Ecole Nationale d'Ingénieurs de Brest

— Cours d'Informatique S4 —

Programmation Orientée Objet (Concepts)

Cédric BUCHE

buche@enib.fr

version du 28 janvier 2014

Informatique S4-POO

Programmation Orientée Objet

— Concepts —

Cédric Buche

École Nationale d'Ingénieurs de Brest (ENIB)

28 janvier 2014

Ces notes de cours accompagnent les enseignements d'informatique du 4^{ieme} 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.

Table des matières

Cours

Avant-propos

Sommaire

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

Module :	*Programmation Orientée Objet — POO*	2013
Responsable :	Cédric Buche (CERV, bureau 30, tel : 02 98 05 89 66, email : `buche@enib.fr`)	S4
Intervenants :	C. Buche (*cb*)	

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

Semaine	LABO	CTD
45	*UML — Classes* *labo* *cb*	
45	*UML — Classes* *labo* *cb*	
46		*UML — Use Case (diag.)* *ctd* *cb*
46		*UML — Use Case (scénarios)* *ctd* *cb*
47	*UML — Use Case (diag.)* *labo* *cb*	
47	*UML — Use Case (scénarios)* *labo* *cb*	
48		*UML — Interactions (com)* *ctd* *cb*
48		*UML — Interactions (seq)* *ctd* *cb*
49	*UML — Interactions (com)* *labo* *cb*	
49	*UML — Interactions (seq)* *labo* *cb*	
50		
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 barême 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.

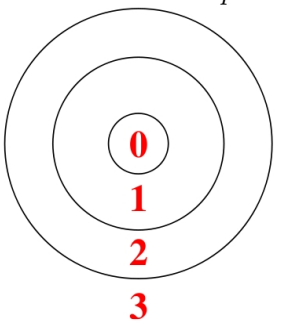
Figure 0.1. *Exemple de QCM*

Informatique Contrôle Continu

Nom/Prenom : *Groupe :*

1. QCM titre 4
   (a) réponse fausse 4.2
   (b) réponse fausse 4.1
   (c) réponse fausse 4.5
   (d) réponse juste 4
   (e) réponse fausse 4.3
   (f) réponse fausse 4.4
2. QCM titre 1
   (a) réponse juste 1
   (b) réponse fausse 1.2
   (c) réponse fausse 1.3
   (d) réponse fausse 1.1
3. QCM titre 2
   (a) réponse juste 2
   (b) réponse fausse 2.1
   (c) réponse fausse 2.2
4. QCM titre 3
   (a) réponse juste 3
   (b) réponse fausse 3.1
   (c) réponse fausse 3.2

Figure 0.2. *Notation : métaphore de la cible*



1 Justification du modèle objet (1 UC)

Sommaire

Objectifs du Cours 1 / Labo 2 :

▷ Justifier le modèle objet

▷ Notion de classe :
- ◇ définition
- ◇ constructeur

▷ Notion d'objet :
- ◇ instanciation
- ◇ 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.

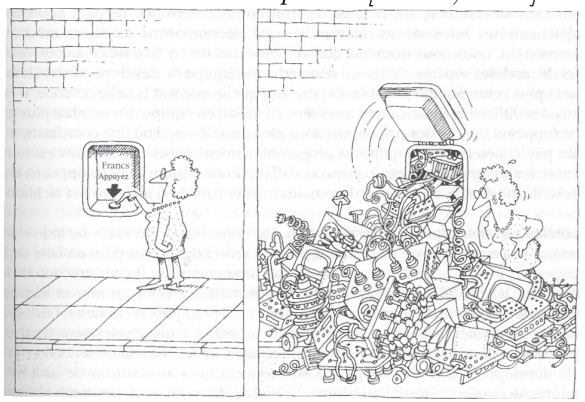
1.1.4 Qualité du logiciel

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

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

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

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"

```
# bernard
num_secu_bernard         = 2881111001
nom_bernard              = "Bernard"
qualification_bernard    = "Ingenieur"
lieu_de_travail_bernard = "Biarritz"


# ...
```

Ce code devient vite fastidieux.
Une autre solution est d'utiliser des conteneurs.

```
##############################################
# Version 2 :
#              utilisation de conteneurs
##############################################

tab_robert=[10573123456,
            "robert",
            "Technicien",
            "Biarritz"]
tab_mickey=[14456464443,
            "mickey",
            "Ingenieur",
            "Brest"]
tab=[]
tab.append(tab_robert)
tab.append(tab_mickey)
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.

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

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

- ▷ *couleur*
- ▷ *puissance*

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)

```
# robert
 e1=Employe()      # objet e1 : instance de la classe Employe
 e1.num_secu         = 10573123456
 e1.nom              = "Robert"
 e1.qualification    = "Technicien"
 e1.Lieu_de_travail = "Biarritz"

 print   (e1.num_secu,
          e1.nom,
          e1.qualification,
          e1.Lieu_de_travail)


# mickey
 e2=Employe()     # objet e2 : instance de la classe Employe
 e2.num_secu         = 14456464443
 e2.nom              = "Mickey"
 e2.qualification    = "Inge"
 e2.Lieu_de_travail = "Brest"
```

1.3.3 Constructeur

Si on considère une nouvelle instance :

```
# norbert
 e3=Employe()

 print   (e3.num_secu,            # ERREUR
          e3.nom,
          e3.qualification,
          e3.Lieu_de_travail)
```

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

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.*

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

```
class Employe:
    def __init__(self):        # Constructeur de la classe Employe
        self.num_secu          = 000000000
        self.nom               = "no_name"
        self.qualification     = "novice"
        self.Lieu_de_travail   = "Paris"

# norbert
e3=Employe()        # creation de e3 : appel implicite de init de Employe

print  (e3.num_secu,            # OK !!!
        e3.nom,
        e3.qualification,
        e3.Lieu_de_travail)
```

```
class Employe:
    def __init__(    self,
                     le_num_secu,
                     le_nom,
                     la_qualification,
                     le_Lieu_de_travail):
        self.num_secu          = le_num_secu
        self.nom               = le_nom
        self.qualification     = la_qualification
        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.

```
class Employe:
    def __init__(    self,
                     le_num_secu          = 000000000,
                     le_nom               = "no_name",
                     la_qualification     = "novice",
                     le_Lieu_de_travail   = "Paris"):
        self.num_secu          = le_num_secu
        self.nom               = le_nom
        self.qualification     = la_qualification
        self.Lieu_de_travail   = le_Lieu_de_travail
############## Programme Principal ##############
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 son contructeur 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-clef
   - (a) `init`
   - (b) `new`
   - (c) `class`
4. Le constructeur
   - (a) ne posséde 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

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*

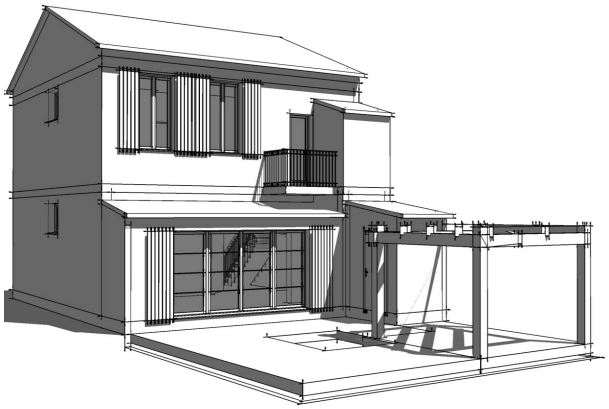
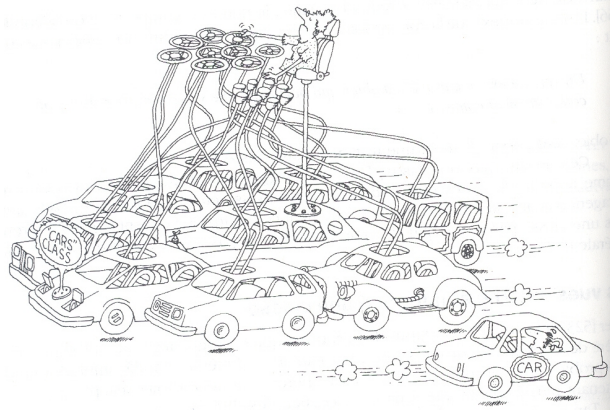


Figure 2.2. *Une classe représente un ensemble d'objets qui partagent une structure commmune et un comportement commun [Booch, 1992]*



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. Concrétement, 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

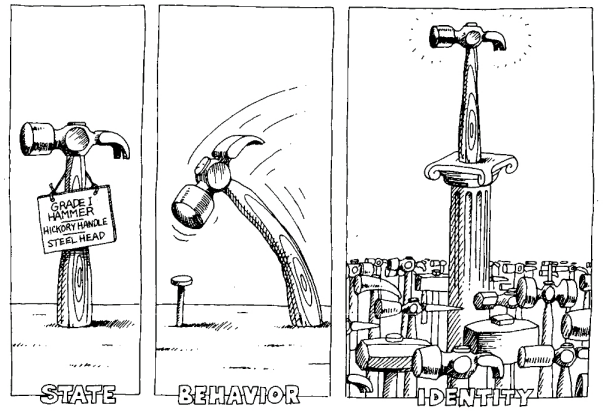
L'état d'un objet :

- ▷ 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          = "pipo"
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()
16
```

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 grâe à un identifiant découlant naturellement du problème (par exemple un produit pourra être repéré par un code, une voiture par un numéro de série ...).

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

Plusieurs techniques peuvent être utilisées :
→ nommer chaque objet (ex. attribut "name")

`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 paramétres 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 crées par l'objet courant au cours de son

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

existence.

```
1
2  class Complexe:
3          def __del__(self):
4                  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'acceder a l'instance
9  I am dying       # elle est susceptible d'etre detruite ce qui
10 >>>              # arrive instantanement 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

Objectifs du Cours 3 / Labo 3 :

- ▷ Encapsulation
- ▷ Régles 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 accés à une instance de la classe `Compte` peut modifier le solde. Ceci n'est pas acceptable.

3.1.2 Modes de contrôle d'accés

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]*

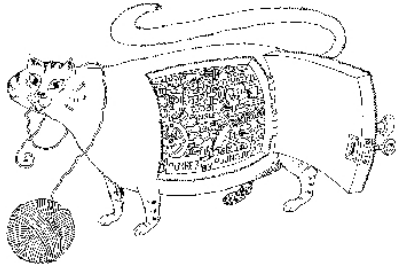


Figure 3.2. *Définir une classe en contrôlant l'accès à certains attributs ou méthodes*

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

```
class Test:

      def __init__(self, laValeurPublic,laValeurPrivee):
          self.valeurPublic    = laValeurPublic
          self.__valeurPrivee = laValeurPrivee


      def f1(self):
          ...
      def __f2(self):
          ...


