Université de Montpellier / Faculté Des Sciences

Modélisation et Programmation par Objets (2e partie)
2017-2018 - HLIN 505

Enseignants
Marianne Huchard, Jessie Carbonnel, Clémentine Nebut, Abdelhak-Djamel Seriai

Ce document est un recueil de notes de cours. Il peut développer des aspects vus en cours plus succinctement ou inversement vous aurez en cours plus de détails sur certains points. Si vous y relevez des erreurs, n'hésitez pas à nous les signaler afin de l'améliorer.

1 Rappels sur les classes en UML et Java

Ce cours fait suite à un cours d'initiation à la modélisation et à la programmation par objets qui en a déjà détaillé les grands principes. Pour en dresser à nouveau un tableau à grands traits, rappelons que la modélisation et la programmation par objets cherchent principalement à introduire dans les représentations informatiques (comme des modèles conceptuels ou du code source) une représentation des concepts du domaine sur lequel porte un logiciel, ou de concepts techniques de l'informatique. L'objectif est de monter en abstraction et d'avoir des représentations relativement stables par rapport à l'évolution des besoins et des technologies.

Ces concepts apparaissent sous deux aspects : un aspect *structurel* qui décrit la forme des objets et un aspect *dynamique* qui décrit leurs comportements. Dans le cadre plus spécifique de la programmation, plutôt que de se trouver face à des appels de fonctions, on va programmer *un monde d'objets s'envoyant des messages*. La notion de *spécialisation/généralisation ou d'héritage* est également le propre des approches à objets, elle sera développée dans le prochain chapitre.

Dans cette section, nous rappelons les deux notions principales des approches à classes, que sont les classes et les instances et leur notation en UML (modélisation) et en Java (programmation) :

— Une classe décrit les aspects structurels et comportementaux d'un ensemble d'instances.
— Une instance se conforme à une classe.

1.1 Classes en UML (rappels)

La figure 1 présente quelques éléments de la modélisation partielle supposée d'un logiciel bancaire. Ce modèle pourrait être représenté avec une association entre les deux classes, mais ce sera revu dans de prochains exemples. Deux concepts (classes UML, présentées dans la notation en trois compartiments : noms / attributs / opérations) y apparaissent :

— les comptes bancaires, décrits par :
  — un numéro, spécifique à chaque compte, qui ne change plus après qu'il ait été attribué au compte

  - un titulaire, spécifique à chaque compte, qui est de type client (les classes sont des types dans notre univers)
  - un solde, spécifique à chaque compte, qui est un nombre réel
  - un plafond, qui est le même pour tous les comptes. Sa valeur, modifiable, étant partagée par tous les comptes, on cherche à ne la stocker qu'une fois.
  - des opérations de création d'instance (constructeurs), précédées du mot-clef UML <<`create`>>
  - des opérations d'accès aux attributs (accesseurs), dont le nom respecte par convention un format particulier ; suivant la sémantique de la classe, seuls certains accesseurs sont fournis
  - d'autres opérations variées, ici la méthode pour créditer un compte
  - pour respecter les convention de Java, une opération `toString` qui retourne une chaîne de caractères décrivant une instance
- les clients, décrits par :
  - un nom, spécifique à chaque client
  - un portefeuille, spécifique à chaque client, qui est une collection de comptes bancaires
  - par souci de simplicité, les opérations de cette classe ne sont pas présentées

Le bas de la figure 1 présente l'attribut partagé `plafond` avec sa valeur, deux instances de `CompteBancaire` et une instance de `Client`. Dans les instances, on observe une valuation des attributs spécifiques (non static).

1.2 Classes en Java (rappels)

Le code Java partiel correspondant est présenté dans les tables 1 et 2. C'est l'occasion de rappeler certaines règles classiques pour une bonne programmation en Java.

TABLE 1 – Class décrivant partiellement des clients

public class Client { private String nom ; private ArrayList<CompteBancaire> portefeuille ; }	 Attributs Collection de comptes

2 Les classes internes (*nested classes)*

Les classes internes constituent l'un des modes de structuration du code. Lors du précédent module, nous avons utilisé les paquetages (*package*) qui jouent également ce rôle en permettant le classement thématique des classes. Les paquetages peuvent s'organiser dans une hiérarchie d'inclusion des paquetages les uns dans les autres, ce qui régit également le nom des classes. Le nom complet d'une classe est en effet constitué du nom du (des) paquetage(s) la contenant. Ces inclusions s'accompagnent des mécanismes de visibilité que nous reverrons au fur et à mesure du cours.

