Introduction à la programmation objet

Interface et héritage

Alice Jacquot — www.lri.fr/~jacquot/ipo

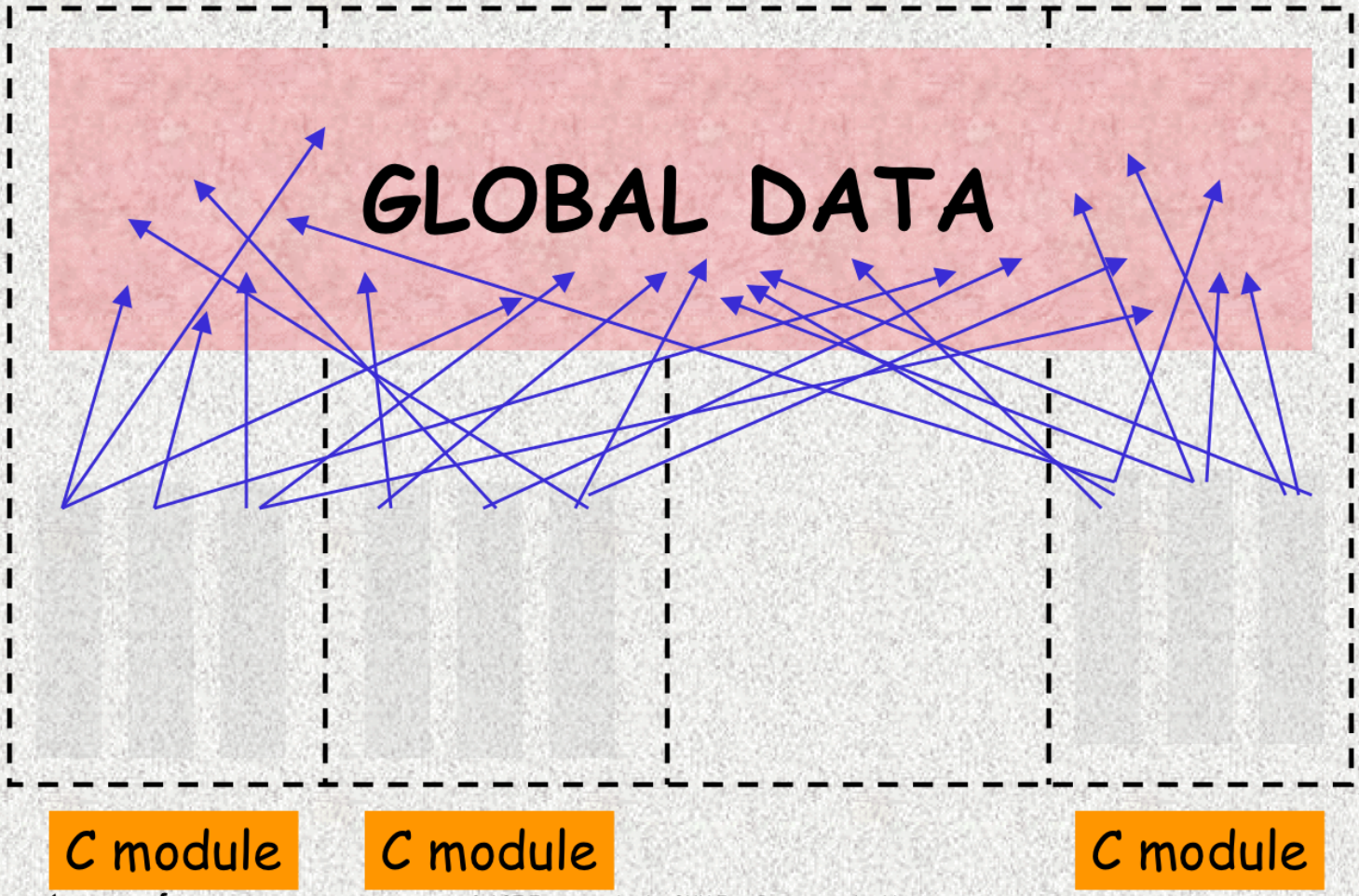
octobre-novembre 2019

Université Paris Saclay & LRI

Première partie I

Rappel : encapsulation

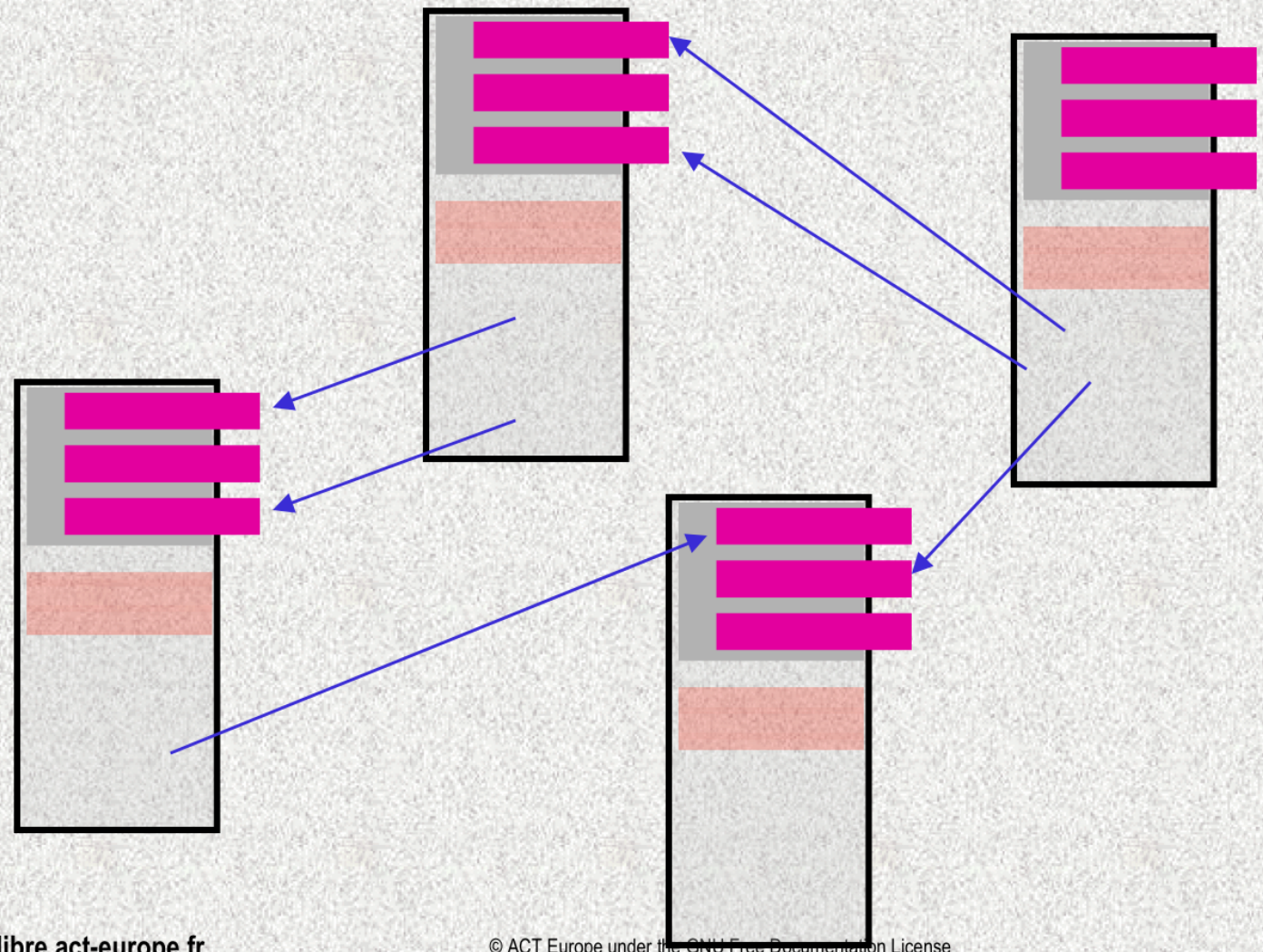
The C model



http://libre.act-europe.fr

© ACT Europe under the GNU Free Documentation License

The object model



http://libre.act-europe.fr

© ACT Europe under the GNU Free Documentation License

Intérêt de la protection de l'information

- permet de découpler les modules qui composent un système.
- les modules peuvent être :
  - développés indépendamment
  - testés indépendamment
  - optimisés indépendamment
  - utilisés indépendamment (et donc ré-utilisés)
  - compris indépendamment
  - modifié indépendamment

Deuxième partie II

Interface

Rappel : interface

Rappel

- interface = décrit ce que fait un objet (et non comment il le fait)
- interface = ensemble des méthodes publiques (et des attributs)

fournisseur du module

utilisateur du module

contrat

sur les services fournis par un module

Retour sur les piles/ensembles

- plusieurs implantation de piles sont envisageables : tableaux, listes chaînées, table circulaire, ...
- en POO : définition d'un objet par son interface (partie publique) et non par la structure de données utilisée (partie privée)
- intérêt de l'encapsulation : possibilité de changer l'implantation si l'interface ne change pas

En pratique

Supposons que l'on ait deux implantations d'une pile : `ArrayStack` et `LinkedListStack`

1re implémentation :

```
ArrayStack s = new ArrayStack(5);
s.push(4);
s.push(3);
```

2e implémentation :

```
LinkedListStack s = new LinkedListStack();
s.push(4);
s.push(3);
```

Problème

Mais : comment définir une méthode comparant le contenu de deux piles ?

