Université de Montpellier / Faculté Des Sciences

Modélisation et Programmation par Objets (2e partie)

2019-2020 - HLIN 505

Cours 4
Interfaces
Annotations

Enseignants

Marianne Huchard, Stéphane Bessy, Marie-Laure Mugnier,

Clémentine Nebut, Abdelhak-Djamel Seriai

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

1 Les interfaces

Cette section propose quelques rappels sur la notion d'interface en Java (le cours complet a été vu dans le module HLIN406).

La notion d'*interface* a été introduite pour définir la partie du *contrat* (ou cahier des charges), pour un ensemble de classes, relative à un ensemble d'opérations que doivent posséder ces classes.

Une interface Java peut contenir plus exactement les éléments suivants.

- Dans les versions de Java jusqu'à Java 1.7 :
  - des méthodes d'instance publiques abstraites (avec les modifieurs `public abstract`),
  - des attributs de classe constant publics (avec les modifieurs `public final static`),
- A partir de Java 1.8 :
  - des méthodes d'instance publiques présentant des comportements par défaut (avec les modifieurs `public` et `default`). Les méthodes de la classe `Object` comme `String toString()` ne peuvent être écrites par défaut.
  - des méthodes statiques (avec les modifieurs `public` et `static`),
  - des types internes publics (comme présenté dans le premier cours).

On remarquera qu'une interface ne contient donc jamais ni constructeur, ni code de méthodes en dehors des méthodes par défaut et des méthodes statiques, ni aucun attribut d'instance ou privé. Les modifieurs (`public`, `abstract`, `final`, ...) sont omis lorsqu'il n'y a pas d'ambiguïté.

1.1 Représentation des interfaces en UML

Le schéma de la figure 1 présente la notion d'interface en UML avec de la multi-spécialisation entre les interfaces, de l'implémentation entre la classe `CompteRemunereAvecCarteBleue` et l'interface `ICompteRemunereAvecCarteBleue`. Notez le stéréotype <<`interface`>>.

Le schéma de la figure 2 montre une autre vue UML sur les interfaces, que l'on retrouve dans les diagrammes de composants et qui montre les services fournis par la classe (ceux de l'interface `ICompteRemunereAvecCarteBleue`) et ceux que la classe requiert (ceux de l'interface `Map`).

1.2 Définition d'une interface en Java

```
public interface ICompte {
   double decouvertMaximum = 500 ;
   double getSolde() ;
   void deposer(double m) ;
   void retirer(double m) ;
   default boolean estAdecouvert() {return this.getSolde()<0 ;}
   default double fermer() {return this.getSolde() ;}
   static void afficheDecouvertMaximum()
      {System.out.println("découvert max = "+decouvertMaximum) ;}
}
```

Une interface peut ensuite être utilisée comme un type pour déclarer une variable, un attribut ou un paramètre.

```
ICompte compte ;
```

Remarquons l'affectation polymorphe qui suit qui montre une forme de sous-typage entre toute interface et la classe `Object`.

```
ICompte compte=null ;
Object o=compte ;
```

1.3 Spécialisation d'une interface en Java

Pour spécialiser une interface par une autre, on utilise le mot-clef `extends`. Une interface peut avoir plusieurs super-interfaces.

```
public interface ICompteRemunere extends ICompte{
   default double fermer(){return this.getSolde()*1.02 ;}
// remarquons qu'on ne pourra pas écrire super.fermer() dans ce contexte
}

public interface ICompteCarteBleue extends ICompte{
   default double fermer(){return this.getSolde()-10 ;}
}

public interface ICompteRemunereAvecCarteBleue
      extends ICompteRemunere, ICompteCarteBleue{
   default double fermer() // obligatoirement redefinie pour eviter les conflits
   {return this.getSolde()-10 ;}
}
```

