

Avrile Floro (n° étudiant : 22000086)

Programmation Orientée Objet

Le 4 avril 2024



Table des matières

1	TP8 - Objet	3
1.1	Le diagramme de classe	3
1.2	Le développement	5
1.2.1	La classe Compte	5
1.2.2	La classe Client	8
1.2.3	La classe Banque	10
1.3	Les modifications apportées aux tests proposés	14
1.3.1	Le test 1 - testAddClient()	14
1.3.2	Le test 2 - testAddCompte()	14
1.3.3	Le test 11 - testAddProprioNotNormal()	14
1.3.4	Le test 12 - testRemoveProprio()	15
1.4	Les tests	16
1.4.1	Test1 - testAddClient()	16
1.4.2	Test2 - testAddCompte()	17
1.4.3	Test3 - testFindClient()	17
1.4.4	Test4 - testFindCompte()	18
1.4.5	Test5 - testCrediter()	18
1.4.6	Test6 - testDebiter()	19
1.4.7	Test7 - testDebiterNotNormal()	19
1.4.8	Test8 - testTransfert	20
1.4.9	Test9 - testTransfertNotNormal()	20
1.4.10	Test10 - testAddProprio()	21
1.4.11	Test11 - testAddProprioNotNormal()	21
1.4.12	Test12 - testRemoveProprio()	22
1.4.13	Test13 - testRemoveCompte()	23
1.4.14	Test14 - testRemoveClient()	23
1.5	L'auto-évaluation	25
	Annexes	26
.1	Le fichier "Main.java"	26
.2	Le fichier "Banque.java"	27
.3	Le fichier "Client.java"	29
.4	Le fichier "Compte.java"	30
.5	Le fichier "IllegalOperationException.java"	32
.6	Le fichier "TestBanque.java"	33

Programmation Orientée Objet

TP8 - Objet

Avrile Floro

Étudiante n°22000086

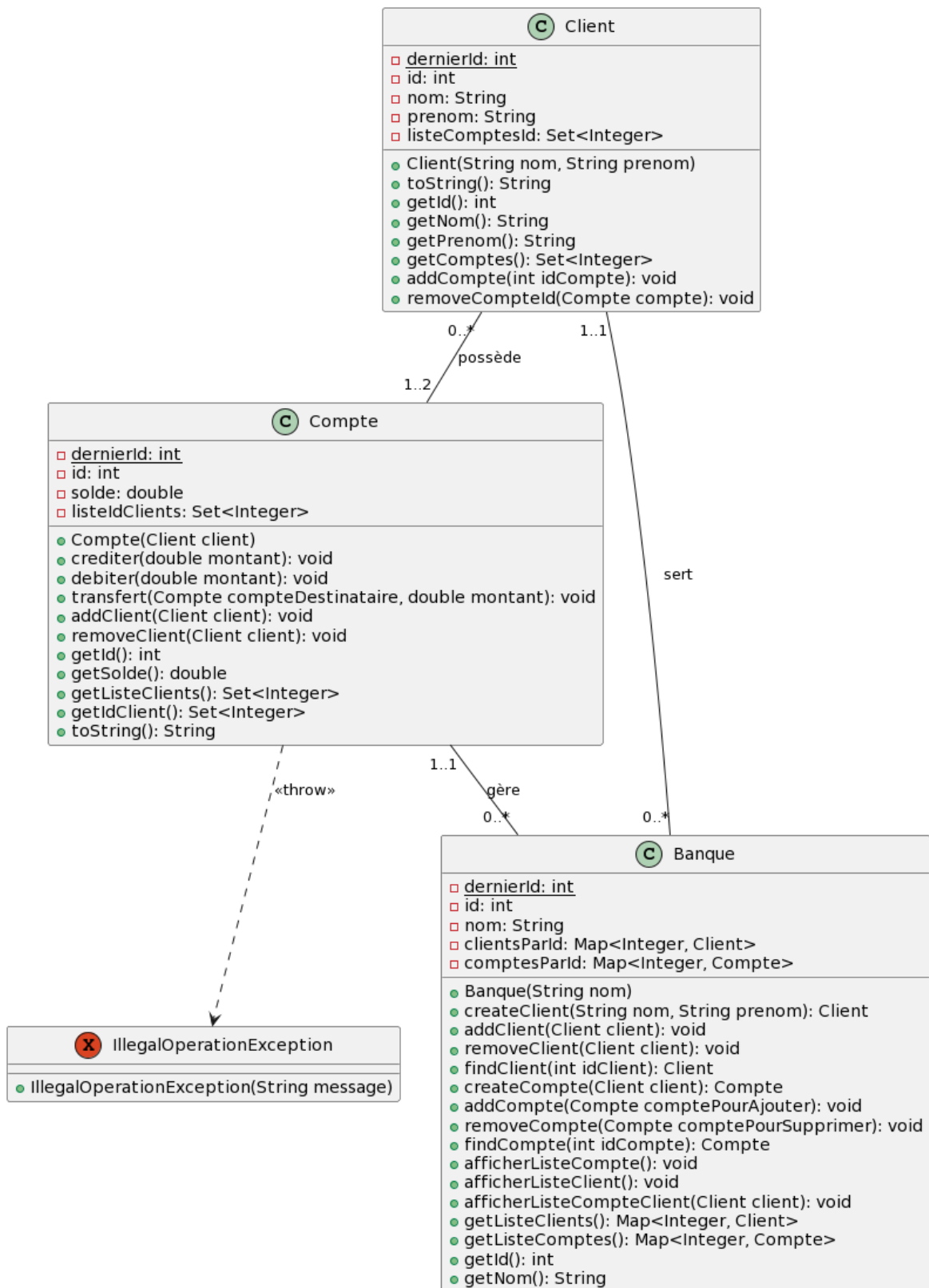
1 TP8 - Objet

1.1 Le diagramme de classe

1. *Modélisation (10-30 minutes)*

- *Représenter sous forme de diagramme de classe le problème*

Dans le cadre de notre diagramme de classe, nous avons utilisé l'icône ● pour représenter une méthode publique et l'icône ◻ pour représenter un attribut privé. Par ailleurs, nous avons souligné les attributs statiques.



1.2 Le développement

1.2.1 La classe Compte

Les attributs

Ayant pour attribut :

Un id dont le numéro sera augmenté automatiquement à la création de chaque nouveau compte

Un solde représentant l'argent disponible sur le compte

Ainsi qu'une liste contenant l'id des clients à qui appartient le compte. Cette liste ne devra pas contenir de doublons.

```
1 import java.util.HashSet;
2 import java.util.Set;
3
4 // définition de la classe Compte
5 public class Compte {
6     // variables statiques et d'instance
7     private static int dernierId; // ID unique pour le dernier compte créé
8     private int id; // ID unique pour ce compte
9     private double solde; // solde du compte
10    private Set<Integer> listeIdClients; // ensemble des IDs des clients associés à ce compte
```

Les méthodes

Un constructeur utilisant un client comme paramètre.

```
12 // constructeur pour initialiser un nouveau compte
13 public Compte(Client client) {
14     this.id = ++dernierId; // incrémente et attribue l'ID
15     this.solde = 0; // initialise le solde à zéro
16     this.listeIdClients = new HashSet<>(); // initialise la liste des clients
17     this.listeIdClients.add(client.getId()); // ajoute le client au compte
18     client.addCompte(this.getId()); // lie le compte au client
19 }
```

Une méthode "crediter" permettant d'augmenter le solde du compte en fonction d'un montant passé en paramètre.

```
49 // méthode pour créditer le compte
50 public void crediter(double montant){
51     this.solde += montant; // ajoute le montant au solde
52 }
```

Une méthode "*debiter*" permettant de diminuer le solde du compte en fonction d'un *montant* passé en paramètre. Cette méthode devra aussi vérifier que le montant voulu être retiré est disponible, sinon une erreur de type "*IllegalOperationException*" devra être déclenchée.

```
54 // méthode pour débiter le compte
55 public void debiter(double montant) throws IllegalOperationException {
56     if (this.solde >= montant) { // vérifie si le solde est suffisant
57         this.solde -= montant; // soustrait le montant du solde
58     } else { // lance une exception personnalisée si le solde est insuffisant
59         throw new IllegalOperationException("Solde insuffisant.");
60     }
61 }
```