Solution : les interfaces

Définition d'une interface

- mot-clé `interface` (s'utilise comme `class`)
- décrit un comportement commun (collection de méthodes)
- définit un nouveau type
- l'interface peut être vue comme une API (protocole de communication) entre deux objets / parties de programme

Implantation d'une interface

- une classe peut implanter une interface
  - mot-clé `implements`
  - doit implanter toutes les méthodes de l'interface
- si la class `A` implante l'interface `B`, les instances de `A` sont de type `A` et de type `B`

L'interface `Stack`

```
public interface Stack {
    public void push(int el);
    public int pop();
    public int peek();
    public boolean isEmpty();
}
```

- une interface n'a pas d'attributs
- une interface correspond à une liste de signatures
- la signature d'une méthode n'est suivie d'aucun code

Implantation i

```
public class ArrayStack implements Stack {

    private int[] data;
    private int lastElement;

    public ArrayStack(int capacity) {
        this.data = new int[capacity];
        this.lastElement = 0;
    }

    public boolean isEmpty() {
        return this.lastElement == 0;
```

Implantation ii

```
}

public void push(int el) {
    this.data[this.lastElement] = el;
    this.lastElement += 1;
}

public int pop() {
    this.lastElement -= 1;
    return this.data[this.lastElement];
}

public int length() {
```

Implantation iii

```
        return this.lastElement;
    }

    public int peek(int i) {
        return this.data[i];
    }
}
```

- donne une implantation de toutes les méthodes définies dans l'interface
- peut ajouter des définitions spécifiques à cette classe (p.ex. le constructeur)

Exemple d'erreur

```
~/java_l2/cours/heritage > javac *.java

ArrayStack.java:1: ArrayStack is not abstract and does not
override abstract method peek() in Stack

public class ArrayStack implements Stack {
       ^
1 error
```

Correction

```
public int peek() {
  return this.peek(this.lastElement);
}
```

au passage : utilisation de la surcharge pour définir des valeurs par défaut aux arguments.

Conséquences

```
Stack s = new ArrayStack(10);
s.push(3);
// ...
```

- `s` est de type `Stack`
- `s` est de type `ArrayStack`

Quelques remarques complémentaires

1. une classe peut implanter plusieurs interfaces :

```
public class A implements B, C, D
```

   - une instance de `A` est de type `A`, `B`, `C` et `D`
   - `A` doit implémenter toutes les méthodes de `B`, `C` et `D`

2. une interface ne peut pas être instanciée
   - normal : elle ne donne pas d'implantation
   - c'est une entité abstraite
   - en pratique :
     - ~~`Stack s = new Stack()`~~
     - `Stack s = new ArrayStack()`

Aspect magique

- Quand une classe utilise une interface...
- ... elle « fonctionnera » avec toutes les classes implantant cette interface...
- ... même celle qui ne sont pas encore écrite.
- on peut « utiliser » une interface même si aucune classe implantant cette interface existe.

Bilan

- permet de définir que plusieurs objets ont un comportement commun
- définition d'un nouveau type
- interface = impose un certain nombre de fonction, mais aucune contrainte sur l'implantation
- interface ne contient pas de code
- en pratique :
  - créer une interface (`interface`)
  - créer les classes qui vont implanter (`implements`) cette interface

