

ÉCOLE NATIONALE D'INGÉNIEURS DE BREST



— Cours d'Informatique S4 —

Programmation Orientée Objet (Concepts)

CÉDRIC BUCHE

buche@enib.fr

version du 28 janvier 2014

Programmation Orientée Objet

— Concepts —

Cédric Buche

École Nationale d'Ingénieurs de Brest (ENIB)

28 janvier 2014



Table des matières

— Cours —	3
Avant-propos	3
1 Justification du modèle objet (1 UC)	8
2 Concepts fondateurs (1 UC)	16
3 Encapsulation (1 UC)	25
4 Hiérarchie (1 UC)	35
5 Collaborations entre objets (1 UC)	44
6 Pour aller plus loin	51
— Labo —	61
7 Rappels python (1 UC)	61
8 Concepts fondateurs (1 UC)	67
9 Encapsulation (1 UC)	74
10 Collaboration et héritage (1 UC)	77
11 Pour aller plus loin	82
— Exercice de synthèse —	85
12 Exercice : Gaia	85
13 Correction : GAIA	89
Références	98

Ces notes de cours accompagnent les enseignements d'informatique du 4^{ième} semestre (S4) de Programmation Orientée Objet (POO) de l'Ecole Nationale d'Ingénieurs de Brest (ENIB : www.enib.fr). Leur lecture ne dispense en aucun cas d'une présence attentive aux cours ni d'une participation active aux travaux dirigés.

Cours

Avant-propos

Sommaire

Objectifs	4
Objectifs du cours	4
Objectifs pour la suite du cursus	4
Organisation	5
Équipe pédagogique	5
Séances de cours et de laboratoire	5
Planning prévisionnel	5
Évaluation	7

Objectifs

Objectifs du cours

L'objectif principal des enseignements d'informatique S4-POO de l'ENIB est l'acquisition des notions fondamentales de la *programmation orientée objet*. Plus précisément, nous étudierons successivement :

1. Le modèle objet :

- ▷ Objet
- ▷ Classe
- ▷ Encapsulation
- ▷ Collaboration
- ▷ Héritage
- ▷ Abstraction

2. La partie statique du langage UML :

- ▷ Diag. de cas d'utilisation
- ▷ Diag. de classes
- ▷ Modèle d'interaction
 - ◊ Diag. de séquence
 - ◊ Diag. de communication

Définition 0.1. *Programmation orientée objet : ou programmation par objet, est un paradigme de programmation informatique qui consiste en la définition et l'assemblage de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre.*

Définition 0.2. *UML : (en anglais Unified Modeling Language, "langage de modélisation unifié") est un langage graphique de modélisation des données et des traitements. C'est une formalisation très aboutie et non-propriétaire de la modélisation objet utilisée en génie logiciel.*

Objectifs pour la suite du cursus

Les notions de POO seront nécessaires pour le métier d'ingénieur en informatique, ainsi que pour la suite de vos études d'ingénieurs notamment pour les modules suivants :

- ▷ Prog. par objets (Java) en S5/S6
- ▷ Prog. par objets (C++) en S5/S6
- ▷ Modèles pour l'ingénierie des systèmes (UML) en S6
- ▷ Stage en entreprise en S8/S10

Organisation

Équipe pédagogique

BOSSER	Anne-Gwen	Cours + Laboratoire	bosser@enib.fr
DESMEULLES	Gireg	Cours + Laboratoire	desmeulles@enib.fr

Séances de cours et de laboratoire

Les enseignements d'informatique S4-POO de l'ENIB sont dispensés lors de 21h de séances de coursTD et de séances de laboratoire :

- ▷ Les cours ont lieu chaque semaine à raison de 3h toutes les 2 semaines
- ▷ Les séances de laboratoire ont lieu à raison de 3h toutes les 2 semaines. Elles se déroulent en salle informatique.

Planning prévisionnel

Semaine	LABO	CTD
37		<i>POO — Justification modèle objet</i> ctd cb
37		<i>POO — Concepts fondateurs</i> ctd cb
38	<i>POO — Rappels python</i> labo cb	
38	<i>POO — Concepts fondateurs</i> labo cb	
39		<i>POO — Encapsulation</i> ctd cb
39		<i>POO — Collaboration et hiérarchie</i> ctd cb
40	<i>POO — Encapsulation</i> labo cb	
40	<i>POO — Collaboration et héritage</i> labo cb	
41		<i>UML — Généralités</i> ctd cb
41		<i>UML — Classes</i> ctd cb
42	<i>UML — Classes</i> labo cb	
42	<i>UML — Classes</i> labo cb	
43		<i>UML — Classes</i> ctd cb
43		<i>UML — Classes</i> ctd cb
44	<i>Vacances</i>	
	<i>Toussaint</i>	

Semaine	LABO	CTD
45	<i>UML — Classes</i> labo cb	
45	<i>UML — Classes</i> labo cb	
46		<i>UML — Use Case (diag.)</i> ctd cb
46		<i>UML — Use Case (scénarios)</i> ctd cb
47	<i>UML — Use Case (diag.)</i> labo cb	
47	<i>UML — Use Case (scénarios)</i> labo cb	
48		<i>UML — Interactions (com)</i> ctd cb
48		<i>UML — Interactions (seq)</i> ctd cb
49	<i>UML — Interactions (com)</i> labo cb	
49	<i>UML — Interactions (seq)</i> labo cb	
50		
51		
51		

Évaluation

Types de contrôle L'évaluation des connaissances et des compétences acquises par les étudiants repose sur 3 types de contrôle : les contrôles d'attention, les contrôles de TD et les contrôles de compétences.

- ▷ *Contrôle d'attention* : il s'agit d'un QCM (questionnaire à choix multiples) auquel il faut répondre individuellement sans document, en 5' en fin de cours, et dont les questions portent sur des points abordés pendant ce cours. Ce type de contrôle teste directement l'acquisition de connaissances. Il permet d'évaluer "à chaud" la capacité d'attention des étudiants et les incite à une présence attentive afin de bénéficier au maximum des heures de présence aux cours.
- ▷ *Contrôle de TD* : il s'agit ici d'inciter les étudiants à réviser et à préparer activement les séances de laboratoire. En début de chaque séance de laboratoire, chaque élève doit répondre sans document en 5' aux questions d'un exercice ou d'un QCM portant sur les notions du laboratoire et du cours précédent.
- ▷ *Contrôle de compétences* : les contrôles de compétences (ou DS) durent 80' pendant une séance de cours.

Notation des contrôles d'attention et de TD Quel que soit le type de contrôle, un exercice cherche à évaluer un objectif particulier. Aussi, la notation exprimera la distance qui reste à parcourir pour atteindre cet objectif :

0 :	"en plein dans le mille!"	→ l'objectif est atteint
1 :	"pas mal!"	→ on se rapproche de l'objectif
2 :	"juste au bord de la cible!"	→ on est encore loin de l'objectif
3 :	"la cible n'est pas touchée!"	→ l'objectif n'est pas atteint

Ainsi, et pour changer de point de vue sur la notation, le contrôle est réussi lorsqu'on a 0 ! Il n'y a pas non plus de 1/2 point ou de 1/4 de point : le seul barème possible ne comporte que 4 niveaux : 0, 1, 2 et 3. On ne cherche donc pas à "grappiller" des points :

- ▷ on peut avoir 0 (objectif atteint) et avoir fait une ou deux erreurs bénignes en regard de l'objectif recherché ;
- ▷ on peut avoir 3 (objectif non atteint) et avoir quelques éléments de réponse corrects mais sans grand rapport avec l'objectif.

Figure 0.1. Exemple de QCM

Informatique

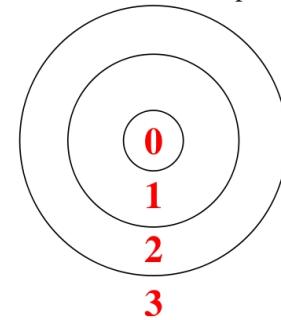
Contrôle Continu

Nom/Prenom :

Groupe :

- | | | | | | | |
|----------------------|---|---|---|---|---|---|
| 1. QCM titre 4 _____ | <input type="checkbox"/> (a) réponse fausse 4.2 | <input type="checkbox"/> (b) réponse fausse 4.1 | <input type="checkbox"/> (c) réponse fausse 4.5 | <input type="checkbox"/> (d) réponse juste 4 | <input type="checkbox"/> (e) réponse fausse 4.3 | <input type="checkbox"/> (f) réponse fausse 4.4 |
| 2. QCM titre 1 _____ | <input type="checkbox"/> (a) réponse juste 1 | <input type="checkbox"/> (b) réponse fausse 1.2 | <input type="checkbox"/> (c) réponse fausse 1.3 | <input type="checkbox"/> (d) réponse fausse 1.1 | | |
| 3. QCM titre 2 _____ | <input type="checkbox"/> (a) réponse juste 2 | <input type="checkbox"/> (b) réponse fausse 2.1 | <input type="checkbox"/> (c) réponse fausse 2.2 | | | |
| 4. QCM titre 3 _____ | <input type="checkbox"/> (a) réponse juste 3 | <input type="checkbox"/> (b) réponse fausse 3.1 | <input type="checkbox"/> (c) réponse fausse 3.2 | | | |

Figure 0.2. Notation : métaphore de la cible



1 Justification du modèle objet (1 UC)

Sommaire

1.1 Complexité	9
1.1.1 Complexité des problèmes	9
1.1.2 Difficulté du contrôle du processus de développement	9
1.1.3 Conséquences d'une complexité sans limites	9
1.1.4 Qualité du logiciel	9
1.2 Programmation procédurale	10
1.2.1 Paradigme de programmation procédurale	10
1.2.2 Limites de la programmation procédurale	10
1.3 Vers la Programmation Orientée Objet (POO)	12
1.3.1 Classe	12
1.3.2 Objet	12
1.3.3 Constructeur	13

Objectifs du Cours 1 / Labo 2 :

- ▷ Justifier le modèle objet
- ▷ Notion de classe :
 - ◊ définition
 - ◊ constructeur
- ▷ Notion d'objet :
 - ◊ instantiation
 - ◊ manipulation

1.1 Complexité

1.1.1 Complexité des problèmes

Les problèmes que les logiciels doivent solutionner comportent souvent des éléments extrêmement complexes. Considérons les exigences auxquelles doit satisfaire l'électronique d'un avion à réacteur ou d'un robot autonome. Le fonctionnement de ces systèmes est déjà difficile à cerner, et pourtant il faut y ajouter des exigences non fonctionnelles (et surtout implicites) telles que la facilité d'emploi, les performances, le coût, la robustesse ...