######## Programme principal #########
t1 = Test()                 # intance (nous sommes en "dehors" de la classe)

print t1.valeurPublic       # OK
print t1.__valeurPrivee     # ERREUR

t1.f1()                     # OK
t1.__f2()                   # ERREUR
```

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

```
class Test:
      .....

     def debit(self, laValeur):
         if     self.__solde - laValeur > 0     :
                    self.__solde    =  self.__solde - laValeur
         else:
                    print("Compte bloque !!!")
```

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

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

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 Régles d'encapsulation

Les régles 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 accés
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 :
  - ◇ debrayer 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

Accés 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__(self):
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  eauChaude = Temperature(310)
4  print eauChaude.getValeurC()
5  print eauChaude.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 :

```
vehicule1 = Voiture();
vehicule1.setLargeur(100) # Largeur de la voiture
vehicule1.mettreAJour()   # Mettons a jour l'affichage de la voiture pour une largeur de 100
vehicule1.setLargeur(150) # Changeons la largeur de la voiture
vehicule1.mettreAJour()   # Mettons a 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 :

```
    def setLargeur(self,value):
        self.__largeur = value
        self.mettreAJour()
```

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

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

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

```
vehicule1 = Voiture();
vehicule1.setLargeur(100) # Largeur de la voiture
# On verifie que la largeur est dans les limites
if vehicule1.getLargeur() < 100 :  vehicule1.setLargeur(100)
else if vehicule1.getLargeur() > 200 : vehicule1.setLargeur(200)
print vehicule1.getLargeur() # Affiche: 100

vehicule1.setLargeur(250) # Changeons la largeur de la voiture
if vehicule1.getLargeur() < 100 :  vehicule1.setLargeur(100)
else if vehicule1.getLargeur() > 200 : vehicule1.setLargeur(200)
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 :

```
    def setLargeur(self,value):
        self.__largeur = value

                # _largeur doit etre comprise entre 100 et 200
        if self.getLargeur() < 100 :       self.__largeur = 100
        else if self.getLargeur() > 200 : self.__largeur = 200

        self.mettreAJour()
```

Le code principal serait alors écrit ainsi :

```
vehicule1 = Voiture();
vehicule1.setLargeur(100) # Largeur de la voiture
# Plus besoin de verifier que la largeur est dans les limites, l'accesseur le fait pour nous !
print vehicule1.getLargeur() #  Affiche: 100

vehicule1.setLargeur(250); # Changeons la largeur de la voiture
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'accés 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 préferentielle, 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

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.

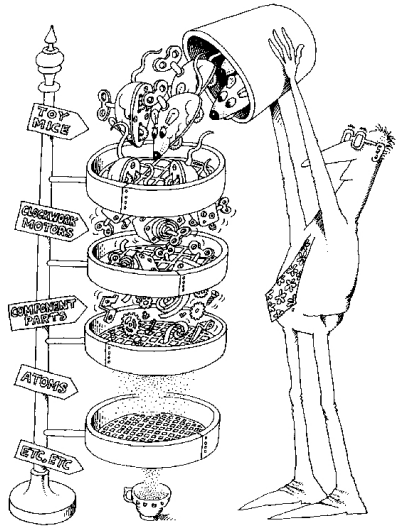
$$\forall x,\ x \in \text{FichierBinaire} \Rightarrow x \in \text{Fichier}$$

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 :

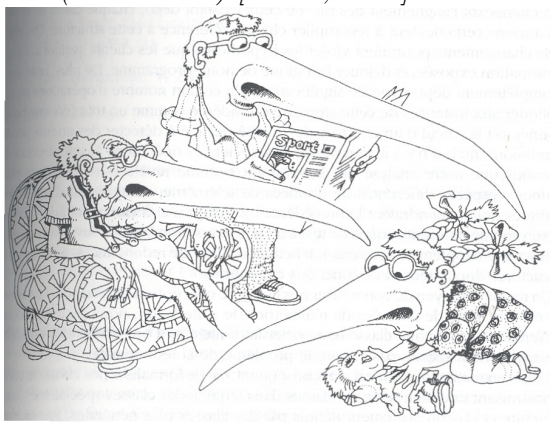
1. l'*enrichissement* : la sous–classe est dotée de nouvelle(s) méthode(s) et/ou de nouveau(x) attribut(s) ;
2. la *surcharge* : une (ou plusieurs) méthode et/ou un attribut hérité est redéfini.

Enrichissement

ajout d'une méthode

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

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* `médor` *instance de* `Chien`
- ▷ *Faite 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

```
######### class Fichier ##########
   class Fichier:

        def __init__(self):                          # Constructeur
           print "creation d'un fichier"

        def ouvrir(self):
            print "ouvrir fichier"


######### class FichierBinaire ##########
   class FichierBinaire(Fichier):

        def __init__(self):                        # Constructeur
            Fichier.__init__(self)

        def ouvrir(self):                           # Surcharge de la methode
            print "ouvrir fichier Binaire"          # "ouvrir"
```

Exemple d'utilisation :

```
 class Engin:
    def __init__(self, braquage=90):
        self.__vitesse = 0
        self.__braquage = braquage

    def virage(self):
        return self.__vitesse / self.__braquage

 #-------------------------------------------
class Sous_marin_alpha(Engin):
    def __init__(self, braquage):
        Engin.__init__(self, braquage)

 #-------------------------------------------
class Torpille(Engin):
    def __init__(self, braquage):
        Engin.__init__(self, braquage)

    def explose(self):
        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* (**-morphe**) 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" specifiant 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
14      def area(self):
15             return "rectangle area"
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 "cirle 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'accés :

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

Règles d'écriture

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

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

```
1  class Telephone:
2          def __init__(self):
3              # donnees prives
4              self.__numero_serie = '126312156'
5
6              # donnees protegees
7              self._code_pin = '1234'
8
9              # donnees publiques
10             self.modele = 'nokia 6800'
11             self.numero = '06 06 06 06 06'
12
13             # methodes privees
14
15             # methodes protegees
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             # methodes 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'
```

```
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 'clik-clak'
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

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

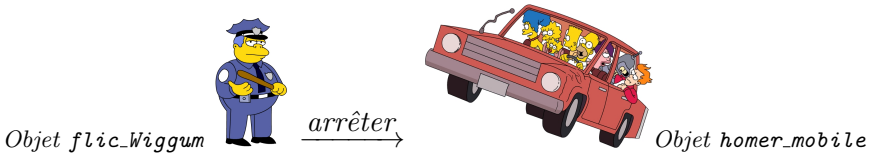
Exemple 5.2. *classe* `Afficheur`

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

```
# Relation multiple

class FilleMars:
      def __init__(self,prenom):
            self.__prenom = prenom

      def affichePrenom(self):
            print self.__prenom
#-------------------------------------------------
class PapaMars:
       def __init__(self):
           self.__filles = []
           self.__filles.append( FilleMars('Pamela') )
           self.__filles.append( FilleMars('Tina') )
           self.__filles.append( FilleMars('Rebecca') )
           self.__filles.append( FilleMars('Amanda') )

       def cri_a_table(self):
          for child in self.__filles:
              child.affichePrenom()
          print ('a table !!!')
```

```
# Relation multiple

class FilleMars:
      def __init__(self,prenom):
            self.prenom = prenom

      def affiche(self):
            print self.prenom
#-------------------------------------------------
class PapaMars:
       def __init__(self):
           self.filles = []
           self.filles.append( FilleMars('Pamela') )
           self.filles.append( FilleMars('Tina') )
           self.filles.append( FilleMars('Rebecca') )
           self.filles.append( FilleMars('Amanda') )

       def cri_a_table(self):
          for nom in self.filles:
              nom.affiche()
```

Exercice 5.1. *classe* `Autonombile`
Implémenter la classe `Autonombile` *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 systèmatiquement 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

Objectifs du Cours 5 / Labo 5 :

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

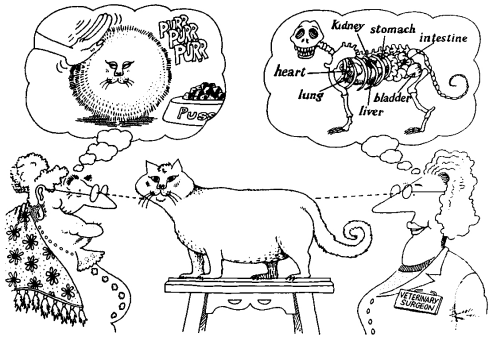
6.1 Abstraction

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 frontiéres 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]*

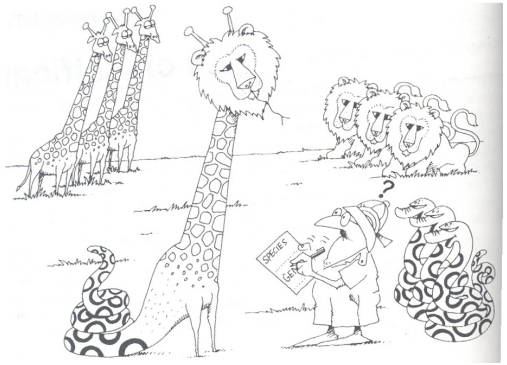


Figure 6.4. *Des obervateurs différents classent le même objet de différentes façons [Booch, 1992]*

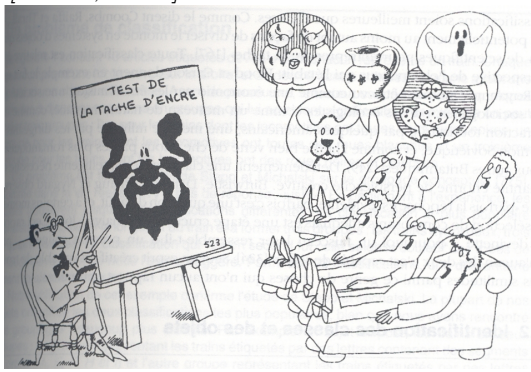
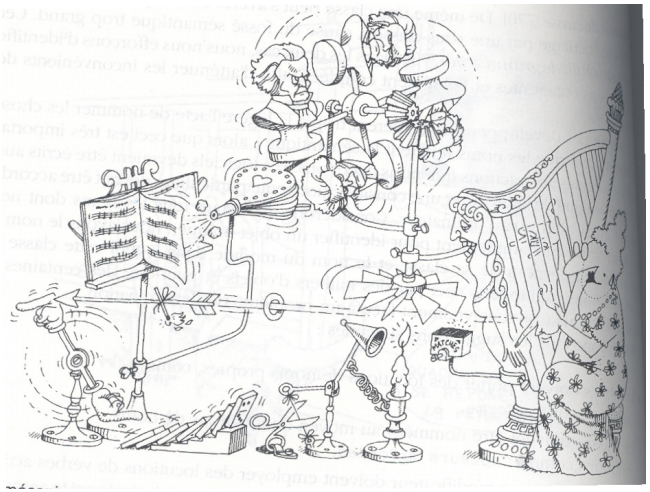
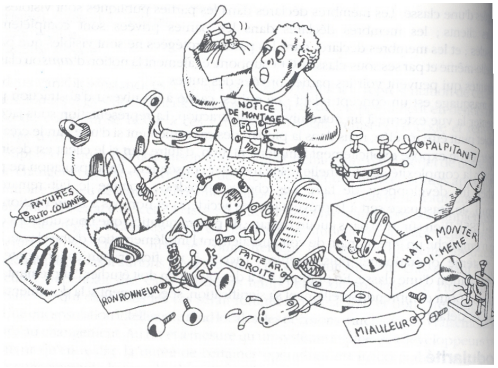


