Interfaces et spécialisation multiple

Conception par Objets, GLIN404

6 mars 2017

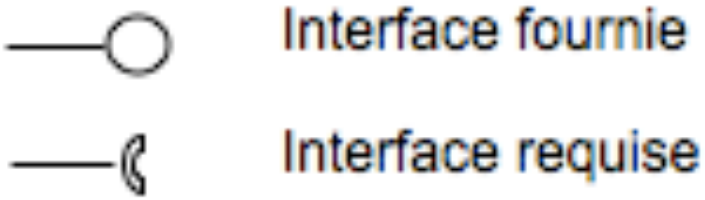
Sommaire

La notion d'interface dans le cadre de la modélisation

- zone de contact
- façade pour une implémentation (classe, programme)
- décrit par exemple pour une classe :
  - ce qu'elle peut offrir comme services à un programme
  - ce qu'elle requiert comme services de son environnement ou du programme
  - l'un des rôles qu'elle peut jouer

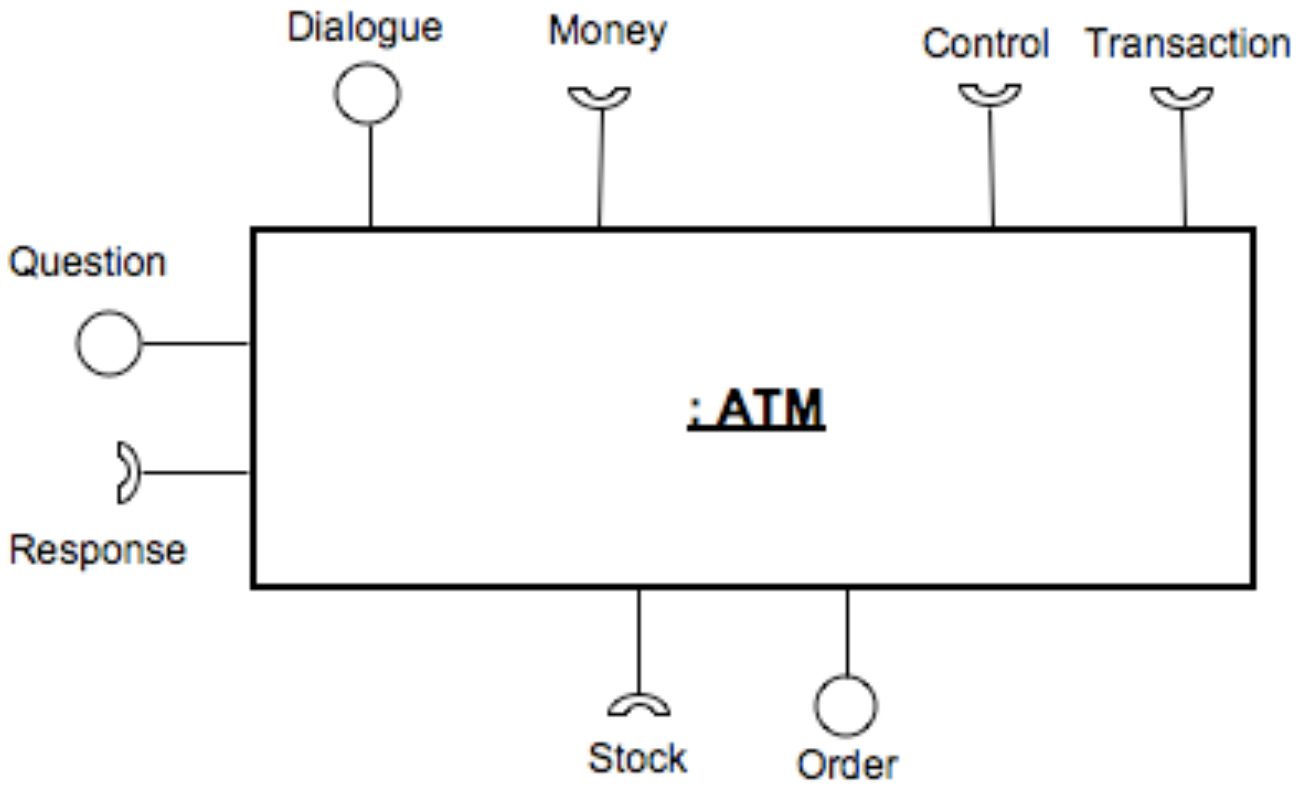
Interface en UML

- Ensemble nommé de propriétés publiques
- Constituant un service cohérent offert par une classe
- Ne spécifie pas la manière dont ce service est implémenté
- Si l'interface déclare un attribut, celui-ci n'est pas forcément implémenté dans la classe, mais doit apparaître aux observateurs