1.1.2 Difficulté du contrôle du processus de développement

La tâche fondamentale de l'équipe de développement de logiciel est de promouvoir une illusion de simplicité afin de protéger l'utilisateur de cette complexité externe arbitraire. Aussi, la taille n'est sûrement pas la principale vertu du logiciel. Ainsi, nous nous efforçons d'écrire moins de code, cependant nous avons parfois à faire face à un énorme volume de spécifications.

Exemple 1.1. *Un système d'exploitation : Tout le monde sait utiliser un système d'exploitation, pour donner un ordre de grandeur il y a 6 millions de lignes de code pour le noyau Linux 2.6.0*

1.1.3 Conséquences d'une complexité sans limites

« Plus un système est complexe, plus il est susceptible d'effondrement » [Peter, 1986]

Est ce qu'un entrepreneur songe à ajouter une nouvelle assise à un immeuble de 100 étages existant ? Ce genre de considérations est pourtant demandé par certains utilisateurs de logiciels.

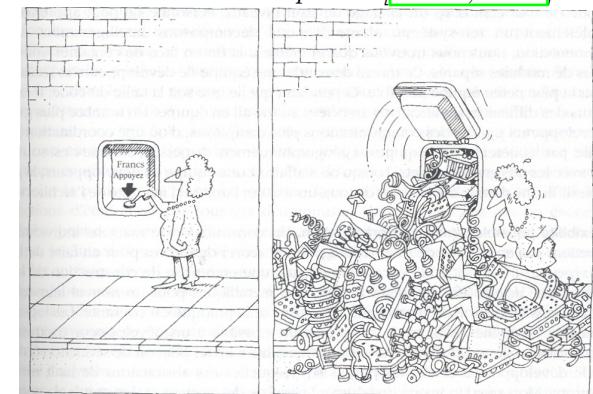
1.1.4 Qualité du logiciel

Le logiciel produit doit présenter les qualités suivantes :

1. **Extensibilité** : faculté d'adaptation d'un logiciel aux changements de spécification.
 - ▷ *simplicité de la conception* : une architecture simple sera toujours plus facile à modifier qu'une architecture complexe.
 - ▷ *décentralisation* : plus les modules d'une architecture logicielle sont autonomes, plus il est probable qu'une modification simple n'affectera qu'un seul module, ou un nombre restreint de modules, plutôt que de déclencher une réaction en chaîne sur tout le système.

Définition 1.1. *Système d'exploitation (SE, en anglais Operating System ou OS) est un ensemble de programmes responsables de la liaison entre les ressources matérielles d'un ordinateur et les applications informatiques de l'utilisateur.*

Figure 1.1. *La tâche de l'équipe de développement de logiciel est de donner l'illusion de la simplicité [Booch, 1992]*



2. Réutilisabilité : aptitude d'un logiciel à être réutilisé en tout ou en partie pour de nouvelles applications.

L'idée à retenir est de "ne pas réinventer la roue!". Il s'agit de programmer *moins* pour programmer *mieux*. C'est la finalité du modèle objet en programmation !! Il apporte une sémantique facilitant ce processus en programmation.

1.2 Programmation procédurale

1.2.1 Paradigme de programmation procédurale

En S1, vous avez appris à manipuler les concepts du paradigme de programmation procédurale. Dans ce modèle, les données et le code sont séparés. Concrètement, on utilise :

1. des variables (globales) qui

- ▷ contiennent vos données
- ▷ sont utilisées par des fonctions/procédures.

2. des fonctions qui

- ▷ peuvent modifier une variable
- ▷ peuvent passer cette variable à une autre fonction/procédure.

Remarque 1.1. *Paradigme : manière d'organiser l'information*

1.2.2 Limites de la programmation procédurale

Prenons l'exemple d'une entreprise située à Biarritz. Robert (num de secu 10573123456), est employé sur un poste de technicien. Voici comment il est possible d'implémenter cette situation en programmation procédurale :

```
1 #####  
2 # Version 1:  
3 #      variables globales  
4 #####  
5 # robert  
6 num_secu_robert      = 10573123456  
7 nom_robert           = "Robert"  
8 qualification_robert = "Technicien"  
9 lieu_de_travail_robert = "Biarritz"
```

Maintenant, nous souhaitons ajouter un autre employé "Bernard"

```
1 # bernard
2 num_secu_bernard      = 288111001
3 nom_bernard          = "Bernard"
4 qualification_bernard = "Ingenieur"
5 lieu_de_travail_bernard = "Biarritz"
6
7
8 # ...
```

Ce code devient vite fastidieux.

Une autre solution est d'utiliser des conteneurs.

```
1 ######
2 # Version 2 :
3 #           utilisation de conteneurs
4 #####
5
6 tab_robert=[10573123456,
7         "robert",
8         "Technicien",
9         "Biarritz"]
10 tab_mickey=[14456464443,
11         "mickey",
12         "Ingenieur",
13         "Brest"]
14 tab=[]
15 tab.append(tab_robert)
16 tab.append(tab_mickey)
17 print tab[0][0]          # num secu robert
```

Sans être le concepteur, comment interpréter facilement les données de chaque conteneur ?

Cela reste délicat. Le paradigme objet propose des réponses.

1.3 Vers la Programmation Orientée Objet (POO)

En POO, nous pouvons définir une structure de donnée particulière par la notion de **classe**.

Ici :

```
1 #####  
2 # Version 3 : structuration des donnees  
3 #####  
4  
5 class Employe:  
6     num_secu  
7     nom  
8     qualification  
9     Lieu_de_travail
```

Définition 1.2. *Classe* : une classe déclare des propriétés communes à un ensemble d'objets.

1.3.1 Classe

On appelle *classe* un ensemble d'objets partageant certaines propriétés. Il s'agit d'un concept *abstrait*, comme par exemple les plans d'une maison.

Exemple 1.2. La classe *Voiture* possède les propriétés suivantes (les **attributs**) :

- ▷ couleur
- ▷ puissance

Définition 1.3. *Attribut* : les attributs sont des entités qui définissent les propriétés d'objets.

1.3.2 Objet

Un objet est une définition de caractéristiques propres à un élément particulier. Il s'agit d'un élément *concret* qui contient les propriétés de sa classe, comme une maison qui suit les plans définis préalablement.

Exemple 1.3. Objets : exemple instance de la classe *Voiture*

1. Voiture "Clio 2007 version roland garros"

- ▷ couleur: verte
- ▷ puissance: 70 Ch

2. Voiture "307 blue lagoon"

- ▷ couleur: bleue
- ▷ puissance: 90 Ch

Mise en place d'**objets** concrets (*instances* de classe)

```
1 # robert
2     e1=Employe()      # objet e1 : instance de la classe Employe
3     e1.num_secu        = 10573123456
4     e1.nom            = "Robert"
5     e1.qualification   = "Technicien"
6     e1.Lieu_de_travail = "Biarritz"
7
8     print (e1.num_secu,
9             e1.nom,
10            e1.qualification ,
11            e1.Lieu_de_travail)
12
13
14 # mickey
15     e2=Employe()      # objet e2 : instance de la classe Employe
16     e2.num_secu        = 14456464443
17     e2.nom            = "Mickey"
18     e2.qualification   = "Inge"
19     e2.Lieu_de_travail = "Brest"
```

Définition 1.4. *Objet* : un objet est l'instanciation d'une classe. Un objet est une définition de caractéristiques propres à un élément.

Définition 1.5. *Instance* : on appelle instance d'une classe un objet avec un comportement et un état, tous deux définis par la classe.

1.3.3 Constructeur

Si on considère une nouvelle instance :

```
1 # norbert
2     e3=Employe()
3
4     print (e3.num_secu,          # ERREUR
5             e3.nom,
6             e3.qualification ,
7             e3.Lieu_de_travail)
```

Ce code amène à une erreur puisque les valeurs de ces attributs ne sont pas précisées.

Les attributs doivent être initialisés avec une valeur « par défaut », nous faisons alors appel à une « fonction » particulière : le **constructeur**.

```

1   class Employe:
2       def __init__(self):      # Constructeur de la classe Employe
3           self.num_secu        = 000000000
4           self.nom            = "no_name"
5           self.qualification   = "novice"
6           self.Lieu_de_travail = "Paris"
7
8   # norbert
9   e3=Employe()      # creation de e3 : appel implicite de init de Employe
10
11 print (e3.num_secu,          # OK !!!
12     e3.nom,
13     e3.qualification,
14     e3.Lieu_de_travail)
```

```

1   class Employe:
2       def __init__(self,
3                     le_num_secu,
4                     le_nom,
5                     la_qualification,
6                     le_Lieu_de_travail):
7           self.num_secu        = le_num_secu
8           self.nom            = le_nom
9           self.qualification   = la_qualification
10          self.Lieu_de_travail = le_Lieu_de_travail
```

On peut alors utiliser les arguments du constructeur pour mettre des valeurs par défaut. On pourra alors instancier des objets « plus facilement » en ne précisant que certaines valeurs.

```

1   class Employe:
2       def __init__(self,
3                     le_num_secu        = 000000000,
4                     le_nom            = "no_name",
5                     la_qualification   = "novice",
6                     le_Lieu_de_travail = "Paris"):
7           self.num_secu        = le_num_secu
8           self.nom            = le_nom
9           self.qualification   = la_qualification
10          self.Lieu_de_travail = le_Lieu_de_travail
11 ###### Programme Principal ######
12 e1 = Employe(le_Lieu_de_travail = "Brest")
```



Pour les puristes, `__init__` est une initialisateur. Pour simplifier, nous l'appellerons constructeur.