Figure 6.5. *Les mécanismes sont les moyens par lequels les objets collaborent pour procurer un certain comportement de plus haut niveau [Booch, 1992]*



6.3 Modularité

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

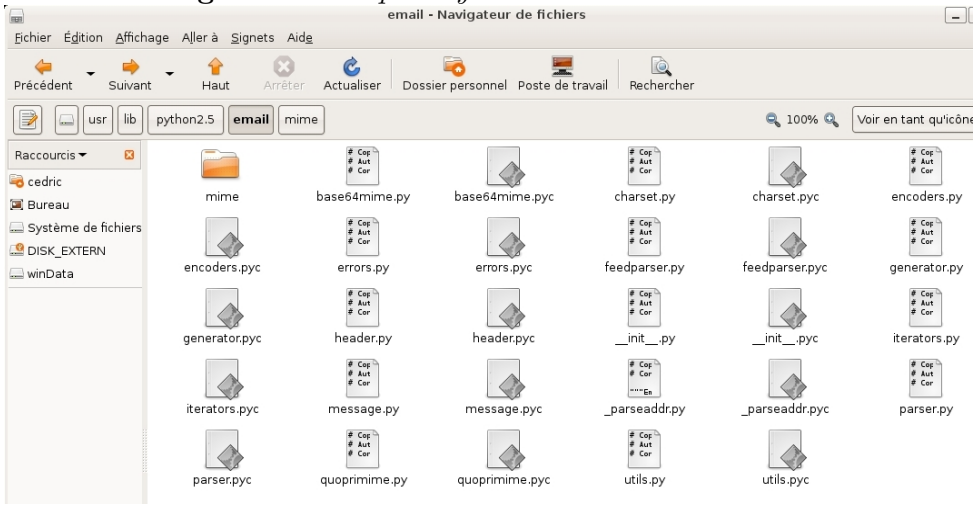


La programmation modulaire repose sur l'utilisation de modules, qui sont des structures permettant de définir des espaces regroupant des éléments définis par le programmeur : ensemble de classes, classes ...

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*

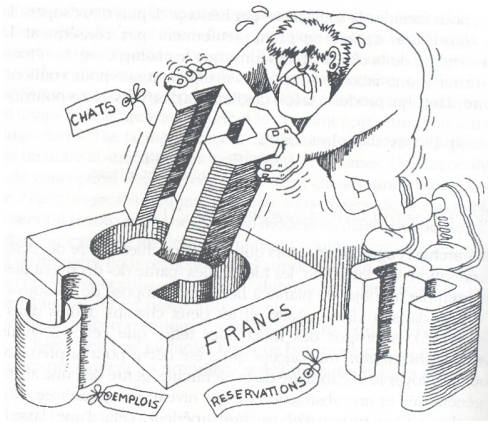
Utiliser ce module dans un autre programme.

◇ 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]*

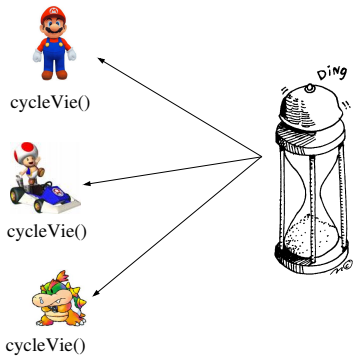



```
1 def estPair(nombre):
2         if not isinstance(nombre, int):
3                 return None
4         return (nombre % 2) == 0
```

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

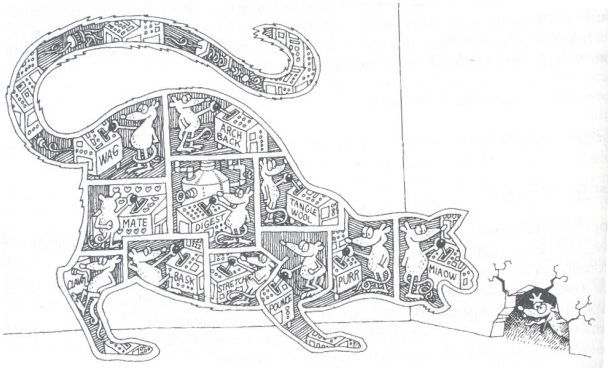
6.5 Simultanéité

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}
```

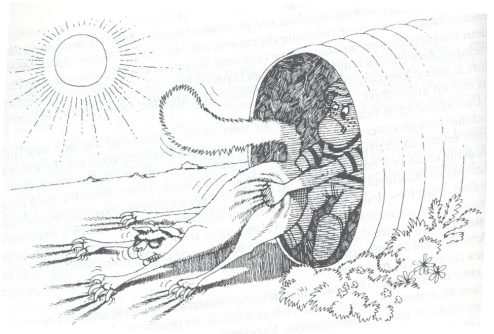
6.6.1 Mise en œuvre

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,datas = None):
20                 Persistent.__init__(self,id)
21                 if datas is not None:
22                         self.datas = datas
```

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

```
# programme principal
instance_de = MaClasse('1')
while True:
    try:
        if hasattr(instance_de,'datas'):
            instance_de.datas = raw_input('valeur (actuelle:%s): ' % instance_de.datas)
        else:
            instance_de.datas = raw_input('nouvelle valeur :')

     except KeyboardInterrupt:      # par exemple "Control-C"
        print 'fin'
        sys.exit(0)
```

Execution

```
>>> python testPersistance.py
chargement...
nouvelle valeur: homer
valeur (actuelle:homer): fin
sauvegarde...

>>> python testPersistance.py
chargement...
valeur (actuelle:homer): bart
valeur (actuelle:bart): fin
sauvegarde...


>>> python testPersistance.py
chargement...
valeur (actuelle:bart): lisa
valeur (actuelle:lisa): fin
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.

— Labo —

7 Rappels python (1 UC)

Sommaire

Objectifs

- ▷ Test : *if*, ==
- ▷ Boucle : *for*, *while*, *range*
- ▷ Tableau : parcours, *len*
- ▷ Récursivité : appel, condition d'arrêt
- ▷ Appel de fonctions : simple, imbriqué

7.1 Boucle

Calcul d'une moyenne

Ecrire un programme qui permette à l'utilisateur de calculer la moyenne de ses notes. Ce programme commencera par demander combien l'utilisateur a eu de notes durant le semestre. Il lira ensuite ces notes une à une pour faire le calcul de la moyenne. Après avoir lu toutes les notes, on affichera la moyenne calculée.

En python, on peut utiliser la fonction **`input`** *pour récupérer une chaîne de caractères tapée au clavier.*

```
1 ma_chaine = input('Taper une chaine: ')
2 print 'La chaine est : ', ma_chaine
```

7.2 Tableau

Recherche dans un tableau

Ecrire une fonction python qui recherche un entier x dans un tableau d'entiers A de longueur N et qui imprime tous les indices i tel que $Ai = x$.

7.3 Appel de fonctions

Considérons la bibliothèque de fonctions suivantes :

```
1  from math import *
2
3  def somme(x1,x2):
4          return x1+x2
5
6  def soustraction(x1,x2):
7          return x1-x2
8
9  def multiplication(x1,x2):
10         return x1*x2
11
12 def division(x1,x2):
13         return x1/x2
14
15 def racine(x):
16         return sqrt(x)
17
18 def puissance(x,exposant):
19         return x**exposant
20
21 def factorielle(x):
22         return fact(x)
23
24 def valeurAbsolue(x):
25         return abs(x)
26
27 def cosinus(x):
28         return cos(x)
29
30 def sinus(x):
31         return sin(x)
32
33 def moyenne(tab):
34         moy = 0.0
35         for i in range(len(tab)):
36                 moy = moy + tab[i]
37         moy  = division(moy,len(tab))
38         return moy
39
40 def logarithme(x):
41         return log(x)
```

Considérons la formule suivante :
formule de l'ecart type :
$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$ où $\bar{x}$ est la moyenne des n échantillons du tableau x

1. Ecrire une fonction qui calcule $(a-b)^2$ en utilisant au maximum la bibliothèque de fonctions. Le prototype de la fonction demandée est le suivant :

   ```
   def fonction1(a_real, b_real) -> result_real
   ```

2. Ecrire une fonction qui calcule $\sum_{i=1}^{n}(x_i - \bar{x})^2$ où $\bar{x}$ est la moyenne des n échantillons du tableau x, en utilisant au maximum la bibliothèque de fonctions et `fonction1`.
   Le prototype de la fonction demandée est le suivant :

   ```
   def fonction2(tab_real) -> result_real
   ```

3. On se propose d'écrire la fonction qui calcule l'ecart type sur un échantillon de n élèments, en utilisant au maximum la bibliothèque de fonctions et `fonction2`. Le prototype de la fonction demandée est le suivant :

   ```
   def ecartType(tab_real) -> result_real
   ```

7.4 Récursivité

Repeat

1. Ecrire une fonction `repete( n,str)` renvoyant la chaîne de caractères *str* répétée n fois : `repete(3, "bla")` donne `blablabla`.
2. Ecrire une fonction `pyramide(n, s)` qui écrit sur la première ligne, 1 fois la chaîne s, sur la deuxième, 2 fois la chaîne s, et ainsi de suite jusque la dernière ligne, où il y aura n fois la chaîne s. Ainsi `pyramide(5, "bla")` donnera
   ```
   bla
   blabla
   blablabla
   blablablabla
   blablablablabla
   ```
3. Quand on lance `pyramide(n, s)`, combien y a-t-il d'appels à `pyramide` ? Combien y a-t-il d'appels à `repete` ? Pour vous aider à répondre dessiner l'arbre des appels pour $n = 3$.

Exercice supplémentaire 7.1. *Recherche dans un tableau par dichotomie*

Ecrire la fonction `rechercheDicho(A,x)` *qui recherche un entier x dans un tableau ordonné d'entiers A de longueur N en utilisant une recherche dichotomique.*

Exercice supplémentaire 7.2. *Valeur la plus proche*

Décrire une fonction récursive qui, étant donné un entier X, détermine la valeur la plus proche de X dans un tableau d'entiers.

7.5 QCM

1. Quel appel est correct pour cette fonction ?

```
def fonction(a,b,c,d=1):
      print a,b,c,d
```

   (a) `fonction(10,5,3)`
   (b) `fonction(a,b,c,d)`
   (c) `fonction(10)`
   (d) `fonction`

2. Combien d'appels à `f` sont effectués ?

```
def f(n):
      if n==1 :
            return 1
      else :
            return n*f(n-1)

f(5)
```

   (a) 10
   (b) 5
   (c) 1
   (d) 6

3. Qu'affiche ce code ?