Une méthode "*transfert*" permettant d'envoyer un *montant* précis passé en paramètre vers un autre *compte* lui aussi précisé en paramètre.

```
64 // méthode pour transférer de l'argent vers un autre compte
65 public void transfert(Compte compte, double montant) throws IllegalOperationException{
66     this.debiter(montant); // débite le montant de ce compte
67     compte.crediter(montant); // crédite le montant sur le compte cible
68 }
```

Une méthode "*addClient*" permettant d'ajouter un *client* passé en paramètre au compte actuel. Cette méthode commencera par vérifier le nombre de clients à qui appartient le compte et déclenchera une erreur de type "*IllegalOperationException*" si jamais le nombre est déjà de 2.

```
70 // méthode pour ajouter un client au compte
71 public void addClient(Client client) throws IllegalOperationException {
72     if (this.listeIdClients.size() >= 2) { // limite à 2 clients par compte
73         throw new IllegalOperationException("Ce compte en banque a déjà deux clients.");
74     } else { // ajoute le client à la liste du compte
75         this.listeIdClients.add(client.getId()); // Ajoute le client à la liste du compte
76         client.addCompte(this.id); // on ajoute le compte à la liste des comptes du client
77     }
78 }
```

Une méthode "*removeClient*" prenant en paramètre un *client* et supprimant celui-ci du compte.

```
81 // méthode pour retirer un client du compte
82 public void removeClient(Client client) {
83     this.listeIdClients.remove(client.getId()); // retire le client de la liste
84 }
```

*Des **getters** pour les attributs id, solde et pour la liste des id client ainsi qu'un **toString** permettant d'afficher toutes ces informations.*

```
21 // getters pour accéder aux propriétés du compte
22 public int getId() {
23     return id;
24 }
25
26 public double getSolde() {
27     return solde;
28 }
29
30 public Set<Integer> getListeClients() {
31     return listeIdClients;
32 }
33
34 // getter pour obtenir les IDs des clients associés au compte
35 public Set<Integer> getIdClient() {
36     return new HashSet<>(listeIdClients); // retourne une copie de la liste des clients
37 }
38
39 // méthode pour représenter le compte sous forme de chaîne de caractères
40 @Override
41 public String toString() {
42     return "Compte{" +
43         "id=" + id +
44         ", solde=" + solde +
45         ", listeIdClients=" + listeIdClients +
46         '}';
47 }
```


1.2.2 La classe Client

Les attributs

Un id, dont le numéro sera augmenté automatiquement à la création de chaque nouveau client

Un nom

Un prénom

Ainsi qu'une liste contenant les id de ses comptes. Cette liste ne devra pas contenir de doublons.

```
1 import java.util.HashSet;
2 import java.util.Set;
3
4 // définition de la classe Client
5 public class Client {
6     // variables statiques et d'instance
7     private static int dernierId; // compteur statique pour l'ID du dernier client créé
8     private int id; // ID unique pour ce client
9     private String nom; // nom du client
10    private String prenom; // prénom du client
11    private Set<Integer> listeComptesId; // ensemble des IDs des comptes associés à ce client
```

Les méthodes

Un *constructeur* prenant en paramètre le *nom* et *prénom* du nouveau client afin d'initialiser ces attributs.

```
13 // constructeur pour initialiser un nouveau client
14 public Client(String nom, String prenom) {
15     this.id = ++dernierId; // incrémente et attribue l'ID au client
16     this.nom = nom; // initialise le nom
17     this.prenom = prenom; // initialise le prénom
18     this.listeComptesId = new HashSet<>(); // initialise l'ensemble des comptes du client
19 }
```

Une méthode *toString* permettant d'afficher tous ces attributs.

```
21 // méthode pour représenter le client sous forme de chaîne de caractères
22 @Override
23 public String toString() {
24     return "Client{" +
25         "id=" + id +
26         ", nom='" + nom + '\'' +
27         ", prenom='" + prenom + '\'' +
28         ", listeComptesId=" + listeComptesId +
```

```
29         '}}';
30     }
```

*Des **getters** pour les attributs id, nom, prénom et pour la liste de comptes.*

```
32     // getters pour accéder aux propriétés du client
33     public int getId() {
34         return id;
35     }
36
37     public String getNom() {
38         return nom;
39     }
40
41     public String getPrenom() {
42         return prenom;
43     }
44
45     public Set<Integer> getComptes() {
46         return listeComptesId; // obtenir la liste des IDs des comptes du client
47     }
```

*Une méthode "**addCompte**" prenant en paramètre **l'id du compte** à ajouter au client.*

```
49     // méthode pour ajouter un compte à la liste des comptes du client
50     public void addCompte(int idCompte) {
51         this.listeComptesId.add(idCompte); // ajoute l'ID du compte à la liste
52     }
```

*Une méthode "**removeCompte**" prenant en paramètre **l'id du compte** à retirer au client.*

```
54     // méthode pour retirer un compte de la liste des comptes du client
55     public void removeCompte(Compte compte) {
56         this.listeComptesId.remove(compte.getId()); // retire l'ID du compte de la liste
57     }
```

1.2.3 La classe Banque

Les attributs

Ayant pour attribut :

Un id, dont le numéro sera augmenté automatiquement à la création de chaque nouvelle banque

Un nom

Un dictionnaire permettant de retrouver un client en fonction de son id

Un dictionnaire permettant de retrouver un compte en fonction de son id

```
1 import java.util.HashMap;
2 import java.util.Map;
3 import java.util.Set;
4
5 // définition de la classe Banque
6 public class Banque {
7     private static int dernierId; // ID unique pour la dernière banque créée
8     private int id; // ID unique de cette banque
9     private String nom; // nom de la banque
10    private Map<Integer, Client> clientsParId; // dictionnaire associant ID de client et
    Client
11    private Map<Integer, Compte> comptesParId; // dictionnaire associant ID de compte et
    Compte
```

Les méthodes

Un constructeur prenant en paramètre le nom de la banque

```
13 // constructeur pour initialiser une nouvelle banque
14 public Banque(String nom) {
15     this.id=++dernierId;
16     this.nom = nom;
17     this.clientsParId = new HashMap<>(); // initialisation du dictionnaire des clients
18     this.comptesParId = new HashMap<>(); // initialisation du dictionnaire des comptes
19 }
```

Une méthode "createClient" qui prend en paramètre le nom et prénom du nouveau client, le crée et le retourne.

```

38 // crée et ajoute un nouveau client à la banque
39 public Client createClient(String nom, String prenom) {
40     Client nouveauClient = new Client(nom, prenom); // création d'un nouveau client
41     clientsParId.put(nouveauClient.getId(), nouveauClient); // ajout du client au
    dictionnaire
42     return nouveauClient;
43 }

```

Une méthode "**addClient**" qui prend un **client** en paramètre et l'ajoute au dictionnaire des clients de la banque.

```

45 // ajoute un client existant à la banque
46 public void addClient(Client client) {
47     clientsParId.put(client.getId(), client);
48 }

```

Une méthode "**removeClient**" qui prend un **client** en paramètre et le supprime du dictionnaire des clients de la banque, ainsi que tous ses comptes non partagés. Attention, tous les comptes ne possédant que ce client doivent être supprimés. Dans le cas de comptes conjoints, seul ce propriétaire doit l'être.

```

50 // supprime un client et ses comptes associés si nécessaire
51 public void removeClient(Client client) {
52     clientsParId.remove(client.getId()); // supprime le client du dictionnaire
53     // parcourt tous les IDs de comptes associés au client
54     for (Integer compteId : client.getComptes()) {
55         Compte compte = comptesParId.get(compteId); // récupère l'objet compte
56         Set<Integer> clientsDuCompte = compte.getListeClients();
57         // récupère tous les clients associés au compte
58         if (clientsDuCompte.size() > 1) { // si le compte a plus d'un client
59             clientsDuCompte.remove(client.getId()); // on supprime seulement la réf à
    notre client
60         } else { // si le compte n'a qu'un seul client, on supprime le compte
61             comptesParId.remove(compteId);
62         }
63     }
64 }

```

Une méthode "**findClient**" qui prend l'id d'un **client** en paramètre et renvoie le client associé.

```

66 // trouve un client par son ID
67 public Client findClient(int id) {
68     return clientsParId.get(id);
69 }

```

Une méthode "**createCompte**" qui prend un **client** passé en paramètre afin de créer un compte et de le retourner.

```

71 // crée et ajoute un nouveau compte pour un client donné à la banque
72 public Compte createCompte(Client client) {
73     Compte nouveauCompte = new Compte(client); // création d'un nouveau compte
74     comptesParId.put(nouveauCompte.getId(), nouveauCompte); // ajout du compte au
    dictionnaire
75     client.addCompte(nouveauCompte.getId()); // associé le compte au client
76     return nouveauCompte;
77 }

