IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# TP8-9 – Liste générique et tri

**Informatique – MAPD**

**Correction**

## Objectifs

- Décider dans quels cas utiliser des patrons de conception (null object, composite, singleton)
- Utiliser la généricité pour implémenter des listes.
- S'appuyer sur le compilateur et le système de type pour résoudre des erreurs dans le code.
- Construire une solution logicielle de manière itérative

Cet exercice sera l'occasion d'aborder un style (paradigme) de programmation fonctionnel pur. À chaque étape vous devez d'abord élaborer un diagramme UML puis le coder et ensuite enrichir tout cela au fur et à mesure des questions.

À chaque étape, vous devez vous poser des questions au sujet des liens (héritage et composition) entre les classes et les interfaces et des adaptations à faire dans votre code.

Les différents exercices sont un guide pour arriver à une solution mais il y a plusieurs variantes et l'on peut y arriver par plusieurs cheminements.

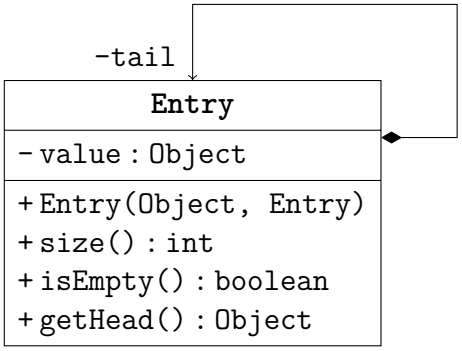
## Notes pédagogiques

- Sujet sur deux séances. Pas mal de choses à faire et parfois très technique.
- Choix du patron, place des méthodes, interfaces, classe abstraite : faut il être plus directif et leur montrer une seule solution ?
- Par exemple définition d'exception ou pas pour certaines méthodes (exemple avec `getTail`)
- Définition du singleton
- Définition d'un accesseur du second élément
- Utilisation de `Comparable<T>`
- Organisation des classes et des interfaces, c'est assez coton

## Exercice 1 (*Liste*)

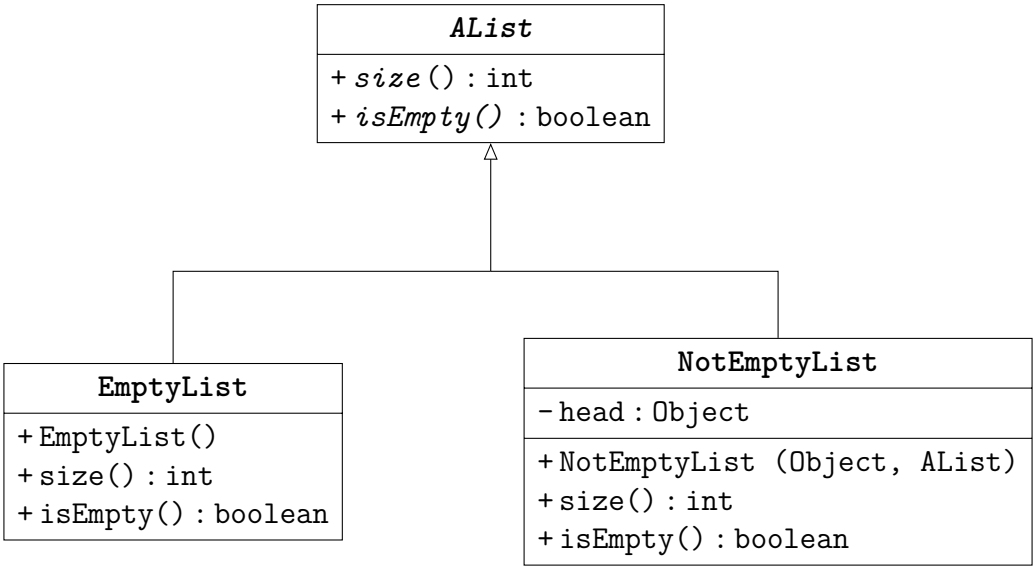
Pour commencer voyons comment construire une liste. Une liste est une succession de « cases » qui contiennent une valeur et qui se termine par une case qui ne contient aucune valeur.

Une solution simpliste consiste à créer une classe comme celle qui suit :




puis de réaliser les méthodes en vérifiant souvent si `tail` vaut **`null`** ou pas.