```
i=1
x=2
while x<10:
      print i
```

   (a) 2
   (b) 111111111
   (c) 11111111
   (d) 1111111
   (e) rien
   (f) 11111111111111111111111 ...

8 Concepts fondateurs (1 UC)

Sommaire

8.1 Exercices préparatoires

8.1.1 Classe ou objet ?

Différencier les classes des objets (instances de classe) dans les propositions suivantes : Footballeur, Zidane, Basket, Sport, Voiture, Clio, Rouge, Bleu, Homme, Couleur, Mégane, Manaudou.

8.1.2 Attributs de classe

Les étudiants sont caractérisés par leur numéro d'étudiant, leur nom, leur prénom, leur date de naissance et leur adresse.

Identifier les attributs de la classe `Etudiant`.

8.2 Classe

Un complexe z est composé d'une partie réelle x et d'une partie imaginaire y, tel que $z = x + i * y$. Nous proposons ici d'implémenter une classe `Complex`.

8.2.1 Définition

▷ Définir une classe `Complex`
▷ Commenter la classe `Complex`

En python, si la première ligne suivant l'instruction `class` est une chaîne de caractères, celle-ci sera considérée comme un commentaire et incorporée automatiquement dans un dispositif de documentation des classes qui fait partie intégrante de Python. Prenez donc l'habitude de toujours placer une chaîne décrivant la classe à cet endroit.

8.2.2 Constructeur

▷ Si cela n'a pas déjà été fait (pas bien), ajouter un constructeur
▷ Ajouter deux attributs `x` et `y`
▷ Ajouter au constructeur la possibilité de spécifier les valeurs de `x` et `y`

8.3 Objet

8.3.1 Instanciation

Nous venons de définir une classe `Complex`. Nous pouvons dès à présent nous en servir pour créer des objets de ce type, par instanciation.

▷ Créer un objet `c1`, instance de la classe `Complex`.

8.3.2 Identité

▷ Effectuer les manipulations avec notre nouvel objet `c1` suivantes :

```
>>> print c1.__doc__
>>> print c1
```

Que remarquez vous ?

8.3.3 Comportement

- ▷ Affecter la valeur 3 à l'attribut `x` et 4 à l'attribut `y` de l'objet `c1`
- ▷ Les fonctions peuvent utiliser des objets comme paramètres.
  Définir une fonction `affiche_complex` qui affiche les valeurs du complex passé en paramétre.
  Testez la avec l'objet `c1`.
- ▷ Créer une méthode (fonction dans la classe) `show` qui affiche les attributs de la classe `Complex`.
  Testez la avec l'objet `c1`.

8.3.4 Manipulation de plusieurs objets

- ▷ Créer une fonction `compareComplex` qui renvoie vrai si chaque attribut de deux objets sont identiques.
- ▷ Créer un objet `c2` de type `Complex`, tester `compareComplex`

8.3.5 Manipulation de plusieurs objets Python

- ▷ Affecter la valeur 3 à l'attribut `x` et 4 à l'attribut `y` de l'objet `c2`.
- ▷ Ajouter une méthode `clone` qui recopie un objet de type `Complex`.
- ▷ Effectuer les manipulations suivantes :

```
>>> print (c1 == c2)
>>> c1=c2; print (c1 == c2)
>>> c3=c1.clone(); print (c1 == c3)
```

  Que remarquez vous ?

Exercice supplémentaire 8.1. *Classe Livre*

1. *Ecrire une classe* `Livre` *avec les attributs suivants :*
   - ▷ `titre` *: Le titre du livre,*
   - ▷ `auteur` *: L'auteur du livre,*
   - ▷ `prix` *: Le prix du livre,*
   - ▷ `annee` *: L'année du livre.*
2. *La classe* `Livre` *doit pouvoir être construites avec les possibilités suivantes :*

- ▷ `Livre()`,
- ▷ `Livre(titre)`,
- ▷ `Livre(titre, auteur)`,
- ▷ `Livre(titre, auteur, prix)`,
- ▷ `Livre(titre, auteur, prix, annee)`

3. *Instancier deux livres de votre choix*
4. *Créer une fonction qui renvoie vrai si le titre de l'instance passée en paramètre correspond au titre demandé.*

Exercice supplémentaire 8.2. *Classe* `Time`

- ▷ *Définir une nouvelle classe* `Time`*, qui nous permettra d'effectuer toute une série d'opérations sur des instants, des durées, etc.*
- ▷ *Ajouter les attributs* `pays`*,* `heure`*,* `minute` *et* `seconde`*.*
- ▷ *Créer un objet* `instant` *qui contient une date particulière.*
- ▷ *Ecrivez une fonction* `affiche_heure()` *qui permet de visualiser le contenu d'un objet de classe* `Time` *sous la forme conventionnelle "heure :minute :seconde".*
- ▷ *Appliquée à l'objet* `instant` *la fonction* `affiche_heure()`
- ▷ *Encapsuler la fonction* `affiche_heure()` *dans la classe* `Time` *(méthode* `affiche`*)*
- ▷ *Instancier un objet* `maintenant`
- ▷ *Tester la méthode*

Une fonction qui est ainsi encapsulée dans une classe s'appelle une méthode.

Exercice supplémentaire 8.3. *Pour obtenir les nombres premiers compris entre 2 et un certain entier N on va construire une liste chaînée d'objets appelés des MangeNombres, chacun comportant deux variables d'instance : un nombre premier et une référence sur le MangeNombres suivant de la liste.*

Le comportement d'un MangeNombres se réduit à l'opération "manger un nombre". Le MangeNombres mp associé au nombre p mange les multiples de p : si on lui donne à manger un nombre q qui n'est pas multiple de p, mp le donne à manger à son MangeNombres suivant, s'il existe. Si mp n'a pas de suivant, celui-ci est crée, associé à q. La création d'un MangeNombres mp provoque l'affichage de p. Définissez la classe MangeNombres et écrivez un programme affichant les nombres premiers entre 2 et N.

Exercice supplémentaire 8.4. *classe Ensemble*

1. *Créez une classe Ensemble pour représenter des sous-ensembles de l'ensemble des entiers compris entre 0 et N - 1, ou N est une constante donnée, avec les méthodes suivantes :*
   - ▷ *cardinal() : nombre d'éléments de l'ensemble,*
   - ▷ *contient(i) : test d'appartenance,*
   - ▷ *ajout(i) : ajout d'un élément à l'ensemble,*
   - ▷ *toString() : transformation de l'ensemble en une chaîne de caractères de la forme {5 ; 6 ; 10 ; 15 ; 20 ; 21 ; 22 ; 30}*

   Les éléments d'un ensemble seront mémorisés dans une variable d'instance de type tableau. Pour essayer la classe Ensemble, écrivez un programme déterminant les nombres différents contenus dans une suite de nombres lue sur l'entrée standard.
2. *Modifiez la classe précédente afin de permettre la représentation d'ensembles d'entiers quelconques, c'est- à-dire non nécessairement compris entre 0 et une constante N connue à l'avance. Faites en sorte que l'interface de cette classe soit identique à celle de la version précédente qu'il n'y ait rien à modifier dans les programmes qui utilisent la première version de cette classe.*

8.4 QCM

1. Comment créer une **méthode** `show()` qui affiche l'attribut `x` de la classe `B` ? _____

(a)
```
class B:
    def __init__(self):
          self.x = 3
    def show(b1):
          print b1.x
```

(b)
```
class B:
    def __init__(self):
          self.x = 3

def show():
    print x
```

(c)
```
class B:
    def __init__(self):
          self.x = 3
    def show(self):
          print self.x
```

(d)
```
class B:
    def __init__(self):
          self.x = 3

def show(b1):
    print b1.x
```

2. Que renvoie la fonction `compareD` ?

```
class D:
    def __init__(self,xinit=10):
         self.x = xinit

def compareD(d1,d2):
       return d1.x == d2.x

d10 = D(2)
d20 = D()
print compareD(d10,d20)
```

(a) Faux

(b) Vrai

(c) Une erreur

(d) Rien

3. Combien de fois *new C* va t'il être affiché ?

```
class C:
  def __init__(self):
       print "new  C"

i = 0
while i < 10:
     c1 = C()
     i = i + 1
```

(a) 0 fois

(b) 2 fois

(c) 10 fois

(d) 1 fois

4. Comment créer une instance de `A` ?

```
class A:
  def __init__ (self):
          print "toto"
```

(a) `a1 = A()`

(b) `a1 = A(10)`

(c) `a1 = A`

(d) `A() = a1`

9 Encapsulation (1 UC)

Sommaire

Dans les exercices qui suivent, utilisez les règles d'encapsulation vues en cours (contrôle d'accés, mutateurs, observateurs ...).

9.1 Domino

Définissez une classe `Domino` qui permette d'instancier des objets simulant les pièces d'un jeu de dominos.

1. Le constructeur de cette classe initialisera les valeurs des points présents sur les deux faces A et B du domino (valeurs par défaut = 0).
2. Deux autres méthodes seront définies :
   ▷ une méthode `affiche_points()` qui affiche les points présents sur les deux faces
   ▷ une méthode `valeur()` qui renvoie la somme des points présents sur les 2 faces.

Exemples d'utilisation de cette classe :

```
>>> d1 = Domino(2,6)
>>> d2 = Domino(4,3)
>>> d1.affiche_points()
 face A : 2 face B : 6
>>> d2.affiche_points()
 face A : 4 face B : 3
>>> print "total des points :", d1.valeur() + d2.valeur()
 15
>>> liste_dominos = []
>>> for i in range(7):
        liste_dominos.append(Domino(6, i))
>>> for j in  liste_dominos:
        j.affiche_points()
```

9.2 CompteBancaire

Définissez une classe `CompteBancaire`, qui permette d'instancier des objets tels que `compte1`, `compte2`, etc.

1. Le constructeur de cette classe initialisera deux attributs `nom` et `solde`, avec les valeurs par défaut `'Dupont'` et `1000`.
2. Trois autres méthodes seront définies :
   ▷ `depot(somme)` permettra d'ajouter une certaine somme au solde
   ▷ `retrait(somme)` permettra de retirer une certaine somme du solde
   ▷ `affiche()` permettra d'afficher le nom du titulaire et le solde de son compte.

Exemples d'utilisation de cette classe :

```
>>> compte1 = CompteBancaire('Duchmol', 800)
>>> compte1.depot(350)
>>> compte1.retrait(200)
>>> compte1.affiche()
Le solde du compte bancaire de Duchmol est de 950 euros.
>>> compte2 = CompteBancaire()
>>> compte2.depot(25)
>>> compte2.affiche()
Le solde du compte bancaire de Dupont est de 1025 euros.
```

9.3 Voiture

Définissez une classe `Voiture` qui permette d'instancier des objets reproduisant le comportement de voitures automobiles.

1. Le constructeur de cette classe initialisera les attributs d'instance suivants, avec les valeurs par défaut indiquées :
   `marque = 'Ford', couleur = 'rouge', pilote = 'personne', vitesse = 0.`
   Lorsque l'on instanciera un nouvel objet `Voiture()`, on pourra choisir sa marque et sa couleur, mais pas sa vitesse, ni le nom de son conducteur.
2. Les méthodes suivantes seront définies :
   ▷ `choix_conducteur(nom)` permettra de désigner (ou de changer) le nom du conducteur

- ▷ `accelerer(taux, duree)` permettra de faire varier la vitesse de la voiture.
  La variation de vitesse obtenue sera égale au produit : `taux` * `duree`. Par exemple, si la voiture accélère au taux de $1,3 m/s^2$ pendant 20 secondes, son gain de vitesse doit être égal à 26 m/s. Des taux négatifs seront acceptés (ce qui permettra de décélérer). La variation de vitesse ne sera pas autorisée si le conducteur est 'personne'.
- ▷ `affiche_tout()` permettra de faire apparaître les propriétés présentes de la voiture, c'est-à dire sa marque, sa couleur, le nom de son conducteur, sa vitesse.

Exemples d'utilisation de cette classe :