```

Une méthode "addCompte" prenant en paramètre un **compte** et l'ajoutant au dictionnaire des comptes disponibles au sein de la banque.

```

80 // ajoute un compte à la banque
81 public void addCompte(Compte comptePourAjouter) {
82     comptesParId.put(comptePourAjouter.getId(), comptePourAjouter);
83 }

```

Une méthode "removeCompte" prenant un **compte** en paramètre, supprimant celui-ci de la banque et des listes de tous les clients possédant celui-ci.

```

85 // supprime un compte de la banque et des clients associés
86 public void removeCompte(Compte comptePourSupprimer) {
87     comptesParId.remove(comptePourSupprimer.getId());
88     for (Client client : clientsParId.values()) { // pour chaque client de la banque
89         if (client.getComptes().contains(comptePourSupprimer.getId())) {
90             // si le client possède le compte à supprimer
91             client.removeCompte(comptePourSupprimer); // on le supprime de sa liste des
    comptes
92         }
93     }
94 }

```

Une méthode "findCompte" permettant de trouver et retourner un compte depuis le dictionnaire de la banque en fonction de son **id** passé en paramètre.

```

96 // trouve un compte par son ID
97 public Compte findCompte(int id) {
98     return comptesParId.get(id);
99 }

```

Une méthode "afficherListeCompte" affichant la liste de tous les comptes de la banque.

```

101 // affiche la liste des comptes de la banque
102 public void afficherListeCompte() {
103     for (Compte compte : comptesParId.values()) {

```

```

104         System.out.printf("%s\n", compte.toString());
105     }
106 }

```

Une méthode "**afficherListeClient**" affichant la liste de tous les clients de la banque.

```

108 // affiche la liste des clients de la banque
109 public void afficherListeClient() {
110     for (Client client : clientsParId.values()) {
111         System.out.printf("%s\n", client.toString());
112     }
113 }

```

Une méthode "**afficheListeCompteClient**" affichant la liste des comptes pour un client passé en paramètre.

```

115 // affiche la liste des comptes d'un client spécifique
116 public void afficherListeCompteClient(Client client) {
117     for (Integer compteId : client.getComptes()) {
118         Compte compte = comptesParId.get(compteId);
119         System.out.printf("%s\n", compte.toString());
120     }
121 }

```

Des **getters** pour les attributs id, nom, dictionnaire de compte et dictionnaire de client.

```

21 // méthodes getters
22 public int getId() {
23     return id;
24 }
25
26 public String getNom() {
27     return nom;
28 }
29
30 public Map<Integer, Client> getListeClients() {
31     return new HashMap<>(clientsParId);
32 }
33
34 public Map<Integer, Compte> getListeComptes() {
35     return new HashMap<>(comptesParId);
36 }

```

1.3 Les modifications apportées aux tests proposés

1.3.1 Le test 1 - testAddClient()

Dans le premier test, nous avons corrigé une faute d'orthographe afin de faire apparaître le nombre de clients au pluriel.

```
50  @Test
51  @Order(1)
52  public void testAddClient() throws Exception {
53      int nbClient = banque.getListeClients().keySet().size();
54      System.out.println("\nTEST 1 : AJOUT CLIENT");
55      System.out.println("Nombre de clients dans la banque : " + (nbClient)); // pluriel
56      System.out.println("Ajout d'un client tété à la banque");
57
58      Client c = new Client("TETE", "tété");
59      banque.addClient(c);
60
61      System.out.println("Nombre de clients dans la banque : " + (banque.getListeClients().
62      keySet().size())); // pluriel
63
64      assertEquals(nbClient+1, banque.getListeClients().keySet().size()); // toto, titi, tutu +
65      tété
66  }
```

1.3.2 Le test 2 - testAddCompte()

Dans le test 2, nous avons corrigé une faute d'orthographe afin de faire apparaître le nombre de comptes au pluriel. Par ailleurs, nous avons corrigé une erreur puisqu'il ne s'agissait pas dans ce test du nombre de clients (comme cela apparaissait initialement dans le test), mais du nombre de comptes.

```
67  @Test
68  @Order(2)
69  public void testAddCompte() throws Exception {
70      int nbCompte = banque.getListeComptes().keySet().size();
71      System.out.println("\nTEST 2 : AJOUT COMPTE");
72      System.out.println("Nombre de comptes dans la banque : " + (nbCompte)); // pluriel
73      System.out.println("Ajout d'un compte à la banque");
74
75      Compte c5 = new Compte(titi);
76      banque.addCompte(c5);
77
78      System.out.println("Nombre de comptes dans la banque : " + (banque.getListeComptes().
79      keySet().size()));
80      // on corrige "nombre de client" en "nombre de comptes" pour respecter la logique
81      assertEquals(nbCompte+1, banque.getListeComptes().keySet().size()); // c1, c2, c3, c4 + c5
82  }
83
84  @Test
85  @Order(3)
86  public void testFindClient() throws Exception {
87      System.out.println("\nTest 3 : Recupérer id");
88  }
```

1.3.3 Le test 11 - testAddProprioNotNormal()

Dans le test 11, nous avons ajouté une ligne de code supplémentaire qui met à jour et affiche le nombre de propriétaires du compte avant de tenter d'ajouter un troisième propriétaire. Cela rend le test plus clair. Cette ligne montre que le nombre de propriétaires passe de 1 à 2. L'exception est levée lorsque le code tente d'ajouter

un troisième propriétaire, ce qui n'est pas autorisé selon la logique du programme.

```
225 @Test
226 @Order(11)
227 public void testAddProprioNotNormal() throws Exception {
228     try {
229         System.out.println("\nTEST 11 : AJOUT PROPRIETAIRE (ANORMAL)");
230         System.out.println("Nombre de proprietaire du compte c1 : [" + (c1.getIdClient().size())
231             + "]");
232         System.out.println("Ajout d'un autre propriétaire");
233         c1.addClient(tutu);
234         System.out.println("Nombre de proprietaire du compte c1 : [" + (c1.getIdClient().size())
235             + "]");
236         // ajout d'une impression pour mise à jour du nombre de proprio
237         c1.addClient(titi);
238         fail();
239     }
```

1.3.4 Le test 12 - testRemoveProprio()

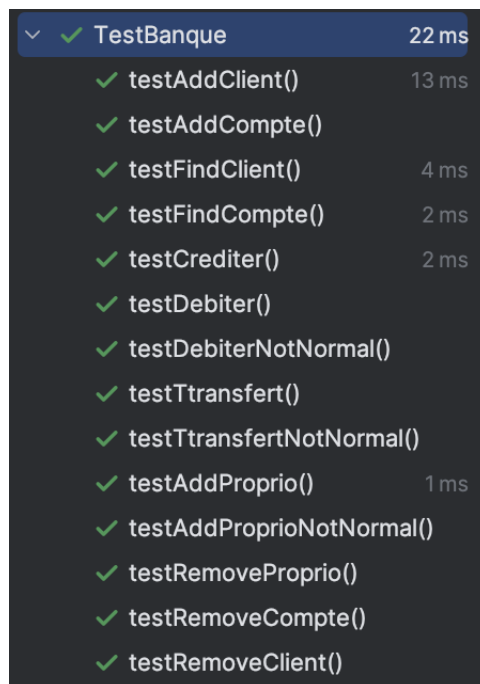
Le test 12, tel qu'il était initialement conçu, ne fonctionnait pas comme prévu. Les tests utilisent l'annotation `@BeforeEach`, qui réinitialise les valeurs avant chaque test, réutilisant ainsi les conditions initiales. Au début de ce test, seul `toto` est propriétaire du compte C1. Le test 12 original comptabilisait le nombre de propriétaires de C1, trouvant un seul propriétaire, `toto`. Ensuite, il tentait de retirer `titi` de la liste des propriétaires, une opération inutile puisque `titi` n'était pas un propriétaire de C1, laissant le nombre de propriétaires inchangé à 1.

