. À travers l'instance `c2`, on ne voit du carré que les méthodes de `Forme` et de `Rectangle`.

```
Forme c1 = new Carre();
// accès aux seuls services de Forme
Rectangle c2 = new Carre();
// accès aux services de Forme et Rectangle
Carre c3 = new Carre();
// accès aux services de Forme, Rectangle et Carre
```



6.4.2. Déclassement : Downcasting

Le déclassement est le fait de convertir une référence sur une classe A vers une de ses classes dérivées B.
Un objet référencé comme un objet de la super-classe est ainsi promu en tant qu'objet de la classe dérivée. On récupère ainsi la possibilité d'utiliser les services de la classe dérivée. Cette opération n'est possible que si l'objet est effectivement du type de la classe dérivée ou d'une classe dérivée de celui-ci.

```
Forme r = new Rectangle();
Rectangle r1 = (Rectangle)r; // OK
Carre c = (Carre)r;        // Erreur à l'exécution, r n'est pas un carré
```

6.5. Visibilité des membres

La notion de visibilité porte sur la restriction d'accès aux membres d'une classe : attributs, méthodes et associations. L'intention est de sécuriser la programmation en protégeant la représentation d'une classe pour se permettre de la faire évoluer sans affecter le reste du programme. La visibilité permet de mettre en œuvre l'encapsulation avec une vérification par le compilateur.

Notation UML :

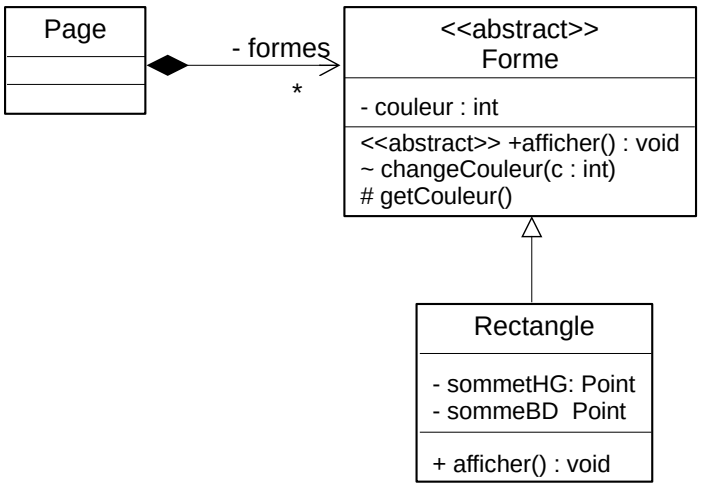
		Accès par des instances				
	Notation UML	Classe elle-même	Classe même pkg	Sous-classe du même pkg	Sous-classe d'un autre pkg	Toute classe
public	+	+	+	+	+	+
protected	#	+	+	+	+	-
package	~	+	+	+	-	-
private	-	+	-	-	-	-

Pour des questions de sécurité de programmation et de maintenance de code, il est important de limiter le plus possible la visibilité des membres.

- La visibilité par défaut doit être **privée**. Il n'y a aucune dérogation possible.
- En particulier, **tous les attributs autres que les constantes et toutes les associations doivent toujours être privés** (cf. encapsulation).
- Les seules méthodes avec une visibilité publique doivent être les services. Les autres méthodes doivent être *private*, *protected* ou *package*.

Exemple : Dans le diagramme ci-après, une instance de la classe `Page`, aura accès au service `afficher()` d'une instance de `Forme` et aux services `changeCouleur()` et `getCouleur()` si la classe `Page` est dans le même paquet que `Forme`. Une instance de la classe `Rectangle` aura accès aux services `afficher()` et `getCouleur()` de `Forme`, et `changeCouleur()` si la classe `Rectangle` est dans le même paquet que `Forme`.

Exemple :



6.5.1. Remarques

- Un objet peut accéder aux attributs privés des autres objets de la même classe.
- Les classes dérivées **ne peuvent pas diminuer la visibilité** d'une méthode redéfinie. C'est une contrainte due à la compilation qui n'a pas moyen de vérifier le respect d'une visibilité plus restrictive. En reprenant l'exemple précédent :

```
Forme f = new Carre();
f.afficher();
```

si `Carre` diminue la visibilité de `afficher()` en privée, alors le compilateur ne peut pas empêcher l'utilisation de la méthode puisque l'objet est de type `Forme` et que sa visibilité est publique.