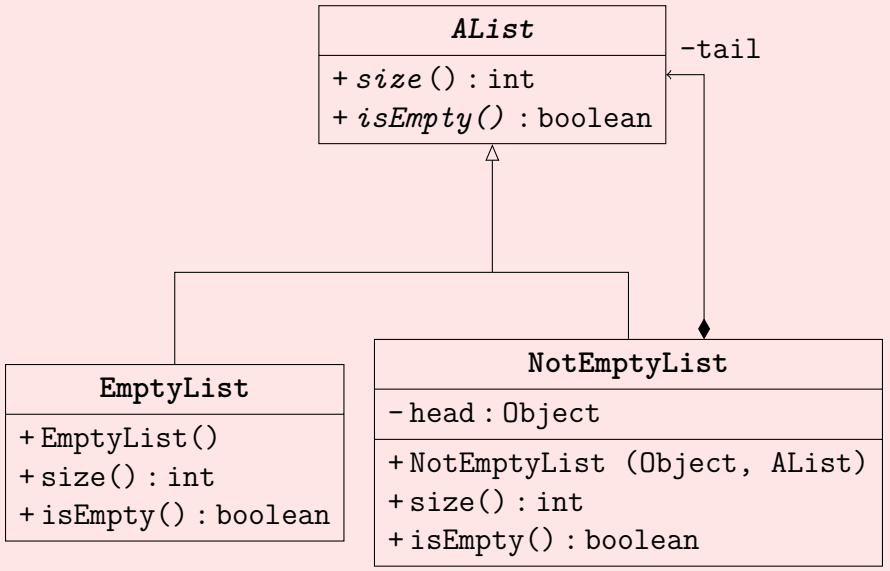
Pour éviter de multiple tests `tail == null`, un patron classique propose de créer une classe pour la valeur spécifique (ici **`null`**). Puis de faire hériter les 2 classes d'une même abstraction. On obtient :




▷ **Question 1.1 :**
**Quel mécanisme permet cet aiguillage entre les méthodes selon que la case est vide ou non ?**

La liaison dynamique, après redéfinition ...

▷ **Question 1.2 :**
**Modifier le diagramme de classe pour qu'une case non vide ait un successeur.**




▷ **Question 1.3 :**
**Implanter et tester en créant des listes de longueur 0, 1, 2.**

Une des difficultés est la façon dont les étudiants perçoivent la construction d'une liste.
On peut commencer par expliquer comment, en fonctionnel, on construit une liste. `EmptyList()` + `NotEmptyList(val, suite)`.
Par exemple :

- Le plus simple : une liste vide : `AList l = new EmptyList();`
- On peut revenir sur :
  - TypeApparent = le plus abstrait possible (ici `AList`)
  - TypeRéel : le plus concret possible (avec TypeRéel <: TypeApparent)
- Une liste avec 1 élément : on crée une liste avec une case qui contient une valeur et une suite...vide. `AList l = new NotEmptyList(7, new EmptyList());`
- On généralise : `AList l = new NotEmptyList(7, new NotEmptyList(5, new EmptyList()));`

L'écriture des méthodes `size()` et `isEmpty()` sont intéressantes : retour d'une constante (intérêt de la liaison dynamique sur null object) et récursivité.

### AList

```
package exercice1;

public abstract class AList {
  public abstract int size();
  public abstract boolean isEmpty();
}
```

### EmptyList

```
package exercice1;
```

```
public class EmptyList extends AList {


  @Override
  public int size() {
    return 0;
  }

  @Override
  public boolean isEmpty() {
    return true;
  }

}
```

## NotEmptyList

```
package exercice1;

public class NotEmptyList extends AList {
  private Object head;
  private AList tail;
  public NotEmptyList(Object head, AList tail) {
    this.head = head;
    this.tail = tail;
  }

  @Override
  public int size() {
    return 1+tail.size();
  }

  @Override
  public boolean isEmpty() {
    return false;
  }

}
```

## Main

```
package exercice1;

public class Main {
  public static void main(String[] args) {
    AList l0 = new EmptyList();
    AList l1 = new NotEmptyList(1,new EmptyList());
    AList l2 = new NotEmptyList(1,new NotEmptyList(2,(new EmptyList())));
    System.out.println("l0 size : "+l0.size()+" empty ? "+l0.isEmpty());
    System.out.println("l1 size : "+l1.size()+" empty ? "+l1.isEmpty());
    System.out.println("l2 size : "+l2.size()+" empty ? "+l2.isEmpty());
```

