Comptes bancaires : une vue simplifiée
Corrigé

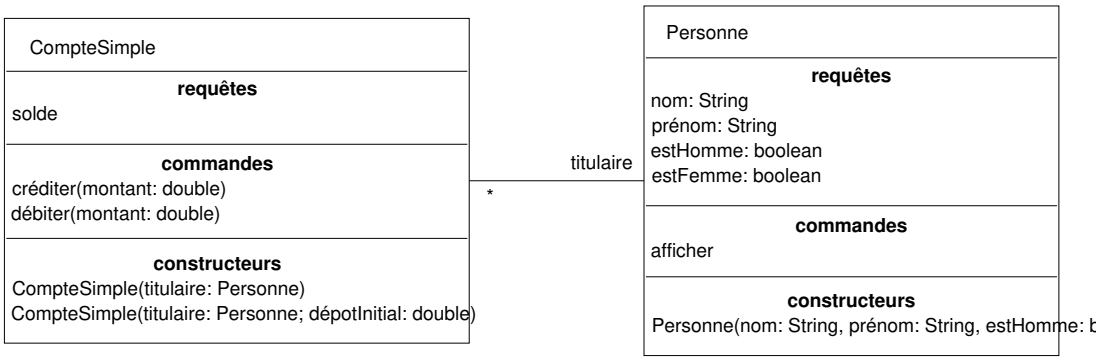
Objectif : Comprendre au travers d'un exemple simplifié de comptes bancaires, les concepts d'encapsulation, de propriétés d'instances et de classes, d'héritage, de surcharge et de redéfinition de méthodes, de polymorphisme et de liaison dynamique.

Exercice 1 : Compte bancaire simple

Nous nous intéressons à un compte simple caractérisé par un solde exprimé en euros, positif ou négatif, et son titulaire. [1] Il est possible de créditer ce compte ou de le débiter d'un certain montant.

1.1. Spécifier la classe `CompteSimple`. Sur le diagramme de classe, on fera apparaître aussi la classe `Personne` (dont la documentation [2] est fournie dans le listing 1).

Solution : Voici le diagramme correspondant. La spécification (documentation et contrats) n'apparaissent pas sur ce diagramme d'analyse mais sont dans le code donné ci-après.



Remarques :

1. On ne doit pas mettre un attribut « titulaire » de type Personne dans CompteSimple. En effet, mettre une relation permet d'expliciter l'architecture du système. C'est particulièrement le cas si on cache les attributs, opérations et constructeurs des classes.

   D'autre part, un attribut correspond à une relation de composition alors qu'ici on a une relation d'association. L'attribut ne convient pas.

   La règle générale est de ne jamais mettre un attribut dont le type est une classe sauf si la classe correspond à un type considéré comme élémentaire. C'est le cas de String, Date, etc.

1. Nous simplifions le problème en considérant que tout compte a un et un seul titulaire correspondant à une personne physique modélisée par une classe Personne fournie.

2. Il s'agit de la documentation telle qu'elle pourrait être engendrée par `javadoc` mais présentée sous la forme d'une classe.

2. La relation utilisée entre `CompteSimple` et `Personne` est une association. Il serait envisageable de mettre une agrégation mais ceci dépend du point de vue que l'on prend. Il me semble que les différentes solutions peuvent être justifiées (et donc se justifier).

   Attention : Mettre une relation d'héritage entre CompteSimple et Personne serait faux. Par exemple, ceci signifierait qu'une personne peut se marier avec un compte simple. Pas très romantique !
3. nous ne connaissons pas les attributs réellement utilisés dans la classe Personne. Il s'agit certainement du prénom, du nom et d'un booléen mais on ne peut pas en être sûr... et on n'a pas besoin de le savoir !
4. Faut-il définir `setSolde` ? La réponse est non. On ne peut pas donner une nouvelle valeur au solde d'un compte sans passer par les opérations créditer ou débiter.
5. Pourrait-on regrouper les deux opérations créditer et débiter en une seule ? Ce n'est pas une bonne idée pour trois raisons : 1) le sujet suggère clairement les deux opérations créditer et débiter ; 2) donner un nom significatif à cette seule méthode ne sera pas simple et 3) avoir deux opérations permet de détecter des erreurs si on fait une opération avec un montant négatif, cette erreur passera inaperçue dans la version avec une seule opération.
6. Prendre un type réel n'est pas une bonne idée car on ne maîtrise pas les arrondis. Il faudrait prendre plutôt des nombres décimaux. Java propose la classe java.math.BigDecimal.
7. Nommer « solde » le paramètre des constructeurs de compte simple serait une mauvaise idée. Ce paramètre correspond au dépôt initial fait à l'ouverture du compte ou l'apport. On peut donc choisir les terme « dépôt initial » ou « apport ».

1.2. Proposer un programme de test de la classe CompteSimple.

Solution : Nous avons défini deux constructeurs sur la classe CompteSimple, il est donc important de tester les deux constructeurs, donc de créer au moins deux comptes.

Dans le texte, il est dit que le solde peut être positif et négatif. Il est donc important de tester un débit avec un montant qui fait passer le solde en négatif.

Remarque : Nous utilisons JUnit pour écrire le programme de test !