6.6. La portée des membres

Les membres d'une classe, attributs, méthodes et associations, peuvent appartenir à une instance ou à une classe. Dans ce dernier cas, l'emplacement mémoire des membres est partagé par toutes les instances et connu au moment de la compilation. Cela a plusieurs implications :

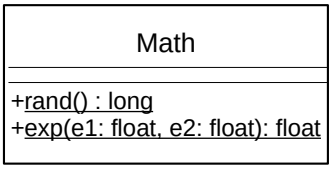
- pour les attributs :
  - Niveau instance : valeur distincte pour chaque instance.
  - Niveau classe : valeur partagée par toutes les instances. Si une instance modifie la valeur d'un attribut de classe, alors toutes les instances partagent cette modification.
- pour les méthodes :
  - Une méthode de classe ne nécessite pas d'instance pour appeler la méthode. Elle s'appelle directement à partir du nom de la classe. Une méthode de classe ne peut utiliser que des attributs, méthodes et associations de classe.

Notation UML

- *Nom de l'attribut ou de la méthode en souligné.*

Notation Java : le mot clé '`static`'.

```
class Math {
  public static int PI = 3.141592;
  public static float exp(float e1, float e2){}
}
```



Pour son utilisation, il n'est pas nécessaire de créer une instance :

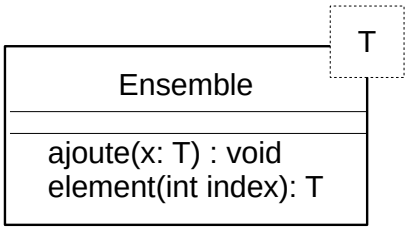
```
Math.PI;
Math.exp(5.0f, 2.0f);
```

6.7. Classe paramétrée

C'est une classe qui utilise un type indéfini au moment de sa déclaration et qui est référencé dans le code par un identificateur. Elle est nommée *generics* en Java et *template* en C++. Le type doit être donné au moment de la définition d'un objet de la classe.

Notation UML

- Le dessin de la casse est augmenté d'un rectangle en haut et à droite qui contient l'identificateur qui remplace le type.



Code Java d'une classe paramétrée

1. Définition de la classe paramétrée

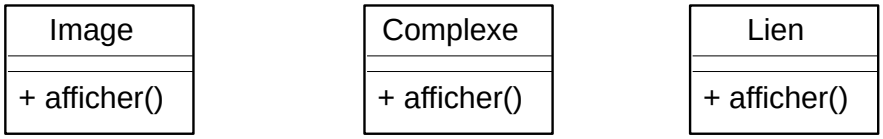
```
public class Ensemble<T> {
  public Ensemble() {}
  public void add(T x) { .. }
  public T get(int index) { .. }
}
```

2. Instanciation de la classe. Tout se passe comme si l'instance e était une instance de la classe `Ensemble` dans laquelle chaque occurrence de `T` a été remplacée par `Client`.

```
Ensemble<Client> e = new Ensemble<Client>();
e.ajoute(new Client());
Client c = e.element(0);
```

6.8. La portée des noms

Il est possible de définir des méthodes de même nom dans des **classes** sans rapport entre elles.
La portée des noms est locale aux classes. Elle permet de garder une écriture locale des classes sans se soucier des conflits de nom avec d'autres classes.
Exemple :

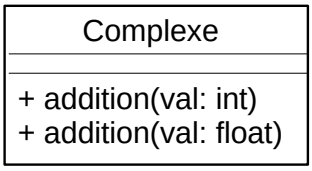


Le choix de la méthode **est fait au moment de la compilation** à partir de la classe de l'objet appelant. La liaison est dite statique (*early binding*)
Par exemple :

```
Image i = new Image();
Complexe c = new Complexe();
c.afficher(); // Appel de la méthode afficher() de Complexe
i.afficher(); // Appel de la méthode afficher() de Image
```

6.9. La surcharge

Dans une même portée, donner un même nom à des méthodes avec des signatures différentes (*overloading*). Cela permet de définir des procédures adaptées aux paramètres.
Exemple :