Pour améliorer ce test, nous avons intégré une étape supplémentaire en ajoutant `titi` comme co-propriétaire du compte C1, portant ainsi le nombre de propriétaires à deux. En retirant ensuite `titi`, le test devient significatif, montrant une réduction du nombre de propriétaires de 2 à 1, ce qui correspond au résultat attendu.

```
245 @Test
246 @Order(12)
247 public void testRemoveProprio() throws Exception {
248     System.out.println("\nTEST 12 : SUPPRIMER PROPRIETAIRE");
249     c1.addClient(titi); // on ajoute un autre propriétaire sinon en raison de @Beforeeach
250     // c'est remis à 1 seul propriétaire (toto) comme au départ et le test est sans intérêt
251     System.out.println("Nombre de proprietaire du compte c1 : [" + (c1.getIdClient().size()) +
252         "]");
253     System.out.println("Suppression du propriétaire titi");
254
255     c1.removeClient(titi);
256
257     System.out.println("Nombre de proprietaire du compte c1 : [" + (c1.getIdClient().size()) +
258         "]");
259     assertEquals(1, c1.getIdClient().size());
260 }
```


1.4 Les tests

L'ensemble des tests ont réussi.



A screenshot of a test runner interface, likely IntelliJ IDEA, showing a list of tests under the 'TestBanque' package. All tests are marked with a green checkmark, indicating they passed. The tests and their durations are as follows:

Test Name	Duration
TestBanque	22 ms
testAddClient()	13 ms
testAddCompte()	
testFindClient()	4 ms
testFindCompte()	2 ms
testCrediter()	2 ms
testDebiter()	
testDebiterNotNormal()	
testTtransfert()	
testTtransfertNotNormal()	
testAddProprio()	1 ms
testAddProprioNotNormal()	
testRemoveProprio()	
testRemoveCompte()	
testRemoveClient()	

FIGURE 1 – Tous les tests sont réussis.

1.4.1 Test1 - testAddClient()

```
50  @Test
51  @Order(1)
52  public void testAddClient() throws Exception {
53      int nbClient = banque.getListeClients().keySet().size();
54      System.out.println("\nTEST 1 : AJOUT CLIENT");
55      System.out.println("Nombre de clients dans la banque : " + (nbClient)); // pluriel
56      System.out.println("Ajout d'un client tété à la banque");
57
58      Client c = new Client("TETE", "tété");
59      banque.addClient(c);
60
61      System.out.println("Nombre de clients dans la banque : " + (banque.getListeClients().
62      keySet().size())); // pluriel
63
64      assertEquals(nbClient+1, banque.getListeClients().keySet().size()); // toto, titi, tutu +
65      tété
66  }
```

```

TEST 1 : AJOUT CLIENT
Nombre de clients dans la banque : 3
Ajout d'un client tété à la banque
Nombre de clients dans la banque : 4

```

FIGURE 2 – Le test 1 a réussi.

1.4.2 Test2 - testAddCompte()

```

67  @Test
68  @Order(2)
69  public void testAddCompte() throws Exception {
70      int nbCompte = banque.getListeComptes().keySet().size();
71      System.out.println("\nTEST 2 : AJOUT COMPTE");
72      System.out.println("Nombre de comptes dans la banque : " + (nbCompte)); // pluriel
73      System.out.println("Ajout d'un compte à la banque");
74
75      Compte c5 = new Compte(titi);
76      banque.addCompte(c5);
77
78      System.out.println("Nombre de comptes dans la banque : " + (banque.getListeComptes().
79      keySet().size()));
80      // on corrige "nombre de client" en "nombre de comptes" pour respecter la logique
81      assertEquals(nbCompte+1, banque.getListeComptes().keySet().size()); // c1, c2, c3, c4 + c5

```

```

TEST 2 : AJOUT COMPTE
Nombre de comptes dans la banque : 4
Ajout d'un compte à la banque
Nombre de comptes dans la banque : 5

```

FIGURE 3 – Le test 2 a réussi.

1.4.3 Test3 - testFindClient()

```

83  @Test
84  @Order(3)
85  public void testFindClient() throws Exception {
86      System.out.println("\nTest 3 : Recupérer id");
87
88      // On recupère l'id de TOTO
89      int idCompte = toto.getId();
90      System.out.println("Clients de la banque " + banque.getListeClients());
91      System.out.println("TOTO ID = " + idCompte);
92
93      // On cherche le client avec cette ID pour verifier que son prenom est bien toto
94      assertEquals("Toto", banque.findClient(idCompte).getPrenom());
95  }

```

```

Test 3 : Recupérer id
Clients de la banque {8=Client{id=8, nom='TOTO', prenom='Toto', li
TOTO ID = 8

```

FIGURE 4 – Le test 3 a réussi.

1.4.4 Test4 - testFindCompte()

```

97  @Test
98  @Order(4)
99  public void testFindCompte() throws Exception {
100      System.out.println("\nTest 4 : Recupérer compte");
101      System.out.println("Comptes de la banque " + banque.getListeComptes());
102      System.out.println("Info compte C1 = " + banque.findCompte(c1.getId()));
103
104
105      // Vérification que le compte C1 n'a qu'un propriétaire
106      assertEquals(1, banque.findCompte(c1.getId()).getIdClient().size());
107
108      // Véification que celui-ci soit bien toto
109      List<Integer> listeClient = new ArrayList<Integer>(banque.findCompte(c1.getId()).
110      getIdClient());
111      int idClient = listeClient.get(0);
112      System.out.println("Clients de la banque " + banque.getListeClients());
113      System.out.println("Proprietaire de C1 = " + banque.findClient(idClient).getPrenom());
114
115      assertEquals("Toto", banque.findClient(idClient).getPrenom());
116  }

```

```

Test 4 : Recupérer compte
Comptes de la banque {16=Compte{id=16, solde=0.0, listeIdClients=[
Info compte C1 = Compte{id=14, solde=0.0, listeIdClients=[11]}
Clients de la banque {12=Client{id=12, nom='TITI', prenom='Titi',
Proprietaire de C1 = Toto

```

FIGURE 5 – Le test 4 a réussi.

1.4.5 Test5 - testCrediter()

```

117  @Test
118  @Order(5)
119  public void testCrediter() throws Exception {
120      int montant = 100;
121      System.out.println("\nTEST 5 : CREDITER");
122      System.out.println("Solde du compte c1 avant ajout : " + c1.getSolde());
123
124      c1.crediter(montant);
125      System.out.println("Solde du compte c1 après ajout de " + (montant) + "€ -> c1 = " + c1.
126      getSolde());
127
128      assertEquals(montant, c1.getSolde());
129  }

```

```
TEST 5 : CREDITER
Solde du compte c1 avant ajout : 0.0
Solde du compte c1 après ajout de 100€ -> c1 = 100.0
```

FIGURE 6 – Le test 5 a réussi.

1.4.6 Test6 - testDebiter()

```
131 @Test
132 @Order(6)
133 public void testDebiter() throws Exception {
134     int montant = 50;
135
136     c2.crediter(100);
137     System.out.println("\nTEST 6 : DEBITER (NORMAL)");
138     System.out.println("Solde du compte c2 avant retrait : " + c2.getSolde());
139
140     c2.debiter(montant);
141
142     System.out.println("Solde du compte c2 après retrait de " + (montant) + "€ -> c2 = " +
143         c2.getSolde());
144
145     assertEquals(montant, c2.getSolde());
146 }
```

```
TEST 6 : DEBITER (NORMAL)
Solde du compte c2 avant retrait : 100.0
Solde du compte c2 après retrait de 50€ -> c2 = 50.0
```

FIGURE 7 – Le test 6 a réussi.

1.4.7 Test7 - testDebiterNotNormal()

```
149 @Test
150 @Order(7)
151 public void testDebiterNotNormal() throws Exception {
152     try {
153         System.out.println("\nTEST 7 : DEBITER (ANORMAL)");
154         System.out.println("Solde du compte c1 avant retrait : " + c1.getSolde());
155         System.out.println("Retrait de 999€");
156         c1.debiter(999);
157         fail();
158     }
159     catch (IllegalOperationException e) {
160         System.out.println(e.getMessage());
161         System.out.println("Le débit d'un montant non possédé à bien été bloqué !\n");
162     }
163 }
```

```

TEST 7 : DEBITER (ANORMAL)
Solde du compte c1 avant retrait : 0.0
Retrait de 999€
Solde insuffisant.
Le débit d'un montant non possédé à bien été bloqué !

```

FIGURE 8 – Le test 7 a réussi.