Notons que dans le programme de test, on peut se demander ce à quoi correspond le booléen `true` utilisé lors de la création de la personne. Pour en connaître la signification, il faut regarder la spécification du constructeur. Il aurait été plus lisible d'utiliser un type énuméré Genre par exemple avec deux valeurs HOMME et FEMME par exemple. Voir `Genre.HOMME` est plus clair que `true`, non ?

```java
import org.junit.*;
import static org.junit.Assert.*;

/** Tests unitaires JUnit pour la classe CompteSimple.  */
public class CompteSimpleTest {

    public static final double EPSILON = 1e-6;
        // précision pour la comparaison entre réels.

    protected CompteSimple c1;
```

Listing 1 – Documentation de la classe Personne

```java
/**
 *  Une personne est simplement définie par son nom, son prénom et son sexe qui
 *  sont les informations nécessaires pour pouvoir la construire.
 */
public class Personne {

    /** Construire une personne à partir de nom, son prénom et son sexe.
     * @param prenom_ le prénom de la personne
     * @param nom_ le nom de la personne
     * @param masculin_ est-ce un homme ?
     */
    public Personne(String prenom_, String nom_, boolean masculin_);

    /** Le nom de la personne.
     * @return le nom de la personne
     */
    public String getNom();

    /** Le prénom de la personne.
     * @return le prénom de la personne
     */
    public String getPrenom();

    /** La personne est-elle un homme ?
     * @return la personne est-elle un homme ?
     */
    public boolean estHomme();

    /** La personne est-elle une femme ?
     * @return la personne est-elle une femme ?
     */
    public boolean estFemme();

    /** Afficher le nom et le prénom. */
    public void afficher();
}

```

```
11      protected CompteSimple c2;
12      protected Personne p1;
13
14      @Before
15      public void setUp() {
16          this.p1 = new Personne("Xavier", "Crégut", true);
17          this.c1 = new CompteSimple(p1, 1000);
18          this.c2 = new CompteSimple(p1);
19      }
20
21      @Test
22      public void testerInitialisationC1() {
23          assertEquals(this.c1.getTitulaire(), this.p1);
24          assertEquals(1000, this.c1.getSolde(), EPSILON);
25      }
26
27      @Test
28      public void testerInitialisationC2() {
29          assertEquals(this.c2.getTitulaire(), this.p1);
30          assertEquals(0, this.c2.getSolde(), EPSILON);
31      }
32
33      @Test
34      public void testerCrediter() {
35          this.c1.crediter(100);
36          assertEquals(1100, this.c1.getSolde(), EPSILON);
37      }
38
39      @Test
40      public void testerDebiter() {
41          this.c1.debiter(100);
42          assertEquals(900, this.c1.getSolde(), EPSILON);
43          this.c1.debiter(250);
44          assertEquals(650, this.c1.getSolde(), EPSILON);
45      }
46
47      @Test
48      public void testerSoldeNegatif() {
49          this.c1.debiter(1200);
50          assertEquals(-200, this.c1.getSolde(), EPSILON);
51          this.c1.crediter(300);
52          assertEquals(100, this.c1.getSolde(), EPSILON);
53      }
54
55  }
```

1.3. Écrire la classe CompteSimple.

Solution : Nous choisissons d'avoir deux attributs, l'un pour le solde, l'autre pour le titulaire. Les requêtes `getSolde()` et `getTitulaire()` sont donc des accesseurs sur ces attributs. Nous ne définissons pas la méthode d'altération `setSolde` car elle n'a pas été identifiée. On doit utiliser `créditer` ou `débiter`.

```
1   /** CompteSimple modélise un compte bancaire simple tenu en euros.
```

```
 * Il est caractérisé par un titulaire et un solde (positif ou négatif)
 * et peut être crédité ou débité d'un certain montant.
 * @author    Xavier Crégut
 */
public class CompteSimple {
    //@ public invariant getTitulaire() != null;
    //@ private invariant titulaire == getTitulaire();
    //@ private invariant solde == getSolde();

    /** Titulaire du compte. */
    private Personne titulaire;

    /** Solde du compte exprimé en euros. */
    private double solde;

    /** Initialiser un compte.
     * @param titulaire le titulaire du compte
     * @param depotInitial le montant initial du compte
     */
    //@ requires leTitulaire != null;    // le titulaire existe
    //@ requires depotInitial >= 0;    // montant initial strictement positif
    //@ ensures getSolde() == depotInitial;    // solde initialisé
    //@ ensures getTitulaire() == leTitulaire;    // titulaire initialisé
    public CompteSimple(Personne leTitulaire, double depotInitial) {
        this.solde = depotInitial;
        this.titulaire = leTitulaire;
    }

    /** Initialiser un compte.
     * Son solde est nul.
     * @param titulaire le titulaire du compte
     */
    //@ requires titulaire != null;        // le titulaire existe
    //@ ensures getSolde() == 0;        // pas de dépôt initial
    //@ ensures getTitulaire() == titulaire;    // titulaire initialisé
    public CompteSimple(Personne titulaire) {
        this(titulaire, 0);
    }

    /** Solde du compte exprimé en euros. */
    public /*@ pure @*/ double getSolde() {
        return this.solde;
    }

    /** Titulaire du compte. */
    public /*@ pure @*/ Personne getTitulaire() {
        return this.titulaire;
    }

    /** Créditer le compte du montant (exprimé en euros).
     * @param montant montant déposé sur le compte en euros
     */
    //@ requires montant > 0;
```

```
55      //@ ensures getSolde() == \old(getSolde()) + montant; // montant crédité
56      public void crediter(double montant) {
57          this.solde = this.solde + montant;
58      }
59
60      /** Débiter le compte du montant (exprimé en euros).
61       * @param montant montant retiré du compte en euros
62       */
63      //@ requires montant > 0;
64      //@ ensures getSolde() == \old(getSolde()) - montant; // montant débité
65      public void debiter(double montant) {
66          this.solde = this.solde - montant;
67      }
68
69      public String toString() {
70          return "solde=" + this.solde + ", titulaire=\"" + this.titulaire + "\"";
71      }
72
73  }
```