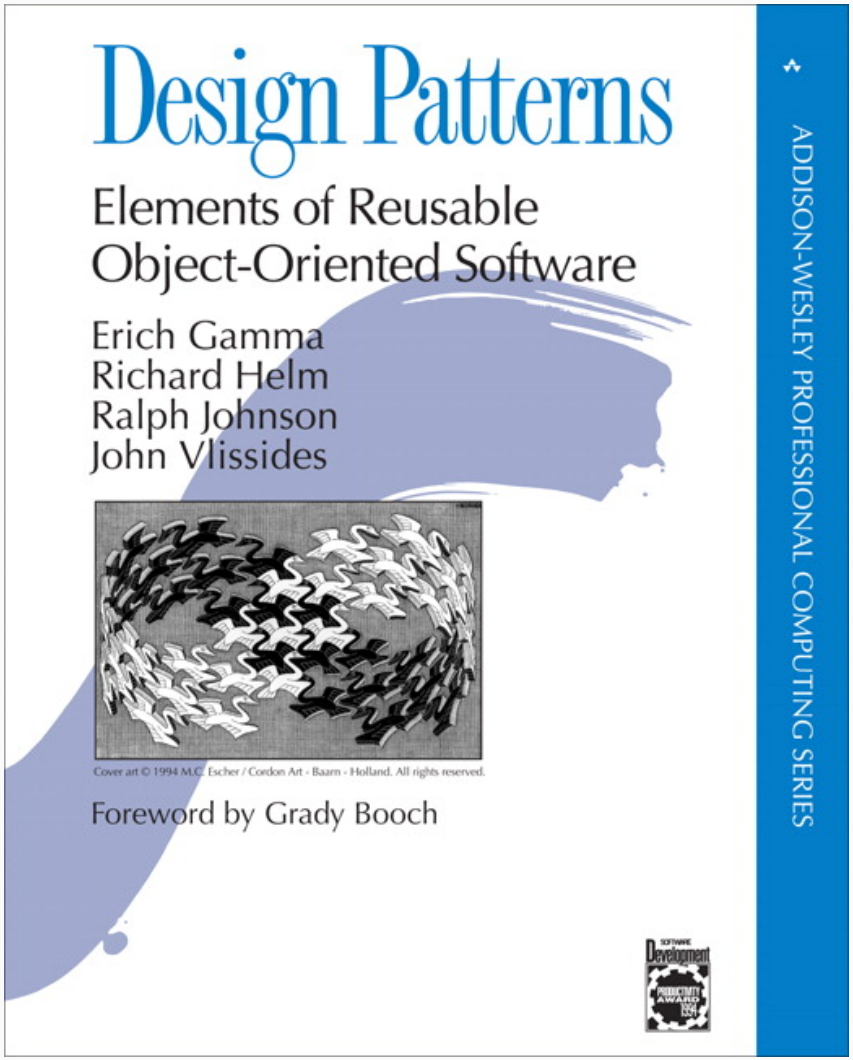
Améliorations possibles

- que faire quand seulement une partie du comportement est commun ?
- peut-on partager du code ?

Troisième partie III

Exemple d'interface : le patron de conception itérateur

Design patterns



- *design patterns* = patron de conception
- résumé d'un problème de conception courant (en POO)
- structures de solutions à ce problème :
  - avantages / inconvénients
  - alternatives
  - ...
- un nom permettant d'identifier un problème et d'en discuter

Motivation des itérateurs

Le problème

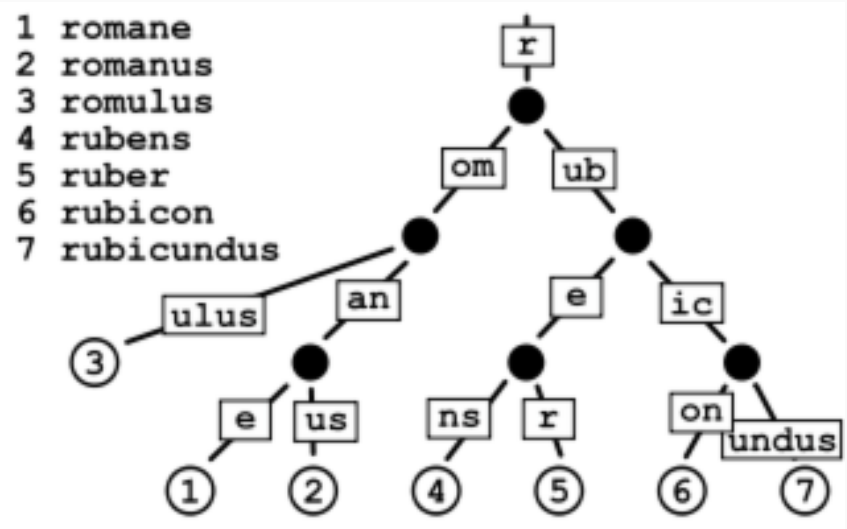
- collection d'objets (tableau)
- objectif parcourir : tous les éléments de la collection

Solution simple

```
double moyenne = 0.0;
for (int i = 0; i < tab.length; i++) {
  moyenne += tab[i];
}
moyenne = moyenne / tab.length
```

Le problème

Et si l'on change de structure de données !



- listes
- table de hachage
- patricia tree/trie
- ...

Plus fondamentalement Abstraction Leak
On ne décrit pas ce que l'on fait, mais comment on le fait

$\Rightarrow$ même problème que l'encapsulation

Les itérateurs (1)

- parcourir tous les éléments contenus dans une collection
- de manière générique : ne dépend pas de la structure interne de la collection
- plus lisible, plus réutilisable, plus robuste

```
double moyenne = 0.0;
for(int value : tab) {
  moyenne += value;
}
moyenne = moyenne / tab.length;
```

- valable pour toutes les collections
- attention : on accède directement à la valeur et pas à l'index

Les itérateurs (2)

Comment assurer la généricité ?

- accéder à l'élément courant
- passer à l'élément suivant
- savoir s'il reste des éléments à parcourir

cf. boucle `for`

Les itérateurs (2)

Comment assurer la généricité ?

- accéder à l'élément courant `next()`
- passer à l'élément suivant `next()`
- savoir s'il reste des éléments à parcourir `hasNext()`

cf. boucle `for`

Iterateur explicite

```
Iterator<Integer> it = tab.iterator();
double moyenne = 0.0;
while (it.hasNext()) {
    Integer value = it.next();
    moyenne += value;
}
```

Itérateur implicite

```
double moyenne = 0.0;
for(int value : tab) {
  moyenne += value;
}
moyenne = moyenne / tab.length;
```

En Java

2 interfaces pour gérer les itérateurs :

- `Iterable<Type>` : indique que l'objet est une collection qui peut être parcourue par un itérateur $\Rightarrow$ création de l'itérateur
- `Iterator<Type>` : indique que l'objet est un itérateur $\Rightarrow$ réalise le parcours à proprement parler.

