Les interfaces

Une interface permet de définir un ensemble de services qu'un client peut obtenir d'un objet. Une interface introduit une abstraction pure qui permet un découplage maximal entre un service et son implémentation. On retrouve ainsi les interfaces au cœur de l'implémentation de beaucoup de bibliothèques et de frameworks. Le mécanisme des interfaces permet d'introduire également une forme simplifiée d'héritage multiple.

Déclaration d'une interface

Une interface se déclare avec le mot-clé **interface**.

```
package com.cgi.udev.compte;

public interface Compte {

}
```

Comme pour une classe, une interface a une portée, un nom et un bloc de déclaration. Une interface est déclarée dans son propre fichier qui porte le même nom que l'interface. Pour l'exemple ci-dessus, le fichier doit s'appeler *Compte.java*.

Une interface décrit un ensemble de méthodes en fournissant uniquement leur signature.

```
package com.cgi.udev.compte;

public interface Compte {

  void deposer(int montant) throws OperationInterrompueException,
                                   CompteBloqueException;

  int retirer(int montant) throws OperationInterrompueException,
                                  CompteBloqueException;

  int getBalance() throws OperationInterrompueException;

}
```

Une interface introduit un nouveau type d'abstraction qui définit à travers ces méthodes un ensemble d'interactions autorisées. Une classe peut ensuite implémenter une ou plusieurs interfaces.

> Les méthodes d'une interface sont par défaut **public** et **abstract**. Il n'est pas possible de déclarer une autre portée que **public**.

```
package com.cgi.udev;

public interface Mobile {

  public abstract void deplacer();

}
```

L'interface ci-dessus est strictement identique à celle-ci :

```
package com.cgi.udev;

public interface Mobile {

  void deplacer();

}
```

Implémentation d'une interface

Une classe signale les interfaces qu'elle implémente grâce au mot-clé **implements**. Une classe concrète doit fournir une implémentation pour toutes les méthodes d'une interface, soit dans sa déclaration, soit parce qu'elle en hérite.

```
package com.cgi.udev.compte;

public class CompteBancaire implements Compte {

  private final String numero;
  private int balance;

  public CompteBancaire(String numero) {
```

```
    this.numero = numero;
  }

  @Override
  public void deposer(int montant) {
    this.balance += montant;
  }

  @Override
  public int retirer(int montant) throws OperationInterrompueException {
    if (balance < montant) {
      throw new OperationInterrompueException();
    }
    return this.balance -= montant;
  }

  @Override
  public int getBalance() {
    return this.balance;
  }

  public String getNumero() {
    return numero;
  }

}
```

L'implémentation des méthodes d'une interface suit les mêmes règles que la redéfinition.

> Si la classe qui implémente l'interface est une classe abstraite, alors elle n'est pas obligée de fournir une implémentation pour les méthodes de l'interface.

Même si les mécanismes des interfaces sont proches de ceux des classes abstraites, ces deux notions sont clairement distinctes. Une classe abstraite permet de mutualiser une implémentation dans une hiérarchie d'héritage en introduisant un type plus abstrait. Une interface permet de définir les interactions possibles entre un objet et ses clients. Une interface agit comme un contrat que les deux parties doivent remplir. Comme l'interface n'impose pas de s'insérer dans une hiérarchie d'héritage, il est relativement simple d'adapter une classe pour qu'elle implémente une interface.

Une interface introduit un nouveau type de relation qui serait du type *est comme un* (*is-like-a*).

Par exemple, il est possible de créer un système de gestion de comptes utilisant l'interface *Compte* . Il est facile ensuite de fournir une implémentation de cette interface pour un compte bancaire, un porte-monnaie électronique, un compte en ligne… t Une classe peut implémenter plusieurs interfaces si nécessaire. Pour cela, il suffit de donner les noms des interfaces séparés par une virgule.

```
package com.cgi.udev.animal;

public interface Carnivore {

  void manger(Animal animal);

}
```

```
package com.cgi.udev.animal;

public interface Herbivore {

  void manger(Vegetal vegetal);

}
```

```
package com.cgi.udev.animal;

public class Humain extends Animal implements Carnivore, Herbivore {

  @Override
  public void manger(Animal animal) {
    // ...
  }

  @Override
  public void manger(Vegetal vegetal) {
    // ...
  }

}
```