En Java, il existe deux catégories principales de classes internes, dont l'une se subdivise en trois à son tour.

- les classes imbriquées statiques (*static nested classes*)

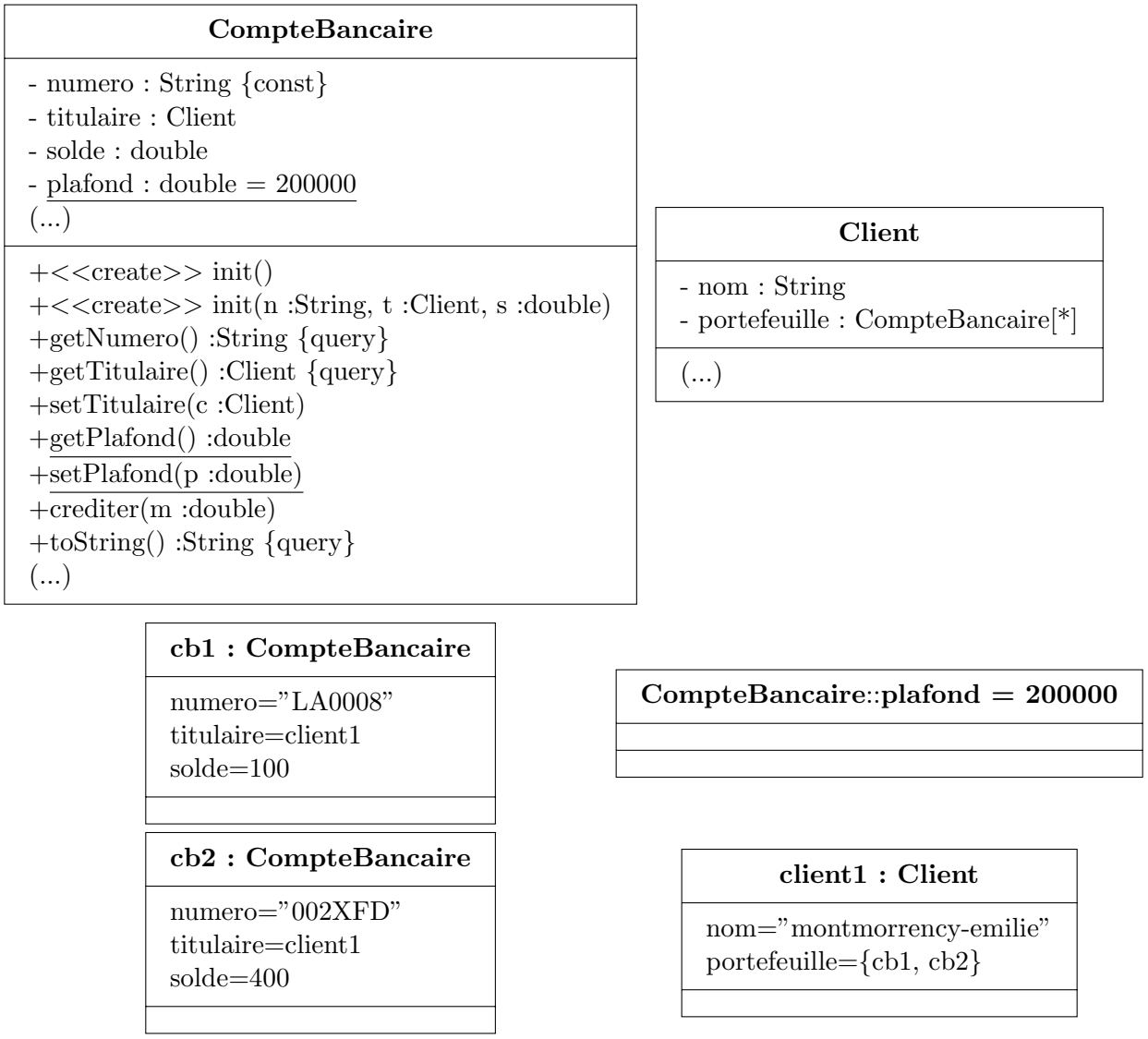

FIGURE 1 – Deux classes `CompteBancaire` et `Client` en UML et quelques instances (ignorer les compartiments vides des boîtes)

