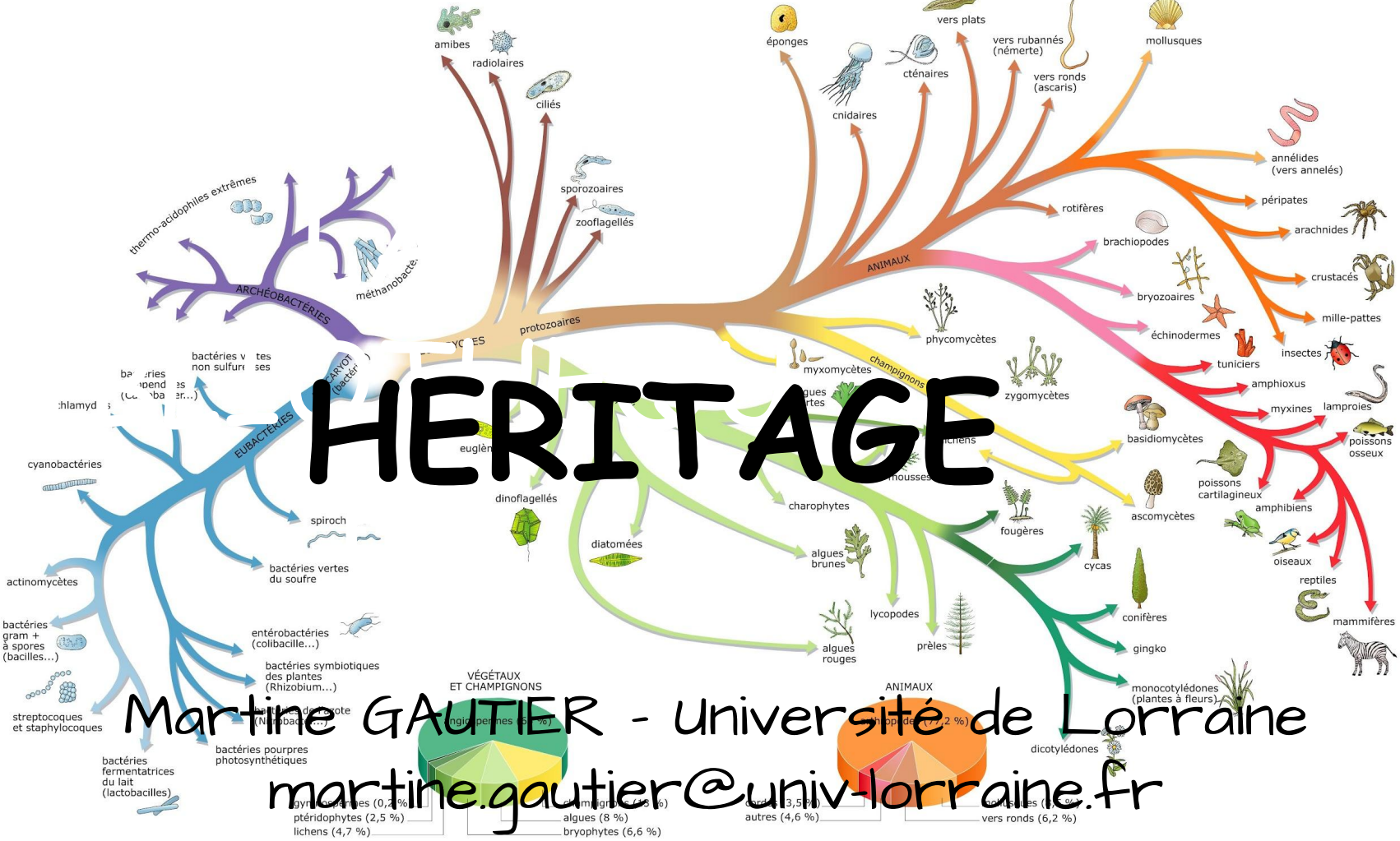


HERITAGE

Martine GAUTIER - Université de Lorraine
martine.gautier@univ-lorraine.fr

Ce qu'on a appris

Ecrire/tester une classe

Composer des classes en utilisant la relation de **clientèle**

→ `Polygone` et `Triangle` sont clientes de `Point`

→ `Ocean` est cliente de `Pingouin`

Ce qu'on ne sait pas faire

Concevoir une application

→ imaginer les classes utiles

→ imaginer les fonctions utiles

→ placer les fonctions dans les bonnes classes

Génie logiciel (Software Engineering)

- Notion due à Margaret Hamilton (1968), conceptrice du système embarqué du programme Apollo 2
- Qualité des logiciels (norme IOS 9216) : 6 groupes d'indicateurs

  → capacité fonctionnelle

  → facilité d'utilisation

  → fiabilité

  → performance

  → maintenabilité

  → portabilité

B.A. BA de la qualité

- Chaque fonction est commentée (balises `javadoc`).
- Les données sont protégées : **encapsulation**

  → les champs sont privés

  → des fonctions `getter/setter` permettent de les consulter/modifier
- Les fonctions d'une classe concernent les instances de la classe.
- Aucun code n'est dupliqué.
- Chaque classe, chaque fonction est testée.

