

PROGRAMMATION ORIENTEE OBJET

LES FONDAMENTAUX

MAXIME KELTSMA

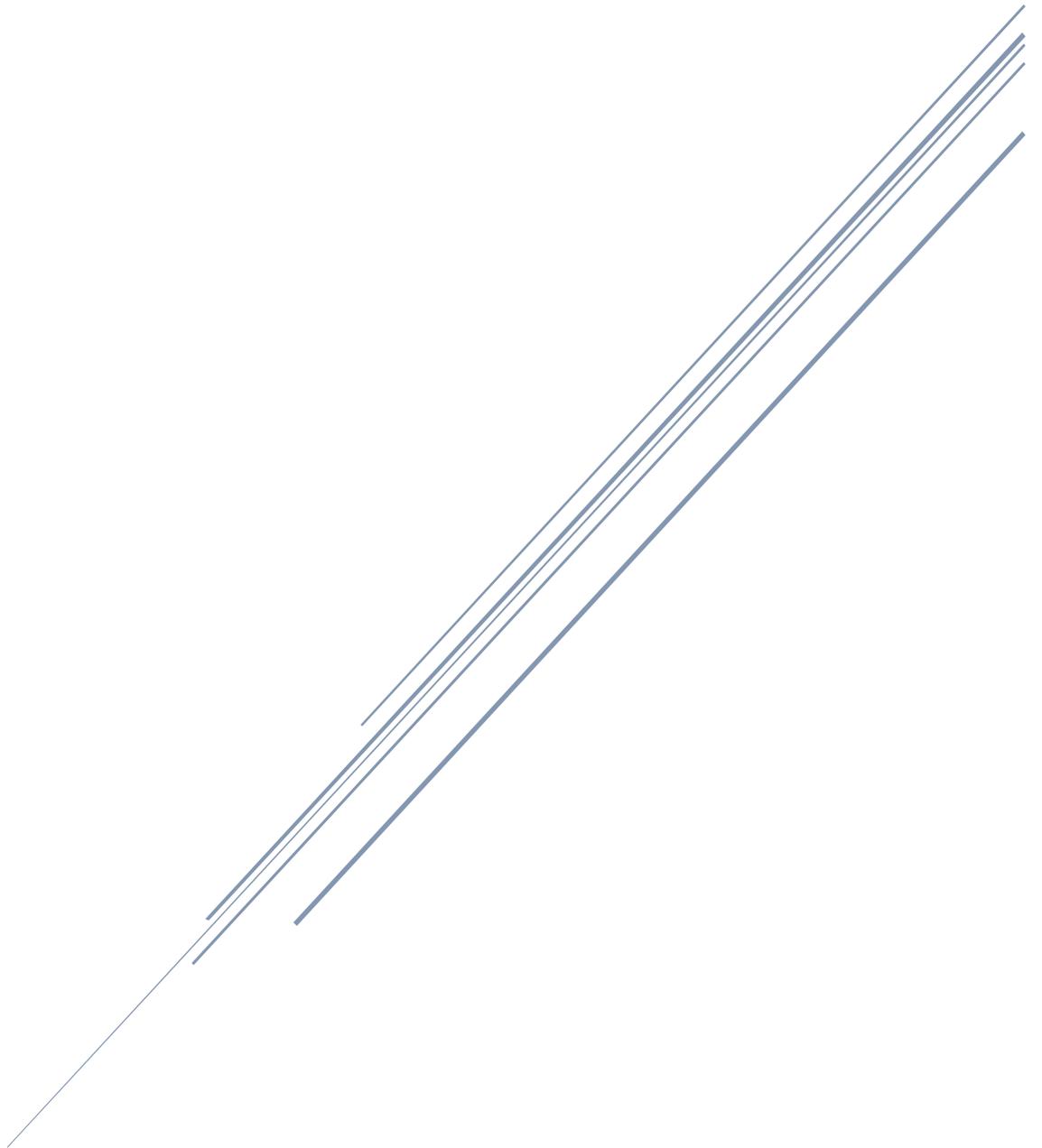


Table des matières

1	Introduction.....	2
2	La POO en deux mots	2
3	Un peu d'histoire	2
4	Classes et objets	3
4.1	Une petite analogie	3
4.2	Classe.....	4
4.3	Objet.....	4
5	Cycle de vie d'un objet	5
5.1	Instanciation.....	6
5.2	Phase active.....	6
5.3	Elimination.....	6
6	Héritage	7
6.1	Héritage simple	7
6.2	Héritage multiple.....	8
7	Modèle de classes	9
8	Encapsulation	10
9	Classes abstraites et interfaces	11
9.1	Classes abstraites	11
9.2	Interfaces.....	13
9.3	Autre types de classes	15
10	Modificateurs de méthodes	16
11	Techniques de polymorphisme	16
11.1	Surcharge et redéfinition de méthodes	16
11.1.1	Surcharge.....	16
11.1.2	Redéfinition	17
11.2	Polymorphisme ad hoc.....	17
11.3	Polymorphisme paramétrique	17
11.4	Polymorphisme d'inclusion	18
12	Lexique	21

1 Introduction

Ce document s'adresse aux personnes souhaitant comprendre les concepts fondamentaux de la programmation orientée objet (POO), ou en éclaircir certains aspects.

Les concepts, présentés de manière très simple, sont communs à tous les langages orientés objets. Aucun exemple de code n'est donné. Le lecteur pourra s'orienter vers d'autres publications concernant un langage en particulier.

2 La POO en deux mots

Tentons une définition de la POO : Il s'agit d'un modèle de programmation informatique basé sur des composants élémentaires appelés objets.

Un programme conçu de cette manière sera matérialisé en moment de son exécution, par un ensemble d'objets évoluant dans la mémoire centrale de l'ordinateur. Le fonctionnement du programme reposera alors sur l'échange de messages entre ces objets.

Nous verrons que ces objets ont un cycle de vie, une structure interne, et qu'ils sont conçus pour exécuter chacun la tâche qu'il leur a été confiée.

Quand on parle de création d'un programme informatique, la phase de programmation est toujours précédée d'une phase d'analyse dans laquelle le concepteur s'efforce de définir les caractéristiques des objets nécessaires au fonctionnement du futur logiciel.

Nous verrons que les principes de la POO relèvent simplement du bon sens et n'ont rien de véritablement complexe.

3 Un peu d'histoire

Les concepts de base de la POO sont apparus dans les années 60 avec le langage Simula-67, puis ont été complétés dans les années 70 avec SmallTalk 71 et SmallTalk 80, eux-mêmes largement inspirés de Simula-67.