MyTable.java i

```java
import java.lang.Iterable;
import java.util.Iterator;

public class MyTable implements Iterable<Integer> {

    // protected => accessible aux classes du package
    protected int[] data;

    public MyTable(int[] data) {
        this.data = data;
    }
```

MyTable.java ii

```
public Iterator<Integer> iterator() {
    return new TableIterator(this);
}

public static void main(String[] s) {
    int[] data = {1, 2, 3};
    MyTable t = new MyTable(data);

    for (Integer value : t) {
        System.out.println(value);
    }
}
```

MyTable.java iii

```
}
```

TableIterator.java i

```
import java.util.Iterator;

public class TableIterator implements Iterator<Integer> {

    private int index;
    private MyTable table;

    public TableIterator(MyTable table) {
        this.index = 0;
        this.table = table;
    }
```

TableIterator.java ii

```
public Integer next() {
    this.index++;
    return this.table.data[index - 1];
}

public boolean hasNext() {
    return this.index < this.table.data.length;
}

public void remove() {
    throw new UnsupportedOperationException();
}
```

TableIterator.java iii

```
}
```

Quatrième partie IV

L'héritage

Les classes, une brique de base

- classe = module ré-utilisable
  - définition d'une interface claire, séparée de l'implémentation
  - module « homogène »
  - module interchangeable (après définition d'une interface)
- ignore un aspect des méthodes de développement habituelles : la plupart du code est écrit en reprenant, imitant, spécialisant et combinant du code existant

Relations entre classes

3 types de relations

1. combinaison de classes existantes
2. spécialisation d'une classe existante
3. extension d'une classe existante

2 concepts :

Composition la classe `A` possède des attributs de type `B` et `C`

Héritage la classe `A` étend/hérite de la classe `B`

Objectif de l'héritage

Ré-utilisabilité (reusability)

- défis : variations et répétitions
- factoriser les parties identiques du code tout en permettant des différences
- tous les tableaux de tableaux ont des comportement commun ; certains types ont des besoins particuliers

Extensibilité (extendibility)

- typage statique
- comment permettre au compilateur de « combiner » des objets de différents types lorsque c'est légitime
- exemple : une classe `Etudiant`, une classe `Enseignant`. Comment représenter les utilisateurs de la cantine ?

Exemple : système d'alerte d'une usine

d'après le cours de Franco Gasperoni

- système de surveillance d'une chaîne de production
- envoie des alertes lorsqu'un problème est détecté
- chaque alerte est décrite par
  - l'heure à laquelle elle a été déclenchée
  - sa cause (le ≪ message d'erreur ≫)
- deux services sont offerts :
  - l'archiver *(log)*
  - déclencher un processus de réparation *(handle)*

Classe correspondante

```
import java.util.Date;

public class Alert {

    // représente la date et l'heure
    private Date timeOfArrival;
    private String cause;

    // ...

    public void log() {
        // ...
    }

    public void handle() {
```