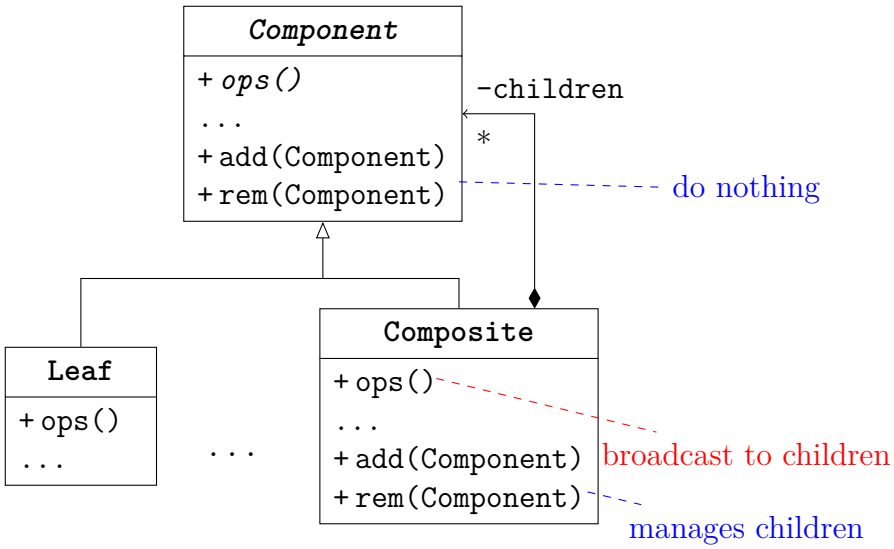



FIGURE 1 – Patron de conception *composite*

```
  }
}
```

## Exercice 2 (*Patron*)

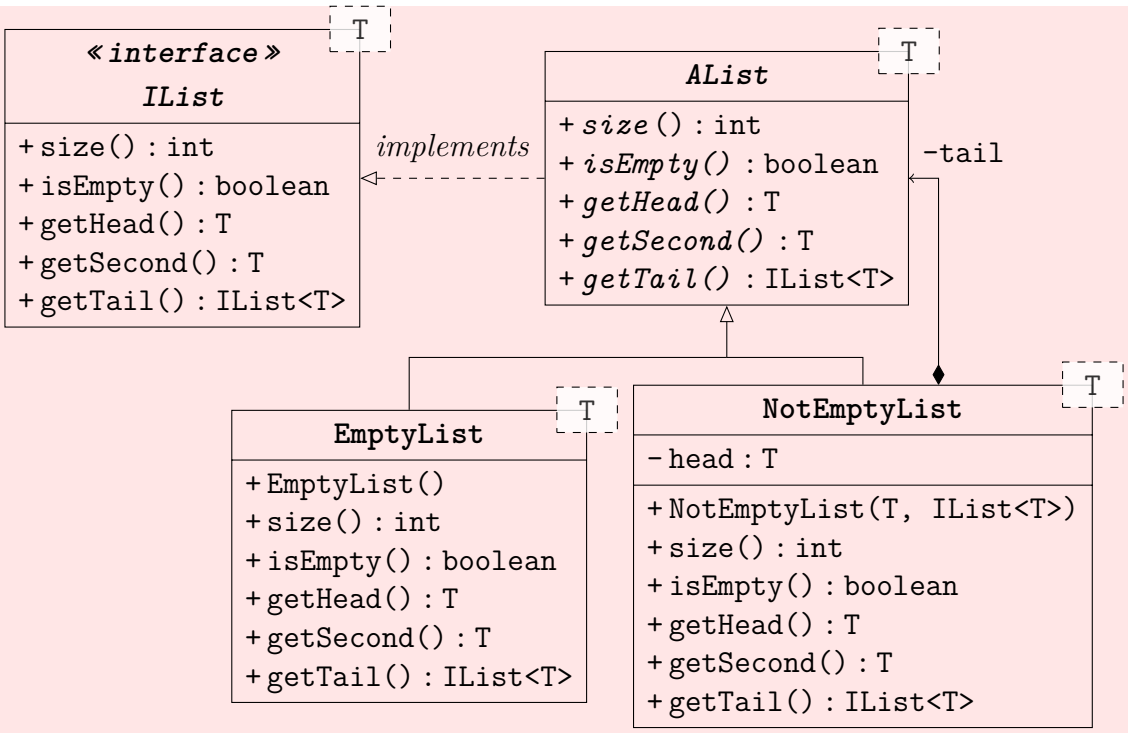
Le patron *composite* est un patron de conception structurel qui permet de concevoir une structure arborescente. La figure 1 présente le principe de ce patron :