Remarque : Mettre l'accesseur sur le titulaire pourrait violer le principe d'encapsulation puisqu'il sera alors possible d'accéder à l'objet titulaire et, potentiellement, modifier son état sans que le le compte n'en ait connaissance. On remarque cependant ici qu'il n'y a pas d'opération de modification sur la classe Personne. Les objets Personne sont donc immuables. On peut donc les partager sans risque de les voir modifier.

Exercice 2 : Compte courant

En fait, une banque conserve, pour chaque compte, l'historique des opérations qui le concernent (on se limite ici aux opérations de crédit et de débit). On souhaite modéliser un tel compte qu'on appelle compte courant. En plus des méthodes d'un compte simple, un compte courant offre des méthodes pour afficher l'ensemble des opérations effectuées (`editerReleve`) ou seulement les opérations de crédit (`afficherReleveCredits`) ou de débit (`afficherRereleveDebits`).

Pour représenter l'historique, on utilisera la classe `Historique` fournie (listing 2). Pour enregistrer une opération, on conservera simplement le montant de l'opération précédé d'un signe indiquant s'il s'agit d'un crédit (montant positif) ou d'un débit (montant négatif).

2.1. Indiquer quelle est la relation entre un compte simple et un compte courant et compléter le diagramme de classes UML.

Solution : On remarque que le compte courant est un sous-type de compte simple : un compte courant pourra être utilisé en lieu et place d'une compte simple. Cette constatation nous permet d'utiliser la relation d'héritage : un compte courant est une spécialisation d'une compte simple.

Attention : Dire qu'il y a les mêmes opérations et que l'on pourra réutiliser du code n'est pas un bon argument !

Important : Dès qu'on a décidé d'utiliser l'héritage, il faut se demander s'il y a des méthodes de la super-classe qui doivent être redéfinies dans la sous-classe.

Le compte courant ajoute la notion d'historique des opérations réalisées. Il faut donc redéfinir les opérations créditer et débiter de compte simple pour les enregistrer dans l'historique.

Listing 2 – Documentation de la classe `Historique`

```java
/** Historique gère un historique chronologique des entiers enregistrés.  */
public class Historique {

    /** Construire un historique vide.  */
    public Historique();

    /** Enregistrer une nouvelle information dans l'historique
     * @param info l'information à enregistrer dans l'historique
     */
    public void enregistrer(double info);

    /** La i<SUP>è</SUP> valeur de l'historique, 1 correspond à la plus
     * ancienne, getNbValeurs() à la plus récente (la dernière).
     *
     * <b>Attention :</b> Cette convention est différente de celle
     * traditionnellement adoptée en Java pour les tableaux et vecteurs !
     *
     * @param i indice de l'opération compris en 1 et getNbValeurs().
     */
    public double getValeur(int i);

    /** Le nombre d'entiers enregistrés dans l'historique
     * @return le nombre d'entiers dans l'historique
     */
    public int getNbValeurs();
}
```

Le diagramme correspondant est donné à la figure 1. Nous n'avons pas fait apparaître les opérations `afficherReleveCredits` et `afficherReleveDebits` qui n'apportent rien de plus. Nous ne gardons que `editerReleve`.