Dans l'exemple précédent, la classe *Humain* implémente les interfaces *Carnivore* et *Herbivore*. Donc une instance de la classe *Humain* peut être utilisée dans une application partout où les types *Carnivore* et *Herbivore* sont attendus.

```
Humain humain = new Humain();

Carnivore carnivore = humain;
carnivore.manger(new Poulet()); // Poulet hérite de Animal

Herbivore herbivore = humain;
herbivore.manger(new Chou());   // Chou hérite de Vegetal
```

Attributs et méthodes statiques

Une interface peut déclarer des attributs. Cependant tous les attributs d'une interface sont par défaut **public**, **static** et **final**. Il n'est pas possible de modifier la portée de ces attributs. Autrement dit, une interface ne peut déclarer que des constantes.

```
package com.cgi.udev.compte;

public interface Compte {

  int PLAFOND_DEPOT = 1_000_000;

  void deposer(int montant) throws OperationInterrompueException, CompteBloqueException;

  int retirer(int montant) throws OperationInterrompueException, CompteBloqueException;

  int getBalance() throws OperationInterrompueException;

}
```

> On peut préciser **public**, **static** et **final** dans la déclaration d'un attribut d'interface :

```
public static final int PLAFOND_DEPOT = 1_000_000;
```

Ceci est strictement équivalent à

```
int PLAFOND_DEPOT = 1_000_000;
```

Une interface peut également déclarer des méthodes **static**. Dans ce cas, il s'agit de méthodes équivalentes aux méthodes de classe et l'interface doit fournir une implémentation pour ces méthodes. Ces méthodes doivent explicitement avoir le mot-clé **static** et elles ont une portée

publique par défaut.

```
package com.cgi.udev.compte;

public interface Compte {

  int PLAFOND_DEPOT = 1_000_000;

  static int getBalanceTotale(Compte... comptes) throws OperationInterrompueException {
    int total = 0;
    for (Compte c : comptes) {
      total += c.getBalance();

    }
    return total;
  }

  void deposer(int montant) throws OperationInterrompueException, CompteBloqueException;

  int retirer(int montant) throws OperationInterrompueException, CompteBloqueException;

  int getBalance() throws OperationInterrompueException;

}
```

Héritage d’interface

Une interface peut hériter d’autres interfaces. Contrairement aux classes qui ne peuvent avoir qu’une classe parente, une interface peut avoir autant d’interfaces parentes que nécessaire. Pour déclarer un héritage, on utilise le mot-clé **extends**.

```
package com.cgi.udev.animal;

public interface Omnivore extends Carnivore, Herbivore {

}
```

Une classe concrète qui implémente une interface doit donc disposer d’une implémentation pour les méthodes de cette interface mais également pour toutes les méthodes des interfaces dont cette dernière hérite.

```
package com.cgi.udev.animal;

public class Humain extends Animal implements Omnivore {

  @Override
  public void manger(Animal animal) {
    // ...
  }

  @Override
  public void manger(Vegetal vegetal) {
    // ...
  }

}
```

L'héritage d'interface permet d'introduire de nouveaux types par agrégat. Dans l'exemple ci-dessus, nous faisons apparaître la notion d'omnivore simplement comme étant à la fois un carnivore et un herbivore.

Les interfaces marqueurs

Comme chaque interface introduit un nouveau type, il est possible de contrôler grâce au mot-clé **instanceof** qu'une variable, un paramètre ou un attribut est bien une instance compatible avec cette interface.

```
Humain bob = new Humain();
if (bob instanceof Carnivore) {
  System.out.println("bob mange de la viande");
}
```