- *Component* est l'abstraction pour tous les éléments qui sont dans la structure, y compris ceux qui ont des sous-éléments. Ses méthodes sont abstraites ;
- *Leaf* représente un élément qui n'a pas de sous-éléments ;
- *Composite* représente un élément ayant des sous-éléments. Cette classe offre des méthodes pour ajouter/supprimer des enfants et implémente le comportement en utilisant les sous-éléments.

▷ **Question 2.1 :**
**En vous inspirant du patron *composite*, proposez un diagramme de classes avec les contraintes suivantes :**

1. **Une interface qui expose au moins : `size()`, `isEmpty()`, `getHead()` (accès au premier élément), `getTail()` (accès au reste de la liste).**
2. **Utilisez la généricité (type plus spécifique que `Object`).**
3. **Pour simplifier, on ne définit pas de `add()` ni de `remove()`.**
4. **Réifiez la liste vide**
   - **peut on utiliser un singleton pour la liste vide ?**
   - **peut-il être générique ?**
5. **Définissez un accesseur du `second` élément.**

Il existe plusieurs solutions. Celle implémentée dans la correction est ci-dessous :




Dans la solution, le Composite est utilisé avec une composition non * mais de cardinalité 1. C'est une adaptation normale et suffisante du patron composite.

**Questions sur le *singleton*.** Pour créer un *singleton* de manière classique :

```
/*
 * First attempt of a singleton
 */
private static final EmptyList<T> single = new EmptyList<T>();
// -> cannot make a static reference on a non static type T
```

On crée une constante statique (un singleton est une constante statique fondamentalement). **Or,** le compilateur doit connaître le type exact pour définir la constante et donc elle ne peut pas être générique.
On pourrait tenter le joker *wildcard*. Ça donne l'impression d'être possible plus longtemps :

```
/*
 * Second attempt ... seems to work ... no compilation problem
 */
public class Singleton<T> extends EmptyList<T>{
  private static EmptyList<?> emptyList = new EmptyList<>();
  private Singleton() {};

  public static EmptyList<?> getInstance(){
              return emptyList;
        }
}


/*
```

```
 * When trying to create a singleton empty list ... for exemple for integers...
 */
 EmptyList<Integer> emptyList = (EmptyList<Integer>) Singleton.getInstance();
 // Type safety: unchecked cast from EmptyList<Capture#1 -of ?> to EmptyList<Integer>
```

Donc, on a un problème de contrôle de typage lorsqu'on essaie d'utiliser le singleton...
Dernier essai : il est possible de créer le singleton comme ci-après.

```
public class Singleton extends EmptyList<Integer> {
     private static final Singleton single = new Singleton();
}
```

Mais il faudrait un singleton par `T`.
Conclusion : pas de singleton sauf à ne pas rendre générique cette classe (ce qui est possible mais pas non plus sans difficultés si on veut conserver `NotEmptyList<T>`).