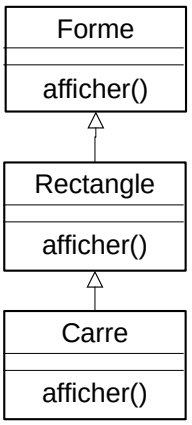


Le choix de la méthode **est fait au moment de la compilation** à partir de la signature (liste des paramètres effectifs). La liaison est dite statique (*early binding*).
Par exemple :

```
Complexe c = new Complexe();
c.addition(5);    // Appel de la méthode addition(val: int)
i.afficher(5.1F); // Appel de la méthode addition(val: float)
i.afficher(5.1D); // Erreur: il n'y a pas de la méthode addition(val: double)
```

6.10. Le polymorphisme

Dans une même hiérarchie, donner le même nom à des méthodes de même **signature** (*overriding*). Le choix de la méthode **est fait à l'exécution** à partir du type de l'objet appelant. La liaison est dynamique (*late binding*).
Dans l'exemple suivant, les trois objets créés sont tous du type `Carre`. Donc quel que soit le type des variables qui référencent les objets (`f`, `r` et `c`), la méthode `afficher()` qui sera appelée sera toujours la méthode la classe `Carre` :




```
Forme f = new Carre();
c.afficher(); // Appel de la méthode afficher de Carre
Rectangle r = new Carre();
r.afficher(); // Appel de la méthode afficher de Carre
Carre c = new Carre();
c.afficher(); // Appel de la méthode afficher de Carre
```

6.10.1. Note

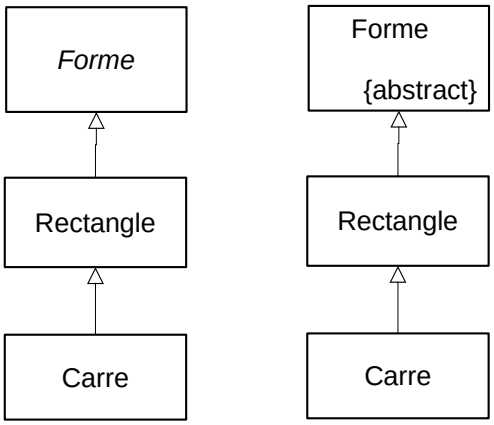
Il est impossible d'accéder à la méthode `afficher()` de `Rectangle` à partir d'une instance de `Carré`, sauf si la méthode `afficher()` de `Carré` fait le lien elle-même (appel de `super.afficher()`). On peut aussi utiliser `final` sur `afficher()` de `Rectangle` pour casser le polymorphisme, mais cela rompt le polymorphisme.

6.11. Classe abstraite

C'est une classe dont on ne peut pas créer d'instances directement. Une classe abstraite est créée pour factoriser des services communs à un ensemble de sous-classes. L'intention est d'obliger à créer des classes dérivées qui auront une représentation concrète dans le domaine.

Notations UML officielle :

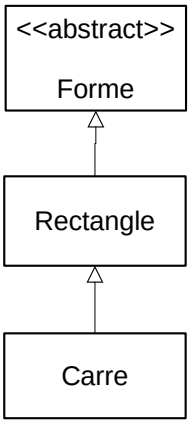
- *Nom en italique*
- *ou ajout d'un attribut {abstract} sous le nom de la classe.*



6.11.1. *Remarques sur cette notation officielle :*

La seconde version avec l'attribut `abstract` pourrait laisser croire que l'on peut associer les mots clés `abstract` et `interface` sur une classe ce qui est faux. La première version est difficilement utilisable pour des diagrammes faits à la main et pas très visible sur un diagramme numérique.

Notation UML personnelle : Puisqu'en UML, nous avons le droit de construire tous les stéréotypes que l'on souhaite, nous avons choisi d'utiliser un stéréotype `<<abstract>>` mis au-dessus du nom de la classe comme le pendant du stéréotype `<<interface>>` (voir section suivante).



Notation Java : le mot clé `abstract`

```
public abstract class Forme() {
}
```

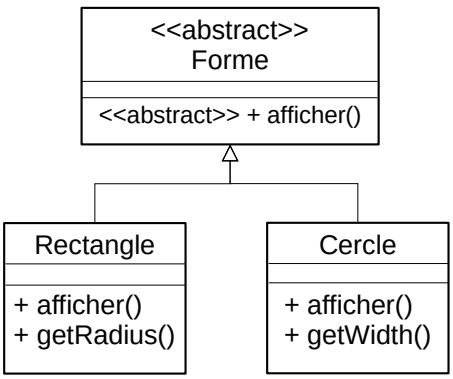
Si on ne peut pas créer d'instance directe, on peut créer des références de ce type par un sur-classement (*upcasting*) :

```
Forme f = new Cercle();
Forme f = new Forme();  // impossible.
Forme f = new Cercle(); // possible, mais f ne peut accéder qu'aux méthodes
                        // de Forme même si c'est un Cercle.
```