En Java, on utilise cette possibilité pour créer des interfaces marqueurs. Une interface marqueur n'a généralement pas de méthode, elle sert juste à introduire un nouveau type. Il est ensuite possible de changer le comportement d'une méthode si une variable, un paramètre ou un attribut implémente cette interface.

```
package com.cgi.udev.animal;

public interface Cannibale {
```

```
}
```

```
package com.cgi.udev.animal;

public class Humain extends Animal implements Omnivore {

  @Override
  public void manger(Animal animal) {
    if (!(animal instanceof Humain) || this instanceof Cannibale) {
      // ...
    }
  }

  @Override
  public void manger(Vegetal vegetal) {
    // ...
  }

}
```

Dans l'exemple ci-dessus, *Cannibale* agit comme une interface marqueur, elle permet à une classe héritant de *Humain* de manger une instance d'humain. Pour cela, il suffit de déclarer que cette nouvelle classe implémente *Cannibale* :

```
package com.cgi.udev.animal;

public class Anthropophage extends Humain implements Cannibale {

}
```

Même si la classe *Anthropophage* ne redéfinit aucune méthode de sa classe parente, le fait de déclarer l'interface marqueur *Cannibale* suffit a modifier son comportement.

Le principe de l'interface marqueur est quelques fois utilisé dans l'API standard de Java. Par exemple, La méthode clone déclarée par Object jette une CloneNotSupportedException si elle est appelée sur une instance qui n'implémente pas l'interface Cloneable. Cela permet de fournir une méthode par défaut pour créer une copie d'un objet mais sans activer la fonctionnalité. Il faut que la classe déclare son intention d'être clonable grâce à l'interface marqueur.

Implémentation par défaut

Il est parfois difficile de faire évoluer une application qui utilise intensivement les interfaces. Reprenons notre exemple du *Compte*. Imaginons que nous souhaitions ajouter la méthode *transférer* qui consiste à transférer le solde d'un compte vers un autre.

```
package com.cgi.udev.compte;

public interface Compte {

  void deposer(int montant) throws OperationInterrompueException,
                                   CompteBloqueException;

  int retirer(int montant) throws OperationInterrompueException,
                                  CompteBloqueException;

  int getBalance() throws OperationInterrompueException;

  void transferer(Compte destination) throws OperationInterrompueException,
                                             CompteBloqueException;

}
```

En ajoutant une nouvelle méthode à notre interface, nous devons fournir une implémentation pour cette méthode dans toutes les classes que nous avons créées pour qu'elles continuent à compiler. Mais si d'autres équipes de développement utilisent notre code et ont, elles-aussi, créé des implémentations pour l'interface *Compte*, alors elles devront adapter leur code au moment d'intégrer la dernière version de notre interface.

Comme les interfaces servent précisément à découpler deux implémentations, elles sont très souvent utilisées dans les bibliothèques et les frameworks. D'un côté, les interfaces introduisent une meilleure souplesse mais, d'un autre côté, elles entraînent une grande rigidité car il peut être difficile de les faire évoluer sans risquer de casser des implémentations existantes.

Pour palier partiellement à ce problème, une interface peut fournir une implémentation par défaut de ses méthodes. Ainsi, si une classe concrète qui implémente cette interface n'implémente pas une méthode par défaut, c'est le code de l'interface qui s'exécutera. Une méthode par défaut doit obligatoirement avoir le mot-clé **default** dans sa signature.

```
package com.cgi.udev.compte;

public interface Compte {
```

```
  void deposer(int montant) throws OperationInterrompueException,
                                   CompteBloqueException;

  int retirer(int montant) throws OperationInterrompueException,
                                  CompteBloqueException;

  int getBalance() throws OperationInterrompueException;

  default void transferer(Compte destination) throws OperationInterrompueException,
                                                     CompteBloqueException {
    if (destination == this) {
      return;
    }
    int montant = this.getBalance();
    if (montant <= 0) {
      return;
    }
    destination.deposer(montant);
    boolean retraitOk = false;
    try {
      this.retirer(montant);
      retraitOk = true;
    } finally {
      if (!retraitOk) {
        destination.retirer(montant);
      }
    }
  }

}
```

Une classe implémentant *Compte* n'a pas besoin de fournir une implémentation pour la méthode *transferer*. La classe *CompteBancaire* que nous avons implémentée au début de ce chapitre continuera de compiler et de fonctionner comme attendu tout en ayant une méthode supplémentaire.