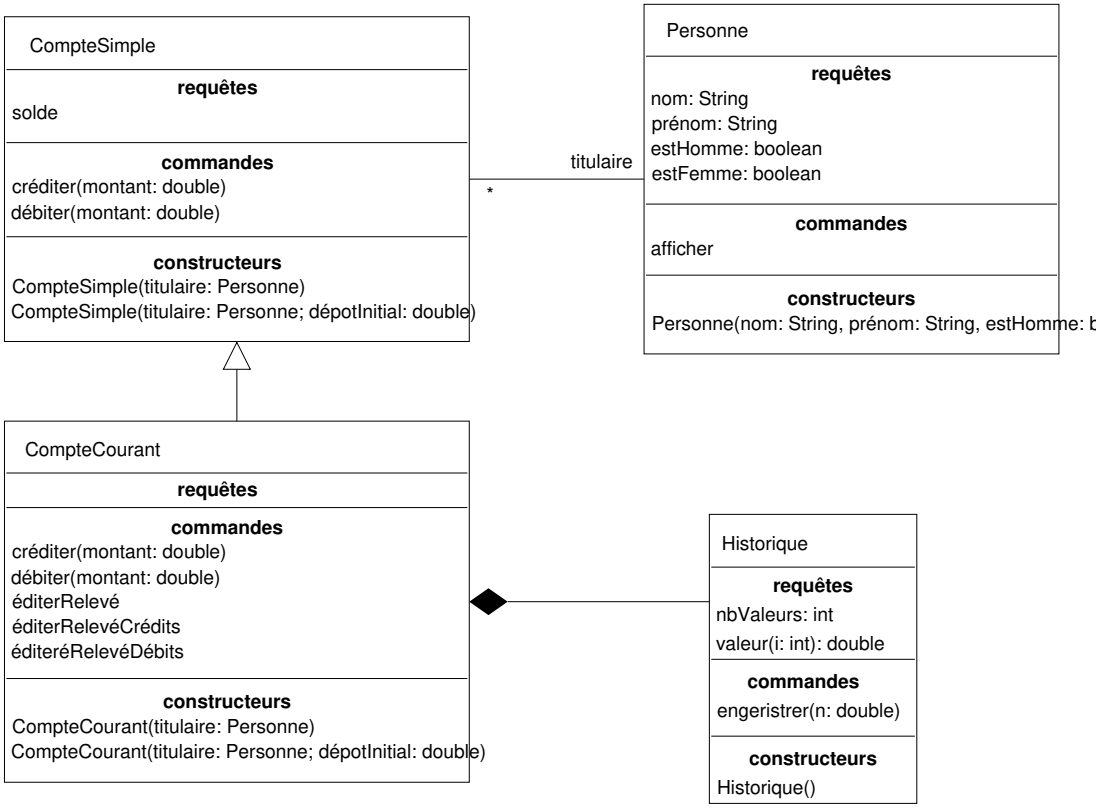


FIGURE 1 – Diagramme de classes intégrant le `CompteCourant`

Remarque : La relation entre CompteCourant et Historique est une relation de composition car l'historique est propre à chaque compte (et, en conséquence ne peut pas être partagé).
Attention : Ceci ne fonctionne qu'à condition que l'historique soit infini (ou circulaire, avec perte des informations les plus anciennes). Ce qui est quasiment le cas avec l'implantation proposée pour l'historique.
Remarque : On ne devrait pas faire apparaître l'historique sur le diagramme d'analyse car il s'agit d'un choix d'implantation qui ne sera pas accessible à l'utilisateur de la classe.

2.2. Écrire la classe `CompteCourant` correspondant à un compte courant.

Solution :

```
1  /** Un compte courant est un compte simple avec un historique des
2   * opérations effectuées.
3   * @author  Xavier Crégut
4   */
5  public class CompteCourant extends CompteSimple {
6
7      /** L'historique des opérations de crédit et débit réalisées. */
```

```
8       private Historique operations;
9
10      /** Construction d'un compte courant dont le solde est nul.
11       * @param titulaire le titulaire du compte
12       */
13      //@ requires titulaire != null;        // le titulaire existe
14      //@ ensures getSolde() == 0;         // pas de dépôt initial
15      //@ ensures getTitulaire() == titulaire;    // titulaire initialisé
16      public CompteCourant(Personne titulaire) {
17          this(titulaire, 0);
18      }
19
20      /** Construction d'un compte courant avec un montant initial.
21       * @param titulaire le titulaire du compte
22       * @param depotInitial le montant initial du compte
23       */
24      //@ requires titulaire != null;    // le titulaire existe
25      //@ requires depotInitial >= 0;    // montant initial strictement positif
26      //@ ensures getSolde() == depotInitial;    // solde initialisé
27      //@ ensures getTitulaire() == titulaire;    // titulaire initialisé
28      public CompteCourant(Personne titulaire, double depotInitial) {
29          super(titulaire, depotInitial);
30          this.operations = new Historique();
31          if (depotInitial > 0) {
32              this.operations.enregistrer(depotInitial);
33          }
34      }
35
36      /** Créditer le compte du montant (exprimé en euros). L'opération
37       * est enregistrée.
38       * @param montant montant déposé sur le compte en euros
39       */
40      @Override public void crediter(double montant) {
41          super.crediter(montant);
42          this.operations.enregistrer(montant);
43      }
44
45      /** Débiter le compte du montant (exprimé en euros). L'opération
46       * est  enregistrée.
47       * @param montant montant retiré du compte en euros
48       */
49      @Override public void debiter(double montant) {
50          super.debiter(montant);
51          this.operations.enregistrer(-montant);
52      }
53
54      /** Éditer le relevé du compte. */
55      public void editerReleve() {
56          System.out.println("----------------------------------------");
57
58          // afficher les caractéristiques du compte
59          System.out.print("Titulaire : ");
60          this.getTitulaire().afficher();
```

```
61          System.out.println();
62
63          // Afficher l'historique des opérations
64          System.out.println("Historique des opérations : ");
65          for (int i = 1, nb = this.operations.getNbValeurs(); i <= nb; i++) {
66              if (this.operations.getValeur(i) > 0) {
67                  System.out.println("  o  Dépôt de " + this.operations.getValeur(i));
68              } else {
69                  System.out.println("  o  Retrait de " + -this.operations.getValeur(i));
70              }
71          }
72
73          // Afficher le solde du compte
74          System.out.println("----------------------------------------");
75          System.out.println("Solde du compte : " + getSolde());
76          System.out.println("----------------------------------------");
77          System.out.println();
78      }
79
80      @Override public String toString() {
81          return super.toString() + ", opérations=" + this.operations;
82      }
83
84  }
```

