Partie III

La programmation orientée objets



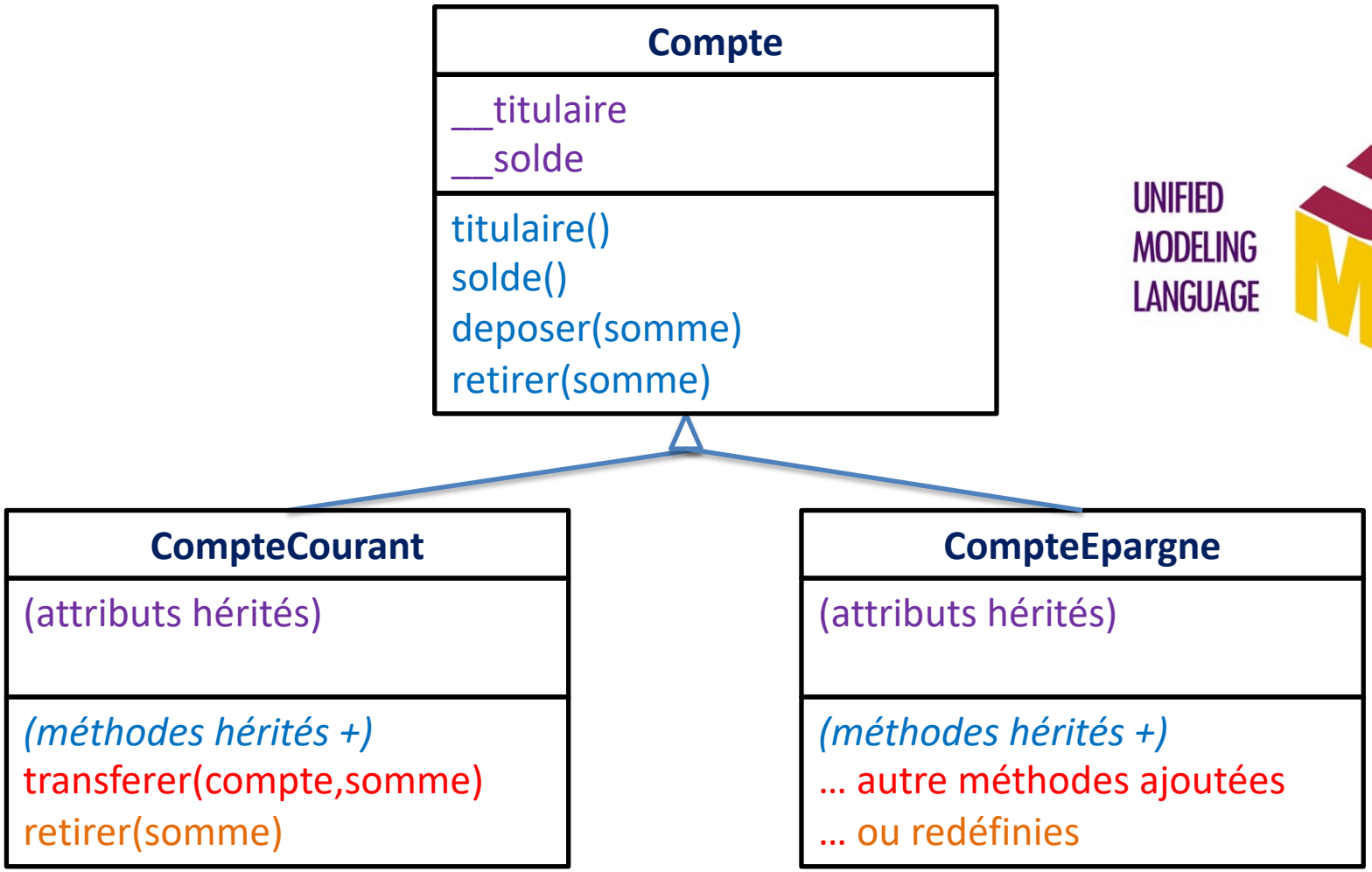
Informatique 1
Introduction à la programmation

Mission 9 : RESTRUCTURATION

Héritage, variables d'instances privées, variables de classes privées, méthodes de classe, les tests unitaires