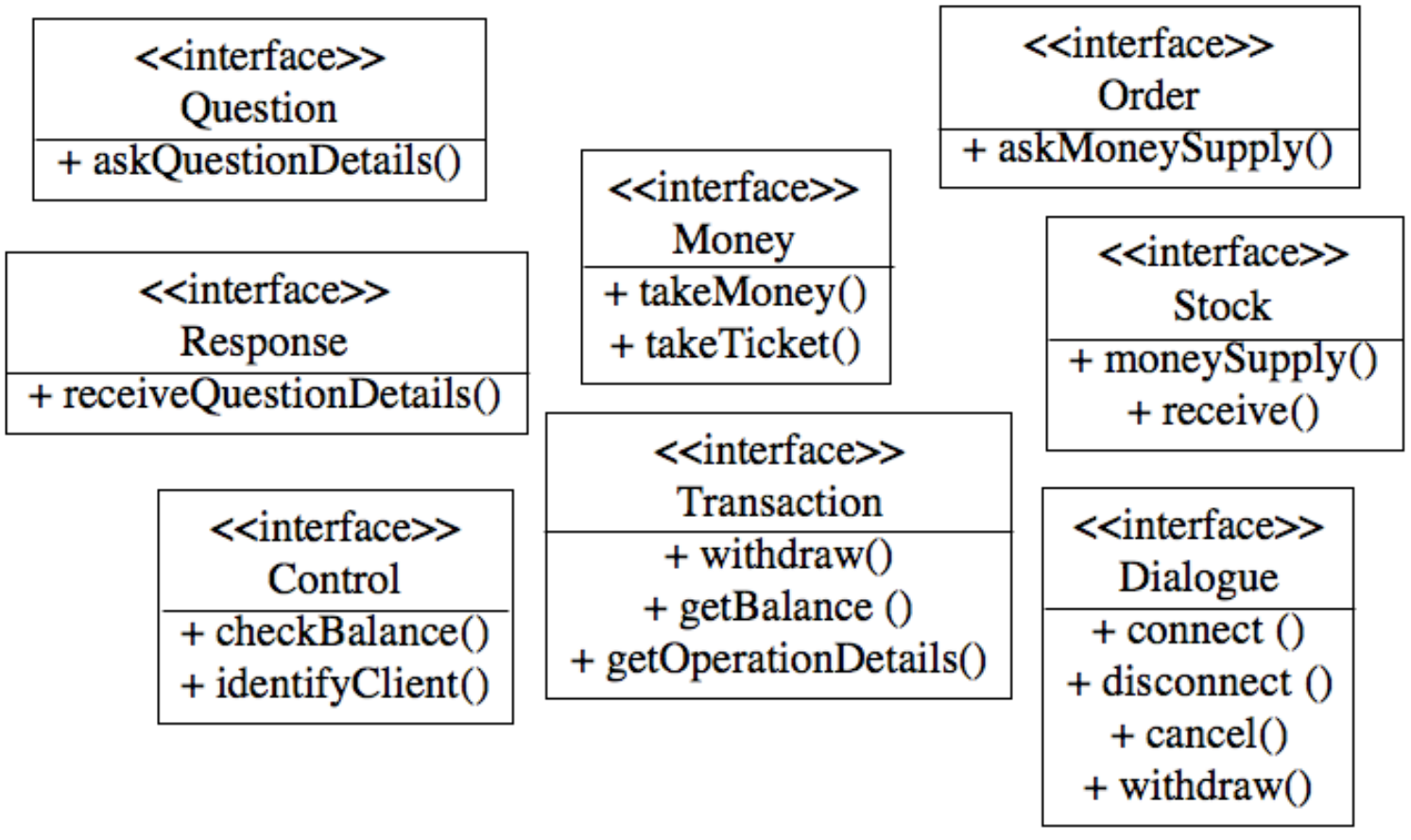


Un composant en UML

Brique logicielle de haut niveau décrite par des interfaces

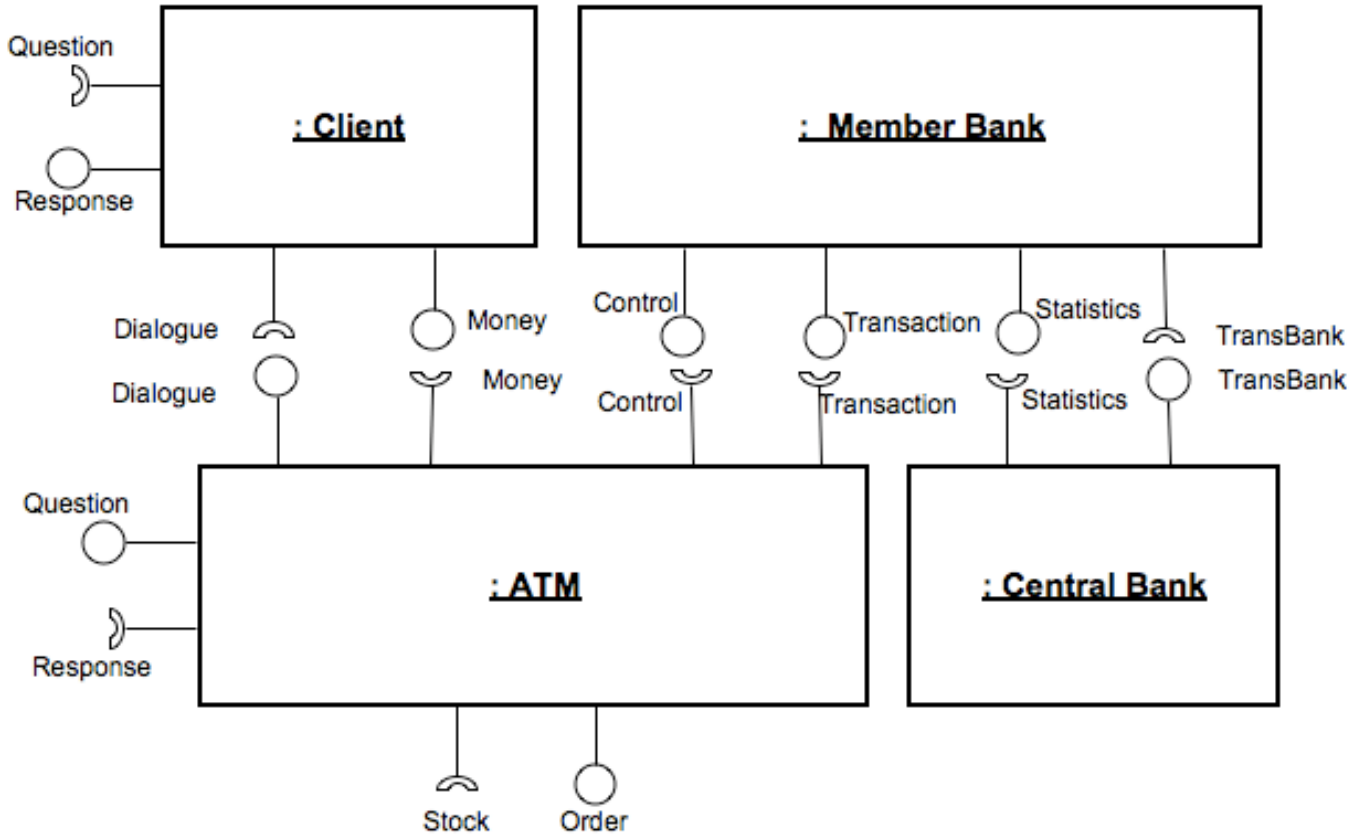


Interfaces en UML



Un assemblage de composants en UML

Architecture de haut niveau d'un logiciel



Interface en Java

- Se limite à présenter des services fournis par une classe
- Un ensemble de :
  - signatures de méthodes publiques (méthodes d'instances abstraites)
  - variables de classes constantes et publiques (constantes statiques)

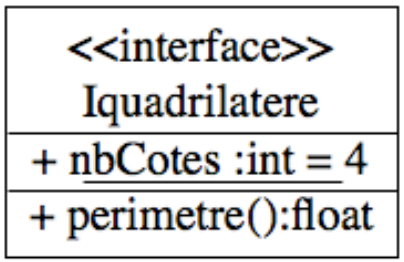
Interface en Java

Améliorer le code :