Durant les années 80 et 90 plusieurs nouveaux langages nativement objet apparaissent : C++, Lisp, Objective C, Java.

A partir des années 90, la POO s'impose comme la technique de programmation de référence dans l'ensemble de l'industrie informatique.

Durant cette épopée, certains langages sont apparus étant nativement orientés objets. D'autres ne l'étaient pas et le sont devenu avec plus ou moins de succès et en présentant parfois certaines particularités.

Aujourd'hui les langages objets sont très nombreux. On trouve parmi les plus répandus : Java, C++, C#, PHP, Objective C, Delphi (Pascal), et même le vénérable Cobol, apparu en 1959, est devenu un langage objet en 2002.

4 Classes et objets

4.1 Une petite analogie

Pour présenter les concepts il nous faut un thème. Choisissons donc un domaine qui se prête bien au jeu de la POO : les bateaux.

Dans le monde des chantiers navals, on construit des bateaux de toutes tailles et pour divers usages.

En premier lieu, un plan est élaboré. Le plan définit toutes les caractéristiques nécessaires à la construction du futur bateau :

- Le nombre de ponts.
- Les mesures.
- Le nombre de salles
- Le nombre de membrures
- Epaisseur de la coque.
- ...

Grâce au plan, il est possible de construire autant de bateaux que nécessaires. Ils seront tous conformes au plan en termes de structure, mais auront des attributs personnalisés comme :

- La couleur de la coque.
- Le nom du bateau.
- L'immatriculation.
- ...

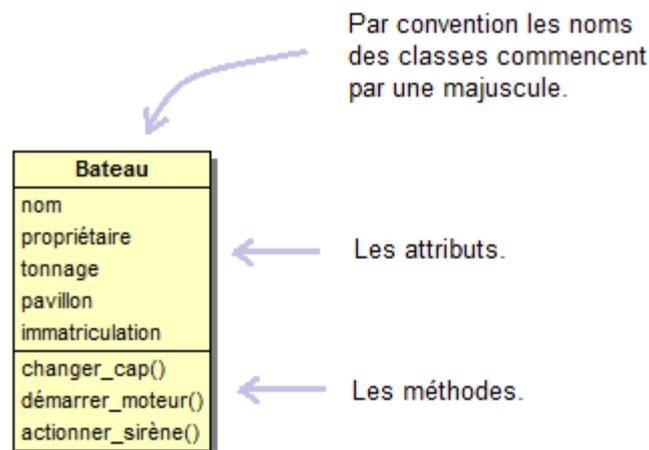
En POO, le plan de construction s'appelle une classe, et les bateaux sont des objets créés à partir de cette classe.

4.2 Classe

La classe décrit la structure des objets qui seront instanciés. Cette structure est composée :

- Des messages que l'objet sera en mesure de traiter. On parle aussi de méthodes. Un objet bateau doit par exemple être en mesure de traiter les messages suivants :
 - démarrer_moteur
 - changer_cap
 - actionner_sirène
- D'attributs : caractéristiques que possèdera l'objet et dont les valeurs pourront être communes à l'ensemble des objets, ou bien le plus souvent propres à chaque objet. On imagine par exemple qu'un objet bateau doit posséder des attributs comme :
 - Un nom
 - Un propriétaire
 - Une immatriculation
 - Un pavillon
 - Un tonnage

Les classes sont représentées de cette manière :



4.3 Objet

Un objet est créé (instancié) en mémoire centrale, par copie d'une classe. Dès lors l'objet devient un élément autonome, structuré selon sa classe d'origine, mais dont les attributs ont des valeurs propres. D'ailleurs tout objet, à sa création, se voit attribuer automatiquement par le langage, un

identifiant unique appelé référence. Cette référence est utilisée en interne par le langage lui-même, pour gérer par exemple l'élimination des objets en fin de vie.

La référence permet également qu'un message arrive bien à l'objet à qui il est adressé. L'objet qui émet le message doit par conséquent posséder la référence de l'objet destinataire.

Il faut distinguer d'une part, la création de la référence, et d'autre part l'attribution d'un objet à cette référence. Il s'agit de deux opérations distinctes. Dans la ligne de code suivante on crée une référence appelée B1 et nous indiquons qu'elle est du type Bateau. A ce stade la référence existe mais ne pointe sur rien :

B1 est un Bateau

Dans la ligne de code suivante, nous instancions un objet Bateau que nous attribuons à notre référence B1 (l'opérateur d'instanciation 'new' est utilisé ici car existant dans plusieurs langages objet majeurs) :

B1 = new Bateau

Les deux lignes peuvent être combinées en une seule :

B1 est un Bateau = new Bateau

Une classe peut générer autant d'objets que nécessaire. La seule limite est la taille de la mémoire centrale de l'ordinateur.

Certaines techniques de programmation permettent d'enregistrer un objet dans une base de données ou sous forme d'un fichier. On parle alors de **persistance des objets** ou **sérialisation**. Le but est de mettre l'objet de côté pour pouvoir l'éliminer de la mémoire centrale et le récupérer ultérieurement.

5 Cycle de vie d'un objet

Les objets vont naître, vivre et mourir dans la mémoire centrale de l'ordinateur, pendant l'exécution d'un programme informatique. L'ensemble de ces objets forment véritablement le programme.

L'activité des objets consiste à échanger des messages. Généralement un objet reçoit un message de la part d'un autre objet, effectue l'action correspondant à ce message et renvoie éventuellement une réponse à l'objet émetteur du message.

Pour qu'un objet A envoie un message à un objet B, il doit connaître sa référence. Le langage, qui gère en interne toutes les références utilisées par le programme, peut constater par exemple qu'un objet n'est plus référencé par aucun autre. Il ne peut plus recevoir de messages. Cet objet est en quelque sorte sorti du programme. La seule solution sera de l'éliminer de la mémoire centrale. De nombreux langages effectuent ce nettoyage de manière automatique.

5.1 Instanciation

Le verbe **instancier** désigne l'action de créer un objet à partir d'une classe. On peut rencontrer l'expression « instance de classe » qui désigne tout simplement un objet.

Durant sa création l'objet exécute notamment une méthode spéciale appelée **constructeur** qui peut contenir certaines actions à effectuer obligatoirement par l'objet. La méthode constructeur() peut également être simplement vide.

C'est bien évidemment le concepteur de la classe qui a défini le contenu de la méthode constructeur. Une fois la méthode constructeur exécutée, l'objet est disponible pour recevoir ses premiers messages.