```
>>> a1 = Voiture('Peugeot', 'bleue')
>>> a2 = Voiture(couleur = 'verte')
>>> a3 = Voiture('Mercedes')
>>> a1.choix_conducteur('Romeo')
>>> a2.choix_conducteur('Juliette')
>>> a2.accelerer(1.8, 12)
>>> a3.accelerer(1.9, 11)
Cette voiture n'a pas de conducteur !
>>> a2.affiche_tout()
Ford verte pilotee par Juliette, vitesse = 21.6 m/s.
>>> a3.affiche_tout()
Mercedes rouge pilotee par personne, vitesse = 0 m/s.
```

10 Collaboration et héritage (1 UC)

Sommaire

10.1 Collaboration

10.1.1 Exercices préparatoires

Pour définir un rectangle, nous spécifions sa largeur, sa hauteur et précisons la position du coin supérieur gauche. Une position est définie par un point (coordonnées x et y).
Quelle(s) classe(s) et quel(s) attribut(s) peuvent être définis pour représenter ces notions ?
A quoi pourra servir le constructeur de chaque classe ?

10.1.2 Collaboration entre objets

Objets composés d'objets

- ▷ Définir la classe `Point` contenant les attributs `x` et `y` (coordonnées)
- ▷ Définir la classe `Rectangle`
- ▷ Instancier un objet `rectangle1` de largeur 50, de hauteur 35, et dont le coin supérieur gauche se situe au coordonnée (12,27)

Constructeur / Destructeur

- ▷ Rafiner au constructeur de la classe `Point` et de la classe `Rectangle` les possibilités que vous jugerez utiles (valeurs par défaut ...)
- ▷ Préciser le destructeur de la classe `Point` et de la classe `Rectangle`

▷ Tester le destructeur de la classe `Rectangle`

Objets comme valeurs de retour d'une fonction Nous avons vu plus haut que les fonctions peuvent utiliser des objets comme paramètres. Elles peuvent également transmettre une instance comme valeur de retour.

▷ Définir la fonction `trouveCentre()` qui est appelée avec un argument de type `Rectangle` et qui renvoie un objet `Point`, lequel contiendra les coordonnées du centre du rectangle.
▷ Tester cette fonction en utilisant l'objet `rectangle1` défini en question 10.1.2.

Modifier un objet Modifier la taille d'un rectangle (sans modifier sa position), en réassignant ses attributs hauteur (hauteur actuelle +20) et largeur (largeur actuelle -5).

10.2 Héritage

10.2.1 Exercices préparatoires

Gestion d'heures complémentaires Chaque enseignant de l'université effectue un certain nombre d'heures d'enseignement dans une année. Suivant le statut de l'enseignant, un certain nombre de ces heures peut-être considéré comme complémentaire. Les heures complémentaires sont payées séparément à l'enseignant. Les volumes horaires sont exprimés en heures entières et le prix d'une heure complémentaire est de 10 Euros.
Le nom et le nombre d'heures total d'un enseignant sont fixés à sa création, puis seul le nom peut être librement consulté (méthode `nom()`).
D'autre part on veut pouvoir librement consulter un enseignant sur son volume d'heures complémentaires (méthode `hc()`) et sur la rétribution correspondante (méthode `retribution()`).
Il y a deux types d'enseignants :

▷ les intervenants extérieurs : toutes les heures effectuées sont complémentaires,
▷ les enseignants de la fac : seules les heures assurées au delà d'une charge statutaire de 192h sont complémentaires.

Questions :

1. Sans écrire de code, modéliser les enseignants : quelles sont les classes ? où sont implémentées les méthodes ? lesquelles sont nouvelles, redéfinies ?

2. Comment modifier le modèle pour y introduire les étudiants de troisième cycle qui assurent des enseignements : toutes les heures effectuées sont complémentaires mais dans la limite de 96 heures.

10.2.2 Héritage simple et multiple

- ▷ Définir une classe `Mammifere`, qui contiendra un ensemble de caractéristiques propres à ce type d'animal (nom, âge, nombre de dents ...).
- ▷ A partir de cette classe, nous pourrons alors dériver une classe `Primate`, une classe `Rongeur`, et une classe `Carnivore`, qui hériteront toutes les caractéristiques de la classe `Mammifere`, en y ajoutant leurs spécificités.
- ▷ Au départ de la classe `Carnivore`, dériver une classe `Belette`, une classe `Loup` et une classe `Chien`, qui hériteront encore une fois toutes les caractéristiques de la classe parente avant d'y ajouter les leurs.
- ▷ Définir une classe `Herbivore` dérivé de la classe `Primate`
- ▷ Définir une classe `Omnivore`, qui hérite de la classe `Carnivore` et de la classe `Herbivore`

10.2.3 Polymorphisme

- ▷ Ecrivez une classe `Forme` qui contiendra une méthode `calculAire`.
- ▷ Faites-en hériter la classes `Carre` contenant un attribut `cote`
- ▷ idem pour la classe `Cercle` contenant un attribut `rayon`.
- ▷ Rédéfinir la méthode `calculAire` pour les classes `Carre` et `Cercle`
- ▷ Définir une fonction `AireTotal` qui à partir d'un tableau de `Forme` calcul l'aire totale

Exercice supplémentaire 10.1. *écrire le code python de l'exercice de préparation sur la gestion d'heures complémenatires*

10.3 QCM

1. Qu'affiche ce code ?

```
class A:
            def __init__(self):
                  print "Creation d'un objet A"
class B(A):
            def __init__(self):
                   A.__init__(self)
                   print "Creation d'un objet B"
class C(B):
            def __init__(self):
                   B.__init__(self)
                   print "Creation d'un objet C"
c1 = C()
```

   (a) Creation d'un objet A , Creation d'un objet B , Creation d'un objet C

   (b) Creation d'un objet C , Creation d'un objet B , Creation d'un objet A

   (c) Creation d'un objet C , Creation d'un objet A , Creation d'un objet B

   (d) Creation d'un objet C

2. Qu'affiche ce code ?

```
class A:
              def __init__(self, xInit = 1):
                           self.x = xInit
class B(A):
             def __init__(self, xInit=0, yInit=0)
                           A.__init__(self, xInit)
                           self.y = yInit
v1 = B()
v2 = B(5, 4)
print v1.x, v1.y, v2.x, v2.y
```

   (a) 0 0 5 4

   (b) 1 0 0 0

   (c) Une erreur

   (d) 0 0 0 0

   (e) 5 4 5 4

3. Qu'affiche ce code ?

```
class A:
            def __init__(self, x):
                        self.x = x
class B(A):
            def __init__(self):
                        A.__init__(self)
b = B(5)
print b.x
```

(a) x

(b) 0

(c) Une erreur

(d) 5

4. Qu'affiche ce code ?

```
class A:
            def __init__(self, xInit = 0.0):
                        self.__x = xInit
a = A(5.0)
print a.__x
```

(a) 5.0

(b) __x

(c) Une erreur

(d) 0.0

5. Qu'affiche ce code ?