▷ **Question 2.2 :**
**Implémentez et testez.**

**Vous pouvez définir une exception comme `UndefinedPlaceException`.**

Ici il faut définir et utiliser une exception (`UndefinedPlaceException`). Elle sera levée systématiquement en cas de liste vide.

```
package exercice2.withininterface;

public class EmptyList<T> extends AList<T> {

  @Override
  public int size() {
    return 0;
  }

  @Override
  public boolean isEmpty() {
    return true;
  }

  @Override
  public T getHead() throws UndefinedPlaceException{
    throw new UndefinedPlaceException();
  }

  @Override
  public IList<T> getTail() throws UndefinedPlaceException{
    throw new UndefinedPlaceException();
  }

  @Override
  public String toString() {
    return "";
```

```
  }

  @Override
  public T getSecond() throws UndefinedPlaceException {
    throw new UndefinedPlaceException();
  }
}

package exercice2.withininterface;

public class NotEmptyList<T> extends AList<T> {

  private T head;
  private IList<T> tail;

  public NotEmptyList(T elt, IList<T> suite) {
    this.head = elt;
    this.tail = suite;
  }

  @Override
  public int size() {
    return 1 + tail.size();
  }

  @Override
  public boolean isEmpty() {
    return false;
  }

  // Never raise UndefinedPlaceException
  @Override
  public T getHead() throws UndefinedPlaceException {
    return this.head;
  }

  // Never raise UndefinedPlaceException
  @Override
  public IList<T> getTail() throws UndefinedPlaceException {
    return this.tail;
  }

  @Override
  public String toString() {
    return head.toString() + ", " + tail.toString();
  }

  // Raises UndefinedPlaceException when getHead() called in an EmptyList
  @Override
  public T getSecond() throws UndefinedPlaceException {
    return tail.getHead();
```

```
  }
}
```

Et une classe `Main` pour tester :

```
package exercice2.withininterface;

public class Main {

  public static void main(String[] args) throws UndefinedPlaceException {
    NotEmptyList<Integer> myList = new NotEmptyList<Integer>(5,
        new NotEmptyList<>(2, new NotEmptyList<>(6, new EmptyList<>())));

    System.out.println("[" + myList + "]");

    System.out.print(myList.getHead() + " -- (" + myList.getTail() + ") -- ");
    System.out.println(myList.getSecond());
    System.out.println();

    myList = new NotEmptyList<Integer>(6, new EmptyList<>());
    System.out.println("[" + myList + "]");
    System.out.print(myList.getHead() + " -- (" + myList.getTail() + ") -- ");
    System.out.println(myList.getSecond());
    System.out.println();

// EmptyList<Integer> anotherList = new EmptyList<Integer>();
// System.out.println(anotherList.getHead());
// System.out.println(anotherList.getTail());
// System.out.println(anotherList.getSecond());


  }
}
```

## Exercice 3 (*Classe Sortable*)

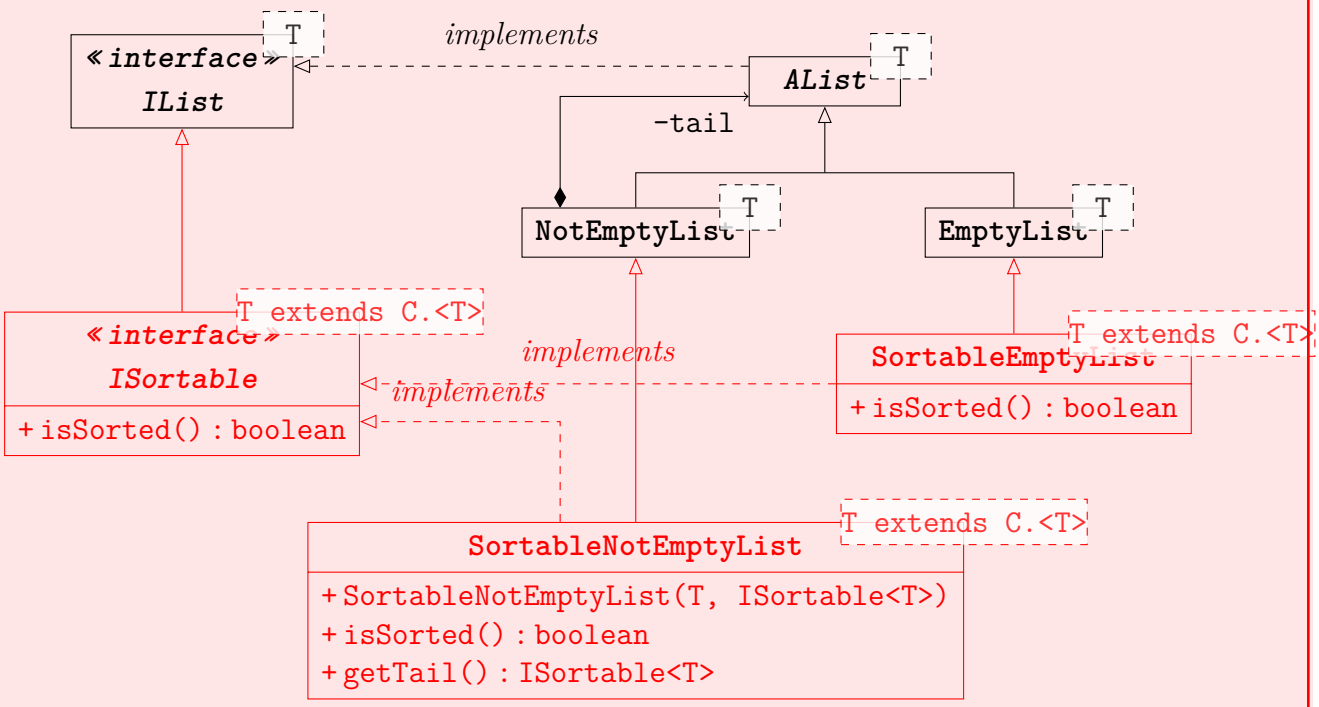
Nous allons introduire la possibilité d'une liste d'être triable. Nous allons observer les conséquences sur la structure et les conséquences du typage.