5.2 Phase active

Pendant sa phase active, l'objet ne fait que répondre aux messages qu'on lui envoie. Chaque message reçu correspond à une méthode qui contient un traitement à effectuer par l'objet. Il peut s'agir d'un traitement extrêmement simple ou beaucoup plus complexe.

Si l'objet reçoit un message qu'il ne connaît pas, alors une erreur d'exécution est générée.

5.3 Elimination

Il arrive un moment où l'objet n'est plus nécessaire. Il y a au moins 3 raisons à cela :

- Le programme se termine.
- Le programme a accompli une certaine tâche, et les objets dont il a eu besoin ne sont plus nécessaires.
- L'objet n'est plus référencé.

Dans ces cas-là, les objets doivent être supprimés de la mémoire centrale. Le but est évidemment de récupérer de l'espace mémoire.

Dans les langages d'anciennes générations, la suppression des objets était effectuée par le développeur. Ce dernier devait veiller à ce que le programme supprime bien les objets après usage. Si cette tâche était mal accomplie, certains objets pouvaient rester indéfiniment en mémoire centrale alors qu'ils n'étaient plus utilisés.

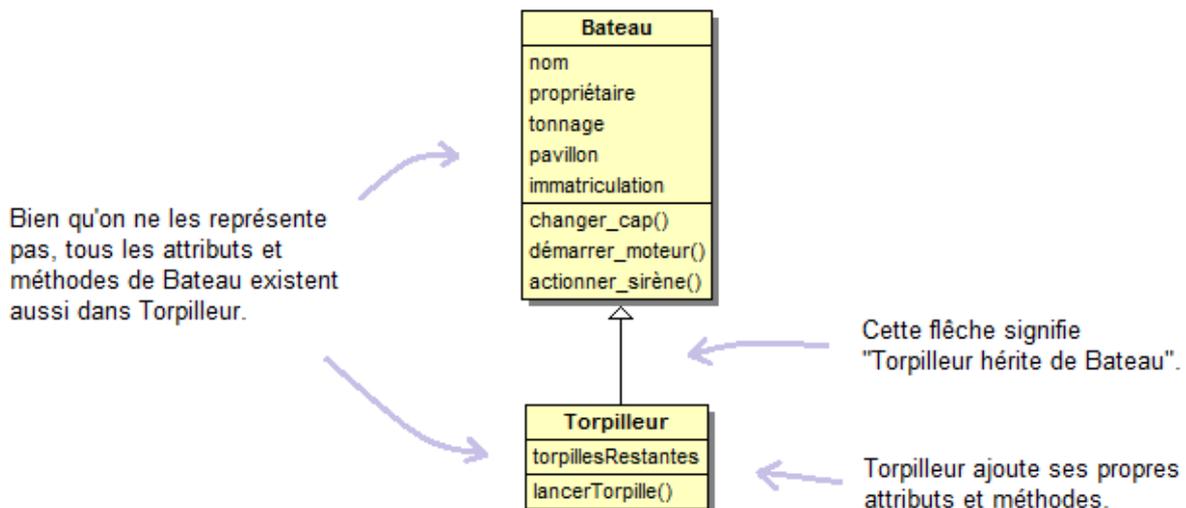
Pour remédier à cela et décharger le développeur de cette tâche ingrate, un système de suppression automatique d'objets est apparu dans les langages objets modernes. Le principe de ce système, appelé ramasse-miettes, est simple : tout objet qui n'est plus référencé est supprimé de la mémoire centrale. Le développeur a cependant toujours la possibilité de supprimer explicitement un objet.

6 Héritage

En POO, l'héritage est un automatisme permettant de récupérer les attributs et les méthodes d'une classe.

Qui hérite ?

Un objet instancié à partir d'une classe, hérite des attributs de sa classe. Mais la puissance de l'héritage se trouve dans l'héritage entre classes. Reprenons notre classe Bateau et créons une nouvelle classe héritant de la classe Bateau :



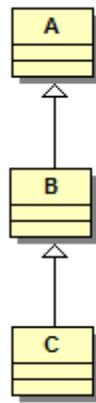
La classe Bateau est la **classe mère** ou **classe parent**.

La classe qui hérite est souvent appelée **classe fille** ou **classe dérivée** ou **sous classe**.

Une classe dérivée ajoute des attributs et des méthodes à son patrimoine hérité. On obtient de ce fait une classe plus spécialisée que ne l'est sa classe mère. Ainsi les classes mères sont conçues pour être générales : elles possèdent des attributs et des méthodes qui concernent toutes les classes filles. A ce titre on constate que dans la classe Bateau, la méthode démarrer_moteur() n'a pas sa place, car elle n'aurait pas de sens par exemple pour une classe fille « Voilier » (à supposer que ce voilier n'a pas de moteur).

6.1 Héritage simple

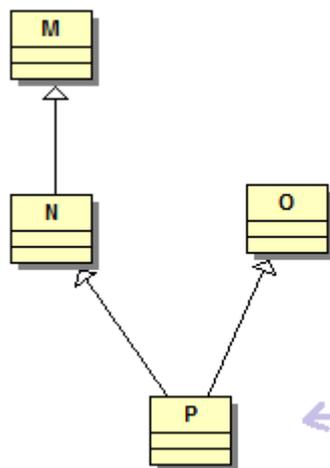
Avec l'héritage simple, une classe n'hérite directement que d'une et une seule classe.



Héritage simple: il peut y avoir plusieurs niveaux d'héritage, mais à chacun d'eux, une classe n'hérite que d'une et une seule classe.