```
class A:
......
            def show(self):
                        print "Je suis un A"
class B(A):
......
            def show(self):
                        print "Je suis un B"
a = B()
a.show()
```

(a) Je suis un B

(b) Je suis un B, Je suis un A

(c) Je suis un A

(d) Une erreur

(e) Je suis un A, Je suis un B

11 Pour aller plus loin

Sommaire

11.1 Abstraction (1 UC)

11.1.1 Exercices préparatoires

Vie parisienne Toute sa vie, voici la journée d'un parisien :

- ▷ métro
- ▷ boulot
- ▷ dodo

Dans le métro, tous les parisiens ont le même comportement.
Dans leur lit, tous les parisiens ont le même comportement.
Le boulot différe pour chacun d'entre eux (parisien commerçant, parisien cadre ...) .
Sans écrire de code, comment représenter ces informations ?

11.1.2 Classe abstraite / méthode abstraite

L'objectif est de définir une classe abstraite destinée à gérer un tableau trié d'éléments et comportant une méthode abstraite `plusGrand(self,a,b)`. Cette méthode devra comparer deux éléments. Pour gérer un tableau trié d'objets d'un certain type, il faudra étendre la classe abstraite en une classe définissant la méthode `plusGrand(self,a,b)` pour le type d'objets en question.

On construira :

1. Une classe abstraite `TableauTrieAbstrait` gérant un tableau d'éléments qui reste toujours trié par ordre croissant par rapport à la relation définie par une méthode abstraite `plusGrand`. Cette classe devra contenir au moins
   - ▷ une méthode abstraite `plusGrand(self,a,b)`

- ▷ une méthode `inserer(self,element)` pour insérer une instance d'élément dans le tableau (il faut faire attention à l'ordre pour ne pas l'insérer n'importe oú)

2. Une classe `TableauTrieEntiers` qui étend la classe `TableauTrieAbstrait` ; cette classe est destinée à gérer un tableau trié d'entier. Il faut essentiellement y définir la méthode `plusGrand(self,a,b)` pour des entiers.
3. Une classe, `TableauTrieChaines` qui étend la classe `TableauTrieAbstrait` ; cette classe est destinée à gérer un tableau trié de chaîne de caractères. Il faut essentiellement y définir la méthode `plusGrand(self,a,b)` en se basant sur le nombre de caractères.

Exercice supplémentaire 11.1. *Vie parisienne . . . suite*
Implémenter les classes de l'exercice préparatoire sur la vie parisienne.
Ajouter les informations suivantes :

- ▷ *la jeunesse de tous les parisiens est la même (cris de bébés, croissance, études)*
- ▷ *à la fin de sa vie, le parisien fait le bilan de sa vie (le commercial compte son pécule, le chercheur regarde combien il avait trouvé de réponses et de problèmes)*
- ▷ *la mort de tous les parisiens est la même (un dernier souffle)*

Exercice supplémentaire 11.2. *Boulangerie*

Un simulateur fait "vivre" trois personnages économiques que sont :

1. *Le Paysan*
2. *Le Meunier*
3. *Le Boulanger*

Les classes à construire sont :

1. *Le Paysan*
   - ▷ *Mange du pain pour vivre*
   - ▷ *Produit du blé*
   - ▷ *Livre du blé à un menier*
   - ▷ *Reçoit du pain*
2. *Le Meunier*
   - ▷ *Mange du pain pour vivre*
   - ▷ *Reçoit du blé*

- ▷ *Produit de la farine en consommant du blé*
- ▷ *Livre de la farine à un boulanger*
- ▷ *Reçoit du pain*

3. *Le Boulanger*
   - ▷ *Mange du pain (de son propre stock) pour vivre*
   - ▷ *Reçoit de la farine*
   - ▷ *Fabrique du pain en consommant de la farine*
   - ▷ *Livre du pain aux meuniers et aux paysans (en fait, à tous ceux qui peuvent en recevoir)*

- ▷ *Etablir les trois "identités de classe" des personnages, en décrivant les attributs nécessaires.*
- ▷ *Peut-on trouver un principe "plus abstrait" qui permette de factoriser certains éléments ?*
  $\rightarrow$ *Etablir cette classe.*
- ▷ *Identifier les comportements communs et les résoudre dans cette classe.*
- ▷ *Modifier et terminer les classes personnages pour tenir compte de cette factorisation.*
- ▷ *Evaluer une manière "intéressante" de procéder aux échanges de matière.*
- ▷ *Ecrire une petite classe "enveloppante" qui mette en scéne trois personnages et les fassent "vivre" dans une boucle en mode "tour par tour".*

Exercice de synthèse

12 Exercice : Gaia

Ce sujet porte sur la modélisation et la simulation de la diffusion de chaleur à la surface de la planète.

12.1 Sujet

12.1.1 Présentation générale des classes de base

Les classes utilisées sont :

1. `Cellule`
   Chacune d'elle possède des caractéristiques physiques (température, coefficient de diffusion, coefficient de perte). La température d'une cellule va conditionner sa couleur. La méthode `update()` permet de calculer la température de la cellule (voir en section 12.1.3 le détail de ce calcul).
2. `Soleil`
   Le soleil est un élément particulier qui est associé à une cellule qu'il chauffe.
3. `Terrain`
   Le terrain est rectangulaire et composé de cellules.
4. `Simulateur`
   Enfin, le simulateur possède une méthode `run()` qui va exécuter régulièrement la méthode `update()` du terrain qui exécutera les méthodes `update()` de chaque cellule (et du soleil).

12.1.2 Détails sur le terrain

Le terrain possède les attributs suivants :

- ▷ `temperatureMoyenne` est la température moyenne du terrain.
- ▷ `TMIN` est une température minimale que nous imposons car les équations d'évolution de la température sont simplifiées et pourrait faire tendre cette température à $-\infty$.
- ▷ `nbcelluleX` est le nombre de cellules constituant la longueur du terrain.
- ▷ `nbcelluleY` est le nombre de cellules constituant la hauteur du terrain.

▷ `cellules` est un conteneur de cellules.

La méthode `update()` du terrain lance cycliquement la méthode `update()` du soleil puis les méthodes `update()` de toutes les cellules.

En plus de la méthode `update()`, il y a les méthodes suivantes :
▷ `celluleDessus()`, `celluleDessous()`, `celluleGauche()`, `celluleDroite()` renvoient les cellules se situant respectivement au dessus, au dessous, à gauche et à droite d'une cellule donnée. Ceci permettra aux cellules de calculer leur température. Ces méthodes prennent en compte le fait que le terrain reboucle sur lui même en haut et en bas (par exemple, la cellule de gauche d'une cellule au bord correspond à la cellule au bout à droite de la même ligne).

12.1.3 Détails sur les cellules

Une cellule possède les attributs suivants :
▷ `terrain` est le terrain qui contient la cellule
▷ `temperature` est la température de la cellule
▷ `coefDiffusion` est le coefficient de diffusion de cette température (permettant de caractériser par exemple le fait que l'eau se réchauffe moins vite que la terre).
▷ `x et y` représentent la position des cellules sur le terrain. Il s'agit du numéro de colonne et de ligne.
▷ `coefPerte` représente le fait que la température à tendance à descendre à cause de l'atmosphère.
▷ `couleur` représente la couleur de la cellule.

La méthode `update()` des cellules calcule la température grâce à la formule suivante :

$$T_x = max(TMIN, \left\{ \{[\sum_{x_i \in Voisins} \frac{T(x_i) - T_x}{nbVoisins}] * coefD_x\} + T_x - \{coefP_x * (-TMIN + T_x)\} \right\})$$

$coefD_x$ et $coefP_x$ étant respectivement les coefficients de diffusion et de perte de la cellule x. Les voisines sont ici au nombre de quatre et correspondent aux cellules de droite, de gauche du dessous et du dessus.

Cette méthode a également en charge de calculer la `couleur` de la cellule, allant de 0 (froid) à 255 (chaud) (méthode `normalize`).

12.1.4 Détails sur le soleil

La méthode `update()` du soleil réchauffe la cellule avec une température max de `TSoleil`, par le biais de la formule suivante :

$$T_x = T_x + coefD_x * (TSoleil - T_x)$$

où T_x est la température de la cellule sur laquelle se trouve le soleil.

Le soleil a des coordonnées `x et y` sur le `terrain` (contenant le soleil).

12.1.5 Détails sur le simulateur

Le simulateur posséde une méthode `run` qui :

1. instancie un soleil avec une température de 500
2. lui donne une position aux coordonnnées (50,50)
3. instancie un terrain de taille 100*100
4. lui donne l'instance du soleil
5. effectue une boucle infinie qui demande la mise à jour du terrain

12.2 Travail demandé

Proposer une implémentation de chacune des classes précédentes respectant le sujet. Instancier ensuite un simulateur. Vous respecterez les principes objets, plus particulièrement sur la visibilité des attributs.

12.2.1 La classe `Cellule`

12.2.2 La classe `Terrain`

12.2.3 La classe `Soleil`

12.2.4 La classe `Simulateur`

Instancier ensuite un simulateur et lancer la méthode `run`

13 Correction : GAIA

```
# -*- coding: utf-8 -


#####################################################
#### auteur : C.Buche
#### date   : 23/03/09
#####################################################

import math

class Terrain:
        pass

#####################################################
class Cellule :
        """ Une Cellule possede des caracteristiques physiques (temperature, coeff. de diffusion, coeff. de perte)      """

                # ------------ constructeurs et destructeurs ------------
        def __init__(self,terrain=Terrain(),x=0,y=0,temperature=0,coefDiffusion=0.7,coefPerte=0.001,couleur=0):
                print "creation d'une Cellule"
                self.__x = x
                self.__y = y
                self.__temperature = temperature
                self.__coefDiffusion = coefDiffusion
                self.__coefPerte = coefPerte
                self.__terrain = terrain
                self.__couleur = couleur

        def __del__(self):
                print "destruction d'une Cellule"

                # ------------ accesseurs et mutateurs ------------
        def getX(self):
                return self.__x

        def getY(self):
                return self.__y

        def setPosition(self,x,y):
                self.__x = x
                self.__y = y

```

```
43          def getTemperature(self):
44                  return self.__temperature
45
46          def setTemperature(self,t):
47                  self.__temperature = t
48
49          def setTerrain(self,t):
50                  self.__terrain = t
51
52          def getTemperature(self):
53                  return self.__temperature
54
55          def getCoefDiffusion(self):
56                  return self.__coefDiffusion
57
58          def getCouleur(self):
59                  return self.__couleur
60
61                  # ------------ methodes ------------
62          def __normaliser(self,x): ## exemple de fonction de normalisation
63                  pente = 0.5
64                  centre =  self.__terrain.getTemperatureMoyenne()
65                  y = 255. / (1 + math.exp(1-(x-centre)*pente) )
66                  y = int(y)
67                  return y
68
69          def update(self):
70                  ##  calcul de la nouvelle temperature
71                  min = self.__terrain.getTMIN();
72                  dTempDiff =  self.__terrain.celluleDessous(self).getTemperature() - self.__temperature;
73                  dTempDiff += self.__terrain.celluleDessus(self).getTemperature() - self.__temperature;
74                  dTempDiff += self.__terrain.celluleGauche(self).getTemperature() - self.__temperature;
75                  dTempDiff += self.__terrain.celluleDroite(self).getTemperature() - self.__temperature;
76                  dTempDiff *= self.__coefDiffusion;
77
78                  dTempPerte = self.__coefPerte * (self.__temperature - min);
79
80                  self.__temperature = dTempDiff + self.__temperature - dTempPerte;
81
82                  if(self.__temperature < min ):
83                          self.__temperature = min
84
85                  ## tranformation temperature vers couleur
86                  self.__couleur = self.__normaliser(self.__temperature)
87
```

```
88                  return self.__temperature;
89
90  ######################################################
91  class Terrain :
92          """ Le terrain est compose de cellules  """
93
94                  # ------------ constructeurs et destructeurs ------------
95          def __init__(self,nbcelluleX=0,nbcelluleY=0,tMIN=10,tMOY=100):
96                  print "creation d'un Terrain"
97                  self.__nbcelluleX = nbcelluleX
98                  self.__nbcelluleY = nbcelluleY
99                  self.__tMIN=tMIN
100                 self.__temperatureMoyenne=tMOY
101                 self.__cellules = []
102                 self.__soleil = None
103                 for i in range(self.__nbcelluleX):
104                         tab = []
105                         for j in range(self.__nbcelluleY):
106                                 tab.append(None)
107                         self.__cellules.append(tab)
108
109                 for i in range(self.__nbcelluleX):
110                         for j in range(self.__nbcelluleY):
111                                 self.__cellules[i][j] = Cellule()
112                                 self.__cellules[i][j].setPosition(i,j)
113                                 self.__cellules[i][j].setTerrain(self)
114
115         def __del__(self):
116                 print "destruction d'un Terrain"
117                 for i in range(self.__nbcelluleX):
118                         for j in range(self.__nbcelluleY):
119                                 del self.__cellules[i][j]
120
121                 # ------------ accesseurs et mutateurs ------------
122         def setSoleil(self,s):
123                 self.__soleil = s
124
125         def getTMIN(self):
126                 return self.__tMIN
127
128         def getTemperatureMoyenne(self):
129                 return self.__temperatureMoyenne
130
131
132         def getnbcelluleY(self):
```

```
133             return self.__nbcelluleY
134
135         def getnbcelluleX(self):
136             return self.__nbcelluleX
137
138             # ------------ methodes ------------
139         def update(self):
140             self.__soleil.update()
141             for i in range(self.__nbcelluleX):
142                 for j in range(self.__nbcelluleY):
143                     self.__cellules[i][j].update()
144
145         def getCellule(self, x, y):
146             if x < 0 :
147                 x = self.__nbcelluleX - 1
148             if x >= self.__nbcelluleX:
149                 x = 0
150             if y < 0:
151                 y = self.__nbcelluleY - 1
152             if y >= self.__nbcelluleY:
153                 y = 0
154             return self.__cellules[x][y]
155
156
157         def celluleDessous(self,cellule):
158             x = cellule.getX()
159             y = cellule.getY()
160             return self.getCellule(x, y + 1)
161
162         def celluleDessus(self,cellule):
163             x = cellule.getX()
164             y = cellule.getY()
165             return self.getCellule(x, y - 1)
166
167         def celluleGauche(self,cellule):
168             x = cellule.getX()
169             y = cellule.getY()
170             return self.getCellule(x-1, y)
171
172         def celluleDroite(self,cellule):
173             x = cellule.getX()
174             y = cellule.getY()
175             return self.getCellule(x+1, y)
176
177
```