▷ **Question 3.1 :**
**Faites évoluer le diagramme de classes pour introduire une interface `ISortable` avec une méthode qui permet de savoir si la liste est triée (`isSorted()`). Ajoutez également les classes concrètes utiles. Notez que pour être triable, les éléments de la liste doivent être `Comparable`...**

**Modifiez ensuite votre code pour respecter le nouveau diagramme de classes.**

La méthode `isSorted` peut-être placée soit dans `IList` soit dans `ISortable`. Dans le premier cas il faudra implémenter la méthode pour les classes `NotEmptyList` et `EmptyList` ce qui demande à vérifier dans le code que les élements de la liste sont comparables...Pour éviter cette gestion,

nous considérons qu'une liste ne peut dire si elle est triable que si elle implémente l'interface `ISortable`...
`C.<T>` est un raccourcis pour `Comparable<T>`. Les modifications pour rapport à la question précédente sont notées en rouge.




```
package exercice3;

public interface ISortable<T extends Comparable<T>> extends IList<T> {

  public boolean isSorted();
}
```

```
package exercice3;

public class SortableEmptyList<T extends Comparable<T>>
extends EmptyList<T> implements ISortable<T> {

  @Override
  public boolean isSorted() {
    return true;
  }

}
```

```
package exercice3;

public class SortableNotEmptyList<T extends Comparable<T>>
extends NotEmptyList<T> implements ISortable<T> {
```

```
  public SortableNotEmptyList(T elt, ISortable<T> suite) {
    super(elt, suite);
  }

  @Override
  public ISortable<T> getTail(){
    try {
      return (ISortable<T>) super.getTail();
    } catch (UndefinedPlaceException e) {
      // Never here
      return null;
    }
  }

  @Override
  public boolean isSorted() {
    // no type check ; we know elements are Comparable
    if (getTail().isEmpty())
      return true;
    else {
      boolean firstSmallerSecond = false;
      try {
        firstSmallerSecond = getHead().compareTo(this.getSecond()) <= 0;
      } catch (UndefinedPlaceException e) {
        // Never here
      }
      return firstSmallerSecond&&getTail().isSorted();
    }
  }

}
```

## Exercice 4 (*Classe Sorted*)

Nous allons introduire maintenant la notion de liste triée.