Différents types d'alertes

low

- rien de particulier à faire

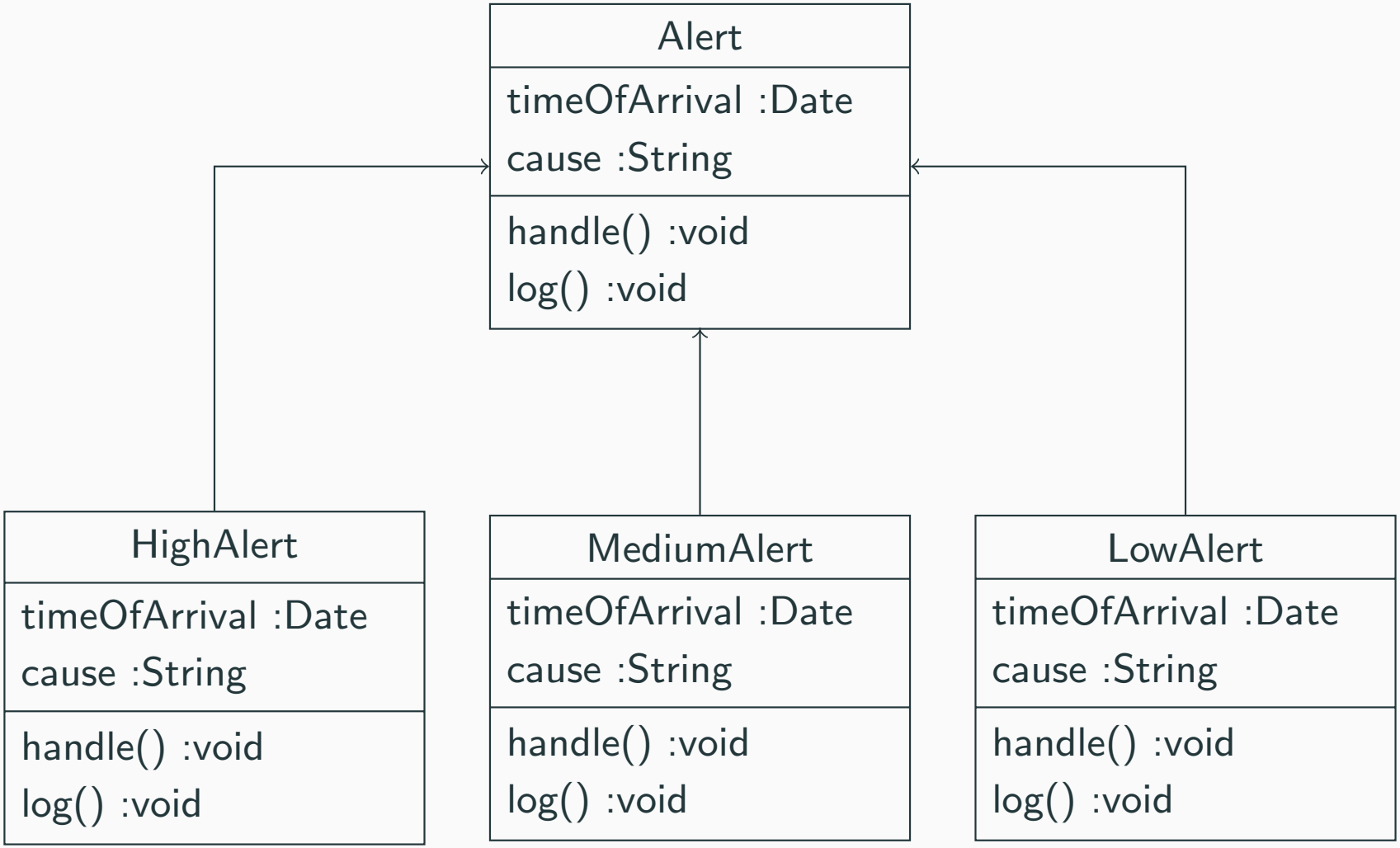
medium

- prévenir un technicien

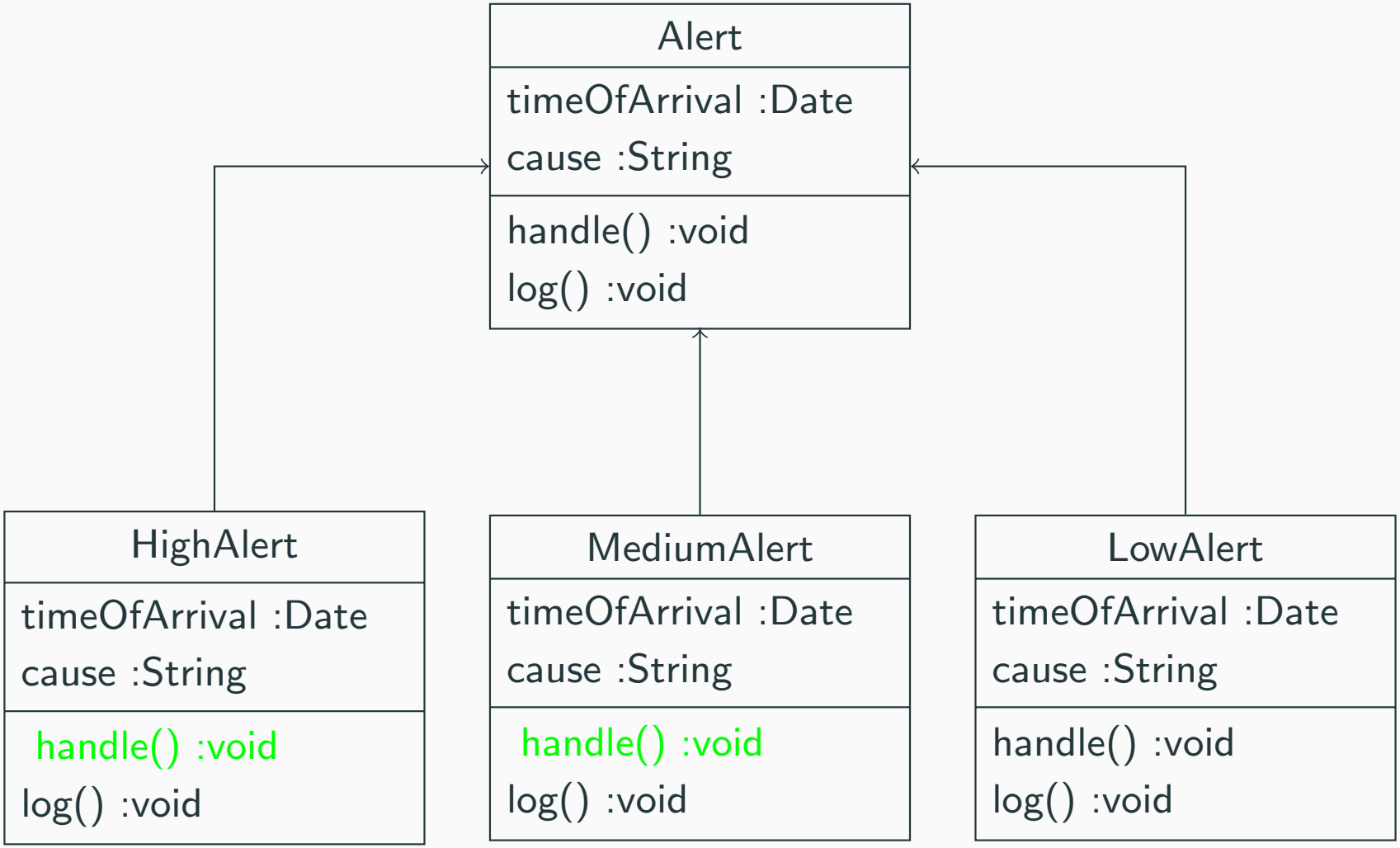
high

- prévenir un ingénieur
- si rien n'est fait avant un certains délais, déclencher une alarme