1.4 Implémentation d'une interface en Java

Une interface est implémentée par une ou plusieurs classes. Une classe, inversement, peut implémenter une ou plusieurs interfaces. Si toutes les méthodes abstraites de l'interface ne sont pas définies dans la classe qui l'implémente, cette classe est abstraite. Dans l'exemple donné, la classe concrète `CompteRemunereAvec-CarteBleue` implémente l'interface `ICompteRemunereAvecCarteBleue` en fournissant un corps à toutes les méthodes abstraites héritées. Elle pourrait aussi redéfinir des méthodes `default` héritées si c'est pertinent.

```
public class CompteRemunereAvecCarteBleue
        implements ICompteRemunereAvecCarteBleue{

   private double solde;
   private Map<String, Client> clients;

   @Override
   public double getSolde() {
      return this.solde;
   }

   @Override
   public void deposer(double m) {
      if (m>0)
         this.solde += m;
   }

   @Override
   public void retirer(double m) {
      if (m>0)
         this.solde -= m;
   }
}

public class Client{.....}
```

2 Les annotations

Les annotations sont des méta-données relatives à un programme. Ces méta-données sont des informations que l'on associe à des éléments de code comme les classes ou les interfaces, les méthodes, les variables ou encore les packages.

Elles s'adressent potentiellement à différents outils :

— Le compilateur peut s'en servir pour détecter des erreurs ou supprimer des warnings.

— Des outils vivants au moment de la compilation ou du déploiement peuvent les exploiter pour générer du code, des fichiers XML de documentation, vérifier des propriétés, etc. C'est le cas de l'outil **javadoc** qui se sert des annotations pour générer la documentation.

— Lors de l'exécution, certaines des annotations peuvent être examinées par introspection, et ainsi être source de nouveaux comportements (qui auront été programmés).

2.1 Définition d'une annotation

Un exemple simple Supposons qu'un groupe de travail rédige de manière collaborative un logiciel et veuille ajouter dans le code source des informations sur les classes telles que :

— l'auteur initial de la classe,

— la personne en charge de son test,

— la responsabilité principale de la classe,

— le niveau d'importance de la classe,
— la version de la classe.

Définir un type d'annotation pour ces informations revient à définir une interface avec une syntaxe un peu particulière :

— le mot-clef `interface` est précédé du caractère `@`
— les opérations :
  — définissent en fait des sortes d'attributs,
  — n'ont pas de paramètre,
  — n'ont pas de clause `throws`,
  — ont pour type de retour possible des types primitifs, les classes `String` et `Class` (que nous verrons dans le cours sur l'introspection), des annotations, des énumérations, des tableaux de dimension 1 des types précédents,
  — une valeur par défaut peut leur être attribuée, elle est dans ce cas introduite par le mot-clef `default`.

Lorsque l'annotation ne contient qu'un attribut, il peut s'appeler `value` de manière à simplifier l'écriture des valeurs ensuite lors de l'utilisation.

Dans l'exemple qui suit, nous utilisons deux annotations différentes : la première décrit la version de la classe (donnée par l'annotation `ClassVersion`), l'autre introduit les autres informations (contenues dans l'annotation `ClassInfo`).

```
public enum NiveauImportance {
   faible, moyen, eleve ;
}

public @interface ClassInfo {
   String createur() ;
   String testeur() ;
   String responsabilite() ;
   NiveauImportance niveau() default NiveauImportance.moyen ;
}

public @interface ClassVersion {
   int value() ;
}
```

Composer des annotations On ne peut pas dériver une annotation d'une autre comme le montre le code ci-dessous. La partie incorrecte concernant la déclaration d'une super-annotation est barrée.

```
public @interface ClassInfoVersionEssai ~~extends ClassInfo~~{
   int value() ;
}
```

Comment construire des annotations par réutilisation d'autres annotations dans ce cas ? Il nous reste le mécanisme de composition dans lequel une annotation peut se composer d'autres annotations. Ci-dessous l'annotation `ClassInfoVersion` se compose des informations de base et du numéro de version par réutilisation des annotations précédemment définies, présentes ici sous forme d'attributs.

```
public @interface ClassInfoVersion{
   ClassInfo info() ;
   ClassVersion version() ;
}
```

2.2 Utilisation d'une annotation

Cas standard Lorsqu'une annotation est utilisée sur un élément (classe, méthode, variable), elle se place avec les modifieurs (visibilité, `static`, `abstract`) et une valeur est affectée à chacun des attributs avec deux cas particuliers :

- on peut ne pas affecter de valeur à un attribut qui a une valeur par défaut.
- si l'interface ne contient qu'un attribut et qu'il s'appelle `value`, le nom de celui-ci peut être omis lors de l'attribution d'une valeur.