Remarque : On appelle `super.créditer(montant)` avant d'enregistrer l'opération. Ceci est important dans le cas où l'opération échoue car dans ce cas, il ne faut pas l'enregistrer dans l'historique... (par exemple si une exception se produit).
Remarque : Une « meilleure » solution serait de définir une classe `Opération` spécialisée en (opération de) Débit et (opération de) Crédit et de conserver une liste d'opérations encapsulée dans une classe `Historique`.
2.3. Le listing 3 propose un exemple d'utilisation des comptes bancaires. Exécuter et commenter ce programme.
Solution : Voici le résultat de l'exécution du programme.

```
1  Solde de cs1 = 1000.0
2  Solde de cc1 = 1100.0
3  ----------------------------------------
4  Titulaire : M. Xavier Crégut
5  Historique des opérations :
6    o  Dépôt de 100.0
7    o  Dépôt de 1000.0
8  ----------------------------------------
9  Solde du compte : 1100.0
10 ----------------------------------------
11
12 Solde de cs = 600.0
13 Solde de cc1 = 600.0
14 ----------------------------------------
15 Titulaire : M. Xavier Crégut
16 Historique des opérations :
```

Listing 3 – Programme utilisant les comptes

```java
/** Programme utilisant les comptes bancaires.  */
public class ExempleComptes {
    public static void main (String argv []) {
        Personne p1 = new Personne("Xavier", "Crégut", true);
        CompteSimple cs1 = new CompteSimple(p1, 0);
        CompteCourant cc1 = new CompteCourant(p1, 100);

        cs1.crediter(1000);
        System.out.println("Solde de cs1 = " + cs1.getSolde());

        cc1.crediter(1000);
        System.out.println("Solde de cc1 = " + cc1.getSolde());
        cc1.editerReleve();

        CompteSimple cs = cc1;
        cs.debiter(500);
        System.out.println("Solde de cs = " + cs.getSolde());
        System.out.println("Solde de cc1 = " + cc1.getSolde());
        cc1.editerReleve();
    }
}
```

```
  o  Dépôt de 100.0
  o  Dépôt de 1000.0
  o  Retrait de 500.0
----------------------------------------
Solde du compte : 600.0
----------------------------------------

```

Le programme a les résultats attendus grâce à la liaison dynamique et la redéfinition des méthode `créditer` et `débiter` (et dans une moindre mesure `afficher`) dans la classe `CompteCourant`.

Les points importants sont :

— cc1.crediter(1000) : on exécute la version de créditer de la classe `CompteCourant` car elle a été redéfinie. Si on a oublié la redéfinition, on s'en rend compte ici car l'historique n'est pas mis à jour (voir `cc1.editerReleve()`).

— cs.débiter(500) : le compilateur sélectionne la méthode `débiter(double)` de la classe `CompteSimple` (type de la poignée `cs`). À l'exécution, la version de la méthode exécutée est celle de `CompteCourant`, classe de l'objet attaché à `cs`. Il s'agit de la liaison dynamique.

Notons qu'à la fin du programme, on ne peut pas faire `cs.editerReleve()` car `cs` est de type `CompteSimple` qui ne possède pas cette méthode.

2.4. Quels tests supplémentaires faudrait-il faire ?

Solution : Le test de la question précédente permet de montrer que l'historique est effectivement géré (quand on considère le CompteCourant comme un CompteCourant ou un CompteSimple).

Cependant, CompteCourant étant une spécialisation de CompteSimple, elle doit réussir les tests de CompteSimple (exercice 1).

```java
/** Tests unitaires JUnit pour la classe CompteSimple.
  * Les comptes courants doivent réussir les tests de compte simple
  * @author    Xavier Crégut
  */
public class CompteCourantTest extends CompteSimpleTest {

    @Override
    public void setUp() {
        this.p1 = new Personne("Xavier", "Crégut", true);
        this.c1 = new CompteCourant(p1, 1000);
        this.c2 = new CompteCourant(p1);
    }

    public static void main(String[] args) {
        org.junit.runner.JUnitCore.main(CompteCourantTest.class.getName());
    }

}
```

Exercice 3 : La banque

Une banque est bien entendu un organisme qui gère un grand nombre de comptes, qu'ils soient simples ou courants.