3 types d'alerte avec un comportement ≪ de base ≫ commun

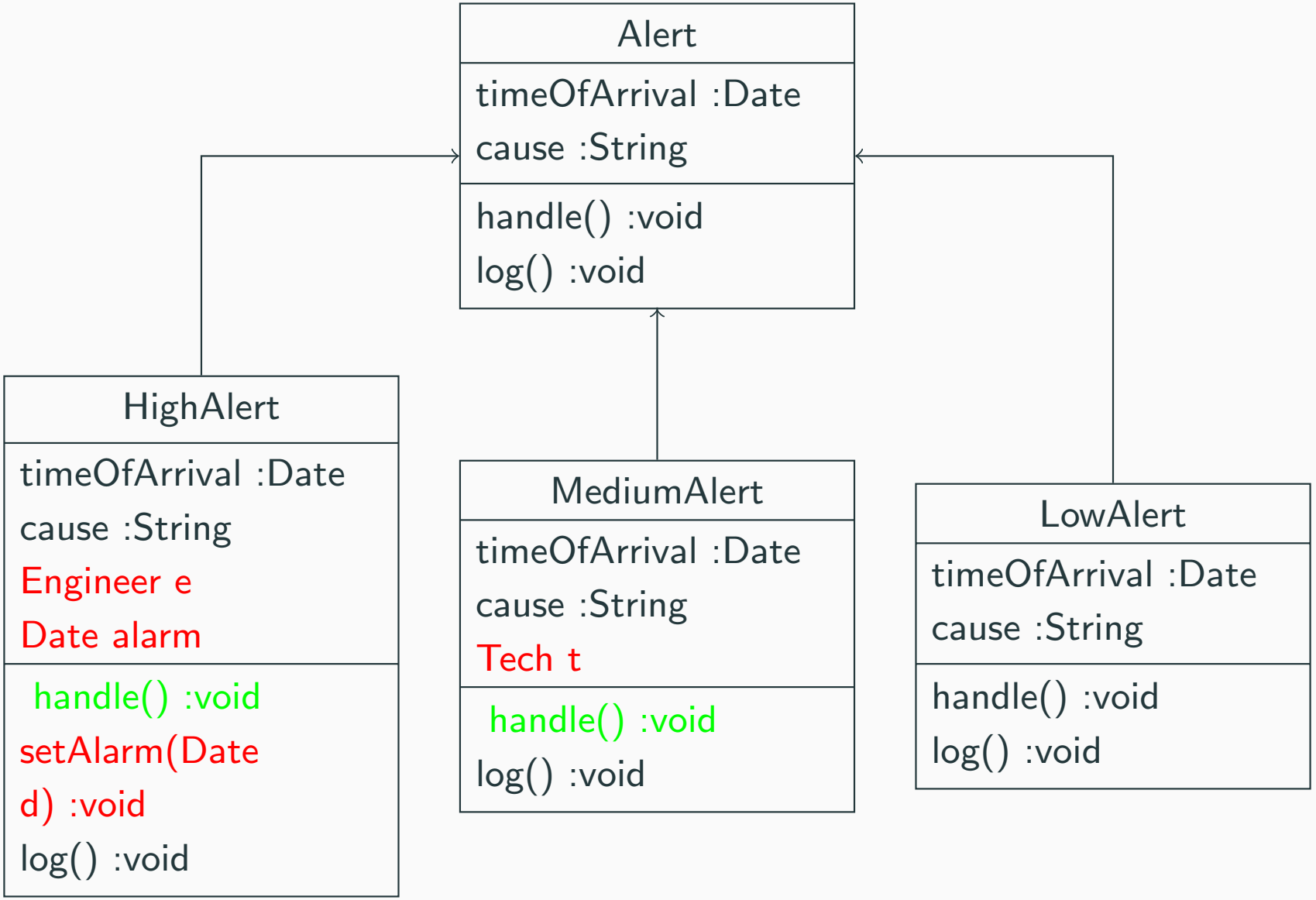


héritage = « copie » le comportement de la classe mère



héritage = « copie » le comportement de la classe mère

redéfinition de méthode



héritage = « copie » le comportement de la classe mère

redéfinition de méthode

ajout d'attributs / méthodes

Définition de l'héritage

```
public class ClasseFille extends ClasseMere
```

- une classe fille hérite d'une classe mère
- la classe fille inclue une partie du code de la classe mère
  - toutes les méthodes publiques (sauf les constructeurs)
  - toutes les méthodes `protected`
- permet de
  - réutiliser du code existant (ni codage, ni debugage)
  - une spécialisation du code (redéfinition d'attributs)
  - une extension du code (redéfinition des méthodes de la classe mère)
- en java : on ne peut hériter que d'une classe (mais on peut implanter plusieurs interfaces)
- héritage transitif : $A \leftarrow B \leftarrow C$

Exemple : HighAlert

```
public class HighAlert extends Alert {

    private Date alarm;
    private Engineer e;

    public void setAlarm(Date d) {
        this.alarm = d;
    }

    public static void main(String[] s) {
        HighAlert a = new HighAlert();
        a.log(); // méthode héritée de Alert
        a.setAlarm(d); // méthode spécifique à HighAlert
    }
```

Redéfinition

Principe

- possibilité de remplacer une méthode de la classe mère par une implémentation spécifique
- spécialisation ou adaptation du comportement d'une classe

En pratique

- méthode de la classe fille avec une signature identique à une méthode de la classe mère
  - même nom
  - même paramètres (type et ordre)
  - même type de retour
- la machine virtuelle appelle systématiquement la méthode la plus spécifique

Surcharge versus redéfinition

Surcharge

- plusieurs méthodes de même nom mais dont les paramètres formels diffèrent en quantité ou en types
- en Java : résolution statique en fonction des types déclarés

Redéfinition

- dans une classe fille, méthode de même nom et de mêmes types de paramètres formels qu'une méthode de la classe mère
- en Java : résolution dynamique, en fonction du type réel de l'appelant

Méthodes et héritages

Hiérarchie d'objets :

- $A \leftarrow B \leftarrow C$
- `A` implante `a`, `B`, `b` et `C` `c` et `b`
- `test` est une instance de `C`