6.2 Héritage multiple

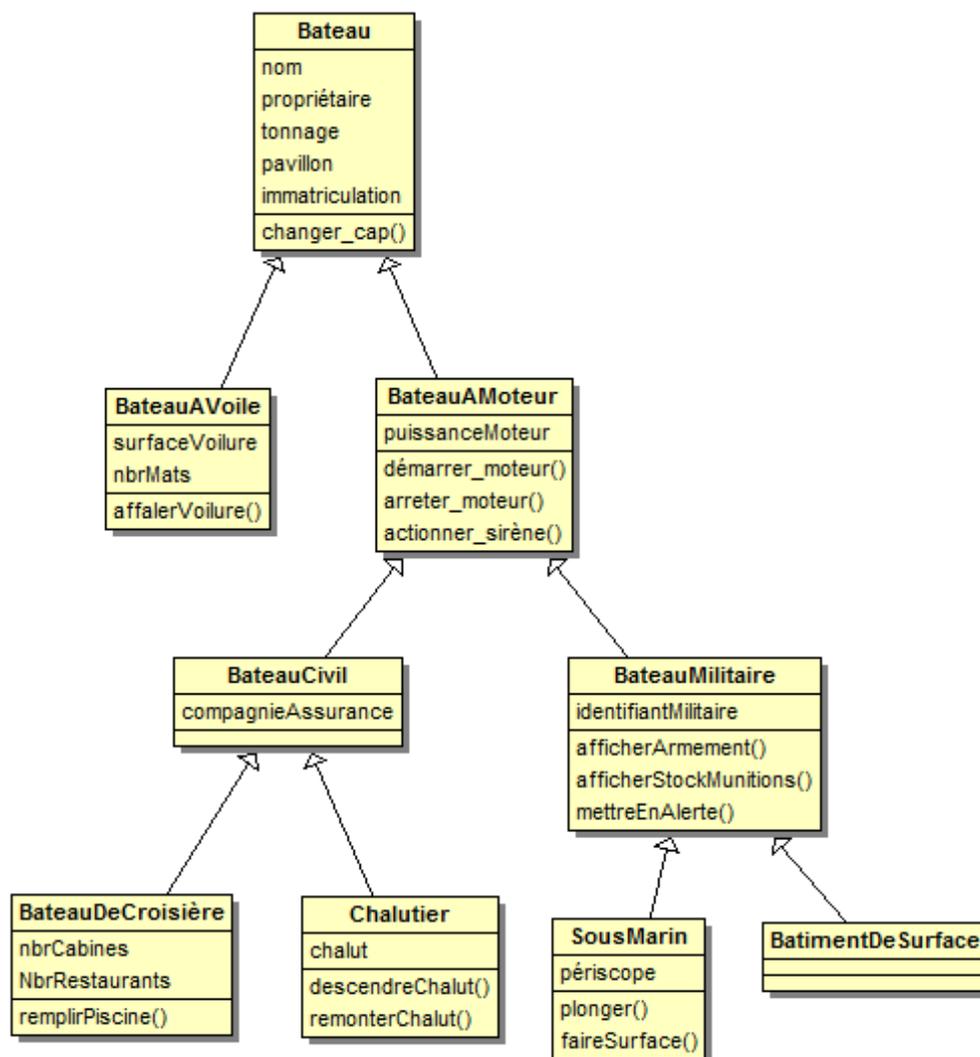
Avec l'héritage multiple, une classe peut hériter directement de plusieurs classes. Cette particularité a tendance à disparaître dans les langages orientés objets récents, car source de confusions et de difficultés de maintenance. A la place est proposé un mécanisme basé sur l'héritage simple, associé à la technique des **interfaces** (voir le chapitre concerné).



Héritage multiple: la classe P hérite à la fois des classes N et O.

7 Modèle de classes

Exploitions maintenant pleinement l'héritage. Le diagramme ci-dessous est un exemple de diagramme de classes. L'analyste élabore cette modélisation pour répondre au mieux aux spécifications du futur logiciel. Un diagramme de classes n'est pas une science exacte. Il y a toujours plusieurs approches, plusieurs variantes possibles.



Quelques remarques :

- Une classe peut avoir plusieurs classes filles directes.
- Il peut y avoir plusieurs niveaux d'héritage (classe mère, classe fille, classe petite fille, etc...)

- Une classe possède implicitement les attributs et méthodes de toutes ses classes parents. La classe Chalutier par exemple, possède ses propres attributs et méthodes, plus ceux de la classe BateauCivil, plus ceux de la classe BateauAMoteur, plus ceux de la classe Bateau.
- Plus on descend dans l'arbre d'héritage, plus les classes se spécialisent et s'enrichissent.
- On constate que la classe BatimentDeSurface n'a pas d'enrichissement : elle n'ajoute ni attributs ni méthodes à son patrimoine hérité. Cela peut être le choix du concepteur à un moment donné, et ne pose pas de problème technique.
- Il est théoriquement possible d'instancier des objets à partir de n'importe laquelle de ces classes, mais nous verrons plus loin la notion de **classe abstraite** dont la particularité est d'interdire la création d'objets, et ne permet que la création de sous classes.

8 Encapsulation

Les objets possèdent des attributs et des méthodes, et dans leur grande générosité mettent ces ressources à la disposition des autres objets. Mais certaines ressources mériteraient d'être protégées. Si notre classe SousMarin avait un attribut contenant le code de mise à feu des missiles, il serait prudent qu'il ne soit pas librement accessible à tous les objets.

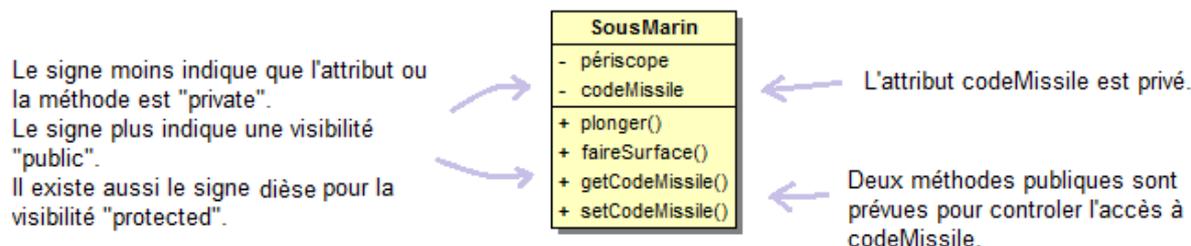
Protéger ainsi les attributs et méthodes d'une classe, s'appelle l'encapsulation. Ces ressources auront un accès contrôlé voire même interdit. Dans ce dernier cas, cela signifie que la ressource est à l'usage exclusif de l'objet lui-même.

L'encapsulation s'appuie sur la visibilité que le concepteur donne aux attributs et méthodes d'une classe. En POO on trouve toujours au moins les 3 niveaux de visibilité suivants :

- **Public** : l'attribut ou la méthode sera accessible aux objets de toutes les classes du programme.
- **Protected** (protégé) : l'attribut ou la méthode sera accessible aux objets de cette classe ainsi qu'aux objets de toutes ses sous classes.
- **Private** (privé) : l'attribut ou la méthode ne sera pas hérité par les sous classes. L'objet instancié à partir de cette classe possèdera la ressource privée, mais celle-ci ne sera pas directement accessible par les autres objets.

Une approche commune pour protéger un attribut ou une méthode, est de le rendre privé tout en prévoyant des méthodes publiques pour en contrôler l'accès.