> L'implémentation par défaut de méthode dans une interface la rapproche beaucoup du fonctionnement d'une classe abstraite. Cependant leurs usages sont différents. L'implémentation d'une méthode dans une classe abstraite est courant car la classe

> abstraite a cette notion de mutualisation de code. Par contre, l'implémentation par défaut de méthode dans une interface est très rare. Elle est réservée pour les types de situations décrits précédemment, afin d'éviter de casser les implémentations existantes.

La ségrégation d'interface

En programmation objet, le principe de ségrégation d'interface stipule qu'un client ne devrait pas avoir accès à plus de méthodes d'un objet que ce dont il a vraiment besoin. L'objectif est de limiter au strict minimum les interactions possibles entre un objet et ces clients afin d'assurer un couplage minimal et faciliter ainsi les évolutions et le refactoring. En Java, le principe de ségrégation d'interface a deux conséquences :

1. Le type des variables, paramètres et attributs doit être choisi judicieusement pour restreindre au type minimum nécessaire par le code.
2. Une interface ne doit pas déclarer *trop* de méthodes.

Le premier point implique qu'il est préférable de manipuler les objets à travers leurs interfaces plutôt que d'utiliser le type réel de l'objet. Un exemple classique en Java concerne l'API des collections. Il s'agit de classes permettant de gérer un ensemble d'objets. Elles apportent des fonctionnalités plus avancées que les tableaux. Par exemple la classe java.util.ArrayList permet de gérer une liste d'objets. Cette classe autorise l'ajout en fin de liste, l'insertion, la suppression et bien évidemment l'accès à un élément selon son index et le parcours complet des éléments.

Un programme qui crée une ArrayList pour stocker un ensemble d'éléments n'utilisera jamais une variable de type ArrayList mais plutôt une variable ayant le type d'une interface implémentée par cette classe.

```
// Utilisation de l'interface List
List maListe = new ArrayList();
```

```
// Utilisation de l'interface Collection
Collection maListe = new ArrayList();
```

```
// Utilisation de l'interface Iterable
Iterable maListe = new ArrayList();
```

Plus une partie d'une application a recours à des interfaces pour interagir avec les autres parties d'une application, plus il est simple d'introduire des nouvelles classes implémentant les interfaces attendues et qui pourront être directement utilisées.

Le second point est lié au principe SOLID de la responsabilité unique. Une interface est conçue pour représenter un type de relation entre la classe qui l'implémente et ses clients. Plus le nombre de méthodes augmente, plus il a de risque que l'interface représente en fait plusieurs types de relation. Dans ce cas, l'héritage entre interfaces et/ou l'implémentation de plusieurs interfaces deviennent une bonne solution pour isoler chaque relation.

Reprenons notre exemple de l'interface *Compte*. Si notre système est composé d'un sous-système de consultation, d'un sous-système de retrait et d'un sous-système de gestion de comptes alors cette interface devrait probablement être séparée en plusieurs interfaces afin d'isoler chaque responsabilité.

Une interface utilisée par le sous-système de consultation :

```
package com.cgi.udev.compte;

public interface CompteConsultable {

  int getBalance() throws OperationInterrompueException;

}
```

Une interface utilisée par le sous-système de retrait :

```
package com.cgi.udev.compte;

public interface OperationDeRetrait {

  int retirer(int montant) throws OperationInterrompueException,
                                  CompteBloqueException;

}
```

Une interface plus complexe utilisée par le système de gestion de comptes :

```
package com.cgi.udev.compte;

public interface Compte extends CompteConsultable, OperationDeRetrait {

  void deposer(int montant) throws OperationInterrompueException,
                                   CompteBloqueException;
```

```
  default void transferer(Compte destination) throws OperationInterrompueException,
                                                     CompteBloqueException {
    if (destination == this) {
      return;
    }
    int montant = this.getBalance();
    if (montant <= 0) {
      return;
    }
    destination.deposer(montant);
    boolean retraitOk = false;
    try {
      this.retirer(montant);
      retraitOk = true;
    } finally {
      if (!retraitOk) {
        destination.retirer(montant);
      }
    }
  }
}
```

L’inversion de dépendance

Lorsque nous avons vu les constructeurs, nous avons vu que nous pouvions réaliser de l’injection de dépendance en passant comme paramètres de constructeur les objets nécessaires au fonctionnement d’une classe plutôt que de laisser la nouvelle instance créer ces objets elle-même. Grâce à la notion d’interface, nous pouvons réaliser une injection de dépendance en découplant totalement l’utilisation de l’objet passé par injection de son implémentation.