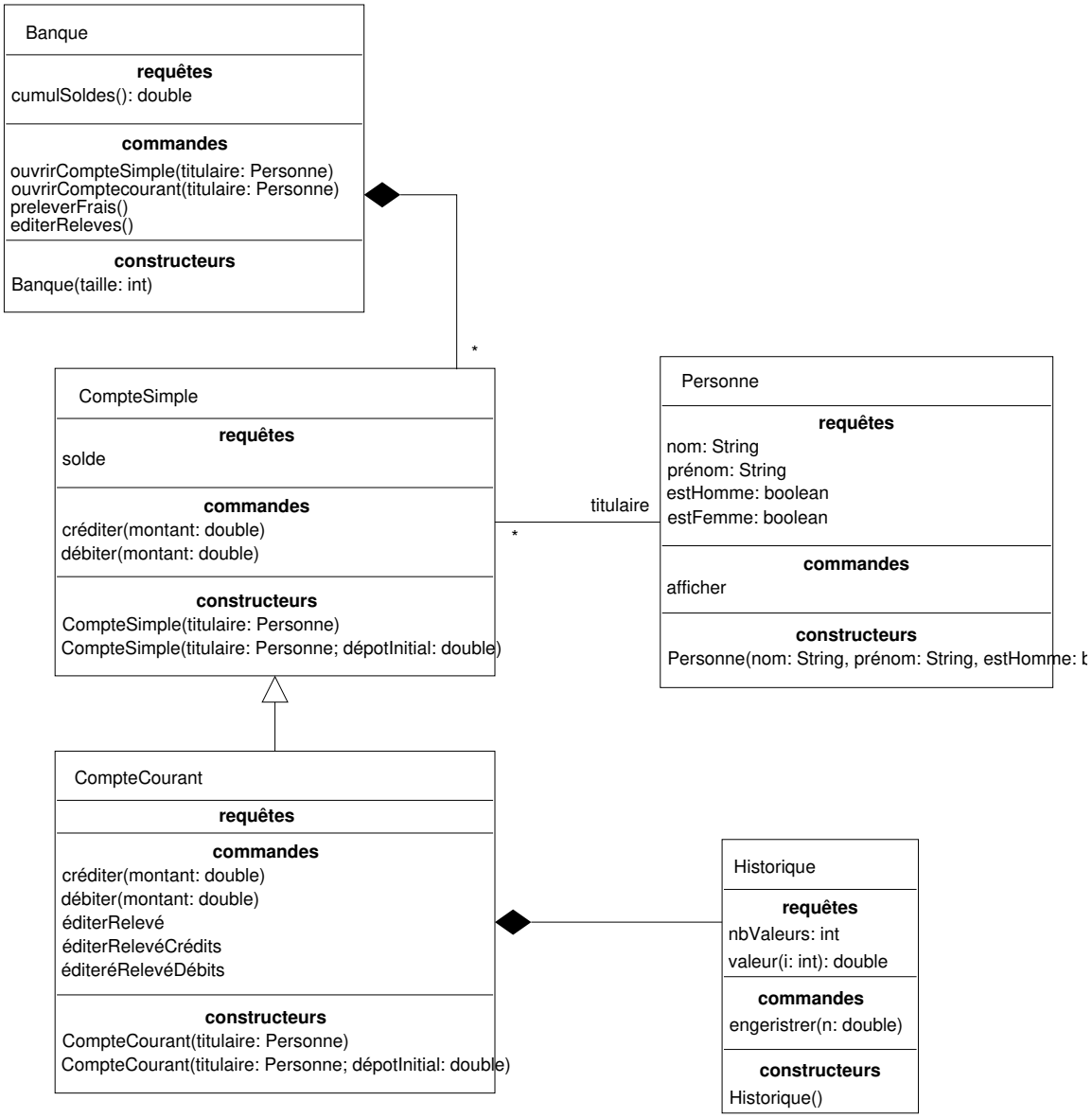
3.1. *État de la classe* `Banque`. Sachant que tous les comptes sont rangés dans un unique tableau, indiquer quels sont les attributs de la classe `Banque`. Donner les constructeurs de la classe `Banque`.

Solution : Il y a au moins deux attributs correspondant respectivement au tableau des comptes et au nombre effectif de comptes gérés par la banque. D'autres attributs seraient certainement nécessaires mais ils ne sont pas explicités dans le sujet (nom de la banque, etc.).

Les attributs sont donc :

```java
    private CompteSimple[] comptes;    // les comptes gérés par la banque
    private int nbComptes;        // le nombre de comptes ouverts
```

En anticipant un peu sur les questions suivantes, on peut compléter le diagramme de classes UML.



3.2. *Ouverture des comptes.* Définir sur la classe `Banque` une méthode pour ouvrir un compte simple et une méthode pour ouvrir un compte courant.

Solution : On crée une banque en précisant sa taille, c'est-à-dire le nombre de comptes qu'elle est capable de tenir. On définit ensuite deux opérations, l'une permettant à une personne de créer un compte simple, l'autre un compte courant.

```
1       /** Construire une nouvelle banque sans clients, donc sans comptes. */
2       public Banque(int taille) {
3           comptes = new CompteSimple[taille];
4           nbComptes = 0;
5       }
6
7       /** Ouvrir un compte courant pour le client p
8        *  avec un dépot initial « apport ».
```

```
9         */
10        public void ouvrirCompteCourant(Personne p, double apport) {
11            this.comptes[this.nbComptes++] = new CompteCourant(p, apport);
12        }
13
14        /** Ouvrir un compte simple pour le client p
15         *  avec un dépot initial « apport ».
16         */
17        public void ouvrirCompteSimple(Personne p, double apport) {
18            this.comptes[this.nbComptes++] = new CompteSimple(p, apport);
19        }
```