Reprenons notre classe SousMarin et donnons-lui l'attribut codeMissile nécessaire à la mise à feu des missiles. Cette donnée sensible aura une visibilité **private**, et sera donc invisible des autres objets. Mais il est nécessaire que cette donnée puisse être lue ou modifiée sous certaines conditions. Nous ajoutons donc 2 méthodes avec une visibilité **public** :



- getCodeMissile() : permet de lire le code missile. Cette méthode pourra par exemple exiger la saisie d'un mot de passe.
- setCodeMissile() : méthode utilisée pour attribuer ou modifier le code missile. Là encore la méthode pourra exiger la saisie d'un mot de passe, et contrôler par ailleurs que le code missile fourni, répond à certains critères.

Ces deux méthodes sont appelées des **accesseurs**. Vues de l'extérieur de l'objet, elles représentent l'unique moyen d'accéder à une donnée encapsulée.

9 Classes abstraites et interfaces

9.1 Classes abstraites

Une classe est définie comme abstraite en utilisant le modificateur **abstract** dans sa définition. Par convention, et pour faciliter leur repérage, le nom des classes abstraites apparaît en italique dans les modèles de classes. Par opposition les classes qui ne sont pas abstraites sont souvent appelées **classes concrètes**.

Les classes abstraites contiennent comme les classes concrètes, des attributs et des méthodes. Mais on peut y trouver également des **méthodes abstraites**. Il s'agit de méthodes « vides » c'est-à-dire

non implémentées. Seul le nom de la méthode, ses paramètres en entrée, et le type de sa valeur de retour, sont définis.

Par ailleurs une classe qui contient au moins une méthode abstraite, doit obligatoirement être elle aussi abstraite.

La particularité des classes abstraites est qu'elles ne peuvent pas créer d'objets. A quoi servent-elles dans ce cas ?

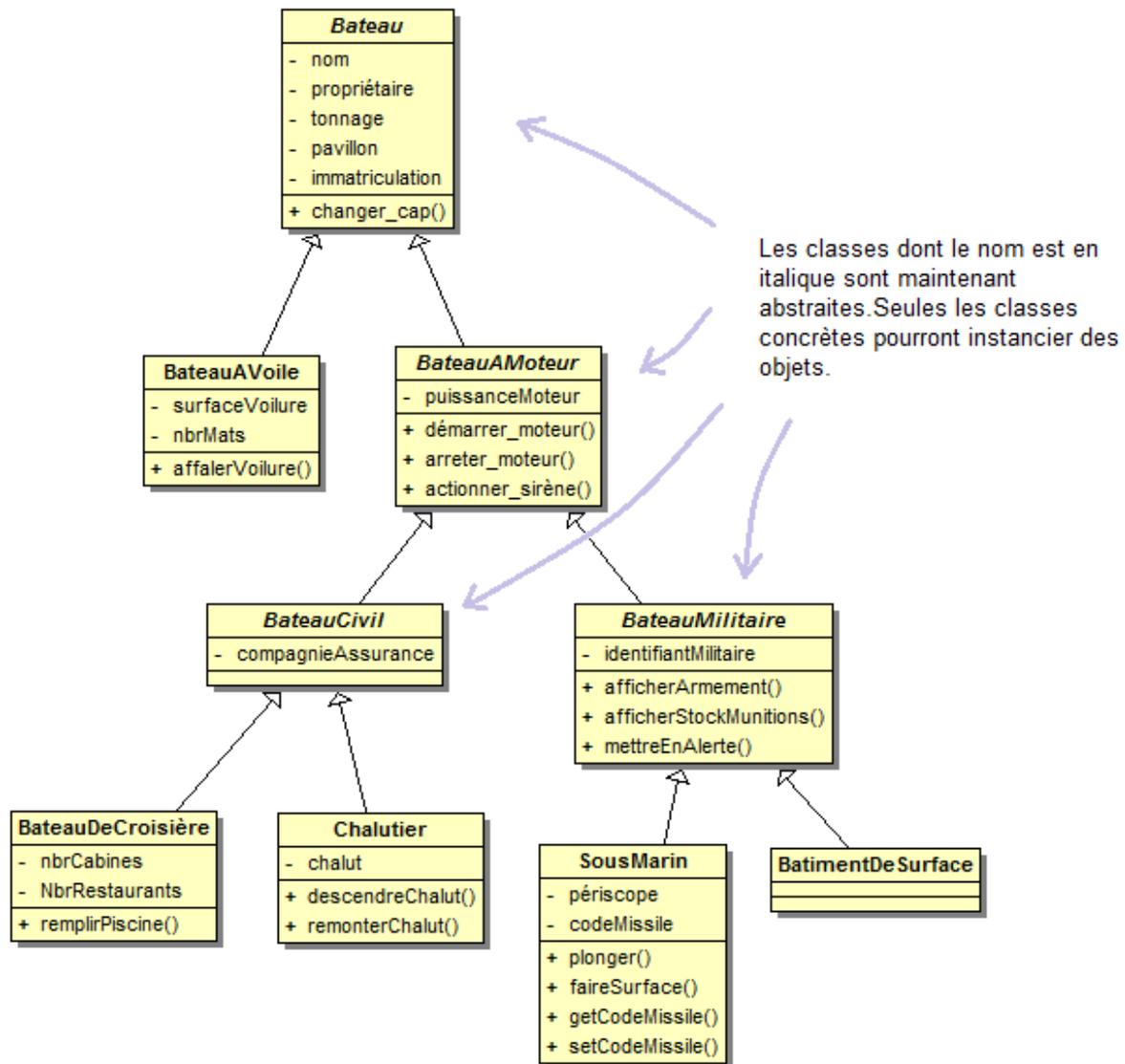
Elles sont utilisées pour déléguer l'implémentation de ses méthodes abstraites, aux sous classes. La classe abstraite joue alors le rôle d'un prototype de classe, partiellement implémenté, dont les sous classes devront terminer le travail d'implémentation. On comprend alors pourquoi les classes abstraites ne doivent pas créer d'objets.

Une classe dérivée d'une classe abstraite n'est pas obligée d'implémenter toutes les méthodes abstraites. Elle peut même n'en implémenter aucune, mais les sous classes restent abstraites tant qu'il reste au moins une méthode abstraite.

Cette technique de délégation est une des formes de polymorphisme (voir le chapitre concerné). En effet les sous classes pourront chacune implémenter une méthode abstraite selon leur besoin spécifique. Une même méthode aura donc un comportement différent selon la sous classe : c'est du polymorphisme.

A titre d'exemple, si nous reprenons notre modèle de classes des bateaux, le concepteur aura peut être déclaré les classes de haut niveau comme abstraites. Seules les classes suffisamment enrichies pourront créer des objets.