Kim Mens

Héritage



UNIFIED MODELING LANGUAGE™

type()

```
class Compte :
    …

class CompteCourant :
    …

class CompteEpargne :
    …

compte_kim = CompteCourant()
compte_tom = CompteEpargne()
```

```
>>> type(compte_kim)
<class '__main__.CompteCourant'>

type(compte_kim) == CompteCourant
#True
type(compte_tom) == CompteEpargne
#True
```

type() retourne la classe de l'objet passé en paramètre

```
type(compte_kim) == Compte   #False
type(compte_tom) == Compte   #False
```

isinstance()

isinstance() vérifie si un objet (premier paramètre) est une instance d'une classe (deuxième paramètre) *ou d'une classe fille de cette classe*

```
class Compte :
    …

class CompteCourant :
    …

class CompteEpargne :
    …

compte_kim = CompteCourant()
compte_tom = CompteEpargne()
```

```
isinstance(compte_kim,CompteCourant)
#True
isinstance(compte_tom,CompteEpargne)
#True


isinstance(compte_kim,Compte) #False
isinstance(compte_tom,Compte) #False
```

type() vs. isinstance ()

même comportement si pas d'heritage

```
class Compte :
    …

class CompteCourant :
    …

class CompteEpargne :
    …

compte_kim = CompteCourant()
compte_tom = CompteEpargne()
```

```
type(compte_kim) == CompteCourant
#True
type(compte_tom) == CompteEpargne
#True

isinstance(compte_kim,CompteCourant)
#True
isinstance(compte_tom,CompteEpargne)
#True

type(compte_kim) == Compte)   #False
type(compte_tom) == Compte)   #False
isinstance(compte_kim,Compte) #False
isinstance(compte_tom,Compte) #False
```

type() vs. isinstance ()

mais … **isinstance** tient compte de l'héritage

```
class Compte :
    …

class CompteCourant(Compte) :
    …

class CompteEpargne(Compte) :
    …

compte_kim = CompteCourant()
compte_tom = CompteEpargne()
```

isinstance() vérifie si un objet (premier paramètre) est une instance d'une classe (deuxième paramètre) ***ou d'une classe fille de cette classe***

```
type(compte_kim) == Compte)    #False
type(compte_tom) == Compte)    #False
isinstance(compte_kim,Compte) #True
isinstance(compte_tom,Compte) #True
```

“if” vs. polymorphisme

à éviter: des “if” sur les types des classes filles

```
class Compte :
    def __str__(self) :
        if type(self) == CompteCourant :
            s = "compte courant"
        elif type(self) == CompteEpargne :
            s = "compte d'épargne"
        else :
            s = "compte"
        return s

class CompteCourant(Compte) :
    …
class CompteEpargne(Compte) :
    …
```



```
compte      = Compte()
compte_kim = CompteCourant()
compte_tom = CompteEpargne()
print(compte)
# compte
print(compte_kim)
# compte courant
print(compte_tom)
# compte d'épargne
```

if vs. “polymorphisme”

mieux: éviter des if grâce au “polymorphisme”

```
class Compte :
    def __str__(self) :
        return "compte"

class CompteCourant(Compte) :
    def __str__(self) :
        return "compte courant"

class CompteEpargne(Compte) :
    def __str__(self) :
        return "compte d'épargne"
```



```
compte     = Compte()
compte_kim = CompteCourant()
compte_tom = CompteEpargne()
print(compte)
# compte
print(compte_kim)
# compte courant
print(compte_tom)
# compte d'épargne
```

if vs. “polymorphisme”

encore mieux: avec utilisation de super()

```
class Compte :
    def __str__(self) :
        return "compte"

class CompteCourant(Compte) :
    def __str__(self) :
        return super().__str__() \
               + " courant"

class CompteEpargne(Compte) :
    def __str__(self) :
        return super().__str__() \
               + " d'épargne"
```



```
compte     = Compte()
compte_kim = CompteCourant()
compte_tom = CompteEpargne()
print(compte)
# compte
print(compte_kim)
# compte courant
print(compte_tom)
# compte d'épargne
```

Variable de classe privée

```
class Compte :
    __taux_interet = 0.02
   …
```

variable de classe privée



```
>>> Compte.__taux_interet
AttributeError: 'Compte' class
has no attribute 'taux_interet'

>>> a = Compte("Kim")
>>> a.__taux_interet
AttributeError: 'Compte' object
has no attribute '__taux_interet'
```

Comment y accéder ?