3.3. *Cumul des soldes des comptes.* Définir une méthode qui donne le cumul des soldes des comptes gérés par une banque. L'accès à d'autres informations pourrait être envisagé : le nombre de comptes débiteurs, la somme des soldes des comptes débiteurs, etc.
Solution : Il suffit de faire la somme des soldes de chacun des comptes. Le polymorphisme permet le traitement. La liaison dynamique n'intervient pas ici car getSolde() n'est pas redéfini pour les CompteCourant.

```
1         /** Cumul du solde des comptes gérés par la banque. */
2         public double getCumulSoldes() {
3             double result = 0;
4             for (int i = 0; i < nbComptes; i++) {
5                 result += comptes[i].getSolde();
6             }
7             return result;
8         }
```

3.4. *Frais de tenue de compte.* Nous considérons que la banque prélève périodiquement des frais de tenue de compte. Écrire une méthode qui débite sur tous les comptes la somme de 2 euros.
Solution : Même remarque que pour la question précédente mais ici la liaison dynamique intervient. En particulier, pour les comptes courants, l'opération de débit sera enregistrée dans l'historique.

```
1         /** Prélèvement des frais bancaires : prélèvement de 2 euros sur chaque
2           * compte.
3           */
4         public void preleverFrais() {
5             for (int i = 0; i < nbComptes; i++) {
6                 comptes[i].debiter(2);
7             }
8         }
```

Question : Est-ce que les -2 euros apparaîtront sur les relevés des comptes courants ?
Oui, bien sûr ! C'est bien la méthode de CompteCourant qui sera appelée même si on manipule un compte courant à travers une poignée de type CompteSimple. C'est la liaison dynamique !

3.5. *Édition des relevés.* Écrire une opération qui édite (à l'écran) les relevés de tous les comptes courants (et seulement des comptes courants, le relevé n'a pas de sens pour un compte simple).
Solution : Il est ici nécessaire de faire de l'interrogation dynamique de type pour reconnaître les comptes courants et alors seulement demander l'édition du relevé.

```java
/** éditer le relevés des comptes courants */
public void editerReleves() {
    for (int i = 0; i < nbComptes; i++) {
        if (comptes[i] instanceof CompteCourant) {
            CompteCourant cc = (CompteCourant) comptes[i];
            cc.editerReleve();
        }
    }
}
```

Remarque :

1. Il ne faut pas oublier le transtypage après le test avec **instanceof** car le type de `cs` reste `CompteSimple` !
2. On pourrait ne pas passer par la variable locale `cc` en faisant :

   ```java
   ((CompteSimple) cs).editerReleve();
   ```

   Il ne faut pas oublier les parenthèses car la priorité de « . » est supérieure à cette du transtypage.

   Cette formulation est moins lisible que l'utilisation d'une variable locale et beaucoup moins pratique si l'on doit utiliser plusieurs opérations du type plus spécifique.
3. Au lieu de tester explicitement le type réel avec **instanceof**, il serait possible de faire directement le transtypage et de récupérer l'exception éventuellement levée (`ClassCastException`). Le test avec **instanceof** me paraît cependant préférable ici.

3.6. *Autre organisation des comptes.* Proposer une autre organisation des comptes qui faciliterait l'édition des relevés. Discuter cette solution.

Solution : L'utlisation de **instanceof** est souvent une mauvaise idée [3]. Le but est donc de trouver une manière décrire simplement les trois méthodes avec une seule boucle et sans instanceof ni transtypage.

Une **première solution** serait de définir une méthode `editerReleve` dans `CompteSimple` avec un code vide. Cette solution n'est pas très satisfaisante car un utilisateur de la classe `CompteSimple` se demandera à quoi correspond cette opération qui ne fait rien. De plus, cette approche ne peut pas être appliquée systématiquement : on ne peut pas (et on ne doit pas !) ajouter dans une superclasse toutes les méthodes qui sont (ou seront) définies sur ses sous-classes. Ceci n'est pas réaliste parce que lorsque l'on définit une classe par spécialisation, on ne devrait pas modifier la classe spécialisée. Cette technique peut toutefois être utile dans certains cas (voir les collections et l'interface java.util.Collection).

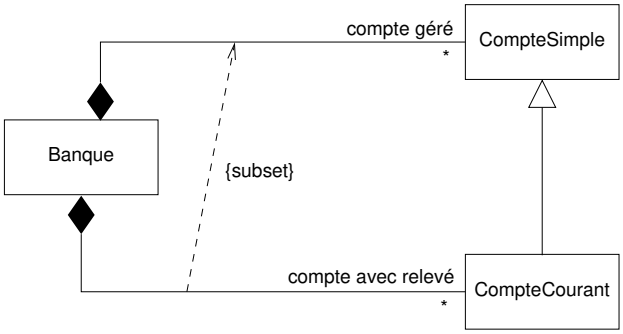
Une **meilleure solution** ici consiste, au niveau de la banque, à conserver deux tableaux de comptes, l'un contenant tous les comptes, l'autre seulement les comptes courants. Nous introduisons une redondance puisqu'un compte courant apparaît dans les deux tableaux : ouvrir un

3. Ici, l'utilisation de **instanceof** serait toutefois légitime mais il serait préférable de définir une interface qui spécifie l'opération `editerReleve`. On pourrait alors tester si un compte réalise cette interface (interface de marquage) et dans l'affirmative, lui donner le type de l'interface pour appeler l'opération `editerReleve`.