En python,
→ le constructeur s'écrit en utilisant la « fonction » `__init__`
→ le premier argument est toujours `self`



En python,
→ `__init__` appelée implicitement à la création d'un objet



En python,
→ le constructeur peut posséder plusieurs arguments
(en plus de `self`)

Exercice 1.1. Classe et instances

- ▷ Écrire la classe `Voiture` en python.
- ▷ Écrire son constructeur avec la possibilité de préciser les valeurs de certains attributs à l'instanciation.
- ▷ Instancier 3 objets issus de la classe `Voiture`

Exercice 1.2. Manipulation d'instances

- ▷ Créer un tableau contenant les instances de la classe `Voiture`
- ▷ Afficher la valeur de l'attribut `couleur` de chaque instance en parcourant ce tableau

QCM

1. Une classe_____
 - (a) est un objet
 - (b) est une fonction particulière
 - (c) déclare des propriétés communes à un ensemble d'objets
2. Appel du constructeur_____
 - (a) il est appelé de façon aléatoire
 - (b) il est automatiquement appelé, s'il existe, dès qu'on crée un objet
 - (c) le constructeur est appelé à la demande
3. Création du constructeur, en python on utilise le mot-clé_____
 - (a) `init`
 - (b) `new`
 - (c) `class`
4. Le constructeur_____
 - (a) ne possède pas d'argument
 - (b) peut posséder des arguments (autre que `self`)
 - (c) doit posséder plusieurs arguments
5. Rôle du constructeur_____
 - (a) il ne sert à rien
 - (b) il permet de documenter le code
 - (c) le rôle principal du constructeur est d'initialiser les attributs

2 Concepts fondateurs (1 UC)

Sommaire

2.1 Qu'est ce qu'une classe ?	17
2.1.1 Classe vs Objets	17
2.1.2 Attributs et méthodes	17
2.2 Qu'est ce qu'un objet ?	18
2.2.1 Classe et objet	18
2.2.2 État	18
2.2.3 Comportement	19
2.2.4 Identité	21
2.2.5 Le chaos des objets	22
2.3 Vie d'un objet	22
2.3.1 Construction	22
2.3.2 Destruction	22

Objectifs du Cours 2 / Labo 2 :

- ▷ Classe :
 - ◊ Attributs
 - ◊ Méthodes
 - ◊ Constructeur / destructeur
- ▷ Objet :
 - ◊ État
 - ◊ Comportement
 - ◊ Identité

2.1 Qu'est ce qu'une classe ?

Je croyais qu'on allait apprendre à créer des objets, pourquoi créer une classe ? Pour créer un objet, il faut d'abord créer une classe ! Par exemple, pour construire une maison, nous avons besoin d'un plan d'architecte. La classe correspond au plan, l'objet à la maison. "Créer une classe", c'est dessiner les plans de l'objet.

2.1.1 Classe vs Objets

Une classe est la description d'une famille d'objets qui ont la même structure et les mêmes comportements. Une classe est une sorte de moule à partir duquel sont générés les objets.

Une classe $\xrightarrow{\text{génération}}$ des objets

2.1.2 Attributs et méthodes

Une classe est constituée :

1. d'un ensemble de variables (appelées *attributs*) qui décrivent la structure des objets ;
2. d'un ensemble de fonctions (appelées *méthodes*) qui sont applicables aux objets, et qui décrivent leurs comportements.

Définition 2.1. *Attribut et méthode*

attributs : *variables contenues dans la classe,
on parle aussi de "variables membres"*

méthodes : *fonctions contenues dans la classe,
on parle aussi de "fonctions membres"
ou de compétences
ou de services*

Figure 2.1. Une *classe* représente le plan

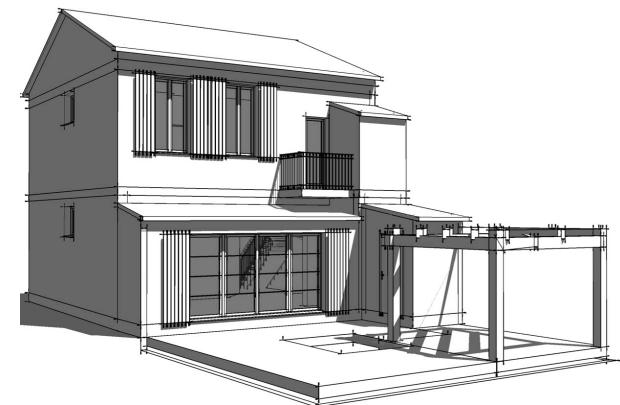
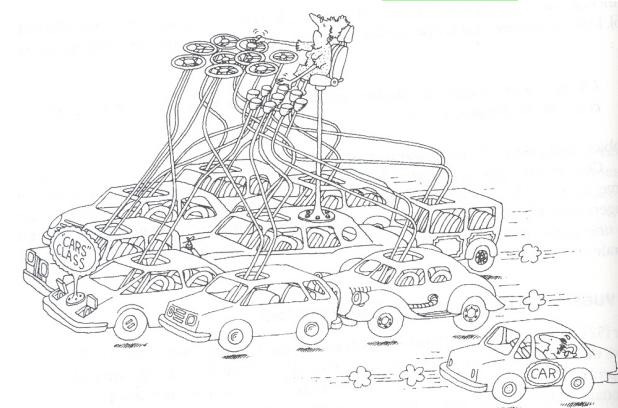


Figure 2.2. Une *classe* représente un *ensemble d'objets* qui partagent une structure commune et un comportement commun [Booch, 1992]



2.2 Qu'est ce qu'un objet ?

2.2.1 Classe et objet

Une classe est une abstraction, elle définit une infinité d'objets. Un objet est caractérisé par un état, des comportements et une identité. Concrètement, il s'agit :

- ▷ Attributs
- ▷ Méthodes
- ▷ Identité

On **crée des classes** pour définir le fonctionnement des objets. On **utilise des objets**.

Remarque 2.1. *On dit qu'un objet est une instance d'une classe*

2.2.2 État

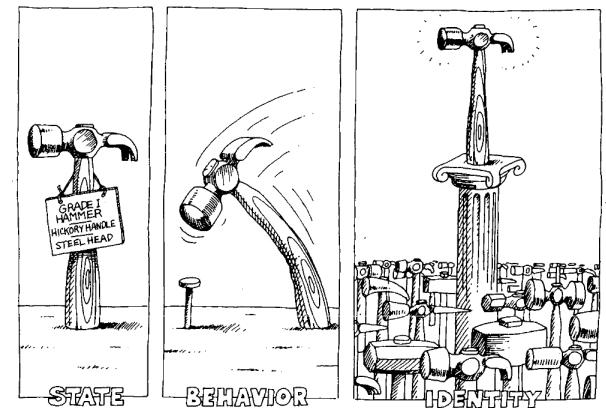
L'état d'un objet :

- ▷ regroupe les valeurs instantanées de tous ses **attributs**
- ▷ évolue au cours du temps
- ▷ est la conséquence de ses comportements passés

Attributs Les attributs sont les caractéristiques de l'objet. Ce sont des variables stockant des informations d'état de l'objet.

```
1 ##### definition classe #####
2 class Voiture:
3     def __init__(self, v1="nomarque", v2="nocolor", v3=0):
4         self.marque = v1
5         self.couleur = v2
6         self.reserveEssence = v3
7
8 ### Objet #####
9 homer_mobile = Voiture()
10
11 ## Etat
12 homer_mobile.marque      = "piro"
13 homer_mobile.couleur      = "rose"
14 homer_mobile.reserveEssence = 30
```

Figure 2.3. *Un objet a un état, il exhibe un comportement bien défini, et il a une identité unique [Booch, 1992]*



Attributs et variables Les attributs caractérisent l'état des instances d'une classe. Ils participent à la modélisation du problème. Les variables sont des mémoires locales à des méthodes. Elles sont là pour faciliter la gestion du traitement. L'accès aux variables est limité au bloc où elles sont déclarées : règle de portée.

2.2.3 Comportement

Méthodes Le comportement d'un objet regroupe toutes ses compétences, il décrit les actions et les réactions d'un objet. Il se représente sous la forme de **méthodes** (appelées parfois fonctions membres).

Les méthodes d'un objet caractérisent son comportement, c'est-à-dire l'ensemble des actions (appelées opérations) que l'objet est à même de réaliser. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures (ou d'agir sur les autres objets). De plus, les opérations sont étroitement liées aux attributs, car leurs actions peuvent dépendre des valeurs des attributs, ou bien les modifier.

Pour un objet "homer_mobile"

- ▷ démarrer
- ▷ arrêter
- ▷ freiner
- ▷ ...

```
1 ##### definition classe #####
2 class Voiture:
3     def __init__(self, v1="nomarque", v2="nocolor", v3=0):
4         self.marque = v1
5         self.couleur = v2
6         self.reserveEssence = v3
7
8     def demarrer(self):
9         print "je demarre"
10    def arreter(self):
11        print "je m'arrete"
12    ...
13
14 ### Objet #####
15 homer_mobile = Voiture()
```



En python, le premier argument du constructeur et d'une méthode est toujours **self**. Ceci est une convention à respecter impérativement !

Exercice 2.1. Implémentez la Classe Fichier.
On peut parler des fichiers (en général) sans faire référence à un fichier particulier, il s'agit donc d'une classe. Un fichier est caractérisé par son nom, sa taille, sa date de création et sa date de modification. Un fichier peut s'ouvrir et se fermer.

```
17     ## Comportement
18 homer_mobile.demarrer()
19 homer_mobile.arreter()
```

Comportement et état L'état et le comportement sont liés.

Exemple 2.1.

homer_mobile.démarrer() ne marchera pas si

homer_mobile.reserveEssence == 0

```
1 ##### definition classe #####
2 class Voiture:
3     def __init__(self):
4         self.marque
5         self.couleur
6         self.reserveEssence
7
8     def demarrer(self):
9         if self.reserveEssence > 0 :
10             print "je demarre"
11         else :
12             print "je ne peux pas demarrer"
```

Exemple 2.2.

Pour la classe "Voiture" : Après 100 kms ...