Des idées nouvelles

- Dans l'application arctique

  → on ajoute d'autres éléments de jeux : bateaux, ours, phoques, *etc.*

  → adapter le code existant

    * ajouter de nouvelles collections d'éléments de jeu

    * modifier le corps des fonctions existantes pour en tenir compte

- Dans le package `geometrie`

  → on ajoute d'autres figures : `Cercle`, `Rectangle`, `Carre`, `Losange`, *etc*

  → beaucoup de fonctions communes

L'héritage, la solution magique

- L'héritage (et les autres mécanismes qui s'y rapportent) est apparu avec les langages objets.
- Améliorer l'extensibilité
- Traiter de façon analogue des objets différents ayant un comportement voisin
- Factoriser du code pour éviter sa duplication
- Réutiliser du code existant

→ Largement répandu dans les Design Patterns

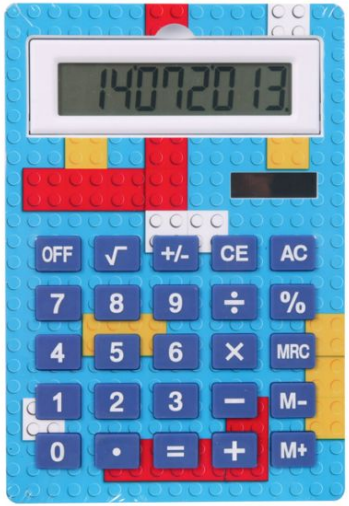
Un exemple pour apprendre le mécanisme

- Application Calculette

  → Fonctionnalité principale : calculer une expression arithmétique

  → Fonctionnalité secondaire : afficher l'expression

20/2 + 37 - (5 + 32)



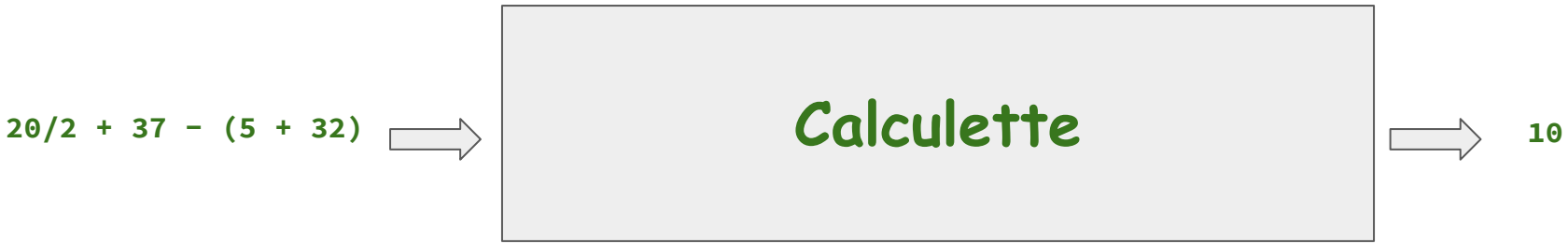
10

((20 / 2) + 37 - (5 + 32))

Deux étapes

- Analyse du texte pour construire une représentation intermédiaire

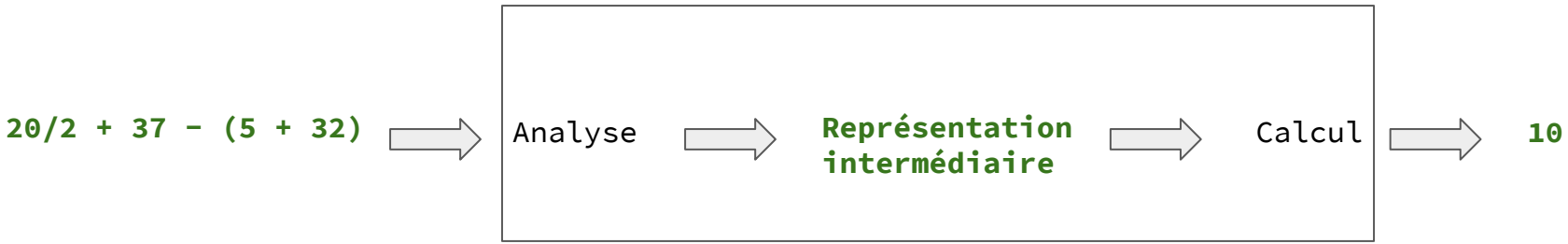
- Calcul / conversion en String



Deux étapes .....