6.12. Méthode abstraite

C'est une méthode sans code. Le code devra forcément être donné dans chaque classe dérivée concrète. L'Intention est d'obliger les classes dérivées à redéfinir la méthode parce qu'elle leur est spécifique.
Notation UML :

- *Nom en italique ou stéréotype `<<abstract>>`*



Notation Java : le mot clé 'abstract'

```
abstract public void afficher();
```

Exemple

```
Forme c1 = new Cercle();
c1.afficher();
```

Conséquences : une méthode abstraite ne peut appartenir qu'à une classe abstraite. Si au moins une méthode est abstraite, la classe doit être abstraite. Par contre, une classe

abstraite peut ne contenir que des méthodes concrètes.

6.13. Interface

6.13.1. Notion de type

Un type définit une liste de services que doit posséder une classe pour être de ce type. Le but est d'obliger une classe de ce type à implémenter ces services afin de garantir que le service existe au moment où on utilise un objet de ce type. L'interface fait le lien entre deux types de classes dont les premières utilisent les services de secondes. Pour cela, elles se basent sur un contrat qu'est la liste des services concernés.

6.13.2. Interface

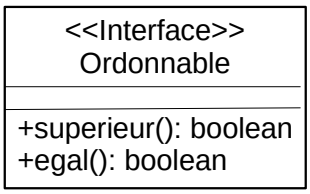
La notion de type est implémentée par une interface. On peut voir une interface comme une classe abstraite où toutes les méthodes sont abstraites. Toutefois, la différence est qu'une classe fait référence à une notion de catégorisation de concepts. Une classe abstraite est une matrice de création de classes dérivées à qui elle donne ses propriétés et le code de ses méthodes concrètes. Un type ne définit que la liste des méthodes publiques que la classe doit définir, mais ne donne pas le code de ces méthodes. Une classe peut implémenter plusieurs types mais ne peut hériter que d'une seule classe.
Une interface :

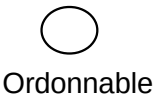
- Une structure dont toutes les **méthodes sont abstraites**.
- Une structure qui **ne possède pas d'attribut**.

Une classe peut implémenter plusieurs interfaces.

Notations UML : deux alternatives.

- *Une classe normale stéréotypée* `<<interface>>` *ou un rond nommé.*





6.13.3. Relation d'implémentation

Une classe qui implémente une interface doit définir toutes les méthodes de l'interface.

- IL est possible de définir une référence d'un type sur un objet d'une classe qui implémente ce type.

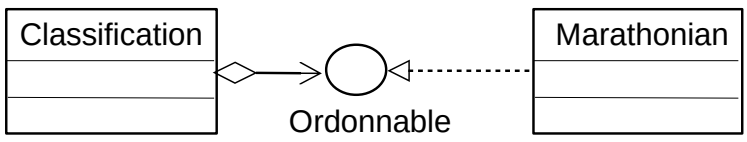
```
Ordonnable o = new Marathonien();
```

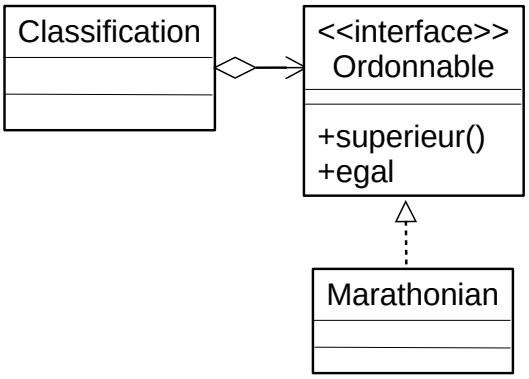
- Par contre, on ne peut pas créer d'objet d'un type donné.

```
~~Ordonnable o = new Ordonnable();~~ // Impossible
```