1.4.8 Test8 - testTransfert

```

166 @Test
167 @Order(8)
168 public void testTtransfert() throws Exception {
169     c3.crediter(100);
170     System.out.println("\nTEST 8 : TRANSFERT (NORMAL)");
171     System.out.println("Solde du compte c3 avant transfert : " + c3.getSolde());
172     System.out.println("Solde du compte c4 avant transfert : " + c4.getSolde());
173     System.out.println("Transfert de 50€ de c3 -> c4");
174
175     c3.transfert(c4, 50);
176
177     System.out.println("Solde du compte c3 : " + c3.getSolde());
178     System.out.println("Solde du compte c4 : " + c4.getSolde());
179
180     assertEquals(50, c3.getSolde());
181     assertEquals(50, c4.getSolde());
182
183 }

```

```

TEST 8 : TRANSFERT (NORMAL)
Solde du compte c3 avant transfert : 100.0
Solde du compte c4 avant transfert : 0.0
Transfert de 50€ de c3 -> c4
Solde du compte c3 : 50.0
Solde du compte c4 : 50.0

```

FIGURE 9 – Le test 8 a réussi.

1.4.9 Test9 - testTransfertNotNormal()

```

186 @Test
187 @Order(9)
188 public void testTtransfertNotNormal() throws Exception {
189     try {
190         System.out.println("\nTEST 9 : TRANSFERT ANORMAL");
191         System.out.println("Solde du compte c1 avant transfert : " + c1.getSolde());
192         System.out.println("Transfert de 999€");
193         c1.transfert(c2, 999);
194         fail();
195     }
196     catch (IllegalOperationException e) {

```

```

197     System.out.println(e.getMessage());
198     System.out.println("Le transfert d'un montant non posséd      bien   t   bloqu   !\n");
199 }
200 }

```

```

TEST 9 : TRANSFERT ANORMAL
Solde du compte c1 avant transfert : 0.0
Transfert de 999  
Solde insuffisant.
Le transfert d'un montant non poss  d      bien   t   bloqu   !

```

FIGURE 10 – Le test 9 a r  ussi.

1.4.10 Test10 - testAddProprio()

```

203 @Test
204 @Order(10)
205 public void testAddProprio() throws Exception {
206     try {
207         System.out.println("\nTEST 10 : AJOUT PROPRIETAIRE (NORMAL)");
208         List<Integer> listeClient = new ArrayList<Integer>(c1.getIdClient());
209         System.out.println("Proprietaire du compte c1 : [" + (c1.getIdClient().size()) + "] = " +
            (banque.findClient(listeClient.get(0))));
210         System.out.print("ajout du proprietaire titi \n");
211
212         c1.addClient(titi);
213         listeClient = new ArrayList<Integer>(c1.getIdClient());
214         System.out.println("Proprietaire du compte c1 : [" + (c1.getIdClient().size()) + "] = " +
            banque.findClient(listeClient.get(0)) + " , " + banque.findClient(listeClient.get(1)));
215
216         assertEquals(2, c1.getIdClient().size());
217     }
218     catch (IllegalOperationException e) {
219         e.printStackTrace();
220         fail();
221     }
222 }

```

```

TEST 10 : AJOUT PROPRIETAIRE (NORMAL)
Proprietaire du compte c1 : [1] = Client{id=29, nom='TOTO', prenom
ajout du proprietaire titi
Proprietaire du compte c1 : [2] = Client{id=29, nom='TOTO', prenom

```

FIGURE 11 – Le test 10 a r  ussi.

1.4.11 Test11 - testAddProprioNotNormal()

```

225 @Test
226 @Order(11)

```

```

227 public void testAddProprioNotNormal() throws Exception {
228     try {
229         System.out.println("\nTEST 11 : AJOUT PROPRIETAIRE (ANORMAL)");
230         System.out.println("Nombre de proprietaire du compte c1 : [" + (c1.getIdClient().size())
231             + "]");
232         System.out.println("Ajout d'un autre proprietaire");
233         c1.addClient(tutu);
234         System.out.println("Nombre de proprietaire du compte c1 : [" + (c1.getIdClient().size())
235             + "]");
236         // ajout d'une impression pour mise à jour du nombre de proprio
237         c1.addClient(titi);
238         fail();
239     } catch (IllegalOperationException e) {
240         System.out.println(e.getMessage());
241         System.out.println("L'ajout d'un 3e beneficiare à bien été bloqué");
242     }
243 }

```

```

TEST 11 : AJOUT PROPRIETAIRE (ANORMAL)
Nombre de proprietaire du compte c1 : [1]
Ajout d'un autre proprietaire
Nombre de proprietaire du compte c1 : [2]
Ce compte en banque a déjà deux clients.
L'ajout d'un 3e beneficiare à bien été bloqué

```

FIGURE 12 – Le test 11 a réussi.

1.4.12 Test12 - testRemoveProprio()

```

245 @Test
246 @Order(12)
247 public void testRemoveProprio() throws Exception {
248     System.out.println("\nTEST 12 : SUPPRIMER PROPRIETAIRE");
249     c1.addClient(titi); // on ajoute un autre proprietaire sinon en raison de @Beforeeach
250     // c'est remis à 1 seul proprietaire (toto) comme au départ et le test est sans intérêt
251     System.out.println("Nombre de proprietaire du compte c1 : [" + (c1.getIdClient().size()) +
252         "]");
253     System.out.println("Suppression du proprietaire titi");
254
255     c1.removeClient(titi);
256
257     System.out.println("Nombre de proprietaire du compte c1 : [" + (c1.getIdClient().size()) +
258         "]");
259     assertEquals(1, c1.getIdClient().size());
260 }

```

```

TEST 12 : SUPPRIMER PROPRIETAIRE
Nombre de proprietaire du compte c1 : [2]
Suppression du proprietaire titi
Nombre de proprietaire du compte c1 : [1]

```

FIGURE 13 – Le test 12 a réussi.

1.4.13 Test13 - testRemoveCompte()

```
261 @Test
262 @Order(13)
263 public void testRemoveCompte() throws Exception {
264     c1.addClient(titi);
265
266     System.out.println("\nTEST 13 : SUPPRIMER COMPTE");
267     System.out.println("Nombre de compte à la banque = " + (banque.getListeComptes().keySet().size()));
268     System.out.println("Suppression du compte c1 ayant pour clients les IdClients : " + c1.getIdClient());
269     System.out.println("Client de la banque possédant ce compte :");
270
271     for(Integer idClient : banque.getListeClients().keySet()) {
272         if(banque.getListeClients().get(idClient).getComptes().contains(c1.getId())) {
273             System.out.println(banque.getListeClients().get(idClient).getPrenom());
274         }
275     }
276
277     // Suppression du compte de la banque
278     banque.removeCompte(c1);
279
280     System.out.println("Nombre de compte à la banque après suppression = " + (banque.getListeComptes().keySet().size()));
281     System.out.println("Client de la banque possédant ce compte :");
282
283     for(Integer idClient : banque.getListeClients().keySet()) {
284         if(banque.getListeClients().get(idClient).getComptes().contains(c1.getId())) {
285             System.out.println(banque.getListeClients().get(idClient).getPrenom());
286         }
287     }
288
289     // verification suppression du compte
290     assertEquals(3, banque.getListeComptes().keySet().size()); // c2 + c3 + c4
291
292     // verification suppression du compte au sein du client
293     assertEquals(0, toto.getComptes().size());
294     assertEquals(2, titi.getComptes().size());
295 }
296 }
```

```
TEST 13 : SUPPRIMER COMPTE
Nombre de compte à la banque = 4
Suppression du compte c1 ayant pour clients les IdClients :[38, 39
Client de la banque possédant ce compte :
Toto
Titi
Nombre de compte à la banque après suppression = 3
Client de la banque possédant ce compte :
```

FIGURE 14 – Le test 13 a réussi.