- Analyse du texte pour construire une représentation intermédiaire

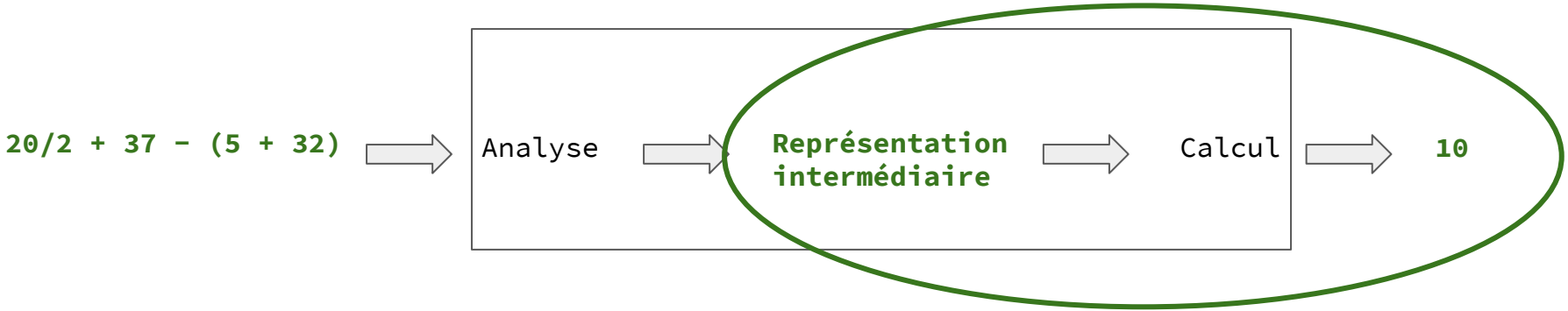
- Calcul / conversion en String



Deux étapes .....

- Analyse du texte pour construire une représentation intermédiaire

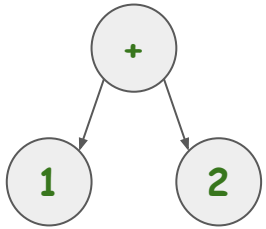
- Calcul / conversion en String



Représentation intermédiaire

Représentation arborescente pour tenir compte de la priorité des opérateurs

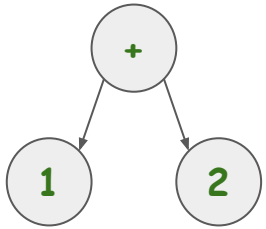
1 + 2

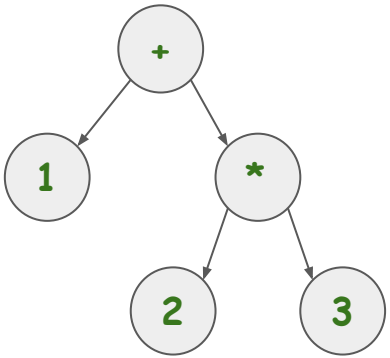


Représentation intermédiaire ….

Représentation arborescente pour tenir compte de la priorité des opérateurs

1 + 2



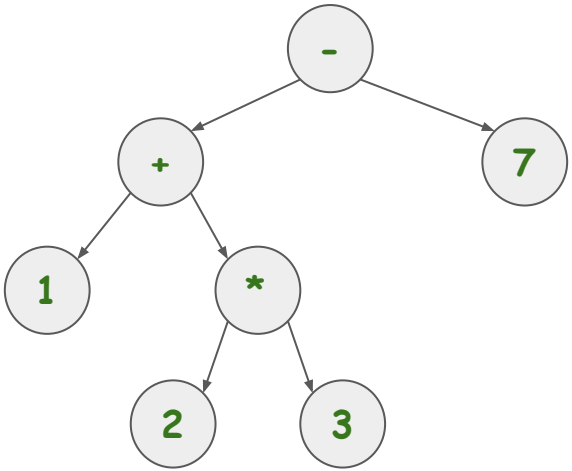


1 + 2 * 3

Représentation intermédiaire …..