A noter que dans cet exemple, les classes abstraites ne contiennent pas de méthodes abstraites (leurs noms seraient en italique). Ce n'est techniquement pas une obligation.



9.2 Interfaces

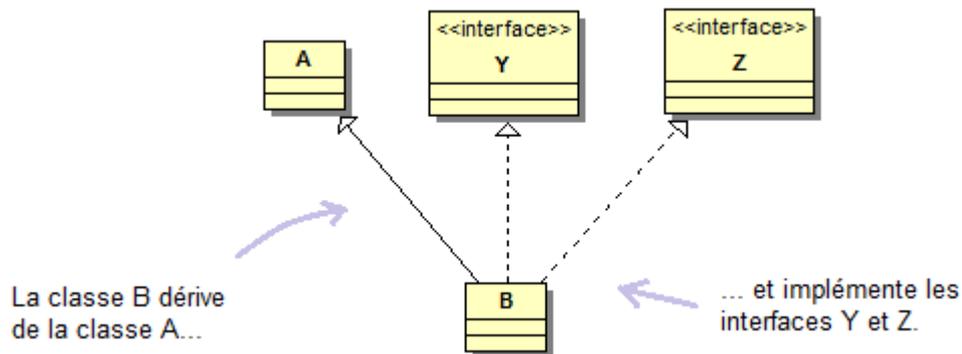
Poussons le concept des classes abstraites à l'extrême : imaginons une classe abstraite qui n'a aucun attribut et ne possède que des méthodes abstraites : on obtient une interface.

Les interfaces se déclarent avec le mot clef **interface**. Ce ne sont donc pas des classes, mais elles offrent les possibilités suivantes :

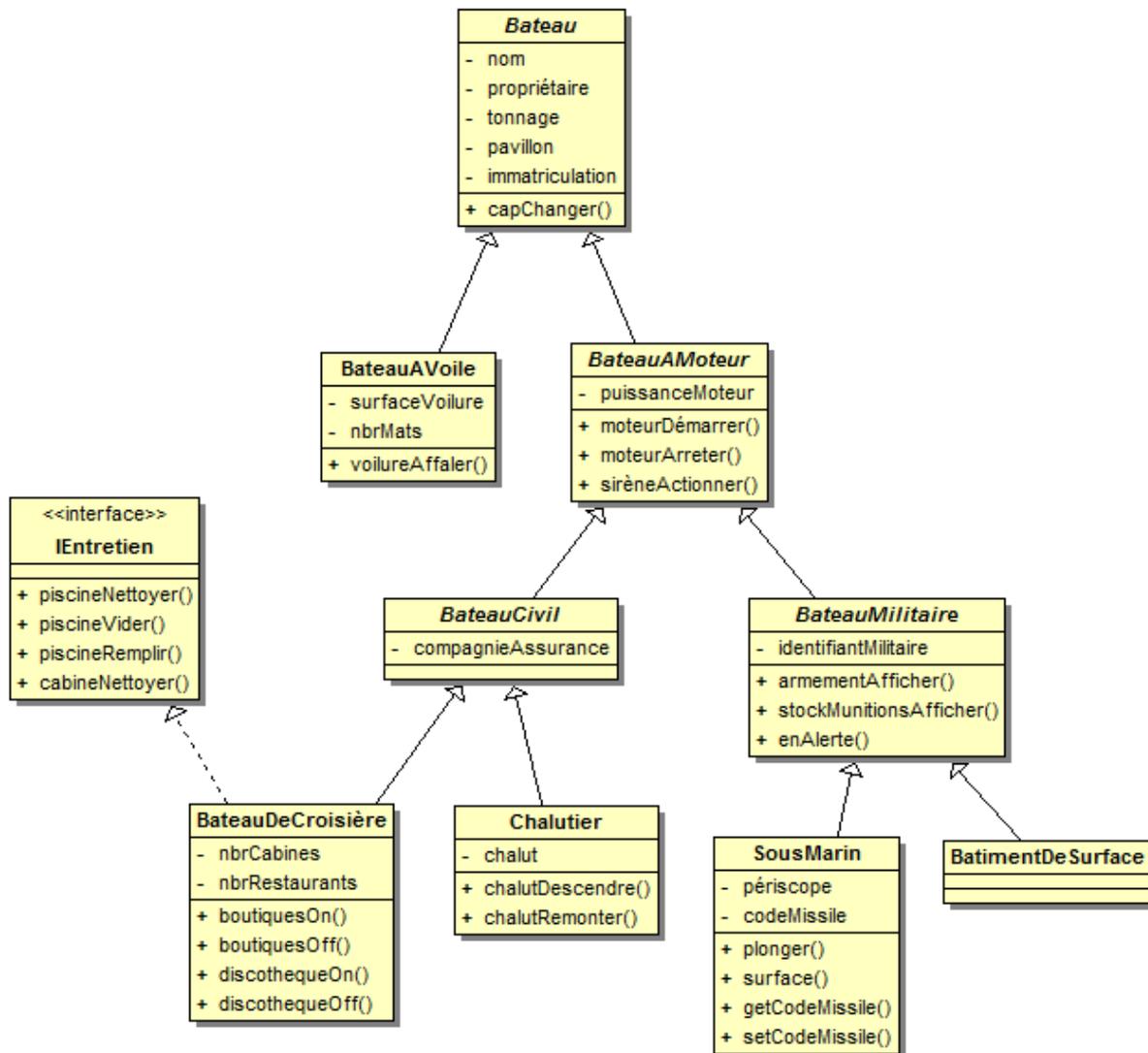
- Elles ne contiennent que des définitions de méthodes (nom de la méthode, paramètres en entrée, type de la valeur de retour).
- Si une classe implémente une interface, alors cette classe sera tenue de définir toutes les méthodes de l'interface. L'interface est donc comme un contrat que la classe doit respecter.

- Comme nous l'avons vu avec l'héritage simple (le plus répandu), une classe ne peut dériver que d'une seule classe mère. Par contre une classe peut implémenter plusieurs interfaces (il n'y a pas de limite). Voilà un outil puissant qui remplace avantageusement l'héritage multiple.
- Il est possible de dériver des interfaces comme on le fait pour les classes.

Les interfaces sont repérées par le stéréotype <<interface>>. Notez que les flèches d'héritage sont continues alors que les flèches d'implémentation d'interfaces, sont en pointillés.



Ci-dessous un exemple plus concret : la classe BateauCroisiere hérite de BateauCivil, mais implémente en même temps l'interface IEntretien, dont elle va devoir définir les méthodes. La classe pourra donc recevoir et traiter de nouveaux messages spécifiques à l'entretien du bateau.