1.4.14 Test14 - testRemoveClient()

```
299 @Test
300 @Order(14)
301 public void testRemoveClient() throws Exception {
302     System.out.println("\n\nTEST 14 : SUPPRIMER CLIENT");
```



```

303 // Ajout du client toto au compte 2
304 c2.addClient(toto);
305 System.out.println("Nombre de client : " + (banque.getListeClients().keySet().size()));
306 System.out.println("Nombre de comptes : " + (banque.getListeComptes().keySet().size()));
307 System.out.print("Compte appartenant uniquement à titi: ");
308 for(Integer idCompte : titi.getComptes()) {
309     if(banque.getListeComptes().get(idCompte).getIdClient().size() == 1) {
310         System.out.print((idCompte) + " ");
311     }
312 }
313
314
315 System.out.println("\nSuppression de titi");
316
317 banque.removeClient(titi);
318 System.out.println("Nombre de client après suppression : " + (banque.getListeClients().
319 keySet().size()));
320 System.out.println("Nombre de comptes après suppression : " + (banque.getListeComptes().
321 keySet().size()));
322
323 assertEquals(2, banque.getListeClients().keySet().size()); // toto + tutu + tété
324 assertEquals(3, banque.getListeComptes().keySet().size()); //c1 + c2 + c4
325 assertEquals(1, c1.getIdClient().size());
326 }

```

```

TEST 14 : SUPPRIMER CLIENT
Nombre de client : 3
Nombre de comptes : 4
Compte appartenant uniquement à titi: 56
Suppression de titi
Nombre de client après suppression : 2
Nombre de comptes après suppression : 3

```

FIGURE 15 – Le test 14 a réussi.

1.5 L'auto-évaluation

Pour finir, votre TP devra être accompagné d'une auto évaluation disponible en commentaire de votre dépôt ou directement dans un readme trouvable dans votre dossier zip. Cette auto évaluation devra :

- soit indiquer la note finale que vous pensez avoir à ce TP,
- soit indiquer un intervalle dans lequel vous pensez que votre note se situe,
- ou détailler chacun des points que vous pensez avoir en fonction du barème fourni ci-dessous.

Auto-évaluation Je m'estime entre 16 et 20 pour les raisons suivantes :

- Mon code respecte intégralement les consignes du TP.
- J'ai ajouté des méthodes supplémentaires pour répondre avec précision aux exigences des tests.
- J'ai apporté des améliorations notables à certains tests, notamment le test 12.
- Le code compile sans aucun avertissement.
- Tous les tests sont passés avec succès, démontrant la fiabilité et la robustesse du code.

Cette auto-évaluation reflète mon engagement à fournir un travail de qualité tout en respectant les consignes données.

Annexes

.1 Le fichier "Main.java"

Voici l'intégralité du fichier Main.java.

```
1 //TIP To <b>Run</b> code, press <shortcut actionId="Run"/> or
2 // click the <icon src="AllIcons.Actions.Execute"/> icon in the gutter.
3 public class Main {
4     public static void main(String[] args) {
5
6     }
7 }
```

.2 Le fichier "Banque.java"

Voici l'intégralité du fichier Banque.java.

```
1 import java.util.HashMap;
2 import java.util.Map;
3 import java.util.Set;
4
5 // définition de la classe Banque
6 public class Banque {
7     private static int dernierId; // ID unique pour la dernière banque créée
8     private int id; // ID unique de cette banque
9     private String nom; // nom de la banque
10    private Map<Integer, Client> clientsParId; // dictionnaire associant ID de client et Client
11    private Map<Integer, Compte> comptesParId; // dictionnaire associant ID de compte et Compte
12
13    // constructeur pour initialiser une nouvelle banque
14    public Banque(String nom) {
15        this.id = ++dernierId;
16        this.nom = nom;
17        this.clientsParId = new HashMap<>(); // initialisation du dictionnaire des clients
18        this.comptesParId = new HashMap<>(); // initialisation du dictionnaire des comptes
19    }
20
21    // méthodes getters
22    public int getId() {
23        return id;
24    }
25
26    public String getNom() {
27        return nom;
28    }
29
30    public Map<Integer, Client> getListeClients() {
31        return new HashMap<>(clientsParId);
32    }
33
34    public Map<Integer, Compte> getListeComptes() {
35        return new HashMap<>(comptesParId);
36    }
37
38    // crée et ajoute un nouveau client à la banque
39    public Client createClient(String nom, String prenom) {
40        Client nouveauClient = new Client(nom, prenom); // création d'un nouveau client
41        clientsParId.put(nouveauClient.getId(), nouveauClient); // ajout du client au dictionnaire
42        return nouveauClient;
43    }
44
45    // ajoute un client existant à la banque
46    public void addClient(Client client) {
47        clientsParId.put(client.getId(), client);
48    }
49
50    // supprime un client et ses comptes associés si nécessaire
51    public void removeClient(Client client) {
52        clientsParId.remove(client.getId()); // supprime le client du dictionnaire
53        // parcourt tous les IDs de comptes associés au client
54        for (Integer compteId : client.getComptes()) {
55            Compte compte = comptesParId.get(compteId); // récupère l'objet compte
56            Set<Integer> clientsDuCompte = compte.getListeClients();
57            // récupère tous les clients associés au compte
58            if (clientsDuCompte.size() > 1) { // si le compte a plus d'un client
59                clientsDuCompte.remove(client.getId()); // on supprime seulement la réf à
notre client
60            } else { // si le compte n'a qu'un seul client, on supprime le compte
61                comptesParId.remove(compteId);
62            }
63        }
64    }
65
66    // trouve un client par son ID
```

```

67     public Client findClient(int id) {
68         return clientsParId.get(id);
69     }
70
71     // crée et ajoute un nouveau compte pour un client donné à la banque
72     public Compte createCompte(Client client) {
73         Compte nouveauCompte = new Compte(client); // création d'un nouveau compte
74         comptesParId.put(nouveauCompte.getId(), nouveauCompte); // ajout du compte au
dictionnaire
75         client.addCompte(nouveauCompte.getId()); // associé le compte au client
76         return nouveauCompte;
77     }
78
79
80     // ajoute un compte à la banque
81     public void addCompte(Compte comptePourAjouter) {
82         comptesParId.put(comptePourAjouter.getId(), comptePourAjouter);
83     }
84
85     // supprime un compte de la banque et des clients associés
86     public void removeCompte(Compte comptePourSupprimer) {
87         comptesParId.remove(comptePourSupprimer.getId());
88         for (Client client : clientsParId.values()) { // pour chaque client de la banque
89             if (client.getComptes().contains(comptePourSupprimer.getId())) {
90                 // si le client possède le compte à supprimer
91                 client.removeCompte(comptePourSupprimer); // on le supprime de sa liste des
comptes
92             }
93         }
94     }
95
96     // trouve un compte par son ID
97     public Compte findCompte(int id) {
98         return comptesParId.get(id);
99     }
100
101     // affiche la liste des comptes de la banque
102     public void afficherListeCompte() {
103         for (Compte compte : comptesParId.values()) {
104             System.out.printf("%s\n", compte.toString());
105         }
106     }
107
108     // affiche la liste des clients de la banque
109     public void afficherListeClient() {
110         for (Client client : clientsParId.values()) {
111             System.out.printf("%s\n", client.toString());
112         }
113     }
114
115     // affiche la liste des comptes d'un client spécifique
116     public void afficherListeCompteClient(Client client) {
117         for (Integer compteId : client.getComptes()) {
118             Compte compte = comptesParId.get(compteId);
119             System.out.printf("%s\n", compte.toString());
120         }
121     }
122
123 }

```

.3 Le fichier "Client.java"

Voici l'intégralité du fichier Client.java.

```
1 import java.util.HashSet;
2 import java.util.Set;
3
4 // définition de la classe Client
5 public class Client {
6     // variables statiques et d'instance
7     private static int dernierId; // compteur statique pour l'ID du dernier client créé
8     private int id; // ID unique pour ce client
9     private String nom; // nom du client
10    private String prenom; // prénom du client
11    private Set<Integer> listeComptesId; // ensemble des IDs des comptes associés à ce client
12
13    // constructeur pour initialiser un nouveau client
14    public Client(String nom, String prenom) {
15        this.id = ++dernierId; // incrémente et attribue l'ID au client
16        this.nom = nom; // initialise le nom
17        this.prenom = prenom; // initialise le prénom
18        this.listeComptesId = new HashSet<>(); // initialise l'ensemble des comptes du client
19    }
20
21    // méthode pour représenter le client sous forme de chaîne de caractères
22    @Override
23    public String toString() {
24        return "Client{" +
25            "id=" + id +
26            ", nom='" + nom + '\'' +
27            ", prenom='" + prenom + '\'' +
28            ", listeComptesId=" + listeComptesId +
29            '}';
30    }
31
32    // getters pour accéder aux propriétés du client
33    public int getId() {
34        return id;
35    }
36
37    public String getNom() {
38        return nom;
39    }
40
41    public String getPrenom() {
42        return prenom;
43    }
44
45    public Set<Integer> getComptes() {
46        return listeComptesId; // obtenir la liste des IDs des comptes du client
47    }
48
49    // méthode pour ajouter un compte à la liste des comptes du client
50    public void addCompte(int idCompte) {
51        this.listeComptesId.add(idCompte); // ajoute l'ID du compte à la liste
52    }
53
54    // méthode pour retirer un compte de la liste des comptes du client
55    public void removeCompte(Compte compte) {
56        this.listeComptesId.remove(compte.getId()); // retire l'ID du compte de la liste
57    }
58
59
60 }
```