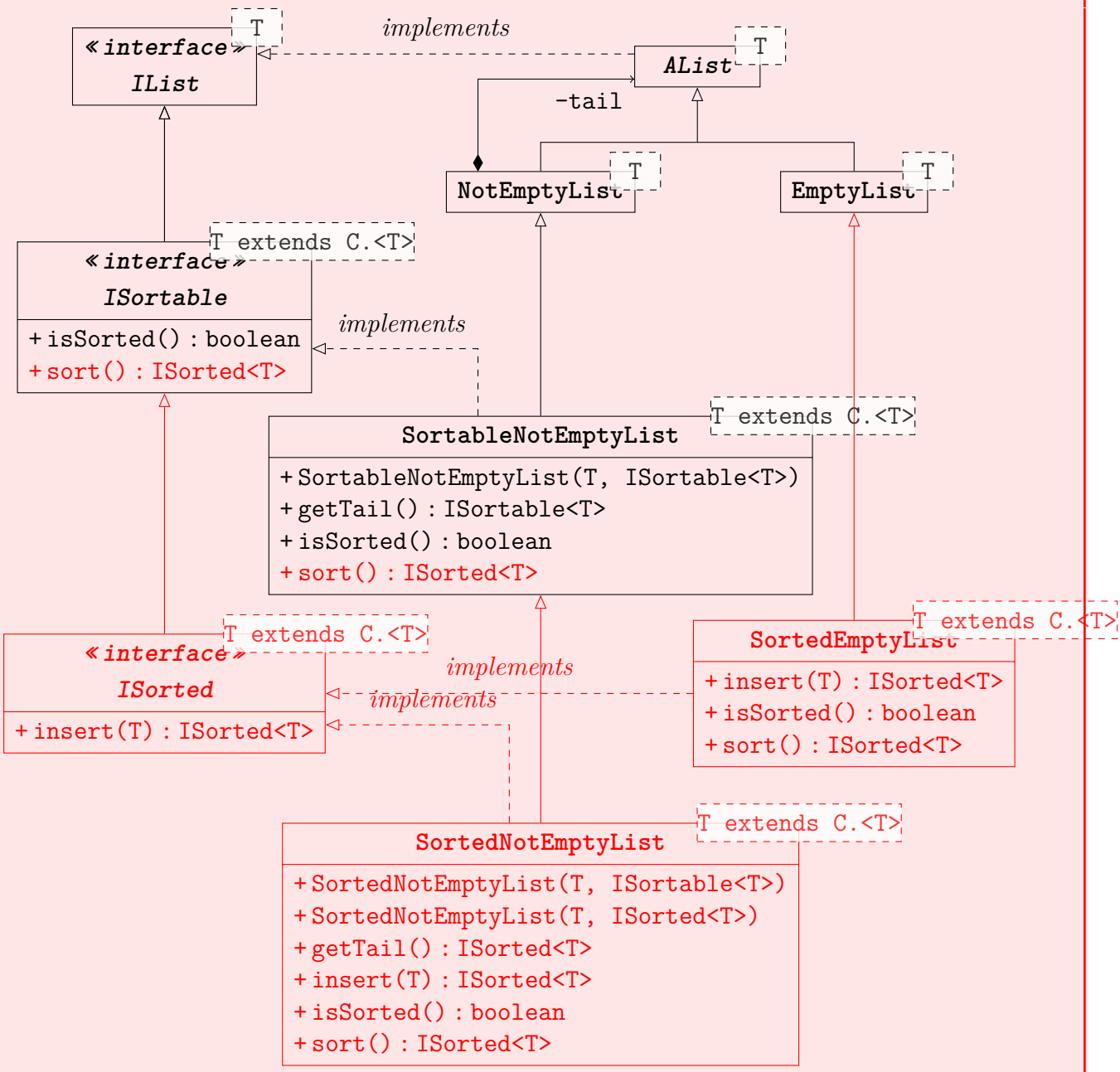
▷ **Question 4.1 :**
**Pour cela, vous devez modifier votre diagramme de classes pour avoir une méthode `sort()` qui produit, à partir d'une liste triable, une nouvelle liste triée (de type `ISorted`). Les listes triées doivent offrir une méthode `insert` qui permet d'implémenter le tri par insertion.**

**Modifiez ensuite votre code pour respecter le nouveau diagramme de classes.**

La méthode `sort` est à ajouter à l'interface `ISortable` : ce sont des instances de ce type qui doivent pouvoir être triées. Le résultat ce sont des listes triées, des listes qui ont des nouvelles propriétés, de là la création d'une nouvelle interface `ISorted`. Elle offre la méthode `insert()` qui permet d'insérer dans la liste qui est déjà triée (tri par insertion) :

— résoudre les cas de base particuliers pour les listes de taille 0 et 1 ;

— ensuite, si je sais que ma liste est triée, insérer un nouvel élément dedans est facile par propagation.