Si nous souhaitons créer une classe pour représenter une transaction bancaire, nous pouvons réaliser l’implémentation suivante :

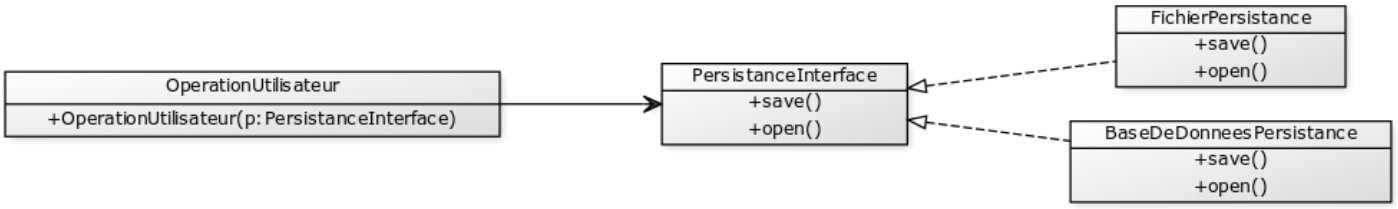
```
package com.cgi.udev.compte;


import java.time.Instant;


public class TransactionBancaire {
```

```
  private final Compte compte;
  private final int montant;
  private Instant date;

  public TransactionBancaire(Compte compte, int montant) {
    this.compte = compte;
    this.montant = montant;
  }

  public void effectuer() throws OperationInterrompueException, CompteBloqueException {
    if (isEffectuee()) {
      return;
    }
    compte.retirer(montant);
    date = Instant.now();
  }

  public void annuler() throws OperationInterrompueException, CompteBloqueException {
    if (! isEffectuee()) {
      return;
    }
    compte.deposer(montant);
    date = null;
  }

  public boolean isEffectuee() {
    return date != null;
  }

  public Instant getDate() {
    return date;
  }
}
```

L'implémentation précédente permet d'effectuer une transaction pour un compte donné et un montant donné et mémorise la date. Elle permet également d'annuler la transaction. Dans cette implémentation, nous avons réalisé une **inversion de dépendance**. La transaction ne connaît pas la nature exacte de l'objet *Compte* qu'elle manipule. La classe *TransactionBancaire* fonctionnera quelle que soit l'implémentation sous-jacente de l'interface *Compte*.

L’inversion de dépendance est un principe de programmation objet qui stipule que si une classe A est dépendante d’une classe B, alors il peut être souhaitable que, non seulement la classe A reçoive une instance de B par injection, mais également que B ne soit connue qu’à travers une interface.

L’inversion de dépendance est très souvent utilisée pour isoler les couches logicielles d’une architecture. Au sein d’une application, nous pouvons disposer d’un ensemble de classes pour gérer des opérations utilisateur et d’un ensemble de classes pour assurer la persistance des informations.



L’architecture logicielle peut utiliser l’inversion de dépendance pour assurer que les opérations utilisateur qui ont besoin de réaliser des opérations persistantes réalisent des appels à travers des interfaces qui sont injectées. D’un côté, on peut imaginer implémenter différentes classes gérant la persistance pour sauver les informations dans des fichiers, dans des bases de données ou sur des serveurs distants (et même nulle part si on souhaite exécuter le code dans un environnement de test). D’un autre côté on peut créer et faire évoluer un système de persistance en ayant une dépendance minimale aux opérations utilisateur puisque le système de persistance doit juste fournir des implémentations conformes aux interfaces.

Revision #1
Created 10 February 2025 20:58:42 by Nicolas
Updated 10 February 2025 21:00:50 by Nicolas