.4 Le fichier "Compte.java"

Voici l'intégralité du fichier `Compte.java`.

```
1 import java.util.HashSet;
2 import java.util.Set;
3
4 // définition de la classe Compte
5 public class Compte {
6     // variables statiques et d'instance
7     private static int dernierId; // ID unique pour le dernier compte créé
8     private int id; // ID unique pour ce compte
9     private double solde; // solde du compte
10    private Set<Integer> listeIdClients; // ensemble des IDs des clients associés à ce compte
11
12    // constructeur pour initialiser un nouveau compte
13    public Compte(Client client) {
14        this.id = ++dernierId; // incrémente et attribue l'ID
15        this.solde = 0; // initialise le solde à zéro
16        this.listeIdClients = new HashSet<>(); // initialise la liste des clients
17        this.listeIdClients.add(client.getId()); // ajoute le client au compte
18        client.addCompte(this.getId()); // lie le compte au client
19    }
20
21    // getters pour accéder aux propriétés du compte
22    public int getId() {
23        return id;
24    }
25
26    public double getSolde() {
27        return solde;
28    }
29
30    public Set<Integer> getListeClients() {
31        return listeIdClients;
32    }
33
34    // getter pour obtenir les IDs des clients associés au compte
35    public Set<Integer> getIdClient() {
36        return new HashSet<>(listeIdClients); // retourne une copie de la liste des clients
37    }
38
39    // méthode pour représenter le compte sous forme de chaîne de caractères
40    @Override
41    public String toString() {
42        return "Compte{" +
43            "id=" + id +
44            ", solde=" + solde +
45            ", listeIdClients=" + listeIdClients +
46            '}';
47    }
48
49    // méthode pour créditer le compte
50    public void crediter(double montant){
51        this.solde += montant; // ajoute le montant au solde
52    }
53
54    // méthode pour débiter le compte
55    public void debiter(double montant) throws IllegalArgumentException {
56        if (this.solde >= montant) { // vérifie si le solde est suffisant
57            this.solde -= montant; // soustrait le montant du solde
58        } else { // lance une exception personnalisée si le solde est insuffisant
59            throw new IllegalArgumentException("Solde insuffisant.");
60        }
61    }
62
63
64    // méthode pour transférer de l'argent vers un autre compte
65    public void transfert(Compte compte, double montant) throws IllegalArgumentException{
66        this.debiter(montant); // débite le montant de ce compte
67        compte.crediter(montant); // crédite le montant sur le compte cible
68    }
69
70    // méthode pour ajouter un client au compte
```

```

71     public void addClient(Client client) throws IOException {
72         if (this.listeIdClients.size() >= 2) { // limite à 2 clients par compte
73             throw new IOException("Ce compte en banque a déjà deux clients.");
74         } else { // ajoute le client à la liste du compte
75             this.listeIdClients.add(client.getId()); // Ajoute le client à la liste du compte
76             client.addCompte(this.id); // on ajoute le compte à la liste des comptes du client
77         }
78     }
79
80
81     // méthode pour retirer un client du compte
82     public void removeClient(Client client) {
83         this.listeIdClients.remove(client.getId()); // retire le client de la liste
84     }
85
86 }

```


.5 Le fichier "IllegalOperationException.java"

Voici l'intégralité du fichier `IllegalOperationException.java`.

```
1 public class IllegalOperationException extends Exception {  
2     // constructeur avec un message d'erreur  
3     public IllegalOperationException(String message) {  
4         super(message);  
5     }  
6 }
```

.6 Le fichier "TestBanque.java"

Voici l'intégralité du fichier TestBanque.java.

```
1 import org.junit.jupiter.api.Test;
2 import org.junit.jupiter.api.TestMethodOrder;
3 import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
4 import org.junit.jupiter.api.Order;
5 import static org.junit.Assert.fail;
6 import static org.junit.jupiter.api.Assertions.assertEquals;
7
8 import java.util.ArrayList;
9 import java.util.List;
10
11 import org.junit.jupiter.api.BeforeEach;
12
13
14
15 @TestMethodOrder(OrderAnnotation.class)
16 public class TestBanque{
17
18     public Banque banque = new Banque("Test Banque");
19
20     public Client toto = new Client("TOTO", "Toto");
21     public Client titi = new Client("TITI", "Titi");
22     public Client tutu = new Client("TUTU", "Tutu");
23
24     public Compte c1 = new Compte(toto);
25     public Compte c2 = new Compte(titi);
26     public Compte c3 = new Compte(titi);
27     public Compte c4 = new Compte(tutu);
28
29
30
31     @BeforeEach
32     // Méthode à exécuter avant chaque tests
33     // Ou @BeforeAll pour exécuter la méthode une seule fois avant tous les tests
34     public void init() {
35
36         // ajout des client à la banque
37         banque.addClient(toto);
38         banque.addClient(titi);
39         banque.addClient(tutu);
40
41         // Ajout du compte dans la banque
42         banque.addCompte(c1);
43         banque.addCompte(c2);
44         banque.addCompte(c3);
45         banque.addCompte(c4);
46     }
47
48
49     // ===== TESTS =====
50     @Test
51     @Order(1)
52     public void testAddClient() throws Exception {
53         int nbClient = banque.getListeClients().keySet().size();
54         System.out.println("\nTEST 1 : AJOUT CLIENT");
55         System.out.println("Nombre de clients dans la banque : " + (nbClient)); // pluriel
56         System.out.println("Ajout d'un client tété à la banque");
57
58         Client c = new Client("TETE", "tété");
59         banque.addClient(c);
60
61         System.out.println("Nombre de clients dans la banque : " + (banque.getListeClients().keySet().size())); // pluriel
62
63         assertEquals(nbClient+1, banque.getListeClients().keySet().size()); // toto, titi, tutu + tété
64     }
65
66
67     @Test
68     @Order(2)
```

```

69 public void testAddCompte() throws Exception {
70     int nbCompte = banque.getListeComptes().keySet().size();
71     System.out.println("\nTEST 2 : AJOUT COMPTE");
72     System.out.println("Nombre de comptes dans la banque : " + (nbCompte)); // pluriel
73     System.out.println("Ajout d'un compte à la banque");
74
75     Compte c5 = new Compte(titi);
76     banque.addCompte(c5);
77
78     System.out.println("Nombre de comptes dans la banque : " + (banque.getListeComptes().
79         keySet().size()));
80     // on corrige "nombre de client" en "nombre de comptes" pour respecter la logique
81     assertEquals(nbCompte+1, banque.getListeComptes().keySet().size()); // c1, c2, c3, c4 + c5
82 }
83
84 @Test
85 @Order(3)
86 public void testFindClient() throws Exception {
87     System.out.println("\nTest 3 : Recupérer id");
88
89     // On récupère l'id de TOTO
90     int idCompte = toto.getId();
91     System.out.println("Clients de la banque " + banque.getListeClients());
92     System.out.println("TOTO ID = " + idCompte);
93
94     // On cherche le client avec cette ID pour verifier que son prenom est bien toto
95     assertEquals("Toto", banque.findClient(idCompte).getPrenom());
96 }
97
98 @Test
99 @Order(4)
100 public void testFindCompte() throws Exception {
101     System.out.println("\nTest 4 : Recupérer compte");
102     System.out.println("Comptes de la banque " + banque.getListeComptes());
103     System.out.println("Info compte C1 = " + banque.findCompte(c1.getId()));
104
105     // Vérification que le compte C1 n'a qu'un propriétaire
106     assertEquals(1, banque.findCompte(c1.getId()).getIdClient().size());
107
108     // Véification que celui-ci soit bien toto
109     List<Integer> listeClient = new ArrayList<Integer>(banque.findCompte(c1.getId()).
110         getIdClient());
111     int idClient = listeClient.get(0);
112     System.out.println("Clients de la banque " + banque.getListeClients());
113     System.out.println("Proprietaire de C1 = " + banque.findClient(idClient).getPrenom());
114
115     assertEquals("Toto", banque.findClient(idClient).getPrenom());
116 }
117
118 @Test
119 @Order(5)
120 public void testCrediter() throws Exception {
121     int montant = 100;
122     System.out.println("\nTEST 5 : CREDITER");
123     System.out.println("Solde du compte c1 avant ajout : " + c1.getSolde());
124
125     c1.crediter(montant);
126     System.out.println("Solde du compte c1 après ajout de " + (montant) + "€ -> c1 = " + c1.
127         getSolde());
128
129     assertEquals(montant, c1.getSolde());
130 }
131
132 @Test
133 @Order(6)
134 public void testDebiter() throws Exception {
135     int montant = 50;
136
137     c2.crediter(100);
138     System.out.println("\nTEST 6 : DEBITER (NORMAL)");
139     System.out.println("Solde du compte c2 avant retrait : " + c2.getSolde());
140
141     c2.debiter(montant);