```
178 ######################################################
179 class Soleil :
180         """ Le Soleil rechauffe la cellule """
181 
182                 # -------------- constructeurs et destructeurs --------------
183         def __init__(self,temperature=0):
184                 print "creation d'un Soleil"
185                 self.__x = 0
186                 self.__y = 0
187                 self.__temperature = temperature
188 
189         def __del__(self):
190                 print "destruction d'un Soleil"
191 
192                 # -------------- accesseurs et mutateurs --------------
193         def setPosition(self,x,y):
194                 self.__x = x
195                 self.__y = y
196 
197         def setTerrain(self,t):
198                 self.__terrain = t
199 
200         def getTerrain(self):
201                 return self.__terrain
202 
203         def getX(self):
204                 return self.__x
205 
206         def getY(self):
207                 return self.__y
208 
209                 # -------------- methodes --------------
210         def update(self):
211                 cellule = self.__terrain.getCellule(self.__x,self.__y);
212                 Tx = cellule.getTemperature();
213                 coefDx = cellule.getCoefDiffusion();
214                 cellule.setTemperature(Tx+coefDx*(self.__temperature-Tx))
215 
216 
217 ######################################################
218 class Simulateur :
219         """ le simulateur execute regulierement la methode update() du terrain  """
220 
221                 # -------------- constructeurs et destructeurs --------------
222         def __init__(self):
```

```
223                 print "creation d'un Simulateur"
224
225         def __del__(self):
226                 print "destruction d'un Simulateur"
227
228                 # ------------ methodes ------------
229         def run(self):
230                 s = Soleil(500)
231                 s.setPosition(50,50)
232                 t = Terrain(20,20,10,100)
233                 t.setSoleil(s)
234                 s.setTerrain(t)
235                 while(True):
236                         t.update()
237
238
239 ##################### Programme principal ########
240 # s = Simulateur()
241 # s.run()
242 ###################################################
243
244
245 ############## Partie graphique #################
246 from Tkinter import *
247
248 class TerrainGraphique(Frame,Terrain):
249         """ Classe representant le terrain en 2D """
250         def __init__(self, master,nbcelluleX=0,nbcelluleY=0,tMIN=10,tMOY=100):
251                 Frame.__init__(self, master)
252                 master.title("GAIA")
253                 self.pack()
254                 self.damier=Canvas(self,bg="black",height=nbcelluleY*10,width=nbcelluleX*10)
255                 self.damier.pack()
256
257                 Terrain.__init__(self,nbcelluleX,nbcelluleY,tMIN,tMOY)
258
259                 self.__rectangles = []
260                 for i in range(self.getnbcelluleX()):
261                         tab = []
262                         for j in range(self.getnbcelluleY()):
263                                 tab.append(None)
264                         self.__rectangles.append(tab)
265
266                 for i in range(self.getnbcelluleX()):
267                         for j in range(self.getnbcelluleY()):
```

```
268                                 self.__rectangles[i][j] = self.damier.create_rectangle(i*10,j*10,i*10+9,j*10+9)
269
270
271                 self.pal=['white','gray', 'green','yellow','gold','orange','pink','red']
272
273                 self.timer = 50
274
275         def setTimer(self,timer):
276                 self.timer = timer
277
278         def update(self):
279                 Terrain.update(self)
280
281                 for i in range(self.getnbcelluleX()):
282                         for j in range(self.getnbcelluleY()):
283                                 couleur = self.getCellule(i,j).getCouleur()
284                                 couleur = (len(self.pal)* couleur) / 255. -1
285                                 couleur = abs(int(couleur))
286                                 couleur = self.pal[couleur]
287                                 self.damier.itemconfigure(self.__rectangles[i][j],fill=couleur)
    ## changer les couleurs
288
289                 fen1.after(self.timer, self.update)
    ## rappel de update
290
291
292 ###################################################
293 flag = 0
294 def start_it():
295     "demarrage de l'animation"
296     global flag
297     flag = flag +1
298     if flag ==1:
299         t.update()
300
301 class SimulateurGraphique :
302         """ le simulateur execute la methode update() du terrain sur click """
303
304                 # ------------ constructeurs et destructeurs ------------
305         def __init__(self,timer=50):
306                 print "creation d'un Simulateur Graphique"
307                 self.timer = timer
308
309
310         def __del__(self):
```

```
311                 print "destruction d'un Simulateur Graphique"
312
313                 # ------------ methodes ------------
314         def run(self,s = Soleil(500)):
315                 s.setPosition(2,2)
316
317                 global t
318                 global fen1
319                 fen1 = Tk()
320
321                 t = TerrainGraphique(fen1,100,100,10,100)
322                 t.setTimer(self.timer)
323                 t.setSoleil(s)
324                 s.setTerrain(t)
325
326                 Button(fen1,text='Quitter',command=fen1.quit).pack(side=RIGHT)
327                 Button(fen1,text='Demarrer',command=start_it).pack(side=LEFT)
328                 fen1.mainloop()
329
330
331
332 ######################################################
333 class SoleilMouvant(Soleil) :
334         """ Le Soleil mouvant rechauffe la cellule """
335
336                 # ------------ constructeurs et destructeurs ------------
337         def __init__(self,temperature=0):
338                 Soleil.__init__(self,temperature)
339
340         def __del__(self):
341                 print "destruction d'un Soleil mouvant"
342
343
344                 # ------------ methodes ------------
345         def update(self):
346                 Soleil.update(self)
347                 t = self.getTerrain()
348                 sizeX = t.getnbcelluleX()
349                 sizeY = t.getnbcelluleY()
350
351                 x = self.getX() +1
352                 y =  ((math.cos((  (x - sizeX/2.) )/sizeX) * sizeY))*1.3 - (sizeY/1.5)
353                 y = int(y)
354                 #y = self.getY()
355                 if x > sizeX : x = 0
```

```
356                 if y > sizeY : y = 0
357                 self.setPosition(x,y)
358
359
360
361
362
363 ##################  Programme principal ########
364
365 timer = 5
366 s = SimulateurGraphique(timer)
367 soleil1 = SoleilMouvant(5000)
368 s.run(soleil1)
```

Solutions Labo

14 Rappels python (1 UC)

```
1  ######## Correction labo 1- POO #####
2
3  ###
4  # 1.1 boucle
5  ###
6  def moyenne():
7      """
8      Calcule la moyenne a partir des notes saisies par l'utilisateur.
9      La premiere saisie correspond aux nombre de notes, les autres correspondent aux notes.
10     Cette fonction utilise input.
11     """
12     nombre=input('nombre de notes: ')
13     print nombre
14     somme=0.0
15
16     for i in range(nombre):
17         note=input("note: ")
18         somme+=note
19
20     moyenne=somme/nombre
21     print moyenne
22
23
24 ###
25 # 1.2 Tableau
26 ###
27 def recherche(t, x):
28     """
29     recherche un entier x dans un tableau d'entiers t
30     et qui imprime tous les indices pour lesquels t[i]==x
31
32     >>> recherche([1,2,1,2,1,2],2)
33     1
34     3
35     5
36     """
37     assert type(t) is list
38     assert type(x) is int
```

```
39      for i in range (len(t)) :
40          if t[i]==x :
41              print i
42
43  ###
44  # Exo supplementaire Tableau
45  ###
46  def rech_dicho(t,x):
47      """
48      recherche dichotomique d'un entier x dans un tableau trie t
49      """
50      assert type(t) is list
51      assert type(x) is int
52
53      a=0
54      b=len(t)
55      while a < b:
56          m=(a+b)/2
57          if x==t[m]:
58              return m
59          elif x>t[m]:
60              a=m+1
61          else:
62              b=m
63      return -1
64
65
66
67  ###
68  # 1.3 Appel de fonctions
69  ###
70  def fonction1(a,b):
71          resultat = puissance(soustraction(a,b),2)
72          return resultat
73  # print fonction1(10,20)
74
75  ########################################
76  def fonction2(tab):
77          moy  = moyenne(tab)
78          for i in range(len(tab)):
79                  resultat = fonction1(tab[i],moy)
80          return resultat
81  # print fonction2([1,2,3,4])
82
83  ########################################
```

```
84  def ecartType(tableau):
85          resultat = racine(division(fonction2(tableau),len(tableau)))
86          return resultat
87  # print ecartType([1,2,3,4])
88
89  ###
90  # 1.4 Recursivite
91  ###
92  def repete(n,s):
93      """
94      fonction recursive renvoyant la chaine s repetee n fois
95      """
96
97      if n>1:
98          return s+repete(n-1,s)
99
100     else:                ## condition d'arret recursivite
101         return s
102
103
104
105 def pyramide(n,s):
106     """
107     fonction recursive imprimant la chaine s une fois sur la premiere ligne, 2 sur
108     la deuxieme, et ainsi de suite jusqu'a la ligne n
109     """
110
111     if n>0:
112         pyramide(n-1,s)
113         print repete(n,s)
114
115 ###
116 ## Exo supplementaire Recursivite
117 ###
118
119 def valproche(t,x,p,i=0):
120     """
121     determine par recursivite la valeur p la plus proche de x dans le tableau t
122     attention p doit avoir une valeur elevee
123     """
124
125     if len(t)==i:               ## condition d'arret recursivite
126         return p
127     else:
128         if abs(t[i]-x)<abs(p-x):
```

```
129                 p=t[i]
130         i+=1
131         return valproche(t,x,p,i)
```

15 Concepts fondateurs (1 UC)

```
#!/usr/bin/python
# -*- coding: latin-1 -*-

#####################################
####### Correction Labo 2 - POO ####
####### Cédric Buche ###############
#####################################


######### Definition des classes ########
class Complex:
        """La classe Complex du labo 2"""       # Documentation de la class
        def __init__(self,xinit=0,yinit=0):     # Constructeur
                self.x = xinit;                 # attribut x
                self.y = yinit;                 # attribut y

        def clone(self,c1):                     # Methode de recopie d'un Complex
                self.x = c1.x
                self.y = c1.y

        def showComplex(self):                  # Methode qui affiche les valeurs
                print self.x                    # des attributs de a classe
                print self.y



######### Definition des fonctions (en dehors de la classe) ############
def affiche_complex(c1):
        print c1.x
        print c1.y


def compareComplex(c1,c2):
        return c1.x == c2.x and c1.y == c2.y



######### Programme Principal ############
c1 = Complex()                          # c1 instance de la classe Complex
c2 = Complex(1,5)                       # c2 instance de la classe Complex

c3 = Complex()                          # c3 instance de la classe Complex
```

```
43 c3.clone(c2)                          # appel de la methode clone de l'objet c3
44
45 c1.x = 3                              # affecter 3 a  l attribut x de c1
46 c1.y = 4                              # affecter 4 a  l attribut x de c1
47
48 affiche_complex(c1)                   # appel de fonction
49 c1.showComplex()                      # appel de methode
50
51 print compareComplex(c1,c2)
52
53
54
55 ########## RESULTAT  du Programme Principal ########
56 ## 3
57 ## 4
58 ## 3
59 ## 4
60 ## False
```

16 Encapsulation (1 UC)

```
#!/usr/bin/python
# -*- coding: ISO-8859-15 -*-

##########################################
#  Compte Bancaire                       #
#  Auteur : Lionel                       #
#  Date de creation : 14.10.07           #
#  Revision : C.Buche : 16.10.07         #
##########################################



class CompteBancaire :
        """ La classe CompteBancaire permet d'instancier des objet tels que des comptes."""