▷ réserve d'essence : **25l**

```
1 ##### definition classe #####
2 class Voiture:
3     def __init__(self ,v1="nomarque" ,v2="nocolor" ,v3=0):
4         self.marque = v1
5         self.couleur = v2
6         self.reserveEssence = v3
7
8     def demarrer(self):
9         if self.reserveEssence > 0 :
10             print "je demarre"
11         else :
```

Exercice 2.2. Gestion du stock

Un Article du stock est défini par 4 champs :

- ▷ sa référence (numéro)
 - ▷ sa désignation (texte)
 - ▷ son prixHT
 - ▷ sa quantité (nombre d'articles disponibles)
- Pour manipuler ces champs, les services suivants sont fournis :*
- ▷ prixTTC
 - ▷ prixTransport (taxe 5% prixHT)
 - ▷ retirer
 - ▷ ajouter

```

12         print "je ne peux pas demarrer"
13
14     def rouler(self ,nbKm):
15         self.reserveEssence = self.reserveEssence - 5*nbKm/100;
16
17
18 #####
19 homer_mobile = Voiture()
20 homer_mobile.reserveEssence = 30
21 homer_mobile.demarrer()
22 homer_mobile.rouler(100)
23 print homer_mobile.reserveEssence

```

2.2.4 Identité

L'objet possède une identité, qui permet de le distinguer des autres objets, indépendamment de son état. On construit généralement cette identité grâce à un identifiant découlant naturellement du problème (par exemple un produit pourra être repéré par un code, une voiture par un numéro de série ...).

Plusieurs techniques peuvent être utilisées :

→ nommer chaque objet (ex. attribut "name")

Définition 2.2. *L'identité est propre à l'objet et le caractérise. Elle permet de distinguer tout objet indépendamment de son état*

print homer_mobile.name



Voiture.1



Voiture.2

→ leur référence

print homer_mobile



0xb7d4e0ec



0xb7d53eec

2.2.5 Le chaos des objets

Attention, ne pas oublier !!



Soient 3 objets :

- ▷ même structures de données (attributs)
- ▷ même comportement (méthodes)

Il faut les décrire de façon abstraite : une classe. Il ne faut pas tomber dans le travers de faire une classe par objet !!

2.3 Vie d'un objet

2.3.1 Construction

A la déclaration d'un objet, les attributs ne sont pas initialisés. Cette phase d'initialisation des attributs est effectuée par une méthode particulière : le constructeur. Une classe peut définir son(ses) constructeur(s) avec des paramètres différents. Notons que le constructeur est appelé lors de la construction de l'objet.

```
1 class Complexe:  
2     def __init__(self, r=0.0, i=0.0):  
3         self.reel = r  
4         self.img = i  
5  
6 ##### Programme principal #####  
7 c1 = Complexe()  
8 c2 = Complexe(0.5)  
9 c3 = Complexe(0.5, 0.2)  
10 c4 = Complexe(i=0.2)
```

2.3.2 Destruction

Le destructeur de la classe est une méthode spéciale appelée lors de la suppression d'une instance. L'objectif est de détruire tous les objets créés par l'objet courant au cours de son



En python,
→ la définition d'un destructeur n'est pas obligatoire

existence.

```
1 class Complexe:  
2     def __del__(self):  
3         print "I'm dying!"
```

On peut se demander si le destructeur est appelé automatiquement. Dans certains langages, un programme s'exécute en tâche de fond afin de supprimer la mémoire associée à un objet. Il s'agit du ramasse-miettes ou "garbage collector".

```
1 class Test:  
2     def __del__(self):  
3         print "I am dying"  
4  
5  
6  
7 >>> t1 = Test() # une instance de Test est cree  
8 >>> t1 = None # t1 ne permet plus d'accéder à l'instance  
9 I am dying # elle est susceptible d'être détruite ce qui  
10 >>> # arrive instantanément ici mais n'est pas garanti
```

Définition 2.3. *Garbage collector* : récupération d'emplacements en mémoire qui ne sont plus utilisés par le système



En python,
→ `__del__` appelée implicitement à la destruction d'un objet

QCM

1. Les attributs sont _____
(a) des variables contenues dans des classes
(b) des variables indépendantes d'une classe
2. L'écriture du destructeur _____
(a) est obligatoire
(b) n'est pas obligatoire
3. Les méthodes sont _____
(a) des fonctions indépendantes d'une classe
(b) des fonctions contenues dans des classes
4. Quel est le rapport entre un objet et une classe ? _____
(a) un objet est une instance de classe
(b) un objet est une intendance de classe
(c) il n'y a aucun rapport
5. Qu'est-ce qu'une classe ? _____
(a) un groupe d'objets qui possèdent les mêmes relations
(b) un groupe d'objets qui ont les mêmes type de propriétés
(c) un groupe d'objets qui ont de caractéristiques non communes
6. Le constructeur _____
(a) permet d'initialiser les valeurs des attributs
(b) ne sert à rien
(c) permet de documenter la classe

3 Encapsulation (1 UC)

Sommaire

3.1 Principes	26
3.1.1 Exemple introductif	26
3.1.2 Modes de contrôle d'accès	26
3.1.3 Intérêt	27
3.2 Règles d'encapsulation	28
3.3 Mise en application	28
3.3.1 Définir les classes	28
3.3.2 Instancier des objets et les utiliser	31
3.4 Intérêt	31

Objectifs du Cours 3 / Labo 3 :

- ▷ Encapsulation
- ▷ Règles d'encapsulation

3.1 Principes

3.1.1 Exemple introductif

Exemple 3.1. Gestion de compte en banque

Il s'agit d'une classe `Compte` possédant un attribut `solde`

```
1 class Compte:
2     def __init__(self):
3         self.solde = 0      # solde nul
4
5     def affiche(self):
6         print "Le solde de votre compte est de " self.solde
7
8 ##### Programme principal #####
9 c = Compte()
10 c.affiche()        # pourquoi pas
11 c.solde = 1000000  # ben tiens !
```

N'importe qui ayant accès à une instance de la classe `Compte` peut modifier le solde. Ceci n'est pas acceptable.

3.1.2 Modes de contrôle d'accès

Lors de la définition d'une classe il est possible de restreindre voire d'interdire l'accès (aux objets des autres classes) à des attributs ou méthodes des instances de cette classe. Comment ? Lors de la déclaration d'attributs ou méthodes, en précisant leur utilisation :

- ▷ **privée** accessible uniquement depuis "l'intérieur" de la classe (ses méthodes, constructeur, destructeur)
- ▷ **public** accessible par tout le monde (ie. tous ceux qui possèdent une référence sur l'objet)

Dans l'exemple précédent, il s'agit de rendre l'attribut `solde` privé.

Figure 3.1. L'encapsulation occulte les détails de l'implémentation d'un objet [Booch, 1992]

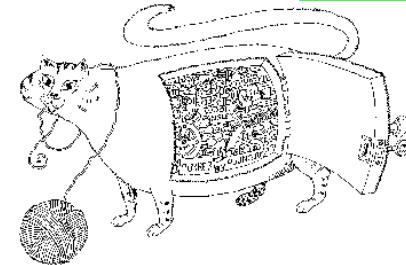


Figure 3.2. Définir une classe en contrôlant l'accès à certains attributs ou méthodes



En python,
→ le préfixe "`__`" rend privé un attribut ou une méthode

```

1 class Test:
2
3     def __init__(self, laValeurPublic, laValeurPrivee):
4         self.valeurPublic = laValeurPublic
5         self.__valeurPrivee = laValeurPrivee
6
7
8     def f1(self):
9         ...
10    def __f2(self):
11        ...
12
13
14 ##### Programme principal #####
15 t1 = Test()          # instance (nous sommes en "dehors" de la classe)
16
17 print t1.valeurPublic      # OK
18 print t1.__valeurPrivee    # ERREUR
19
20 t1.f1()                # OK
21 t1.__f2()                # ERREUR

```

Ce mécanisme permet de protéger l'accès à certaines données, par exemple la modification du solde :

```

1 class Test:
2     .....
3
4     def debit(self, laValeur):
5         if self.__solde - laValeur > 0 :
6             self.__solde = self.__solde - laValeur
7         else:
8             print("Compte bloqué !!!")

```

3.1.3 Intérêt

L'encapsulation est donc l'idée de cacher l'information contenue dans un objet et de ne proposer que des méthodes de manipulation de cet objet. Ainsi, les propriétés et axiomes associés aux informations contenues dans l'objet seront assurés/validés par les méthodes de l'objet et ne seront plus de la responsabilité de l'utilisateur extérieur. L'utilisateur extérieur ne pourra



Attention le préfixe "`__`" d'un attribut n'a rien à voir avec `__init__`.

Définition 3.1. *L'encapsulation empêche l'accès aux données par un autre moyen que les services proposés. L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.*

pas modifier directement l'information et risquer de mettre en péril les axiomes et les propriétés comportementales de l'objet.

L'objet est ainsi vu de l'extérieur comme une boîte noire ayant certaines propriétés et ayant un comportement spécifié. La manière dont ces propriétés ont été implémentées est alors cachée aux utilisateurs de la classe.

Voici les principaux avantages qu'il faut retenir du principe d'encapsulation proposé en POO :

1. **protéger**

→ ne pas permettre l'accès à tout dès que l'on a une référence de l'objet

2. **masquer l'implémentation**

→ toute la décomposition du problème n'a besoin d'être connue du programmeur client

3. **évolutivité**

→ possibilité de modifier tout ce qui n'est pas public sans impact pour le programmeur client

3.2 Règles d'encapsulation

Les règles à respecter sont les suivantes :

1. Rendre **privés les attributs** caractérisant l'état de l'objet. Si nous voulons respecter le concept d'encapsulation, nous devons donc conserver la portée de cet attribut et définir des accesseurs pour y avoir accès
2. Fournir des *méthodes publiques* permettant de accéder/modifier l'attribut
 - ▷ accesseurs
 - ▷ mutateurs

3.3 Mise en application

3.3.1 Définir les classes

```
1 #####  
2 # Programme Python      #  
3 # Auteur : .....      #  
4 # Date creation : ... #  
5 #####  
6
```

```

7  class X :
8
9      # --- Documentation ---
10
11
12      # --- Constructeur et destructeur ---
13
14      # --- Methodes publiques: Acces aux attributs ---
15          # accesseurs
16          # mutateurs
17
18      # --- Methodes publiques: Operations ---
19
20      # --- Methodes privees ---

```

Documentation

- ▷ Conception :
 - ◊ décrire la classe et ses fonctionnalités
- ▷ Debuggage :
 - ◊ débrayer des parties de code, marquer des points douteux, signer un correctif
- ▷ Maintenance :
 - ◊ décrire les corrections/ajouts/retraits et les signer
- **Indispensable** dans un code
 - ◊ description, pas des "faut que j'aille prendre un café" ...