Ci-dessous, les annotations définies dans la section précédente sont appliquées à différentes classes.

```
@ClassInfo(
createur=”justine”,
testeur = ”lucien”,
responsabilite=”representer les voitures”) // on utilise la valeur par défaut pour le niveau
@ClassVersion(1) // on peut aussi écrire value=1
public class Voiture {
.....
}

@ClassVersion(0)   // on voit que l'on peut mettre les annotations
@ClassInfo(        // dans n'importe quel ordre
createur=”justine”,
testeur = ”lucien”,
responsabilite=”representer les personnes”,
niveau=NiveauImportance.eleve)
public class Personne {
.....
}

@ClassInfoVersion( // on donne une valeur à chaque attribut de l'annotation composite
info=@ClassInfo(
createur=”justine”,
testeur = ”lucien”,
responsabilite=”representer les personnes”,
niveau=NiveauImportance.eleve),
version=@ClassVersion(0))
public class Garage {
.....
}
```

Annoter un package Il est moins simple d'annoter un package en Java puisqu'un package correspond à un ensemble de classes. Pour réaliser une telle annotation, on crée un fichier `package-info.java` qui sera

placé à la racine du package, là où l'on place habituellement les fichiers décrivant les classes, les interfaces, les énumérations du package, etc. Ce fichier commence par la déclaration de l'annotation, puis contient le nom du package annoté. Ci-dessous on commence par déclarer une annotation (comme vu précédemment), puis on l'utilise sur un package).

```
// Dans un fichier PackageMesInfos.java

public @interface PackageMesInfos{
String objectif() ;
}
```

```
// Dans un fichier package-info.java

@PackageMesInfos ( // annotation
objectif = "tester les annotations"
)
package annotationsEssais ; // nom du package annoté
```

2.3 Annotations existantes

Premiers exemples L'API propose différentes annotations pré-définies, par exemple :

- `@Deprecated` indique que l'élément provient d'une ancienne version et a été remplacé. Son usage provoque l'apparition d'un *warning*, mais le code continue en principe à fonctionner. Cela assure la compatibilité avec les anciennes versions de Java. La documentation javadoc (par une annotation `@deprecated`) indique en principe quel autre élément utiliser à la place de l'élément devenu obsolète.
- `@Override` indique qu'une méthode en redéfinit une autre (au sens de la liaison dynamique). Lorsque l'annotation est placée sur une méthode, le compilateur vérifie qu'une méthode de même nom et de signature compatible (au sens de la redéfinition) existe dans une super-classe. Si dans la classe `Voiture` on ajoute :

```
@Override
public boolean equals(Voiture o){
return true ;
}
```

  On aura une erreur de compilation puisque cette méthode ne redéfinit pas correctement la méthode `public boolean equals(Object o)`. Si on enlève l'annotation, l'erreur disparaît.
- `@SuppressWarnings` a un attribut `value` de type tableau de chaînes de caractères (`String[] value`) qui contient les *warnings* que l'on ne souhaite pas voir au moment de la compilation, comme "unused", "unchecked", etc. L'exemple ci-dessous montre l'usage de l'annotation pour faire disparaître un *warning* dû à la définition d'une méthode privée non utilisée dans le reste de la classe.

```
@SuppressWarnings("unused")
private void crypter()
{ /* à écrire plus tard*/
System.out.println("cryptage") ;
}
```

Annotations d'annotations Parmi les annotations existantes, notons également celles qui portent sur d'autres annotations et en précisent différents aspects. Elles doivent être importées depuis le package `java.lang.annotation` :

— `@Inherited` indique que l'annotation sera héritée par les sous-classes.
— `@Documented` indique que l'annotation devrait apparaître dans la documentation générée par les outils de documentation (comme javadoc).
— `@Retention` indique la portée et le moment de leur utilisation avec : `SOURCE` pour celles qui sont écartées par le compilateur et utiles pour les outils manipulant le code source, `CLASS` pour celles stockées dans le fichier .class, `RUNTIME` pour celles qui stockées dans le fichier .class et accessibles par introspection pendant l'exécution. Ces valeurs sont les valeurs du type énuméré `RetentionPolicy`. Un exemple est donné plus loin. La valeur par défaut est `RetentionPolicy.CLASS`.
— `@Target` restreint la cible à `TYPE`, `FIELD`, `METHOD`, `ANNOTATION_TYPE`, etc. Un exemple est donné plus loin.
— `@Repeatable` permet de répéter une utilisation d'annotation. Ce mécanisme existe depuis Java 1.8. L'exemple qui suit montre une annotation qui peut être répétée, et sa répétition.

```
// une annotation qui peut être répétée
@Repeatable(Intervenants.class)
public @interface Intervenant {
String value() ;
}

// avec obligatoirement une annotation précisant le conteneur des valeurs
public @interface Intervenants {
Intervenant[] value() ;
}

// et son utilisation
@Intervenant("marianne")
@Intervenant("jessie")
@Intervenant("clémentine")
public class GarageUrbain extends Garage{
.....
}
```