- on décrit des types de manière plus **abstraite** qu'avec les classes et par conséquent ces types sont plus **réutilisables** ;
- c'est une technique pour **masquer l'implémentation** puisqu'on découple la partie publique d'un type de son implémentation ;
- on peut écrire du **code plus générique** (plus général), au sens où il est décrit sur ces types plus abstraits ;
- les relations de spécialisation entre les interfaces (d'une part) et entre les classes et les interfaces (d'autre part) relèvent de la **spécialisation multiple**, ce qui facilite l'organisation des types d'un programme.

Sommaire

Une interface quadrilatère en UML



Définition d'une Interface

```
public interface Iquadrilatere
{
  public static final int nbCotes = 4;
  public abstract float perimetre();
}
```

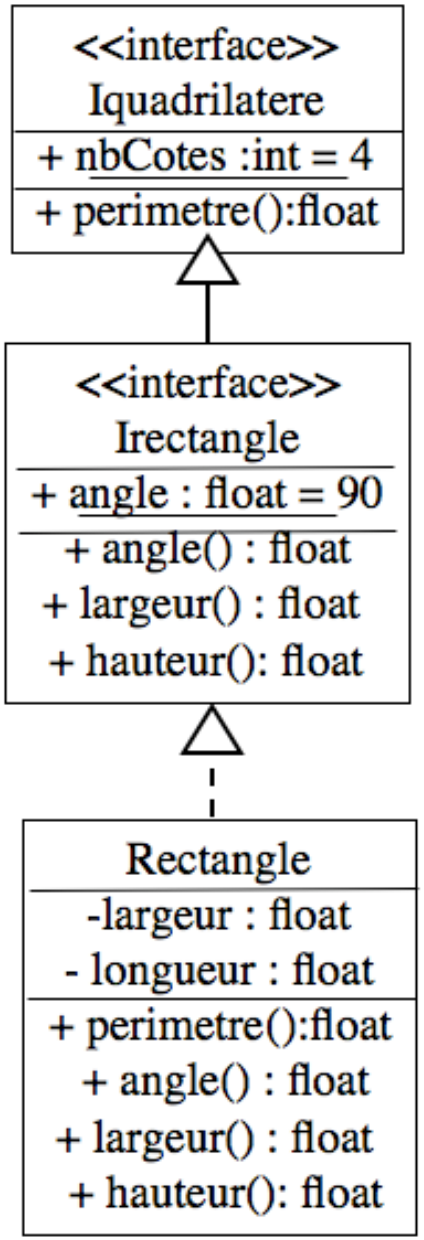
ou (en enlevant les mots-clefs obligatoires)

```
public interface Iquadrilatere
{
  int nbCotes = 4;
  float perimetre();
}
```

Spécialisation d'une interface

```
public interface Irectangle extends Iquadrilatere
{
float angle = 90;
float angle();
float largeur();
float hauteur();
}
```

Interface et réalisation par une classe en UML



Interface et réalisation par une classe en Java / solution 1

L'implémentation utilise deux attributs de type réel

```
public class Rectangle implements Irectangle
{
private float largeur,hauteur;
public Rectangle(){}
public Rectangle(float l, float h){largeur=l; hauteur=h;}

public float perimetre(){return 2*largeur()+2*hauteur();}
public float angle(){return Irectangle.angle;}
public float largeur(){return largeur;}
public float hauteur(){return hauteur;}
}
```

Interface et réalisation par une classe en Java / solution 2

L'implémentation uilise un tableau de deux réels

```
class RectangleTab implements Irectangle
{
private float tab[]=new float[2];
public RectangleTab(){}
public RectangleTab(float l, float h){tab[0]=l; tab[1]=h;}

public float perimetre(){return 2*largeur()+2*hauteur();}
public float angle(){return Irectangle.angle;}
public float largeur(){return tab[0];}
public float hauteur(){return tab[1];}
}
```

Limite des interfaces

On ne peut pas factoriser de code commun dans une interface
On peut introduire une classe (abstraite) pour cette factorisation

```
abstract public class RectangleAbs implements Irectangle
{
public RectangleAbs(){}
public float perimetre(){return 2*largeur()+2*hauteur();}
public float angle(){return Irectangle.angle;}
}
```