Constructeur et destructeur

- ▷ **Constructeur**
 - ◊ Définir l'état initial de l'objet
 - Initialisation des attributs définis dans la classe
 - ◊ Créer les objets dont l'existence est liée à celle de l'objet
 - ◊ Types de constructeur
 - par défaut, par recopie ...
- ▷ **Destructeur**
 - ◊ Détruire tous les objets créés par l'objet courant au cours de son existence

Accès aux attributs : `getAttrName` et `setAttrName`

- ▷ **Objectifs**
 - ◊ Maintien de l'intégrité
 - ◊ Limitation des effets de bord
- ▷ **Comment ?**
 - ◊ Attribut : toujours un membre **privé**
 - ◊ `getAttrName` et `setAttrName` : en fonction des contraintes d'intégrité

Exemple 3.2. classe *Temperature*

```

1  class Temperature:
2      """
3          La classe Temperature represente une grandeur physique liee
4          a la notion immediate de chaud et froid
5          """
6      # — Constructeur et destructeur
7      def __init__(self, laValeur = 0):
8          self.__valeur = laValeur
9
10     def __del__():
11         print "destruction"
12
13     # — Methodes publiques: Acces aux attributs
14     #    # accesseurs
15     def getValeur(self):
16         return self.__valeur
17     #    # mutateurs
18     def setValeur(self, value):
19         self.__valeur = value
20
21     # — Methodes publiques: Operations
22     def getValeurC(self):
23         return self.__valeur - 273.16
24
25     def getValeurF(self):
26         return 1.8 * self.getValeurC() + 32

```

3.3.2 Instancier des objets et les utiliser

```
1 # Exemple d'utilisation de cette classe
2
3 eauChaud = Temperature(310)
4 print eauChaud.getValeurC()
5 print eauChaud.getValeurF()
6
7 eauFroide = Temperature(5)
8 print eauFroide.getValeurC()
9 print eauFroide.getValeurF()
```

3.4 Intérêt

Lorsque vous utilisez des accesseurs, vous n'êtes pas obligé de vous contenter de lire un attribut dans un getter ou de lui affecter une nouvelle valeur dans un setter : en effet, il est tout à fait possible d'ajouter du code supplémentaire, voire de ne pas manipuler d'attribut en particulier !

```
1 class Voiture :
2     """
3     La classe Voiture definie par sa largeur
4     """
5     # —— Constructeur et destructeur
6     def __init__(self, laValeur = 0):
7         self.__largeur = laValeur
8
9     # —— Methodes publiques: Acces aux attributs
10    # accesseurs
11    def getLargeur(self):
12        return self.__largeur
13    # mutateurs
14    def setLargeur(self, value):
15        self.__largeur = value
16
17    # —— Methodes publiques: Operations
18    def mettreAJour(self):
19        ...
```

Dans un premier temps, imaginons que dans notre classe, nous disposions d'une méthode `mettreAJour()` qui redessine l'affichage de notre objet en fonction de la valeur de l'attribut

`largeur`. Pour spécifier la largeur de la voiture, nous procéderions ainsi :

```
1  vehicule1 = Voiture();
2  vehicule1.setLargeur(100) # Largeur de la voiture
3  vehicule1.mettreAJour()  # Mettons à jour l'affichage de la voiture pour une largeur de 100
4  vehicule1.setLargeur(150) # Changeons la largeur de la voiture
5  vehicule1.mettreAJour()  # Mettons à jour l'affichage de la voiture pour une largeur de 150
```

Grâce aux accesseurs, il est possible de l'appeler automatiquement dès que l'on modifie la largeur de l'objet :

```
1  def setLargeur(self, value):
2      self._largeur = value
3      self.mettreAJour()
```

Ainsi, au lieu d'avoir à appeler manuellement la méthode `mettreAJour()`, il suffit de modifier la largeur :

```
1  vehicule1 = Voiture();
2  vehicule1.setLargeur(100) # Largeur de la voiture
3  # L'affichage de la voiture est automatiquement mis à jour dans l'accesseur setLargeur !
4  vehicule1.setLargeur(150) # Changeons la largeur de la voiture
5  # Encore une fois, il n'y a rien d'autre à faire !
```

Maintenant, nous aimerions limiter les valeurs possibles de l'attribut `largeur`; disons qu'il doit être supérieur à 100 et inférieur à 200.

```
1  vehicule1 = Voiture();
2  vehicule1.setLargeur(100) # Largeur de la voiture
3  # On vérifie que la largeur est dans les limites
4  if vehicule1.getLargeur() < 100 :  vehicule1.setLargeur(100)
5  else if vehicule1.getLargeur() > 200 :  vehicule1.setLargeur(200)
6  print vehicule1.getLargeur() # Affiche: 100
7
8  vehicule1.setLargeur(250) # Changeons la largeur de la voiture
9  if vehicule1.getLargeur() < 100 :  vehicule1.setLargeur(100)
10 else if vehicule1.getLargeur() > 200 :  vehicule1.setLargeur(200)
11 print vehicule1.getLargeur() # Affiche: 200
```

Vous remarquerez que c'est plutôt fastidieux. Bien sûr, nous pourrions utiliser une fonction, mais il vaut mieux mettre dans la classe `Voiture` ce qui appartient à la classe `Voiture`! Encore une fois, les accesseurs nous facilitent grandement la tâche ; voyez plutôt :

```
1 def setLargeur(self , value):
2     self.__largeur = value
3
4         # largeur doit etre comprise entre 100 et 200
5     if self.getLargeur() < 100 :      self.__largeur = 100
6     else if self.getLargeur() > 200 : self.__largeur = 200
7
8     self.mettreAJour()
```

Le code principal serait alors écrit ainsi :

```
1 vehicule1 = Voiture();
2 vehicule1.setLargeur(100) # Largeur de la voiture
3 # Plus besoin de verifier que la largeur est dans les limites , l'accesseur le fait pour nous !
4 print vehicule1.getLargeur() # Affiche: 100
5
6 vehicule1.setLargeur(250); # Changeons la largeur de la voiture
7 print vehicule1.getLargeur() # Affiche: 200
```

Avouez que c'est extrêmement pratique ! Appliquer cette façon de faire le plus souvent possible, cela vous rendra service

QCM

1. Une méthode **privée** est _____
(a) accessible de n'importe quelle classe
(b) accessible seulement à partir de la même classe
2. L'encapsulation permet _____
(a) d'avoir un contrôle complet sur ces données et sur des contraintes à imposer à ces données
(b) de documenter le code
(c) de contrôler l'accès aux données du constructeur
3. Une méthode **publique** est _____
(a) accessible de n'importe quelle classe
(b) accessible seulement à partir de la même classe
4. Un accesseur est une méthode permettant de _____
(a) de modifier le contenu d'un attribut privé
(b) récupérer le contenu d'un attribut privé
5. De façon préférentielle, le nom de l'accesseur commence par le préfixe _____
(a) **get**
(b) **new**
(c) **self**
(d) **init**
(e) **del**
(f) **set**
6. Les méthodes permettant de modifier les attributs _____
(a) sont appelées accesseurs
(b) sont appelées mutateurs

4 Hiérarchie (1 UC)

Sommaire

4.1 Hiérarchie	36
4.1.1 Héritage	36
4.1.2 Spécialisation	37
4.1.3 L'héritage multiple	40
4.1.4 Retour sur l'encapsulation	42

4.1 Hiérarchie

Exemple 4.1. Fichier

Il existe plusieurs sortes de fichier :

- ▷ fichier binaire → FichierBinaire
- ▷ fichier ASCII → FichierASCII

Certaines propriétés sont générales à toutes les classes (nom, taille). Certaines sont spécifiques à une classe ou un groupe de classes spécialisées (executer, imprimer)

4.1.1 Héritage

Définition 4.1. La relation d'héritage

▷ *Principe*

les caractéristiques des classes supérieures sont héritées par les classes inférieures.

▷ *Mise en œuvre*

les connaissances les plus générales sont mises en commun dans des classes qui sont ensuite spécialisées par définitions de sous-classes successives contenant des connaissances de plus en plus spécifiques.

Point de vue ensembliste : la relation d'héritage décrit une inclusion d'ensembles.

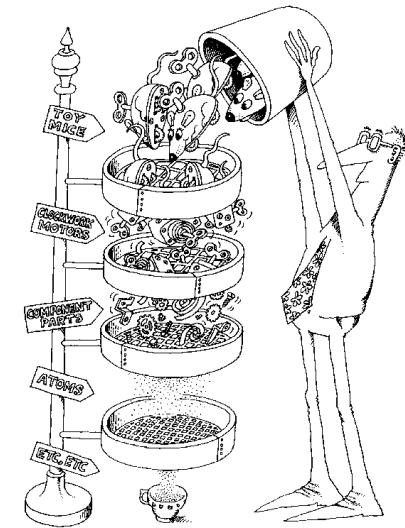
$$\forall x, x \in \text{FichierBinaire} \Rightarrow x \in \text{Fichier}$$

Point de vue conceptuel : la relation d'héritage indique une spécialisation.

FichierBinaire est une sorte de Fichier

Exemple 4.2. FichierBinaire et FichierASCII héritent de Fichier

Figure 4.1. Les entités constituent une hiérarchie [Booch, 1992]



4.1.2 Spécialisation

La spécialisation d'une classe peut être réalisée selon deux techniques :

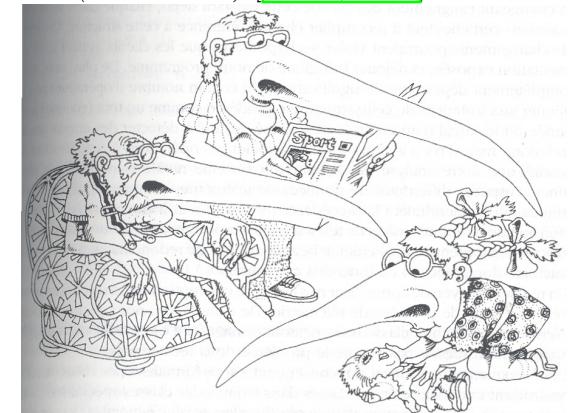
1. l'*enrichissement* : la sous–classe est dotée de nouvelle(s) méthode(s) et/ou de nouveau(x) attribut(s) ;
 2. la *surcharge* : une (ou plusieurs) méthode et/ou un attribut hérité est redéfini.