Pour réduire la portée de l'annotation `ClassVersion` aux classes (on ne peut plus l'appliquer ensuite à une méthode) et pour pouvoir la consulter lors de l'exécution par des méthodes d'introspection, on la complète de cette manière :

```
import java.lang.annotation.* ;

@Target(ElementType.TYPE)
@Retention(value=RetentionPolicy.RUNTIME)
public @interface ClassVersion {
int value() ;
}
```

Annotations pour javadoc Enfin un autre ensemble d'annotations que l'on rencontre couramment sont les annotations utilisées par l'outil javadoc ou par eclipse.

Parmi les annotations de la javadoc, on trouvera par exemple, `@param`, `@return` ou `@see`, qui vont provoquer la génération des sections correspondantes dans les pages de documentation. Les commentaires

javadoc sont introduits par `/**`.

```
public class GarageCampagne extends Garage{
/**
* Cette méthode décrit la procédure pour faire admettre
* un véhicule dans un garage de campagne
*
* @param numImmat est le numéro d'immatriculation du véhicule
* @return un numéro d'autorisation
* @see les procédures officielles à : www...../.../...
*/
public String admission(String numImmat){
// .....
return ".....";
}
}
```

2.4 Interroger une annotation à l'exécution

Pour les plus curieux, cette section, qui n'est pas au programme du cours sur les annotations, vient donner un premier aperçu de l'exploitation des annotations en pratique dans le cas des annotations prévues pour l'exécution. Pour connaître à l'exécution les annotations qui ont été posées sur des éléments et leurs attributs, on utilise l'introspection. Ce mécanisme qui sera réellement développé dans un prochain cours, permet, pendant l'exécution d'un programme, d'accéder à différentes informations sur les classes et les méthodes et de manipuler des objets. Il nous intéresse ici pour accéder aux informations concernant les annotations.

Il comprend :

- l'interface `Annotation`, qui est implicitement spécialisée par les annotations (on ne l'étend pas directement). Elle contient la méthode
  - `Class<? extends Annotation> annotationType()` qui retourne une représentation du type d'annotation (de type `Class`)
- l'interface `AnnotatedElement`, qui est implémentée par les représentations de classes (Class), de méthodes (Method), d'attributs (Field), etc. Elle contient les méthodes :
  - `<T extends Annotation> getAnnotation(Class<T> annotationType)` qui retourne l'annotation du type passé en paramètre (ou null)
  - `Annotation[] getAnnotations()` qui retourne les annotations attachées à l'élément (incluant héritées)
  - `AnnotatedElement.getAnnotationsByType(Class<T>)` vient avec Java 8 pour les annotations répétées
  - `Annotation[] getDeclaredAnnotations()` qui retourne toutes les annotations attachées à l'élément (propres)
  - `boolean isAnnotationPresent (Class<? extends Annotation>annotationType)` qui retourne vrai si une annotation du type passé en paramètre est attachée à l'élément

Nous montrons un exemple très simplifié d'une étape de leur mise en œuvre pour réaliser un outil de test unitaire. L'objectif général est :

- d'embarquer dans les classes des méthodes de test unitaire,
- que les programmeurs annotent les méthodes de test pour les distinguer des autres,

— que l'outil de test utilise les annotations pour tester la classe suivant différents niveaux de risque.
Le code complet sera vu plus tard (lors du cours sur l'introspection) et on en donne ici une première idée.