L'opérateur **instanceof** est à éviter quand on teste plusieurs types possibles pour un même objet.

compte courant l'ajoute au deux tableaux et une opération de clôture du compte le supprimerait des deux tableaux. Les méthodes s'écrivent facilement, soit en travaillant sur la liste de tous les comptes (cumul des comptes et calcul des intérêts), soit la liste des comptes courants (éditer les relevés).

Le diagramme UML est alors le suivant.



Exercice 4 : Numéro de compte

Pour les différencier, chaque compte possède un numéro unique. On suppose que les numéros sont des entiers et qu'ils sont attribués par ordre croissant en commençant à 10001. Les numéros de compte sont donc 10001, 10002, 10003, etc. Dans la suite, nous envisageons deux solutions pour attribuer les numéros de compte.

4.1. On souhaite que l'attribution du numéro de compte soit de la responsabilité des classes `CompteSimple` et `CompteCourant`. Indiquer les modifications à apporter à ces classes.

Solution : Si le compte est responsable de gérer son numéro, ceci signifie qu'il doit être initialiser lors de la création de chaque nouveau compte. Il suffit alors de le traiter au niveau de `CompteSimple`. En effet, `CompteCourant` hérite de `CompteSimple` et récupèrera donc le numéro de Compte.

Le numéro de compte doit être initialisé à chaque création, dans les constructeurs de la classe `CompteSimple`. Comme le second constructeur appelle le premier, il est suffisant d'adapter le premier constructeur en introduisant un simple appel à `initialiserNumero()`.

```
1       /** Initialiser un compte.
2        * @param titulaire le titulaire du compte
3        * @param depotInitial le montant initial du compte
4        */
5       //@ requires leTitulaire != null;    // le titulaire existe
6       //@ requires depotInitial >= 0;    // montant initial strictement positif
7       //@ ensures getSolde() == depotInitial;    // solde initialisé
8       //@ ensures getTitulaire() == leTitulaire;    // titulaire initialisé
9       public CompteSimple(Personne leTitulaire, double depotInitial) {
10          this.solde = depotInitial;
11          this.titulaire = leTitulaire;
12          this.initialiserNumero();
13      }
```

Chaque compte possède un attribut d'instance qui est son numéro. Il suffit de le définir sur la classe `CompteSimple`.

Pour initialiser le numéro d'un compte, il faut se souvenir du dernier numéro de compte attribué. Cette information est une information commune à tous les comptes. Nous la définissons donc comme un attribut de classe dans la classe `CompteSimple`.

```
1       /** Numéro du compte. */
2       private String numero;
3           // Il est essentiel que le numéro de compte soit privé pour
4           // que la classe puisse contrôler son changement et donc
5           // garantir son unicité.
6           //
7           // Ceci se généralise à toutes les données d'une classe.
8           // C'est la protection des attributs en écriture.
9
10      /** Le dernier numéro attribué à un compte. */
11      private static int dernierNumero = 10000;
12           //  dernierNumero est un « attribut de classe » car il est
13           //  partagé par toutes les instances (les objets) de la classe
14           //  compte.  C'est lui qui permet d'affecter un numéro unique
15           //  à un compte car il contient le dernier numéro utilisé.
16
```

Enfin, nous pouvons définir la méthode `initialiserNumero()`.

```
1       /** Affecter un numéro unique au compte. */
2       private void initialiserNumero() {
3           dernierNumero++;
4           this.numero = String.valueOf(dernierNumero);
5       }
```

Notons que cette dernière méthode est bien une méthode d'instance (et non de classe) car elle manipule l'état de l'objet (le numéro du compte). Il se trouve qu'elle manipule également des attributs de classe (`dernierNumero`).

On aurait pu définir une méthode qui retourne le nouveau numéro à utiliser. Dans ce cas, la méthode aurait été de classe ! Attention, c'est une méthode qui renvoie une valeur et modifie un attribut. Ceci n'est pas forcément conseillé !

Remarque : Nous n'avons en fait modifié que la classe `CompteSimple`. Les classes dérivées telles que `CompteCourant` récupèrent automatiquement le numéro de compte et son initialisation (par l'appel obligatoire du constructeur de la classe `CompteSimple`).

Remarque : Cette solution n'est pas réaliste, notamment parce que les comptes bancaires consistuent une information persistante qui doit être conservée après chaque redémarrage du système informatique.

4.2. On suppose maintenant que c'est la banque qui gère et attribue les numéros de compte. Indiquer les modifications à apporter aux classes `CompteSimple` et `CompteCourant`.

Solution : Dans ce cas, il faut rajouter un paramètre au niveau des constructeurs de la classe `CompteSimple` et donc dans les constructeurs de la classe `CompteCourant`.

Notons que le numéro de compte n'a qu'un accesseur et aucun modifieur. Même la banque ne peut changer le numéro d'un compte. Elle doit le clôturer et en ouvrir un autre.