Enrichissement

ajout d'une méthode

```
1 ##### class Fichier #####
2 class Fichier:
3     """Definition classe Fichier"""
4
5     def __init__(self):
6         print "creation d'un fichier"
7
8
9 ##### class FichierBinaire #####
10 class FichierBinaire(Fichier):
11     """Definition classe FichierBinaire"""
12
13     def __init__(self):
14         Fichier.__init__(self)
15
16     def imprimer(self)
17         # Documentation
18
19         # Constructeur
20
21         # Enrichissement
```

Figure 4.2. Une sous-classe peut hériter de la structure et du comportement de sa superclasse (classe mère) [Booch, 1992]



ajout d'un attribut

```

1 ##### class Fichier #####
2     class Fichier:
3         """Definition classe Fichier"""
4
5         def __init__(self):
6             print "creation d'un fichier"
7
8
9 ##### class FichierBinaire #####
10    class FichierBinaire(Fichier):
11        """Definition classe FichierBinaire"""
12
13        def __init__(self):
14            Fichier.__init__(self)
15            self.__codage=32

```

Surcharge

d'un attribut

```
1 ##### class Fichier #####
2     class Fichier:
3         def __init__(self):                      # Constructeur
4             self.nom = "fichier_sans_nom"
5
6
7
8
9 ##### class FichierBinaire #####
10    class FichierBinaire(Fichier):
11
12        def __init__(self):                      # Constructeur
13            Fichier.__init__(self)
14            self.nom = "fichierBinaire_sans_nom" # Surcharge de l'attribut "nom"
```

Exercice 4.1. Héritage

- ▷ Ecrire une classe *Animal* avec 2 attributs (*couleur*, *poids*) et 2 méthodes (*afficheDureeDeVie* et *crier*)
 - ▷ Ecrire une classe *Chien* (classe enfant de *Animal*)
 - ▷ Instancier *mé dor* instance de *Chien*
 - ▷ Faites le *crier*
 - ▷ Modifier la classe *Chien* pour préciser
 - ◊ que sa durée de vie est de 15ans
 - ◊ et son poids de 30kg par défaut

d'une méthode

```
1 ##### class Fichier #####
2 class Fichier:
3
4     def __init__(self):           # Constructeur
5         print "creation d'un fichier"
6
7     def ouvrir(self):
8         print "ouvrir fichier"
9
10
11 ##### class FichierBinaire #####
12 class FichierBinaire(Fichier):
13
14     def __init__(self):           # Constructeur
15         Fichier.__init__(self)
16
17     def ouvrir(self):            # Surcharge de la methode
18         print "ouvrir fichier Binaire"
# "ouvrir"
```

Exemple d'utilisation :

```
1 class Engin:
2     def __init__(self, braquage=90):
3         self.__vitesse = 0
4         self.__braquage = braquage
5
6     def virage(self):
7         return self.__vitesse / self.__braquage
8
9 #
10 class Sous_marin_alpha(Engin):
11     def __init__(self, braquage):
12         Engin.__init__(self, braquage)
13
14 #
15 class Torpille(Engin):
16     def __init__(self, braquage):
17         Engin.__init__(self, braquage)
18
19     def exploser(self):
20         print "Explosion !!!!!"
```

Exercice 4.2. Gestion du stock

La classe *Article* contient un *prixHT*. Ajouter une classe *Vetement* contenant les mêmes informations et services que la classe *Article*, avec en plus les attributs *taille* et *coloris*. Ajouter une classe *ArticleDeLuxe* contenant les mêmes informations et services que la classe *Article*, si ce n'est une nouvelle définition de la méthode *prixTTC* (taxe différente).

4.1.3 L'héritage multiple

L'idée est de regrouper au sein d'une seule et même classe les attributs et méthodes de plusieurs classes. Pour celà, cette classe doit hériter de plusieurs superclasses.

Exemple 4.3. *La classe Omnivore hérite des 2 superclasses*

```
1  class Herbivore:
2      ...
3      def mangerHerbe(self):
4          print "je mange de l'herbe"
5
6  #
7  class Carnivore:
8      ...
9      def mangerViande(self):
10         print "je mange de la viande"
11 #
12 class Omnivore(Herbivore, Carnivore): # Heritage multiple
13     ...
14     def manger(self):
15         if rand(10)%2 == 0 :
16             self.mangerHerbe()
17         else:
18             self.mangerViande()
```

Polymorphisme A un **même service** peuvent correspondre, dans les classes dérivées, des **méthodes différentes**. Chaque classe dérivée définit la *forme* (**-morphé**) sous laquelle elle remplit le service (méthode). Cette forme peut donc être *multiple* (différente d'une classe à l'autre) (**poly-**)

Exemple 4.4. La méthode `affiche()` est différente selon qu'il s'agisse d'un *Rectangle* ou d'un *Cercle*.

```
1 class Shape:                      # classe "abstraite" spécifiant une interface
2     def __init__(self):
3         print "je suis une shape"
4
5     def area(self):
6         return "shape area"
7
8 class Rectangle(Shape):
9     def __init__(self):
10        Shape.__init__(self)
11        print "je suis un Rectangle"
12
13    def area(self):
14        return "rectangle area"
15
16
17 class Circle(Shape):
18     def __init__(self):
19        Shape.__init__(self)
20        print "je suis un cercle"
21    def area(self):
22        return "circle area"
23
24 ##### programme principal
25 shapes = []
26 shapes.append(Rectangle())
27 shapes.append(Circle())
28 for i in shapes:
29     print i.area()                  # appel polymorphe
```

Notons qu'un algorithme manipulant des `Shape` peut utiliser la méthode `area()`.

4.1.4 Retour sur l'encapsulation

Contrôle d'accès :

- ▷ publiques
- ▷ privées
- ▷ **protégées** : les attributs ne sont visibles que des sous-classes

Règles d'écriture

- ▷ publiques : `attr1, attr2`
- ▷ privées : `__attr1, __attr2`
- ▷ **protégées** : `_attr1, _attr2`

```
1 class Telephone:  
2     def __init__(self):  
3         # donnees privées  
4         self.__numero_serie = '126312156'  
5  
6         # donnees protégées  
7         self._code_pin = '1234'  
8  
9         # données publiques  
10        self.modele = 'nokia 6800'  
11        self.numero = '06 06 06 06 06'  
12  
13        # méthodes privées  
14  
15        # méthodes protégées  
16        def _chercherReseau(self):  
17            print 'Reseau FSR, bienvenue dans un monde meilleur ...'  
18  
19        def _recupMessage(self):  
20            print 'Pas de message'  
21  
22        # méthodes publiques  
23        def allumer(self, codePin):  
24            print self.modele  
25            if self._code_pin == codePin :  
26                print 'Code pin OK'  
27                self._chercherReseau()  
28                self._recupMessage()  
29            else:  
30                print 'mauvais code pin'
```



En python,

→ Les attributs et méthodes **protégées** sont accessibles normalement, le préfixe a pour seul objectif d'informer sur leur nature

```

31
32 # programme principal
33 nokia = Telephone()
34 nokia.allumer('1524')
35 nokia.allumer('1234')


---


1 class TelephonePhoto(Telephone):
2     def __init__(self):
3         Telephone.__init__(self)
4         print 'telephone + mieux'
5
6     # methodes publiques
7     def prendre_photo(self):
8         print 'clic-kak'
9
10    # Test
11    def fonction_test(self):
12        print self.__numero_serie # ERREUR
13        print self._code_pin      # OK sous-classe de Telephone
14
15
16 # programme principal
17 nokia = TelephonePhoto()
18 nokia.allumer('1234')
19 nokia.prendre_photo()
20
21
22
23 tel_sagem = Telephone()
24 print tel_sagem.__numero_serie # ERREUR
25 print tel_sagem._code_pin     # NON !!

```

5 Collaborations entre objets (1 UC)

Sommaire

5.1 Héritage vs associations	44
5.2 Relation simple	44
5.3 Relation multiple	47
5.4 Destruction dans le cadre d'association	48

5.1 Héritage vs associations

La relation d'héritage trompe souvent les néophytes sur la sémantique de la relation d'héritage, particulièrement lorsque l'héritage est multiple.

Exemple 5.1. classe *Automobile*

*Il est tentant de définir une classe Automobile à partir des classes Carrosserie, Siege, Roue et Moteur. Il s'agit d'une erreur conceptuelle car l'héritage multiple permet de définir de nouveaux objets par **fusion** de la structure et du comportement de plusieurs autres, et non par **composition**. Une automobile est composée d'une carrosserie, d'un moteur, de sièges et de roues ; son comportement n'est en aucun cas l'union des comportements de ces différentes parties.*

*La classe Automobile ne peut être définie comme une spécialisation de la classe Roue, ou l'une des autres classes, car une automobile **n'est pas** une roue.*

5.2 Relation simple

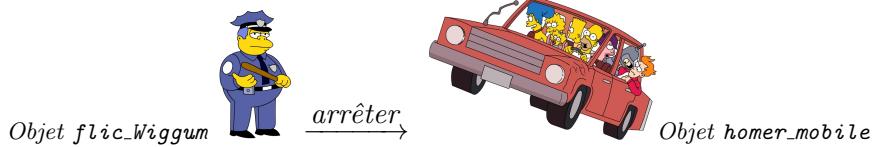
Exemple 5.2. classe *Afficheur*

La classe Afficheur se sert de la classe Calculateur pour fournir une interface à l'utilisateur.

```
1 # Relation simple
2 class Calculateur():
3     """ classe de calculs """
4     def somme(self, args = []):
5         resultat = 0
```

```
6         for arg in args:
7             resultat += arg
8         return resultat
9 #
10 class Afficheur():
11     """ classe de gestion de l'interface """
12     def __init__(self):
13         self.__calculateur = Calculateur()
14     def somme(self, args = []):
15         resultat = self.__calculateur.somme(args)
16         print 'le resultat est %d' % resultat
```