Nous introduisons tout d'abord un type d'annotation pour les méthodes de test. Ces annotations seront accessibles lors de l'exécution.

```
enum NiveauRisque
   { faible, moyen, eleve ; }

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test     { NiveauRisque risque() ; }
```

Une classe en cours de développement peut alors être annotée comme suit (les méthodes m0, m1, m3, m5 et m7 sont annotées avec des niveaux de risques différents) :

```
public class Foo {

@Test(risque=NiveauRisque.eleve)
public static void m0() throws MonException {
System.out.println("m0") ;
System.out.println(("truc" instanceof String)) ;
throw new MonException("truc") ;
}

@Test(risque=NiveauRisque.faible)
public static void m1() {System.out.println("m1") ;}

public static void m2() {System.out.println("m2") ;}

@Test(risque=NiveauRisque.moyen)
public static void m3() {
throw new RuntimeException("Boom") ;
}

public static void m4() {System.out.println("m4") ;}

@Test(risque=NiveauRisque.moyen)
public static void m5() {System.out.println("m5") ;}

public static void m6() {System.out.println("m6") ;}

@Test(risque=NiveauRisque.eleve) public static void m7() {
throw new RuntimeException("Crash") ;
}

public static void m8() {System.out.println("m8") ;}

}
```

Une classe (très simplifiée comme nous l'avons dit) de l'outil de test est présentée ci-dessous. Sa méthode `main` prend en paramètre un nom de classe. Elle récupère les méthodes de la classe `Foo`, et si ce sont des méthodes portant l'annotation de test, elle affiche leur nom et la valeur du niveau de risque.

```
public static void main(String[] args) throws Exception {
   for (Method m : Class.forName("annotationsEssais.Foo").getMethods()) {
      if (m.isAnnotationPresent(Test.class))
         System.out.println(m.getName()+" "
         +((Test)m.getAnnotation(Test.class)).risque());
   }
}
```

Lors de l'exécution, on verra l'affichage suivant :

```
m0 eleve
m1 faible
m3 moyen
m5 moyen
m7 eleve
```

D'autres détails et un code plus complet seront présentés dans un prochain cours sur l'introspection.