        #---------- Constructeur destructeur ----------#

        def __init__(self,nom_int='Dupont',solde_int=1000):
                self.__nom      = nom_int
                self.__solde    = solde_int


        #---------- Acces aux attributs ----------#

                #accesseurs
        def getSolde(self):
                return self.__solde
        def getNom(self):
                return self.__nom

                #mutateurs
        def setNom(self,name):
                self.__nom = name

        def __setSolde(self,valeur):            # Atention mutateur Prive
                self.__solde = valeur           # on ne peut pas modifier son solde de l'exterieur


        #---------- Methodes publiques : Operations ----------#

        def depot(self, somme):
                if somme > 0 and somme < 5000:          # on considere qu'il n'est pas possible
```

```
43                          value = self.getSolde()+somme   # de deposer plus de 5000 Euros en une fois
44                          self.__setSolde(value)
45
46          def retrait(self, somme):
47                  if somme < 0 and somme > -5000:         # on considere qu'il n'est pas possible
48                          value = self.getSolde()-somme   # de retirer plus de 5000 Euros en une fois
49                          self.__setSolde(value)
50
51
52          def affiche(self):
53                  print 'Le solde du compte bancaire de ',self.getNom(), ' est de ',self.getSolde(), ' euros.'
54
55
56
57  #------ Programme principal -------------#
58
59  c1 = CompteBancaire('Duchmol', 800)
60  c1.depot(350)
61  c1.retrait(200)
62  c1.affiche()
63
64
65  c2 = CompteBancaire()
66  c2.depot(25)
67  c2.affiche()
68
69  c3 = CompteBancaire()
70  c3.affiche()
71  c3.depot(100000)                # ne fait rien
72  c3.affiche()
73  c3.__setSolde(100000)           # ERREUR
```

17 Collaboration et héritage (1 UC)

```
#!/usr/bin/python
# -*- coding: latin-1 -*-

#####################################
####### Correction Labo 3 - POO ####
####### Cedric Buche ###############
#####################################


######### Definition des classes ########
class Point:
        """La classe Point du labo 3"""          # Documentation de la class
        def __init__(    self,                   # Constructeur
                         xinit=0,
                         yinit=0):
                self.__x = xinit
                self.__y = yinit

        def getX(self): return self.__x
        def getY(self): return self.__y

        def __del__(self):                       # Destructeur
                print "mort d'un point"

#-------------------------
class Rectangle:
        """La classe Rectangle du labo 3"""              # Documentation de la class
        def __init__(    self,                           # Constructeur
                         largeurInit=0,
                         hauteurInit=0,
                         coinSupGaucheInit=Point() ):
                self.__largeur = largeurInit
                self.__hauteur = hauteurInit
                self.__coinSupGauche = coinSupGaucheInit


        def getLargeur(self):           return self.__largeur
        def getHauteur(self):           return self.__hauteur
        def getCoinSupGauche(self):     return self.__coinSupGauche

        def setLargeur(self,l): self.__largeur = l
        def setHauteur(self,h): self.__hauteur = h
```

```
43
44          def __del__(self):                                  # Destructeur
45                  print "mort d'un rectangle"
46                  del self.__coinSupGauche
47
48
49
50
51  ######### Definition des fonctions (en dehors de la classe) ############
52  def trouverCentre(rect1):
53          xCentre = rect1.getCoinSupGauche().getX() + rect1.getLargeur()/2
54          yCentre = rect1.getCoinSupGauche().getY() + rect1.getHauteur()/2
55          pointTmp = Point(xCentre,yCentre);
56          return pointTmp
57
58
59
60  ######### Programme Principal ############
61  p1 = Point()                                # p1 instance de la classe Point
62  p2 = Point(1,5)                             # p2 instance de la classe Point
63  print "attention je vais tuer un point !!!"
64  del p1
65  #-----------------------------------------
66  r1 = Rectangle()
67  r2 = Rectangle(10,20,p2)
68  #-----------------------------------------
69  p3 = trouverCentre(r2)
70  print p3.x
71  print p3.y
72  #-----------------------------------------
73  r2.setHauteur(r2.getHauteur() +20)
74  r2.setLargeur(r2.getLargeur() -5)
75  print r2.getHauteur()
76  print r2.getLargeur()
77
78
79  print "attention je vais tuer un rectangle !!!"
80  del r2
81
82  print "FIN DU PROGRAMME PRINCIPAL"
83
84  ########## RESULTAT  du Programme Principal #########
85  ## attention je vais tuer un point !!!
86  ## mort d'un point
87  ## 6
```

```
88 ## 15
89 ## 40
90 ## 5
91 ## attention je vais tuer un rectangle !!!
92 ## mort d'un rectangle
93 ## FIN DU PROGRAMME PRINCIPAL
94 ## mort d'un point
95 ## mort d'un point
96 ## mort d'un rectangle
97 ## mort d'un point
```

```
#######################################
######## Correction Labo 5 - POO ######
######## Cedric Buche ##################
#######################################
class Mammifere:
        def __init__(self,nom="",age="",nbDent=0):
                self._nom = nom
                self._age = age
                self._nbDent = nbDent
        def affiche(self):
                print "Je m'appelle " + self._nom
                print "J'ai " + str(self._age) +" ans"
#-------------------------------------------------------
class Primate(Mammifere):
        def __init__(self,nom="",age="",nbDent=0,couleur="blanc"):
                Mammifere.__init__(self,nom,age,nbDent)
                self._color=couleur
        def crier(self):
                print "je crie"
        def ditBonjour(self):
                print "Coucou"
#-------------------------------------------------------
class Rongeur(Mammifere):
        def __init__(self,nom="",age="",nbDent=0,typeDent="canines"):
                Mammifere.__init__(self,nom,age,nbDent)
                self._dentType = typeDent
        def grignoter(self):
                print "je grignote"
class Carnivore(Mammifere):
        def __init__(self,nom="",age="",nbDent=0):
                Mammifere.__init__(self,nom,age,nbDent)
        def mangerViande(self):
                print "je mange de la viande rouge"
#-------------------------------------------------------
class Bellette(Carnivore):
        def __init__(self,nom="",age="",nbDent=0):
                Mammifere.__init__(self,nom,age,nbDent)
#-------------------------------------------------------
class Loup(Carnivore):
        def __init__(self,nom="",age="",nbDent=0,dominateur=False):
                Mammifere.__init__(self,nom,age,nbDent)
                self._domin = dominateur
        def chasser(self):
                print "je chasse ma proie"
```

```
45 #-------------------------------------------------------------
46 class Chien(Carnivore):
47         def __init__(self,nom="",age="",nbDent=0,race="",dangerous=False):
48                 Mammifere.__init__(self,nom,age,nbDent)
49                 self._race=race
50                 self._dangereux=dangerous
51 #-------------------------------------------------------------
52 class Herbivore(Primate):
53         def __init__(self,nom="",age="",nbDent=0):
54                 Mammifere.__init__(self,nom,age,nbDent)
55         def mangerBio(self):
56                 print "je mange des legumes moi messieurs"
57         def ditBonjour(self):
58                 print "Bonjour, j'aime la nature"
59 #-------------------------------------------------------------
60 class Omnivore(Carnivore,Herbivore):                    ## tres dangereux
61         def __init__(self,nom="",age="",nbDent=0):
62                 Carnivore.__init__(self,nom,age,nbDent)
63                 Herbivore.__init__(self,nom,age,nbDent)
64 ###################### Programme principal ###################################
65 m1 =  Mammifere("titi",25,10)
66 m2 =  Mammifere(nom="tata")
67
68 p1 = Primate("titi",25,10,"vert")
69 p1.crier()
70 p1.affiche()
71
72
73 o1 = Omnivore("tutu",25,10)
74 o1.mangerBio()
75 o1.ditBonjour()
```

18 Pour aller plus loin (1 UC)

```
#!/usr/bin/python
# -*- coding: latin-1 -*-

#####################################
####### Correction Labo 6 - POO ####
####### CÃ©dric Buche ###############
#####################################
class TableauTrieAbstrait:
    def __init__(self):
        self.__tableautrie=[]
    #-------------------------------------------------------
    def plusGrand(self,x,y):              #EST CE QUE X EST SUP  Y ?
        raise "methode abstraite"
    #-------------------------------------------------------
    def inserer(self,element):
        position_trouver=False
        for i in range(len(self.__tableautrie)):
            if(self.plusGrand(element,self.__tableautrie[i])==False):
                self.__tableautrie.insert(i,element)
                position_trouver=True
                break

        if(position_trouver==False):
            self.__tableautrie.append(element)
    #-------------------------------------------------------
    def supprimer(self,elements):
        i = len(self.__tableautrie)-1
        while i>=0 :
                if(elements==self.__tableautrie[i]):
                        del self.__tableautrie[i]
                i=i-1
    #-------------------------------------------------------
    def afficher(self):
        print self.__tableautrie

#####################################
class TableauTrieEntiers(TableauTrieAbstrait):
    def __init__(self):
        TableauTrieAbstrait.__init__(self)

    def plusGrand(self,x,y):
        return x>y
```

```
43
44 T=TableauTrieEntiers()
45 T.inserer(5)
46 T.inserer(4)
47 T.inserer(5)
48 T.inserer(10)
49 T.inserer(2)
50 T.inserer(5)
51 T.afficher()
52 T.supprimer(5)
53 T.afficher()
54
55 ##############################################################################
56 class TableauTrieChaines(TableauTrieAbstrait):
57     def __init__(self):
58         TableauTrieAbstrait.__init__(self)
59
60     def plusGrand(self,x,y):
61         return len(x)>len(y)
62
63 T2=TableauTrieChaines()
64 T2.inserer("ferz")
65 T2.inserer("greqq")
66 T2.inserer("e")
67 T2.inserer("gfreqfgq")
68 T2.inserer("ee")
69 T2.inserer("geqrfqefreqf")
70 T2.afficher()
71 T2.supprimer("ee")
72 T2.afficher()
```

Références

[Booch, 1992] Booch, G. (1992). *Conception orientée objets et applications*. Addison-Wesley.

[Brooks, 1987] Brooks, F. (1987). No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4) :10–19.

[Peter, 1986] Peter, L. (1986). *The peter pyramid*. New York, NY :William Morrow.