Exemple 5.3. *Policier*



```
1 ##### definition classe #####
2 class Voiture:
3     def __init__(self):
4         self.__marque
5     def setMarque(self,m):
6         self.__marque = m
7     def getMarque(self):
8         return self.__marque
9
10    def demarrer(self):
11        print "je demarre"
12
13    def arreter(self):
14        print "je m'arrete"
15
16
17 class Flic:
18     def __init__(self):
19         self.__nom
20     def setNom(self,m):
21         self.__nom = m
22     def getNom(self):
23         return self.__nom
24
25     def donnerOrdreStop(self,vehicule): ## instance de Voiture
26         print "arrete toi !"
27         vehicule.arreter()
28
29
30
31 #### Objet #####
32 homer_mobile = Voiture()
33 flic_Wiggum = Flic()
34 flic_Wiggum.donnerOrdreStop(homer_mobile)
```

5.3 Relation multiple

```
1 # Relation multiple
2
3 class FilleMars:
4     def __init__(self, prenom):
5         self.__prenom = prenom
6
7     def affichePrenom(self):
8         print self.__prenom
9 #
10 class PapaMars:
11     def __init__(self):
12         self.__filles = []
13         self.__filles.append( FilleMars('Pamela') )
14         self.__filles.append( FilleMars('Tina') )
15         self.__filles.append( FilleMars('Rebecca') )
16         self.__filles.append( FilleMars('Amanda') )
17
18     def cri_a_table(self):
19         for child in self.__filles:
20             child.affichePrenom()
21         print ('a table !!!')
```

```
1 # Relation multiple
2
3 class FilleMars:
4     def __init__(self, prenom):
5         self.prenom = prenom
6
7     def affiche(self):
8         print self.prenom
9 #
10 class PapaMars:
11     def __init__(self):
12         self.filles = []
13         self.filles.append( FilleMars('Pamela') )
14         self.filles.append( FilleMars('Tina') )
15         self.filles.append( FilleMars('Rebecca') )
16         self.filles.append( FilleMars('Amanda') )
17
18     def cri_a_table(self):
19         for nom in self.filles:
20             nom.affiche()
```

Exercice 5.1. classe *Autonomobile*
Implémenter la classe *Autonomobile* à partir des classes *Carrosserie*, *Siege*, *Roue* et *Moteur*.

```
21     print ('a table !!!')
```

5.4 Destruction dans le cadre d'association

Objets perdus Le destructeur doit détruire les attributs alloués dynamiquement en mémoire avant que l'objet ne soit supprimé (je vous rappelle que le destructeur est automatiquement appelé lorsqu'un objet va être supprimé). Les objets ne font pas exceptions.

```
1  class Roue:
2      def __init__(self):
3          print "Me voici !"
4
5      def __del__(self):
6          print "Je meurs !"
7
8
9  class Voiture:
10     def __init__(self):
11         self.__roue1 = Roue()
12         self.__roue2 = Roue()
13         self.__roue3 = Roue()
14         self.__roue4 = Roue()
15
16     def __del__(self):
17         print "Je suis mort!"
18         del self.__roue1
19         del self.__roue2
20         del self.__roue3
21         del self.__roue4
```

Objets partagés Attention à ne pas systématiquement détruire tous les objets.

```
1  class Humain:
2      def __init__(self,nom):
3          self.__nom = nom
4
5  class Roue:
6      def __init__(self):
7          print "Me voici !"
8
9      def __del__(self):
```

```
10         print "Je meurs !"
11
12
13 class Voiture:
14     def __init__(self, proprio):
15         self.__roue1 = Roue()
16         self.__roue2 = Roue()
17         self.__roue3 = Roue()
18         self.__roue4 = Roue()
19         self.__conducteur = proprio("")
20
21     def __del__(self):
22         print "I'm dying!"
23         del self.__roue1
24         del self.__roue2
25         del self.__roue3
26         del self.__roue4
```

QCM

2. La classe B hérite de la classe A.
Si A possède 3 méthodes et que B en possède 2 qui lui sont propres, combien de méthodes différentes un objet de type B pourra-t-il utiliser ? _____
- (a) 3
- (b) 5
- (c) 2
3. L'héritage multiple est un mécanisme dans lequel une classe _____
- (a) peut hériter des méthodes et des attributs de plus d'une super-classe
- (b) peut hériter des méthodes et des attributs d'une seule super-classe
- (c) peut hériter des méthodes d'une des classes filles
4. La portée "protégée" empêche l'accès aux méthodes et attributs qui suivent depuis l'extérieur de la classe, sauf... _____
- (a) dans les classes filles
- (b) dans le programme principal
- (c) dans la classe mère
5. La classe qui hérite d'une autre classe est aussi appelée la classe... _____
- (a) Mère
- (b) Fille
- (c) Petite-fille

6 Pour aller plus loin

Sommaire

6.1 Abstraction	52
6.2 Classification	53
6.3 Modularité	54
6.3.1 Paquet en python	55
6.3.2 Module en python	55
6.3.3 Classe dans un module en python	55
6.4 Typage	56
6.5 Simultanéité	57
6.6 Persistance	58
6.6.1 Mise en œuvre	58

Objectifs du Cours 5 / Labo 5 :

- ▷ Abstraction
- ▷ Classification
- ▷ Modularité
- ▷ Typage
- ▷ Simultanéité
- ▷ Persistance

6.1 Abstraction

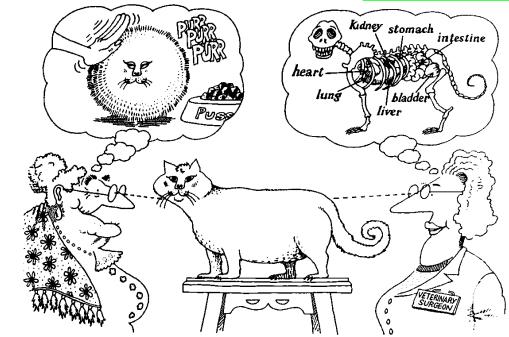
Un service peut être déclaré sans qu'une méthode y soit associée : **service abstrait**. La méthode associée au service est définie dans une ou plusieurs classes dérivées. On ne peut pas instancier d'objets d'une classe possédant un service abstrait : **classe abstraite**.

Une abstraction fait ressortir les caractéristiques essentielles d'un objet (qui le distinguent de tous les autres genres d'objets) et donc procure des frontières conceptuelles définies, relativement au point de vue de l'observateur.

```

1  class AbstractClass :
2
3      ...
4
5      # Force la classe fille a definir cette methode
6      def getValue(self):
7          raise NotImplementedError()
8
9      # methode commune
10     def printOut(self):
11         print self.getValue();
12
13
14 class ConcreteClass1(AbstractClass) :
15
16     ...
17
18     def getValue(self):
19         return "ConcreteClass1"
20
21 class ConcreteClass2(AbstractClass) :
22
23     ...
24
25     def getValue(self):
26         return "ConcreteClass2"
27
28 ##### Programme principal #####
29 class1 = ConcreteClass1()
30 class1.printOut()
31
32 class2 = ConcreteClass2()
33 class2.printOut()
```

Figure 6.1. L'abstraction se concentre sur les caractéristiques essentielles d'un objet, selon le point de vue de l'observateur [Booch, 1992]



En python,
→ attention, on peut instancier une classe abstraite

Figure 6.2. Les classes et les objets doivent être au juste niveau d'abstraction : ni trop haut ni trop bas [Booch, 1992]



6.2 Classification

Figure 6.3. La classification est le moyen par lequel nous ordonnons nos connaissances

[Booch, 1992]

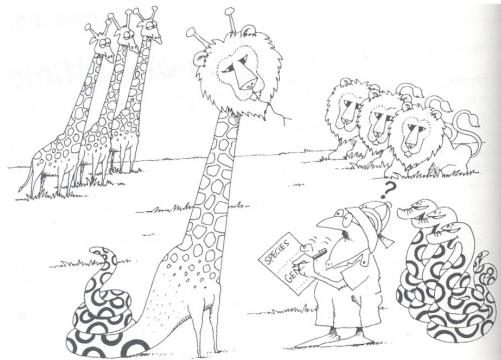


Figure 6.4. Des observateurs différents classent le même objet de différentes façons

[Booch, 1992]

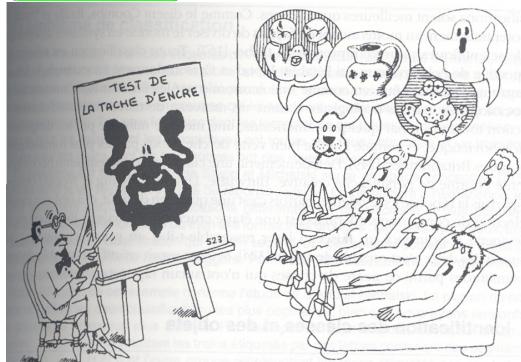
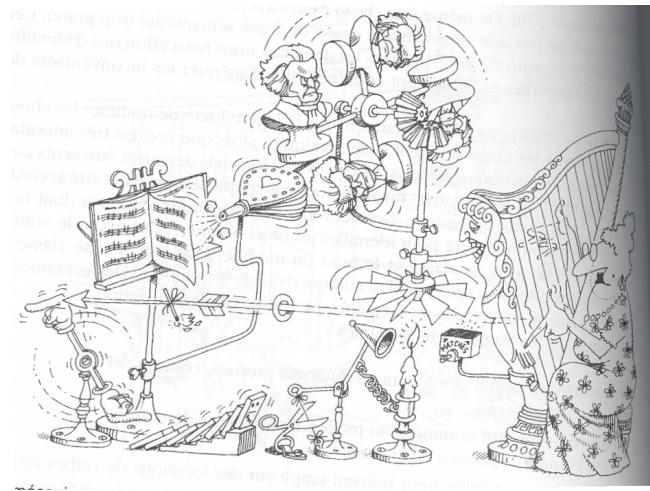
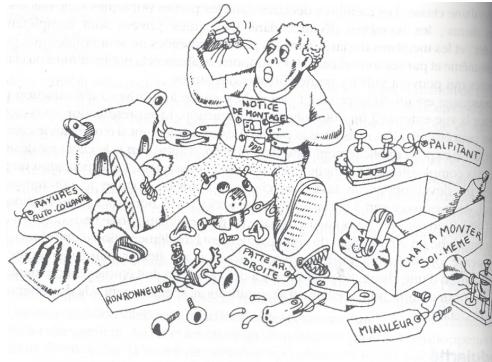


Figure 6.5. Les mécanismes sont les moyens par lesquels les objets collaborent pour procurer un certain comportement de plus haut niveau [Booch, 1992]