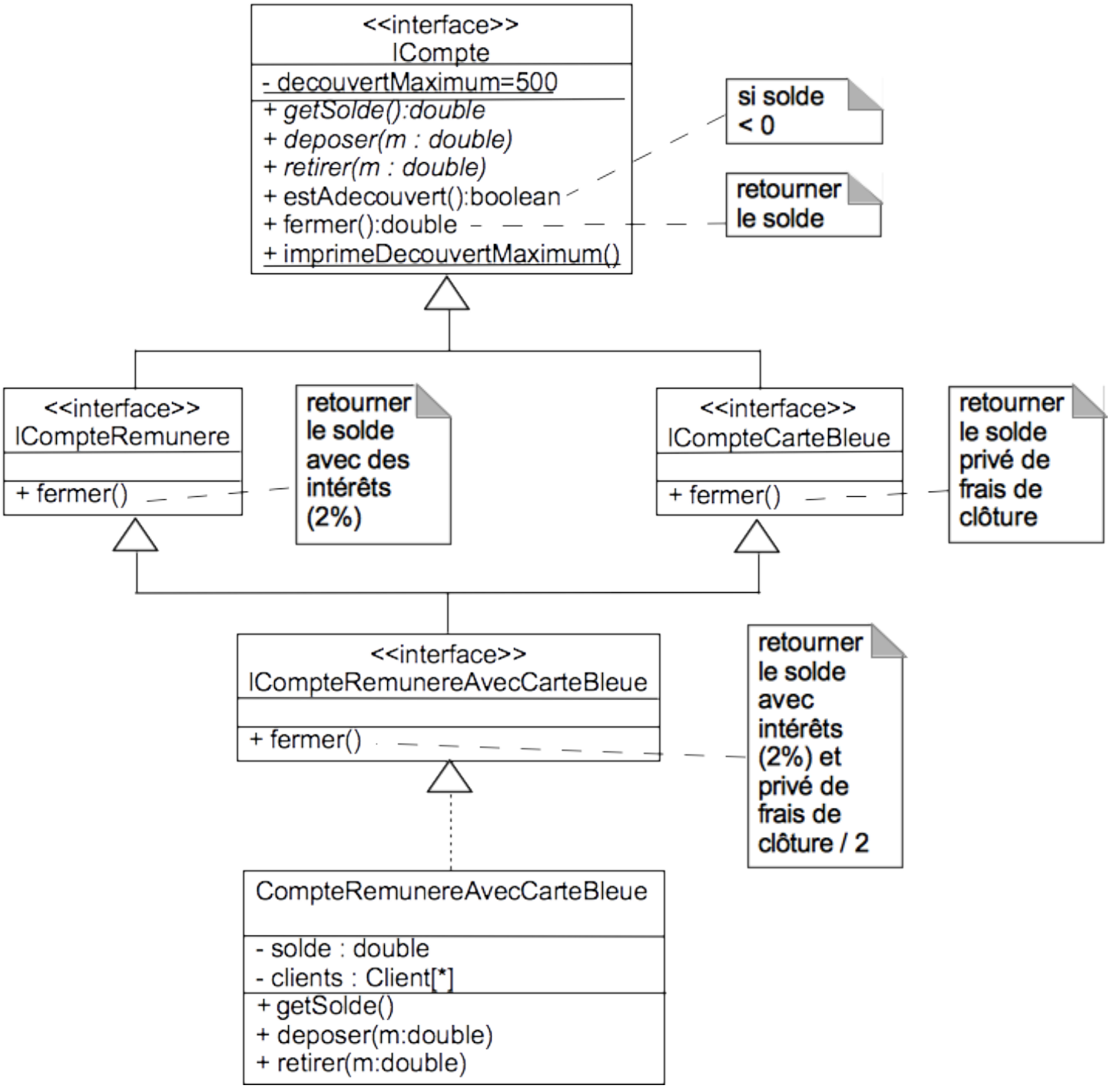

FIGURE 1 – Représentation des interfaces et des relations de spécialisation, de réalisation (implémentation)

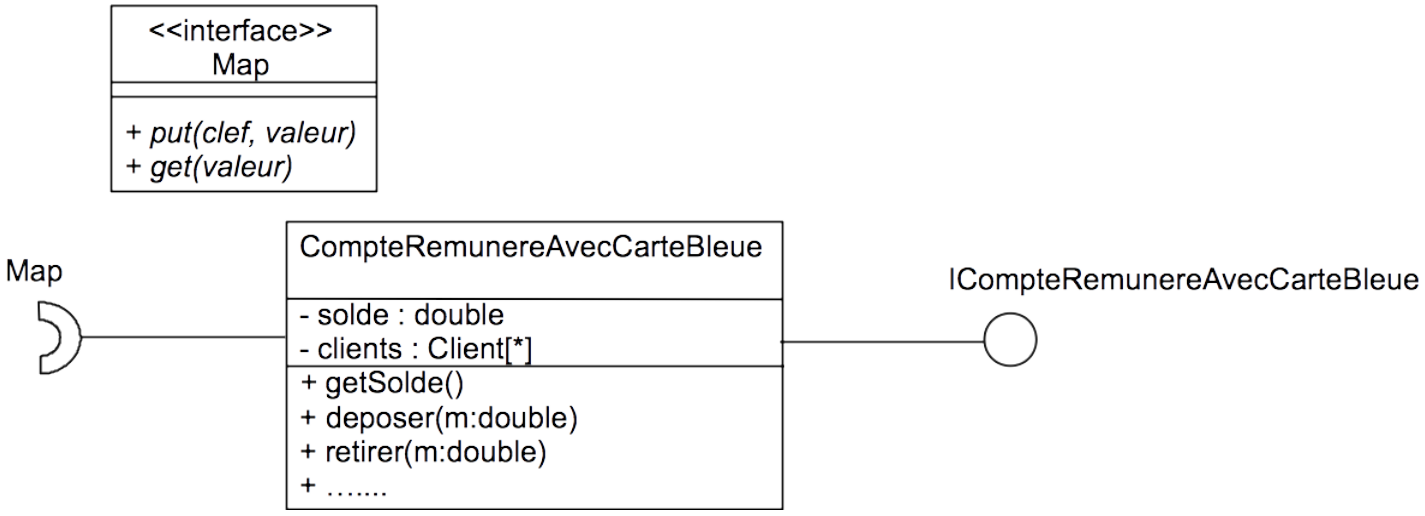

FIGURE 2 – Représentation des interfaces d'une classe avec la notation des diagrammes de composants