Abstraite car :

- Elle ne propose pas de modèle d'implémentation en mémoire (pas d'attributs)
- Elle n'implémente pas `largeur` ni `hauteur`

Interface et réalisation par une classe en Java
Retour sur la solution 1

```
public class Rectangle extends RectangleAbs
{
private float largeur,hauteur;
public Rectangle(){}
public Rectangle(float l, float h){largeur=l; hauteur=h;}

public float largeur(){return largeur;}
public float hauteur(){return hauteur;}
}
```

Interface et réalisation par une classe en Java
Retour sur la solution 2

```
public class RectangleTab extends RectangleAbs
{
private float tab[]=new float[2];
public RectangleTab(){}
public RectangleTab(float l, float h){tab[0]=l; tab[1]=h;}

public float largeur(){return tab[0];}
public float hauteur(){return tab[1];}
}
```

Sommaire

Description de comportements génériques

Noter : seules des interfaces et méthodes abstraites sont invoquées

```
public class StockRectangle
{
Vector<Irectangle> listeRectangle = new Vector<Irectangle>();

public void ajoute(Irectangle r){listeRectangle.add(r);}

public float sommePerimetres()
{
  float sp=0;
  for (int i=0; i<listeRectangle.size(); i++)
  {sp+=listeRectangle.get(i).perimetre();}
  return sp;
}
}
```

Description de comportements génériques

Noter : on peut mettre des rectangles de toutes sortes dans le stock de rectangles

```
public class TestStockRectangle
{
  public static void main (String [] arg)
  {
        StockRectangle st = new StockRectangle();
        st.ajoute(new RectangleTab(3,8));
        st.ajoute(new Rectangle(2,9));
.....
  }
}
```

Sommaire

Spécialisation multiple

- moins contraignante
- plus naturelle pour exprimer des relations de classification quelconques
- permet une factorisation maximale dans tous les cas

Complétons nos interfaces

```
public interface ILosange extends Iquadrilatere
{
float cote();
}
```

Un carré est à la fois un rectangle et un losange :

```
public interface ICarre extends IRectangle, ILosange
{
}
```

Discussion

- En utilisant seulement des classes (pour lesquelles on n'a que de l'héritage simple), les carrés ne pourraient être à la fois des rectangles et des losanges, comme c'est pourtant le cas dans le domaine des mathématiques. Conséquence : une méthode admettant des losanges en paramètre ne pourrait admettre des carrés !
- Toutes les propriétés ne pourraient pas factorisées, certaines seraient redondantes, comme `cote()` ou `angle()`.
- La multi-spécialisation que nous avons pu faire sur les interfaces limite ces problèmes.

Sommaire

Interfaces marqueurs

- vides d'opérations et de constantes de classes
- précisent la sémantique et indiquent dans quel contexte leurs objets peuvent être utilisés
- implémenter ces interfaces marqueurs n'est pas toujours suffisant pour obtenir le comportement attendu, mais c'est nécessaire

cloneable les objets peuvent être clonés : on peut leur appliquer une méthode `clone`, `protected` dans la classe `Object`, et qui doit être redéfinie `public` dans la classe concernée.
serializable les objets peuvent être « sérialisés » c'est-à-dire écrits dans un flux de données. `readObject` et `writeObject` sont réécrites si on désire une sérialisation particulière.

Comparaison d'objets et tris

L'API définit une interface `Comparable` dont le code est le suivant.

```
public interface Comparable<T>
{
int compareTo(T o);
}
```

`compareTo` retourne -1, 0, ou 1 suivant si l'objet receveur est plus petit, égal ou plus grand que le paramètre. Cette opération est notamment utilisée pour les opérations de tri de la classe `Collections`

Collections et itérateurs

L'API 1.5 a ajouté pour les collections une interface bien pratique qui se définit comme suit.

```
public interface Iterable<T>
{
Iterator<T> iterator();
}
```

Collections et itérateurs

Elle permet de parcourir les objets qui l'implémentent avec les méthodes usuelles de `Iterator`, en l'occurrence `hasNext()` et `next()` :

```
public float sommePerimetres()
{
  float sp=0;
  Iterator<Irectangle> It=listeRectangle.iterator();
  while (It.hasNext()){sp+=It.next().perimetre();}
  return sp;
}
```