6.3 Modularité

Figure 6.6. La modularité regroupe les entités en unités discrètes [Booch, 1992]

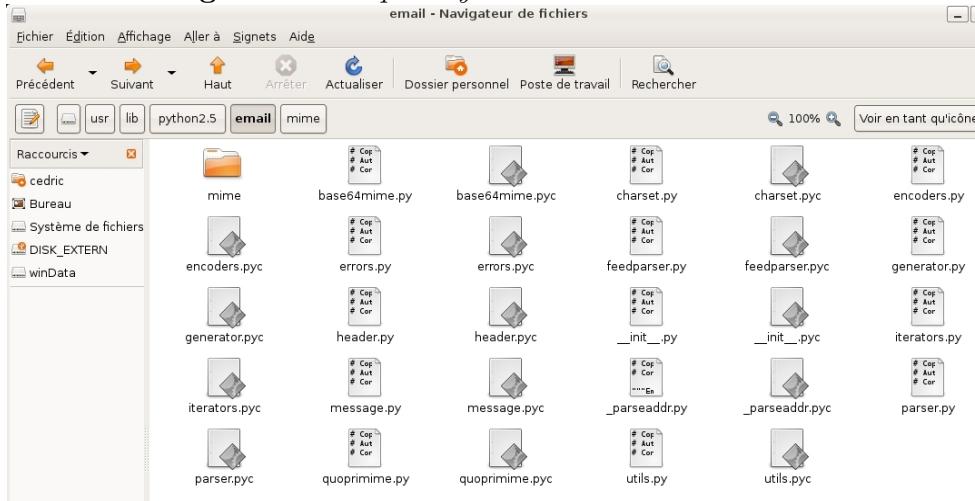


La programmation modulaire repose sur l'utilisation de modules, qui sont des structures permettant de définir des espaces regroupant des éléments définis par le programmeur : ensemble de classes, classes ...

6.3.1 Paquet en python

- ▷ répertoire avec un fichier `__init__.py` qui définit les modules qu'il contient (exemple :
`/usr/lib/python/xml/`)
- ▷ `from paquetage`

Figure 6.7. Paquet Python



6.3.2 Module en python

- ▷ un fichier regroupant variables, classes et fonctions
- ▷ extension ".py"
- ▷ `from paquetage import Module`
- ▷ attention la liste path de sys
 - ◊ `sys.path.append(Module)`

6.3.3 Classe dans un module en python

- ▷ Une classe
 - ◊ `from paquetage import Class`

Exercice 6.1. Gestion d'objets

On souhaite pouvoir créer des vecteurs de R^3 . La mise en œuvre se fera dans le module `vector` qui contiendra la classe `vec3`. On choisit de représenter les vecteurs par des tableaux de 3 réels.

Mettez en œuvre les méthodes suivantes :

- ▷ `string_of()` : renvoie une chaîne de caractère qui représente le vecteur;
- ▷ `scale(k)` : renvoie le vecteur qui résulte du produit du réel k par le vecteur;
- ▷ `len()` : renvoie la norme euclidienne du vecteur

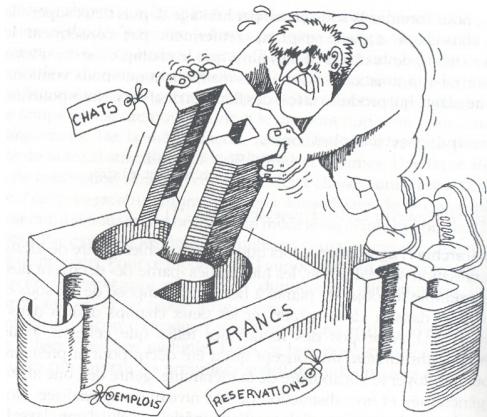
Utiliser ce module dans un autre programme.

```
◊ from turtle import Pen  
p1 = Pen()
```

6.4 Typage

Dans la programmation par objet, chaque objet peut être typé. Le type définit la syntaxe (comment l'appeler ?) et la sémantique (qu'est ce qu'il fait ?) des messages auxquels peut répondre un objet.

Figure 6.8. Un typage fort empêche le mélange des entités [Booch, 1992]



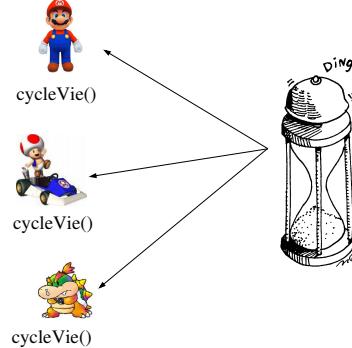
```
1 def estPair(nombre):  
2     if not isinstance(nombre, int):  
3         return None  
4     return (nombre % 2) == 0
```



En python,
→ `isinstance(X,type)` permet de vérifier le type d'un élément

6.5 Simultanéité

Exemple 6.1. *Ordonnanceur d'activités : class MyTimer*



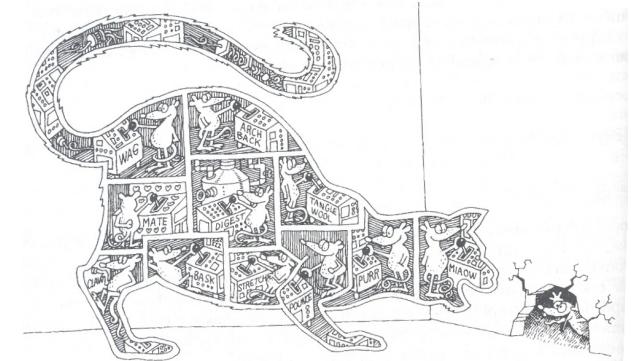
```

1 import threading
2 import time
3
4 class MyTimer:
5     def __init__(self, tempo, targets = []):
6         self.__targets = targets
7         self.__tempo = tempo
8
9     def start(self):
10        self.__timer = threading.Timer(self.__tempo, self.__run)
11        self.__timer.start()
12
13    def stop(self):
14        self.__timer.cancel()
15
16    def __run(self):
17        for i in self.__targets:
18            i.cycleVie()
19
20 #####
21 # Programme principal #
22
23 bots.append( Animat2D() )
24 bots.append( Animat2D() )
25
26 a = MyTimer(1,bots)
27 a.start()

```

Figure 6.9. La simultanéité permet à des objets différents d'agir en même temps

[Booch, 1992]



6.6 Persistance

La sauvegarde de l'état d'un objet peut s'effectuer selon 2 étapes :

1. sauvegarder la valeur des attributs de chaque objet
2. à partir de ces valeurs, on peut reconstruire l'objet

```
1 class Dummy:  
2     def __init__(self):  
3         self.member1="abcd"  
4         self.member2=0  
5         self.member3=['a','e','i','o','u']  
6  
7 d1 = Dummy()  
8 print d1.__dict__
```

```
1 ##### Test #####  
2 >>> python testDic.py  
3 {'member1': 'abcd', 'member3': ['a', 'e', 'i', 'o', 'u'], 'member2': 0}
```

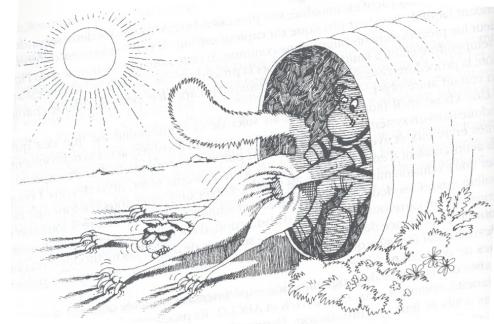
6.6.1 Mise en œuvre

Persistance de toutes les instances de classes dérivées d'une classe mère.

Remarque 6.1. *shelve charge les données sauvegardées, puis les rend disponibles à chaque instance.*

```
1 import shelve  
2 import atexit  
3 import sys  
4  
5 donnees = None  
6 instances = []  
7  
8 def _charge_objets():  
9     print 'chargement...'  
10    global donnees  
11    donnees = shelve.open('objets.bin')  
12
```

Figure 6.10. La persistance préserve l'état et la classe d'un objet à travers le temps et l'espace [Booch, 1992]



En python,
→ `__dict__` permet de récupérer les valeurs des attributs d'un objet

```

13 def _sauvegarde():
14     print 'sauvegarde...'
15     global donnees
16     for instance in instances:
17         donnees[instance.getId()] = instance.__dict__
18     if donnees is not None:
19         donnees.close()
20
21 # chargement des objets
22 _charge_objets()
23
24 # _sauvegarde sera appelee a la fermeture du programme
25 atexit.register(_sauvegarde)

```

```

1 class Persistent:
2
3     def __init__(self,id):
4         global donnees
5         if id in donnees:
6             self.__dict__ = donnees[id]
7         else:
8             self._id = id
9             instances.append(self)
10
11    def getId(self):
12        return self._id
13
14
15
16
17 class MaClasse(Persistent):
18
19     def __init__(self,id,data = None):
20         Persistent.__init__(self,id)
21         if data is not None:
22             self.datas = data

```

La classe de base `Persistent` charge les données sauvegardées dans son attribut `__dict__`, puis s'enregistre comme instance. Lorsque le programme se termine, les données de chaque instance sont sérialisées grâce à un appel provoqué par `atexit`.

Utilisation

```
1 # programme principal
2 instance_de = MaClasse('1')
3 while True:
4     try:
5         if hasattr(instance_de, 'datas'):
6             instance_de.datas = raw_input('valeur (actuelle:%s): ' % instance_de.datas)
7         else:
8             instance_de.datas = raw_input('nouvelle valeur :')
9
10    except KeyboardInterrupt:      # par exemple "Control-C"
11        print 'fin'
12        sys.exit(0)
```

Execution

```
1 >>> python testPersistance.py
2 chargement...
3 nouvelle valeur: homer
4 valeur (actuelle:homer): fin
5 sauvegarde...
6
7 >>> python testPersistance.py
8 chargement...
9 valeur (actuelle:homer): bart
10 valeur (actuelle:bart): fin
11 sauvegarde...
12
13
14 >>> python testPersistance.py
15 chargement...
16 valeur (actuelle:bart): lisa
17 valeur (actuelle:lisa): fin
18 sauvegarde...
```

Le mécanisme présenté ici ne sauvegarde les données qu'à la fermeture du programme. Il peut être intéressant dans certaines situations de provoquer cette sauvegarde à chaque modification de données.