— les classes imbriquées non statiques
  — classes internes (*inner classes*)
  — classes locales (*local classes*)
  — classes anonymes (*anonymous classes*)

Parmi les usages principaux de ces classes, on peut noter qu'elles servent à :

— grouper des classes qui sont toujours utilisées ensemble, alternativement aux paquetages et souvent en identifiant une classe principale qui contient les autres.
— augmenter l'encapsulation
  — suivant la visibilité choisie, la classe imbriquée peut être cachée et seulement accessible par la classe englobante.
— faciliter la communication
  — sous certaines conditions qui seront précisées, la classe imbriquée peut accéder à la partie privée de la classe englobante et inversement.

2.1 Les classes imbriquées statiques (*static nested classes*)

Les classes imbriquées statiques respectent les principes généraux suivants :

— Elles sont utilisées principalement comme un moyen de ranger les classes les unes dans les autres pour des raisons thématiques.
— Il n'y a pas de lien particulier entre une instance de la classe imbriquée et une instance de la classe englobante.
— à la visibilité près, la classe imbriquée pourrait être du même niveau que la classe englobante.

La table 3 détaille des extraits d'un exemple de classe statique (`Cellule`) imbriquée dans une autre classe, dite englobante, ici la classe `Liste`.

On peut y observer les phénomènes suivants :

— La classe englobante peut accéder aux attributs de la classe imbriquée, même s'ils sont privés.
— La classe imbriquée peut accéder aux attributs **statiques** de la classe englobante, même s'ils sont privés. Elle ne peut accéder aux autres attributs, quelle que soit leur visibilité, que une instance créée explicitement.
— Pour créer une instance de la classe imbriquée, on utilise le plus souvent le nom de la classe englobante suivi du '.' suivi du nom de la classe imbriquée.

Dans l'API Java, on trouve des exemples de classes imbriquées statiques dans la description des collections (paquetage `java.util`), comme les classes décrivant les entrées d'un dictionnaire associatif (`map`), incluses dans la classe abstraite décrivant les `map`s.

```
public abstract class AbstractMap<K,V> {
    public static class AbstractMap.SimpleEntry<K,V> {
    // .....
    }
    public static class AbstractMap.SimpleImmutableEntry<K,V> {
    // .....
    }
...
}
```

2.2 Les classes internes (*inner classes*)

Une classe interne s'utilise lorsque chacune de ses instances est associée à une instance de la classe englobante. Ces instances associées partagent une certaine "intimité". Une instance de la classe interne peut accéder aux attributs et aux méthodes de l'instance associée de la classe englobante. Cet accès peut se faire également en sens inverse.

Par conséquent, une stratégie usuelle de création des instances consistera à créer tout d'abord une instance de la classe englobante, puis la (ou les) instance(s) de la classe interne. En programmant, on doit veiller à la logique des créations, en particulier à ce qu'une instance englobante référence sa (ses) propre(s) instance(s) interne(s) et ne référence pas une instance interne à une autre instance englobante.

Une classe interne ne peut contenir :

— d'attribut static (sauf s'il est en plus final),

— de méthode statique.

La table 4 présente un exemple où une classe `Personne` contient une classe interne `adresse`. Une instance d'adresse est naturellement associée à une seule instance de personne (implicite) dans cette modélisation. Une personne pourrait avoir plusieurs adresses associées dans un développement de l'exemple non présenté ici.

2.3 Les classes locales (*local classes*)

Une classe locale est une classe interne créée dans un bloc (méthode, corps d'une itération, etc.) et dont la portée sera restreinte à ce bloc. Elle permet de structurer ce bloc en lui ajoutant des déclarations de types qui ne sont pas utiles en d'autres endroits du programme.

Elle ne peut comporter de visibilité, elle peut seulement être abstraite ou final (non spécialisable).