Collections et itérateurs

... mais également avec une forme particulière de l'instruction `for` :

```
public float sommePerimetres()
{
  float sp=0;
  for (Irectangle r:listeRectangle)
   {sp+=r.perimetre();}
  return sp;
}
```

Hiérarchie des collections

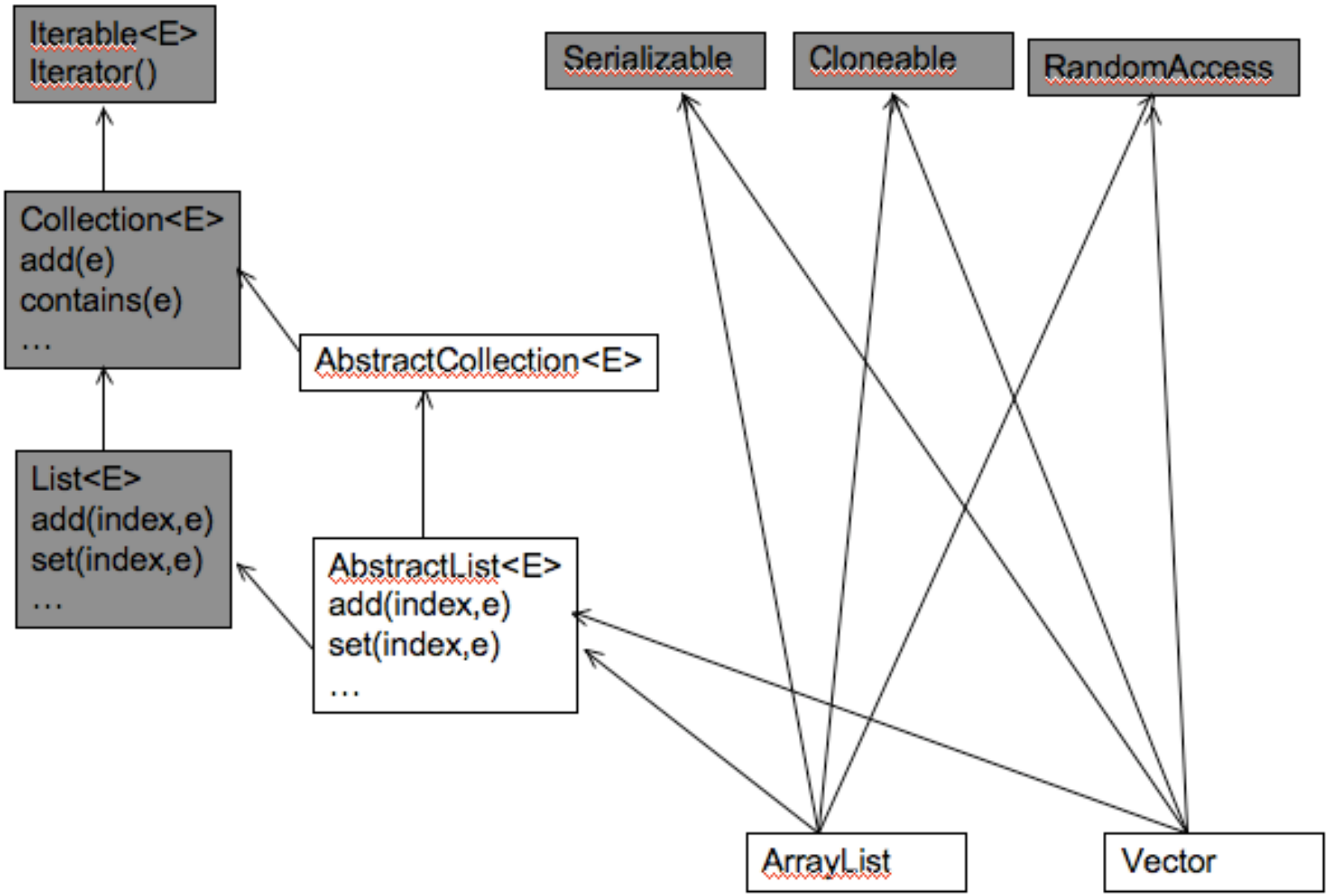


Figure – Extrait des collections