Méthode de classe

Nouveau

```
class Compte :
    __taux_interet = 0.02

    @classmethod
    def taux_interet(cls):
        return cls.__taux_interet

    @classmethod
    def set_taux_interet(cls,nouveau_taux):
        cls.__taux_interet = nouveau_taux
  …
```

Comment y accéder ? …

Par une méthode de classe

Reçoit la classe comme paramètre implicit

```
Compte.taux_interet()
# 0.02
Compte.set_taux_interet(0.04)
Compte.taux_interet()
# 0.04
```

Envoyé à la classe !

Méthode mutateur

Méthode accesseur

Qu’est-ce qu’un **test unitaire** ?

- ***Comment tester son code?***
- Le principe :
  - Chaque test unitaire vérifie automatiquement qu’une petite partie du programme (fonction, méthode) se comporte comme prévu.
- Idée clé :
  1. On appelle la function/méthode avec des valeurs connues.
  2. On compare le résultat attendu et le résultat obtenu.
  3. Si ça ne correspond pas → le test échoue.
- Pourquoi faire ?
  - Détecter les erreurs tôt.
  - Éviter de casser du code déjà correct.
  - Avoir confiance dans la qualité du programme.



Tests unitaires avec `assert`

- Même si des frameworks spécialisés pour les tests unitaires existent :
  - untittest
  - pytest
- Pour faire simple nous utiliserons que des simples **`assert`**
- Un test unitaire par “unité” dans votre programme (p.ex. une fonction)
- Chaque test unitaire est implémenté par un ensemble de **`assert`**
- Pour vérifier le comportement correct de la fonction dans plusieurs cas de figure

Exemple simple avec `assert`

- Exemple d'un test unitaire pour une fonction addition :

```
def ajout(a, b):
    return a + b
```

- Test unitaire = un ensemble d'assertions pour cette fonction :

```
assert ajout(2,3)==5, "Échec: 2+3  doit donner 5"
assert ajout(-1,1)==0,"Échec: -1+1 doit donner 0"
assert ajout(0,0)==0, "Échec: 0+0  doit donner 0"
print("Tous les tests passés avec succès!")
```

- Si tous les `assert` sont vrais → le programme continue.
- Si un test échoue → Python affiche le message d'erreur correspondant.
- `assert` signifie : Je m'attends à ce que cette condition soit vraie.

Comment raisonner comme un testeur?

- Tester, c’est prévoir :
  - Les cas normaux → usage attendu.
  - Les cas limites → valeurs extrêmes, 0, négatifs, etc.
  - Les cas erronés → entrées invalides si pertinent.
- Bon réflexe :
  - Relancer les tests après chaque modification du code.
  - Les tests servent de filet de sécurité contre les régressions.



Tests unitaires en Programmation Orientée Objet

Quand on travaille avec des classes, on teste les méthodes ou attributs d'un objet.

Compte.py

```
class Compte :
  def __init__(self, titulaire):
    self.__titulaire = titulaire
    self.__solde = 0
  def titulaire(self):
    return self.__titulaire
  def solde(self):
    return self.__solde
  def deposer(self, somme):
    self.__solde += somme
    return self.solde()
  def retirer(self, somme):
    if self.solde() >= somme :
      self.__solde -= somme
      return self.solde()
    else :
      return "Solde insuffisant"
```

Compte_test.py

```
# Tests unitaires
c = Compte("Kim")
try:
  c.__solde
except:
    None
```

test1

```
assert c.titulaire() == "Kim"
```

test2

```
assert c.solde() == 0
```

test3

```
c.deposer(100)
assert c.solde() == 100
```

test4

```
c.retirer(50)
assert c.solde() == 50
```

test5

```
resultat = c.retirer(100)
assert resultat == "Solde
insuffisant"
```

test6

Avantages dans le contexte objet

- Les tests unitaires permettent de :
  - Vérifier chaque méthode séparément.
  - Garantir le bon état interne de l'objet après une action.
  - Faciliter la maintenance et les évolutions du code.
- Bonne pratique :
  - Chaque classe devrait avoir un petit ensemble de tests (au moins une par méthode) qui prouvent qu'elle fonctionne isolément.
- Remarque :
  - des frameworks spécialisés pour les tests unitaires existent : untittest, pytest, …