Dans l'exemple présenté (table 5), dans une classe `Autorisation` une méthode de construction d'un numéro de téléphone `saisieNumTel` déclare une classe interne `NumTel` décrivant le format d'un numéro et une méthode de saisie.

Le bloc englobant a accès aux attributs et méthodes de la classe locale, même s'ils sont privés.

La classe locale a accès aux paramètres et aux variables du bloc, à condition qu'ils soient spécifiés `final` ou non modifiés dans les faits, et comme toute classe interne, à son instance englobante implicite.

2.4 Les classes anonymes (*anonymous classes*)

Les classes anonymes sont des variantes des classes locales, qui ne possèdent pas de nom. Ce sont des expressions. Elles se construisent sur la base d'une classe support (ou d'une interface) préalablement déclarée, et qui peut être abstraite.

Dans l'exemple précédent, si on ne désire pas donner un nom à la classe `NumTel`, on peut la déclarer de manière anonyme (table 6). Par souci de clarté du code, on commence par définir une classe abstraite `Saisissable` qui représente les objets disposant d'une méthode de saisie sur un Scanner. Puis on retrouve la classe `Autorisation`, dont la méthode de saisie contiendra l'expression de définition et d'instanciation simultanée de la classe interne anonyme.

Les classes internes anonymes sont très utilisées lorsque l'on fait de la programmation graphique événementielle, ce qui sera présenté dans un prochain chapitre.

TABLE 2 – Classe décrivant partiellement des comptes bancaires

```
public class CompteBancaire {

  private final String numero;
  private Client titulaire;
  private double solde;
  private static double plafond = 200000;
    // un constructeur porte le nom de sa classe
  public CompteBancaire()
      { this.numero = "compte par defaut" ; }
  public CompteBancaire(String num, Client tit, double solde) {
    this.numero = num;
    this.titulaire = tit;
    this.solde = solde;
  }

  public Client getTitulaire() { return titulaire; }
  public void setTitulaire(Client titulaire)
  { this.titulaire = titulaire; }
  public double getSolde() {return solde; }

  public static double getPlafond() { return plafond; }
  public static void setPlafond(double plafond) {
    CompteBancaire.plafond = plafond;
  }

  public String getNumero() { return numero; }

  public void crediter(double montant){
    if (montant >=0) this.solde += montant;
    else System.out.println("Erreur");
  }
  public String toString()
  {
    return this.numero+" "+this.titulaire
        +" "+this.solde;
  }
  public static void main(String[] a){
    Client cl1 = new Client();
    CompteBancaire cb1 =
        new CompteBancaire("LA0008",cl1,100);
    CompteBancaire cb2 =
        = new CompteBancaire("002XFD",cl1,400);
    System.out.println(cb1);
  }
}
```

Entête de la classe
Attributs privés
numero : Constant, spécifique
titulaire : Spécifique
solde : Spécifique
plafond : Partagé
Constructeurs publics
Toujours un constructeur
sans paramètres,
et un ou plusieurs constructeurs avec
des paramètres suivant la sémantique
de construction.
ici, on crée un compte après son client
`this` désigne l'instance receveur
Accesseurs suivant sémantique
Nom codifié

Pas de setSolde

Accesseurs static
quand attribut static

Pas de setNumero
Autres méthodes
on modifie le solde
par des méthodes
qui effectuent des contrôles

Méthode conventionnelle `toString`
Transformation d'une instance
en chaîne de caractères

Programme principal
avec des créations d'instances

TABLE 3 – La classe Cellule est une classe imbriquée statique de la classe Liste

public class Liste {	`Liste` est la classe englobante
private Cellule premier ;	Elle a ses propres attributs, `premier`
private static int nbListes ;	et `nbListes` (ce dernier est static)
(...)	
private static **class Cellule {**	La class imbriquée `Cellule` est statique
	Elle peut avoir différentes visibilités, ici `private`
private int valeur ;	`Cellule` a ses propres attributs `valeur`
private Cellule suivante ;	et `suivante`
(...)	
public String essai(){	`Cellule` a sa propre méthode `toString`
Liste l=new Liste() ;	On peut accéder aux membres static
return ""+nbListes	de la classe englobante (comme `nbListes`).
~~//+premier ;~~	Mais pas à `premier` qui n'est pas static
+" "+l.premier ;	sauf sur une instance de `Liste`
} // fin classe Cellule	créée explicitement
public Liste() {}	Constructeur de `Liste`
public void ajouteTete(int v){	Méthode d'ajout dans une `Liste`
Liste.Cellule c = new Liste.Cellule() ;	Création d'instance de la classe imbriquée
c.valeur=v ;	La classe englobante peut accéder
c.suivante=premier ;	aux attributs privés de la classe imbriquée
premier=c ;	
}	
public static void main(String[] a){	
Liste l = new Liste() ;	
}	
} // fin classe Liste	