Quelle méthode est exécutée ?

- `test.c()`
- `test.a()`
- `test.b()`

Méthodes et héritages

Hiérarchie d'objets :

- $A \leftarrow B \leftarrow C$
- `A` implante `a`, `B`, `b` et `C` `c` et `b`
- `test` est une instance de `C`

Quelle méthode est exécutée ?

- `test.c()` extension : on ajoute une méthode spécifique à la classe fille
- `test.a()` utilisation
- `test.b()` redéfinition : on redéfinit une méthode pour adapter à la classe fille

Le mot-clé `super`

`super`

- `super.methode()` et `super.attribut` permettent d'accéder aux méthodes/attributs `public` ou `protected` de la classe mère même en cas de redéfinition
- `super(...)` permet d'accéder aux constructeurs de la classe mère

Héritage et constructeur

- s'il n'y pas de constructeur dans la classe fille : appelle automatiquement le constructeur de la classe mère
- sinon, le constructeur de la classe fille doit appeler explicitement le constructeur de la classe mère $\Rightarrow$ la première ligne du constructeur doit être un `super(...)`

Utilisation des annotations

- introduites dans Java 1.5
- métadonnées associées à des éléments Java (classe, fonction, ...)
- les plus courantes : `@SupressWarnings`, `@Overrides`, `@Deprecated`
- `@Overrides` pour indiquer que l'on veut surcharger une méthode
  - documentation
  - on est sûr de surcharger une méthode

Polymorphisme

Types d'une instance

- une instance de `HighAlert` a deux types : `Alert` et `HighAlert`
- une instance de classe a le type de la classe et de toutes les classes dont celle-ci hérite

Conséquence : polymorphisme

- polymorphisme = qui peut prendre plusieurs formes
- en POO : référence qui peut-être attachée à des objets de différents types.
- exemple : une référence de type `Alert` peut décrire les trois types d'alertes

Attention

- lors d'un attachement polymorphique, il n'y a pas modification de la mémoire du programme
- seul le type de la référence est modifiée

Structures de données polymorphiques

```
List<Alert> alerts = new ArrayList<Alert>();
for (Alert a : alerts) {
  a.log();
}
```

- `alerts` peut contenir les 3 types d'alertes
- dynamic binding la méthode appelée ne peut être déterminée qu'au moment de l'exécution

Bilan

- héritage = permet de partager du code
- objectifs :
  - faciliter la ré-utilisabilité
  - code plus sûr
  - meilleure architecture logicielle