Il faut garantir, à la construction de la liste, qu'elle est triée. Une solution est d'utiliser une méthode privée `placeHead()` pour placer la tête au bon endroit.

```
package exercice4;

public interface ISortable<T extends Comparable<T>> extends IList<T> {

  public boolean isSorted();
  public ISorted<T> sort();
}
```

```java
package exercice4;



public class SortableNotEmptyList<T extends Comparable<T>>
extends NotEmptyList<T> implements ISortable<T> {

  public SortableNotEmptyList(T elt, ISortable<T> suite) {
    super(elt, suite);
  }

  @Override
  public ISortable<T> getTail(){
    try {
      return (ISortable<T>) super.getTail();
    } catch (UndefinedPlaceException e) {
      // Never here
      return null;
    }
  }

  @Override
  public boolean isSorted() {
    // no type check ; we know elements are Comparable
    if (getTail().isEmpty())
      return true;
    else {
      boolean firstSmallerSecond = false;
      try {
        firstSmallerSecond = getHead().compareTo(this.getSecond()) <= 0;
      } catch (UndefinedPlaceException e) {
        // Never here
      }
      return firstSmallerSecond&&getTail().isSorted();
    }
  }
  @Override
  public ISorted<T> sort() {
    // recursive attempt .... create a new object
    return getTail().sort().insert(this.head);
  }

}

package exercice4;

public interface ISorted<T extends Comparable<T>> extends ISortable<T> {

  // return a ISorted plus one element
  public ISorted<T> insert(T elt);
```

```
}

package exercice4;

public class SortedEmptyList<T extends Comparable<T>>
extends EmptyList<T> implements ISorted<T> {

  @Override
  public ISorted<T> sort() {
    return this;
  }

  @Override
  public ISorted<T> insert(T elt) {
    return new SortedNotEmptyList<T>(elt, this);
  }

  @Override
  public boolean isSorted() {
    return true;
  }

}

package exercice4;

public class SortedEmptyList<T extends Comparable<T>>
extends EmptyList<T> implements ISorted<T> {

  @Override
  public ISorted<T> sort() {
    return this;
  }

  @Override
  public ISorted<T> insert(T elt) {
    return new SortedNotEmptyList<T>(elt, this);
  }

  @Override
  public boolean isSorted() {
    return true;
  }

}

package exercice4;

public class SortedNotEmptyList<T extends Comparable<T>>
extends SortableNotEmptyList<T> implements ISorted<T> {
```

```
public SortedNotEmptyList(T elt, ISortable<T> suite) {
  super(elt, suite.sort());
  placeHead();
}

public SortedNotEmptyList(T elt, ISorted<T> suite) {
  super(elt, suite);
  placeHead();
}

// utility for sorting at constructor time
private void placeHead() {
  T first = this.head;
  try {
    T second = this.getSecond();
    // 0 ; -1 or 1
    int res = first.compareTo(second);
    if (res > 0) {// switch and propagate
      setHead(second);
      ((NotEmptyList<T>) getTail()).setHead(first); // exception ensure correctness
      ((SortedNotEmptyList<T>) getTail()).placeHead();
    }
  } catch (UndefinedPlaceException e) {
    // no second ; do nothing
  }
}

@Override
public ISorted<T> sort() {
  return this; // sorted by construction
}

// Warning : insertion on place.
// no use of constructors because of loss of reference
// a Sortable cannot become a Sorted... so future use of sort() isSorted() not optimal.
@Override
public ISorted<T> insert(T elt) {
  T first = this.head;
  // 0 ; -1 or 1
  int res = elt.compareTo(first);
  // no duplication
  if (res == 0)
    return this;
  else if (res < 0)
    return new SortedNotEmptyList<T>(elt, this);
  else
    return new SortedNotEmptyList<T>(first, new SortedNotEmptyList<T>(elt, getTail()));
}

public boolean isSorted() {
  return true; // by definition
```

```
  }

  // In order to insert... need to redefine getTail... covariant return.
  @Override
  public ISorted<T> getTail() {
    return ((ISorted<T>) super.getTail());
  }
}
```

## Exercice 5 (*Pour aller plus loin*)

Questions ouvertes :

▷ **Question 5.1 :**

1. **Définir la méthode `getHead` dans la sous-classe `NotEmptyList`, si oui que faut-il faire ? Comparez votre solution avec celle de la correction.**
2. **Est-il utile d'avoir une classe abstraite, où et à quoi elle peut servir ?**
3. **Essayez des variantes en rendant les méthodes les plus abstraites possibles (plus haut dans la hiérarchie d'héritage).**