Notation UML

- *Flèche fermée avec trait en pointillé.*



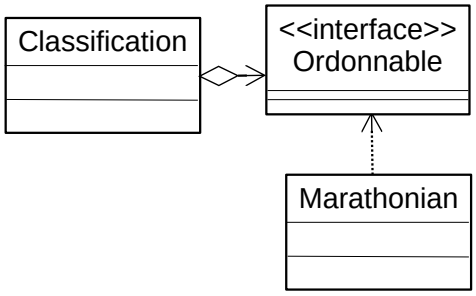


6.13.4. Notes

- Un marathonien n'est pas un `Ordonnable`, mais il possède la capacité d'être ordonné. Il est de type `Ordonnable`, c'est-à-dire qu'il doit fournir les services `egal()` et `superieur()`.
- On a le droit de faire des références vers les types abstraits, mais on ne peut pas créer des types abstraits.
- La notation d'interface joue un rôle central en gestion de projet informatique. C'est un point de consensus pour l'intégration de parties développées par des équipes différentes. Une équipe présente une interface qui liste l'ensemble des services qu'il faut implémenter pour utiliser leur travail. Nous verrons l'utilisation concrète des interfaces l'année prochaine.

6.13.5. Interface en Java

```
public interface Ordonnable {
  public boolean superieur();
  public boolean egal();
}
public class Marathonian implements Ordonnable {
  public boolean superieur() {..}
  public boolean egal() {..}
  // méthodes spécifiques
}
public class Classification {
  ArrayList<Ordonnable> coureurs;
  …
  coureurs.get(1).superieur(coureurs.get(2));
}
```



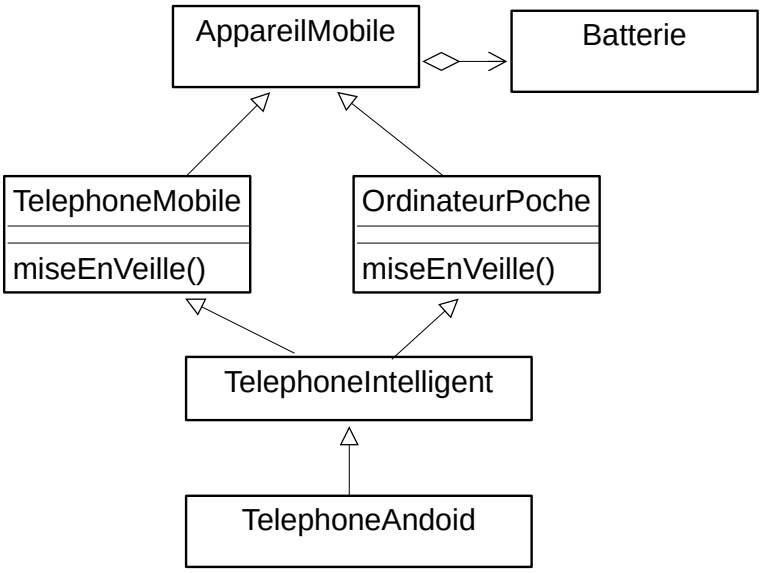
6.14. Cas de l'héritage multiple

L'héritage multiple est possible en UML, mais son emploi est déconseillé pour les difficultés qui peuvent apparaître lors de l'implémentation avec un langage de programmation.

6.14.1. Problème du diamant (*Diamond Problem*)

En considérant, la modélisation ci-dessous :

- Combien de batteries possède un téléphone Android ?
- Quelle méthode `miseEnVeille()` est appelée à partir d'un téléphone Android ?



6.14.2. Réponse

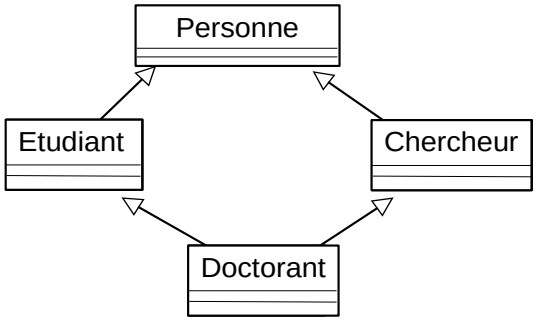
En C++, la réponse est 2 batteries sauf si l'une des deux branches d'héritage est marquée par `virtual`, auquel cas la réponse est 1.
Le choix de la méthode `miseEnVeille()` se fait en indiquant le chemin absolu de la méthode.
Tout cela apparaît contradictoire avec le paradigme objet. La modélisation n'est plus locale. Il faut connaître la hiérarchie pour l'utiliser et s'arranger avec des artifices du langage pour régler les problèmes d'ambiguïté et d'homonymie.