9.3 Autre types de classes

Nous avons vu que le modificateur **abstract** permettait de définir une classe comme abstraite.

Il existe 3 autres modificateurs de classes :

- **Final** : Une classe déclarée **final** ne pourra pas avoir de sous classe, c'est-à-dire qu'on interdit l'héritage. Cette classe ne pourra que créer des objets. C'est en quelques sortes l'inverse des classes abstraites.
- **Private** : Cette classe n'est accessible qu'à partir du fichier où elle est définie. Elle sera invisible des autres classes et objets.
- **Public** : (modificateur par défaut) : la classe est visible par toutes les autres.

Notez que les modificateurs **abstract**, **final**, **public** et **private**, sont mutuellement exclusifs.

10 Modificateurs de méthodes

Les méthodes aussi ont leurs modificateurs :

Final : Toute méthode définie avec le modificateur **final** ne pourra être ni surchargée ni redéfinie dans les sous classes.

Private : la méthode ne sera visible que par la classe, et sera invisible des autres classes, y compris les classes dérivées.

Protected : la méthode sera visible par la classe ainsi que ses sous classes.

Static : la méthode ne sera pas copiée dans les objets créés par la classe. Elle n'existera donc qu'en un seul exemplaire, dans la classe elle-même, et sera utilisée communément par tous les objets créés par cette classe.

Public : la méthode sera visible par toutes les classes (modificateur par défaut).

11 Techniques de polymorphisme

En POO le polymorphisme (littéralement : qui peut prendre plusieurs formes), est un terme générique qui indique qu'un traitement peut changer de nature selon le contexte dans lequel il s'exécute.

Le polymorphisme, qui est un des fondement de la POO et en particulier des **design patterns**, ces modèles de programmation objet, recouvre plusieurs aspects.

11.1 Surcharge et redéfinition de méthodes

Avant d'entrer dans le vif du sujet, définissons les deux manières de créer plusieurs méthodes ayant le même nom :

11.1.1 Surcharge

Dans une même classe, il est possible de créer plusieurs méthodes de même nom, à condition que leurs signatures soient différentes. La signature est définie par le nombre et le type des paramètres en entrée, ainsi que le type de la valeur de retour de la méthode. C'est la **surcharge de méthode**.

Au moment où la méthode est appelée, le langage choisira automatiquement la méthode qui correspond au nombre et au type des paramètres fournis dans l'appel.

11.1.2 Redéfinition

La redéfinition d'une méthode ne peut intervenir que dans le cadre d'une relation d'héritage. La classe fille va redéfinir une méthode de la classe mère, en lui donnant une nouvelle implémentation qui viendra remplacer celle de la classe mère. La méthode redéfinie garde la même signature.

La méthode redéfinie peut éventuellement intervenir, non pas en remplacement, mais en complément du traitement de la méthode de la classe mère. Dans ce cas la méthode redéfinie commencera par appeler la méthode de la classe mère, puis exécutera le traitement complémentaire

11.2 Polymorphisme ad hoc

Il s'agit d'une forme de polymorphisme automatique dans lequel le langage, au moment de l'exécution ou de la compilation, va décider de la conduite à tenir selon le type des données impliquées dans le traitement.

Prenons par exemple l'opérateur « + » qui dans certains langages peut avoir deux interprétations :

- Additionner des valeurs.
- Concaténer des chaînes de caractères.

Selon les lignes de codes suivantes, un travail implicite d'adaptation du traitement, voire de conversion de données, sera effectué automatiquement par le langage :

- **23 + 7** : ici il le programme doit additionner deux entiers. Les données étant du même type, aucune conversion n'est nécessaire.
- **16,3 + 8** : ici le programme doit additionner une valeur réelle et un entier. L'entier doit être converti en valeur réelle et le résultat sera une valeur réelle. Le développeur n'a pas à se soucier de ces conversions, elles seront effectuées par le polymorphisme ad hoc.
- **'Il est minuit' + 'docteur Schweitzer.'** : ici le programme doit concaténer deux chaînes de caractères. Le polymorphisme ad hoc va automatiquement adapter le traitement au type des arguments.

11.3 Polymorphisme paramétrique

Le polymorphisme paramétrique correspond à la surcharge de méthode indiquée plus haut : une méthode est définie plusieurs fois dans la même classe avec le même nom de méthode, mais avec des signatures différentes.

Reprenons notre classe Submarine. Elle possède une méthode `dive()`. Précisons qu'elle prend en paramètre, un entier qui donne la profondeur à atteindre :

```
dive(depth : integer)
```

Surchargeons cette méthode en ajoutant une seconde méthode `dive()` prenant en paramètres, la profondeur ainsi qu'un booléen indiquant s'il s'agit d'une plongée rapide :

```
dive(depth : integer, fastDive : boolean)
```

Nous avons maintenant deux méthodes `dive()` dans la même classe. Elles ont des signatures différentes, et bien évidemment des implémentations également différentes.

Lors de l'appel d'une méthode `dive()`, le langage sera capable de choisir automatiquement celle qui convient, en fonction des paramètres passés dans l'appel : c'est du polymorphisme paramétrique.

11.4 Polymorphisme d'inclusion

Ce type de polymorphisme s'appuie sur l'héritage. Dans le chapitre concernant les objets, nous avons vu que pour créer une référence il fallait indiquer son type. Au moment d'attribuer un objet à une référence plusieurs cas peuvent se produire :

1. Le type de l'objet instancié, est différent de celui de la référence, et il n'y a pas de lien d'héritage. Dans ce cas une erreur sera lancée à la compilation ou à l'exécution selon le langage utilisé.
2. Le type de l'objet instancié, est le même que celui de la référence. Dans ce cas bien évidemment la référence et l'objet sont compatibles :

```
B1 est un BateauAVoile = new BateauAVoile
```

3. Le type de l'objet instancié, est une sous classe du type de la référence :

```
B2 est un BateauAMoteur = new Chalutier
```