TABLE 4 – La classe Adresse est une classe interne de la classe Personne

public class Personne {	
private String nom ;	
private AdressePersonne ad ;	référence vers une instance de la classe interne
public Personne(String nom, String ville, String pays) {	
this.nom = nom ;	
ad = this.new AdressePersonne(ville, pays) ;	création d'une instance interne
}	
public String toString(){	
return nom+" habite "+ad ;	code standard avec partage des responsabilités sur toString
//return nom+" habite "+ad.ville ;	accès à un attribut de l'instance interne
}	
private class AdressePersonne{	
private String ville="là", pays="ailleurs" ;	
public AdressePersonne(String ville, String pays)	
{this.ville = ville ; this.pays = pays ;}	
public String toString(){	
return ville+" "+pays ;	code standard avec partage des responsabilités sur toString
//return Personne.this.nom+this.ville+... ;	accès à un attribut de l'instance englobante
//return nom+this.ville+... ;	accès à un attribut de l'instance englobante (alternative montrant que l'instance englobante est implicite)
}	
}	
public static void main(String[] args) {	
Personne j = new Personne ("Jacky","Boston","USA") ;	
System.out.println(j) ;	
}	
}	

TABLE 5 – La classe NumTel est une classe locale de la méthode saisieNumTel

public class Autorisation { private int num ; public Autorisation() {} public String saisieNumTel(Scanner sc) { System.out.println("saisie num tel") ; String numComplet ; String code="+" ; **class NumTel{** **private String indicatif ;** **private String numerolocal ;** **public String saisie(Scanner sc){** **System.out.println("saisie indicatif") ;** **indicatif = sc.next() ;** **System.out.println("saisie numero") ;** **numerolocal = sc.next() ;** **//~~numComplet = indicatif+" "+... ;~~** **return code+indicatif+" "+numerolocal ;** **//+Autorisation.this.num ;** **//+num ;** **}** **}** // fin classe NumTel NumTel n = new NumTel() ; numComplet = n.saisie(sc) ; return numComplet ; }//fin saisieNumTels public static void main(String[] args) { Scanner sc = new Scanner(System.in) ; Autorisation a = new Autorisation() ; System.out.println(a.saisieNumTel(sc)) ; } }	 pas de visibilité indiquée on ne peut affecter une valeur à numComplet ici on peut utiliser la variable locale `code` accès possible à l'instance implicite idem (comme dans une classe interne) création d'un objet de la classe locale

TABLE 6 – Classe anonyme dans la méthode saisieNumTel

public class Saisissable{ public String saisie(Scanner sc){ System.out.println("On peut me saisir !") ; return "" ; } } public class AutorisationAnonyme {	définition de la classe support mais ici la classe `Object` aurait pu servir de support également
public String saisieNumTel(Scanner sc) { System.out.println("saisie num tel") ; String code="+" ; Saisissable n = **new Saisissable(){** **private String indicatif ;** **private String numerolocal ;** **public String saisie(Scanner sc){** **System.out.println("saisie indicatif") ;** **indicatif = sc.next() ;** **System.out.println("saisie numero") ;** **numerolocal = sc.next() ;** **return code+indicatif+" "+numerolocal ;** **}** **} ; // fin classe anonyme** return n.saisie(sc) ; }//fin saisieNumTels	définition de la classe anonyme et instanciation en même temps
public static void main(String[] args) { Scanner sc = new Scanner(System.in) ; AutorisationAnonyme a = new AutorisationAnonyme() ; System.out.println(a.saisieNumTel(sc)) ; } }	