6.14.3. Avons-nous besoin de l'héritage multiple ?

L'erreur consiste à utiliser l'héritage pour résoudre des problèmes de partage de ressources communes.
Par exemple, un doctorant :

- possède une procédure d'inscription comme un étudiant.
- peut assister à des séminaires de recherche comme un chercheur.

La solution de facilité est d'utiliser l'héritage multiple pour récupérer les services des deux super-classes. Mais cela est une grave erreur de modélisation.



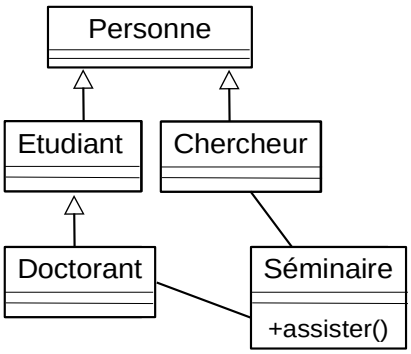
6.14.4. L'héritage multiple est auto-contradictoire

L'héritage ne devrait être utilisé que pour la catégorisation. Il n'y a donc qu'une classe d'appartenance.
En reprenant l'exemple, un doctorant est un étudiant à part entière, mais pas un chercheur à part entière, bien qu'il partage la possibilité d'assister aux séminaires. En autre, il n'est pas fonctionnaire. Mais avec cette

modélisation, il bénéficie des prérogatives d'un fonctionnaire alors qu'il ne l'est pas. Il se trouve aussi affecté par des modifications du statut de fonctionnaire alors qu'il ne devrait pas.
La notion d'héritage multiple n'est ni un concept fondateur ni même nécessaire à la conception orientée objet. Tous les langages à objets n'implémentent pas l'héritage multiple (notamment les langages orientés développement logiciel les plus récents : Java, C#, Scala, Ruby). L'héritage multiple est utilisé pour d'autres principes que la généralisation, par exemple en C++ pour implémenter la notion d'interface qui est absente du langage.

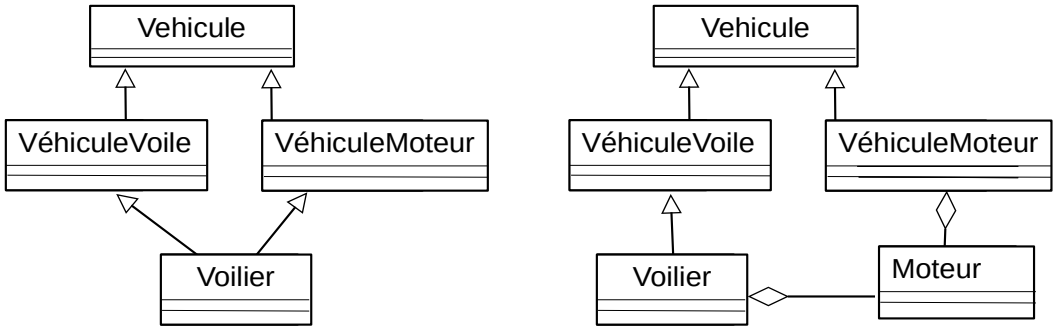
6.14.5. Solution sans héritage multiple

Pour implanter le partage de ressources communes avec des sous-classes strictement exclusives, il faut créer des classes avec les services et les propriétés partagées.
Par exemple, pour partager la méthode assister aux séminaires, on fera une classe partagée par les classes avec uniquement cette méthode. Transformer une méthode en classe, s'appelle faire une abstraction hypostatique.



L'abstraction hypostatique est aussi utilisée pour le partage d'attributs.

- Par exemple, le voilier est d'abord un véhicule à voile qui possède un moteur comme les véhicules à moteur. Le moteur est externalisé et partagé par les deux classes.



Certains langages informatiques proposent une représentation formelle de cette notion de partage de méthode. Par exemple, Python ou Dart propose la notion de `mixin` et Php celle de `trait`.

7. Que retenir de ce chapitre ?

Le paradigme objet définit plusieurs concepts importants :

- Classe et objet
- Relations

    - Association
        - Standard
        - Agrégation
        - Composition
        - Dépendance
    - Héritage et polymorphisme
- Classe et Type

Dans l'utilisation de ces concepts, le développeur doit respecter deux principes fondamentaux :

- Restreindre le plus possible la visibilité des membres pour respecter le principe d'encapsulation.
- Exprimer le plus de choses statiquement pour qu'elles soient vérifiables par le compilateur.