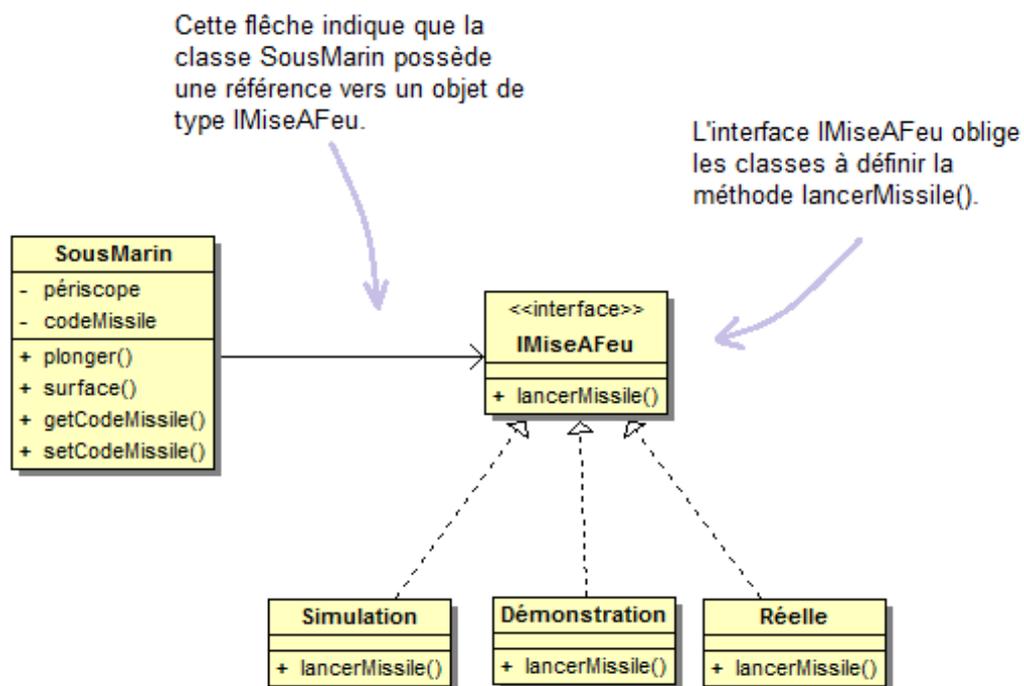
La ligne de code est correcte car la classe `Chalutier` est indirectement une classe fille de `BateauAMoteur`. En POO, l'habit fait le moine. B2 croit manipuler un objet `BateauAMoteur` alors qu'un fait il s'agit concrètement d'un `Chalutier`, et de fait ce n'est pas aberrant car un `Chalutier` est un `BateauAMoteur` car il en a hérité toutes les caractéristiques.

Grâce à cela, il va être possible d'introduire un polymorphisme d'inclusion : prenons par exemple la méthode `moteurDémarrer()` définie dans la classe `BateauAMoteur`, et supposons qu'elle ait été redéfinie dans la classe `Chalutier` pour prendre en compte certaines spécificités de la manière de démarrer le moteur d'un chalutier. Et bien quand le message `moteurDémarrer()` sera envoyé à la référence `B2`, c'est la méthode `moteurDémarrer()` de la classe `Chalutier` qui sera exécutée sans que `B2` n'en sache rien.

On peut résumer en disant que vu de `B2` :

- Le type apparent est `BateauAMoteur`.
- Le type réel (caché) est `Chalutier`

Et ce n'est pas tout : le polymorphisme d'inclusion fonctionne également très bien avec les interfaces. Imaginons la situation suivante :



Généralement dans ce type de montage, ce n'est pas la classe `SousMarin` qui instancie l'un des 3 objets de mise à feu, mais un objet de plus haut niveau qui contrôle justement ce genre de choses. Au final, l'objet `SousMarin` possède bien une référence sur un objet `IMiseAFeu`, sans savoir quel est l'objet concret qui a été instancié, parmi les 3 possibles.

Pour lancer son missile, l'objet `SousMarin` doit invoquer la méthode `lancerMissile()` sur son objet `IMiseAFeu`. Evidemment les implémentations de `lancerMissile()` sont différentes dans les 3 classes. Le sous-marin lui, n'a même pas connaissance de l'existence des 3 classes. Il ne voit qu'un objet de type `IMiseAFeu`.

Cette méthode `lancerMissile()` qui peut avoir plusieurs implémentations, c'est encore du polymorphisme d'inclusion.

12 Lexique

Classe : Regroupement de méthodes et d'attributs, servant de modèle pour instancier des objets.

Classe abstraite : Classe ne pouvant pas instancier d'objets et ne pouvant que servir de classe mère pour des sous classes.

Classe concrete : Par opposition aux classes abstraites, les classes concrètes peuvent instancier des objets.

Constructeur : Méthode existant dans toute classe, et exécutée au moment de la création d'un objet. Le développeur y place généralement des opérations préparatoires.

Design patterns : Il existe une vingtaine de patterns. Ce sont des modèles de programmation objet, chacun répondant à une problématique particulière.

Destructeur : Méthode existant dans toute classe, et exécutée juste avant la destruction d'un objet. Le développeur y place généralement des opérations de nettoyage.

Encapsulation : technique de programmation objet visant à contrôler voire interdire l'accès à certains attributs ou méthodes d'une classe.

Héritage : Relation entre deux classes. La classe qui hérite, récupère le patrimoine de sa classe mère.

Instancier : Pour une classe, le fait de créer un objet.

Interface : ensemble de méthodes non implémentées, définies dans un fichier. Une classe implémentant une interface, est tenue de définir toutes les méthodes de l'interface.

Méthode abstraite : Méthode non implémentée définie dans une classe, et dont l'implémentation est déléguée aux sous classes.

Objet : Une classe instanciée (créée) des objets. Les objets sont des copies de la classe.

Polymorphisme : littéralement : qui a plusieurs formes. Se dit d'une méthode apparaissant dans plusieurs classes d'une même famille, avec le même nom mais avec des implémentations différentes.

Persistence : Le fait de stocker une information sur un support autre que la mémoire centrale de l'ordinateur, ce qui permet de récupérer cette information même après un redémarrage de l'ordinateur.

Redéfinition : Une méthode est redéfinie quand une nouvelle implémentation lui est donnée dans une sous classe.

Signature d'une méthode : ensemble des informations qui permettent de distinguer les méthodes entre elles : nom de la méthode, nom et type des paramètres en entrée, type de la valeur de retour.

Surcharge : Une méthode est surchargée quand, dans la même classe, une méthode de même nom est créée mais avec une signature différente.

Sérialisation d'un objet: enregistrement d'un objet sous forme d'un fichier, ce qui permet de recharger cet objet ultérieurement, ou de transmettre cet objet via le réseau.

Visibilité : périmètre plus ou moins restreint associé à un attribut, une méthode ou une classe, à l'aide des mots clef réservés, et qui détermine qui pourra accéder ou pas à cet élément.

Fin du document

Copyright 2012 www.smaltek.fr

Toute reproduction totale ou partielle interdite sans l'autorisation de l'auteur.