```

```

142     System.out.println("Solde du compte c2 après retrait de " + (montant) + "€ -> c2 = " +
143     c2.getSolde());
144
145     assertEquals(montant, c2.getSolde());
146 }
147
148
149 @Test
150 @Order(7)
151 public void testDebiterNotNormal() throws Exception {
152     try {
153         System.out.println("\nTEST 7 : DEBITER (ANORMAL)");
154         System.out.println("Solde du compte c1 avant retrait : " + c1.getSolde());
155         System.out.println("Retrait de 999€");
156         c1.debiter(999);
157         fail();
158     }
159     catch(IllegalOperationException e) {
160         System.out.println(e.getMessage());
161         System.out.println("Le débit d'un montant non possédé à bien été bloqué !\n");
162     }
163 }
164
165
166 @Test
167 @Order(8)
168 public void testTtransfert() throws Exception {
169     c3.crediter(100);
170     System.out.println("\nTEST 8 : TRANSFERT (NORMAL)");
171     System.out.println("Solde du compte c3 avant transfert : " + c3.getSolde());
172     System.out.println("Solde du compte c4 avant transfert : " + c4.getSolde());
173     System.out.println("Transfert de 50€ de c3 -> c4");
174
175     c3.transfert(c4, 50);
176
177     System.out.println("Solde du compte c3 : " + c3.getSolde());
178     System.out.println("Solde du compte c4 : " + c4.getSolde());
179
180     assertEquals(50, c3.getSolde());
181     assertEquals(50, c4.getSolde());
182
183 }
184
185
186 @Test
187 @Order(9)
188 public void testTtransfertNotNormal() throws Exception {
189     try {
190         System.out.println("\nTEST 9 : TRANSFERT ANORMAL");
191         System.out.println("Solde du compte c1 avant transfert : " + c1.getSolde());
192         System.out.println("Transfert de 999€");
193         c1.transfert(c2, 999);
194         fail();
195     }
196     catch(IllegalOperationException e) {
197         System.out.println(e.getMessage());
198         System.out.println("Le transfert d'un montant non possédé à bien été bloqué !\n");
199     }
200 }
201
202
203 @Test
204 @Order(10)
205 public void testAddProprio() throws Exception {
206     try {
207         System.out.println("\nTEST 10 : AJOUT PROPRIETAIRE (NORMAL)");
208         List<Integer> listeClient = new ArrayList<Integer>(c1.getIdClient());
209         System.out.println("Proprietaire du compte c1 : [" + (c1.getIdClient().size()) + "] = " +
210         (banque.findClient(listeClient.get(0))));
211         System.out.print("ajout du proprietaire titi \n");
212
213         c1.addClient(titi);
214         listeClient = new ArrayList<Integer>(c1.getIdClient());
215         System.out.println("Proprietaire du compte c1 : [" + (c1.getIdClient().size()) + "] = " +
216         banque.findClient(listeClient.get(0)) + " , " + banque.findClient(listeClient.get(1)));

```

```

215         assertEquals(2, c1.getIdClient().size());
216     }
217 }
218 catch(IllegalArgumentException e) {
219     e.printStackTrace();
220     fail();
221 }
222 }
223
224
225 @Test
226 @Order(11)
227 public void testAddProprioNotNormal() throws Exception {
228     try {
229         System.out.println("\nTEST 11 : AJOUT PROPRIETAIRE (ANORMAL)");
230         System.out.println("Nombre de proprietaire du compte c1 : [" + (c1.getIdClient().size())
231             + "]");
232         System.out.println("Ajout d'un autre proprietaire");
233         c1.addClient(tutu);
234         System.out.println("Nombre de proprietaire du compte c1 : [" + (c1.getIdClient().size())
235             + "]");
236         // ajout d'une impression pour mise à jour du nombre de proprio
237         c1.addClient(titi);
238         fail();
239     } catch (IllegalArgumentException e) {
240         System.out.println(e.getMessage());
241         System.out.println("L'ajout d'un 3e beneficiare à bien été bloqué");
242     }
243 }
244
245 @Test
246 @Order(12)
247 public void testRemoveProprio() throws Exception {
248     System.out.println("\nTEST 12 : SUPPRIMER PROPRIETAIRE");
249     c1.addClient(titi); // on ajoute un autre proprietaire sinon en raison de @Beforeeach
250     // c'est remis à 1 seul proprietaire (toto) comme au départ et le test est sans intérêt
251     System.out.println("Nombre de proprietaire du compte c1 : [" + (c1.getIdClient().size()) +
252         "]");
253     System.out.println("Suppression du proprietaire titi");
254
255     c1.removeClient(titi);
256
257     System.out.println("Nombre de proprietaire du compte c1 : [" + (c1.getIdClient().size()) +
258         "]");
259     assertEquals(1, c1.getIdClient().size());
260 }
261
262 @Test
263 @Order(13)
264 public void testRemoveCompte() throws Exception {
265     c1.addClient(titi);
266
267     System.out.println("\nTEST 13 : SUPPRIMER COMPTE");
268     System.out.println("Nombre de compte à la banque = " + (banque.getListeComptes().keySet().
269         size()));
270     System.out.println("Suppression du compte c1 ayant pour clients les IdClients : " + c1.
271         getIdClient());
272     System.out.println("Client de la banque possédant ce compte :");
273
274     for(Integer idClient : banque.getListeClients().keySet()) {
275         if(banque.getListeClients().get(idClient).getComptes().contains(c1.getId())) {
276             System.out.println(banque.getListeClients().get(idClient).getPrenom());
277         }
278     }
279
280     // Suppression du compte de la banque
281     banque.removeCompte(c1);
282
283     System.out.println("Nombre de compte à la banque après suppression = " + (banque.
284         getListeComptes().keySet().size()));
285     System.out.println("Client de la banque possédant ce compte :");
286
287     for(Integer idClient : banque.getListeClients().keySet()) {

```

```

284         if(banque.getListeClients().get(idClient).getComptes().contains(c1.getId())) {
285             System.out.println(banque.getListeClients().get(idClient).getPrenom());
286         }
287     }
288
289     // verification suppression du compte
290     assertEquals(3, banque.getListeComptes().keySet().size()); // c2 + c3 + c4
291
292     // verification suppression du compte au sein du client
293     assertEquals(0, toto.getComptes().size());
294     assertEquals(2, titi.getComptes().size());
295
296 }
297
298
299 @Test
300 @Order(14)
301 public void testRemoveClient() throws Exception {
302     System.out.println("\n\nTEST 14 : SUPPRIMER CLIENT");
303
304     // Ajout du client toto au compte 2
305     c2.addClient(toto);
306     System.out.println("Nombre de client : " + (banque.getListeClients().keySet().size()));
307     System.out.println("Nombre de comptes : " + (banque.getListeComptes().keySet().size()));
308     System.out.print("Compte appartenant uniquement à titi: ");
309     for(Integer idCompte : titi.getComptes()) {
310         if(banque.getListeComptes().get(idCompte).getIdClient().size() == 1) {
311             System.out.print((idCompte) + " ");
312         }
313     }
314
315     System.out.println("\nSuppression de titi");
316
317     banque.removeClient(titi);
318     System.out.println("Nombre de client après suppression : " + (banque.getListeClients().
319     keySet().size()));
320     System.out.println("Nombre de comptes après suppression : " + (banque.getListeComptes().
321     keySet().size()));
322
323     assertEquals(2, banque.getListeClients().keySet().size()); // toto + tutu + tété
324     assertEquals(3, banque.getListeComptes().keySet().size()); //c1 + c2 + c4
325     assertEquals(1, c1.getIdClient().size());
326 }

```