Représentation arborescente pour tenir compte de la priorité des opérateurs et de l'ordre d'évaluation gauche-droite

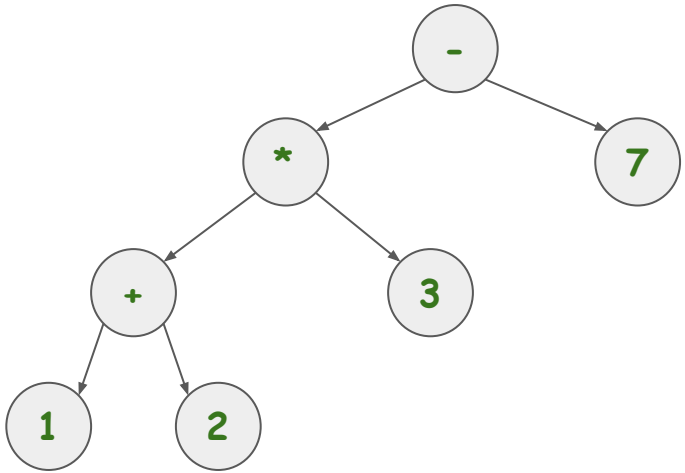
```
1 + 2 * 3 - 7
```



Représentation intermédiaire ……

Représentation arborescente pour tenir compte de la priorité des opérateurs et de l'ordre d'évaluation gauche-droite ; les parenthèses disparaissent.

(1 + 2) * 3 - 7



Représentation intermédiaire .....

Comment représenter cette structure en C ?

```
struct noeud {
   char op ;
   int val ;
   struct noeud *gauche ;
   struct noeud *droit ;
} ;
typedef struct noeud Noeud ;
```

Représentation intermédiaire ……

Comment représenter

Codification de l'opérateur +, - , …
ou vide pour une constante

```
struct noeud {
   char op ;
   int val ;
   struct noeud *gauche ;
   struct noeud *droit ;
} ;
typedef struct noeud Noeud ;
```

Représentation intermédiaire .....

Comment représenter cette stru

```
struct noeud {
   char op ;
   int val ;
   struct noeud *gauche ;
   struct noeud *droit ;
} ;
typedef struct noeud Noeud ;
```

Uniquement pour un opérande

Représentation intermédiaire .....

Comment représenter cette structure en C ?

```
struct noeud {
   char op ;
   int val ;
   struct noeud *gauche ;
   struct noeud *droit ;
} ;
typedef struct noeud Noeud ;
```

Indéfinis pour un opérande

Représentation intermédiaire .....

Comment calculer la valeur ?

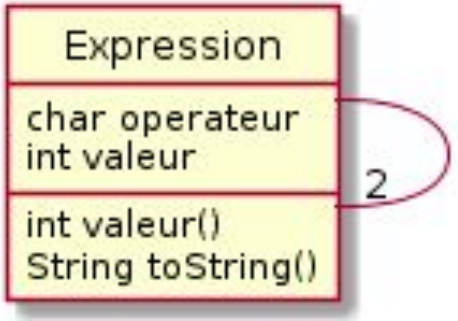
```
struct noeud {
   char op ;
   int val ;
   struct noeud *gauche ;
   struct noeud * droit ;
} ;
typedef struct noeud Noeud ;
```

```
int evaluer(Noeud noeud) {
  int resultat ;
  switch (noeud.op) {
    case ' ' : resultat = val ; break ;
    case '+' : resultat = evaluer(*noeud.gauche)
                        + evaluer(*noeud.droit) ;
               break ;
    case '-' : resultat = evaluer(*noeud.gauche)
                        - evaluer(*noeud.droit) ;
               break ;
          ...
  }
  return resultat ;
}
```

Représentation intermédiaire .....

Faire la même chose en Java ?

```
package calc ;
public class Expression {
   private char operateur ;
   private int valeur ;
   private Expression gauche ;
   private Expression droit ;
   public int valeur() ;
   public String toString() ;
  ...
}
```

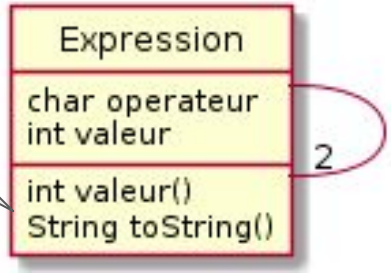


Représentation intermédiaire .....

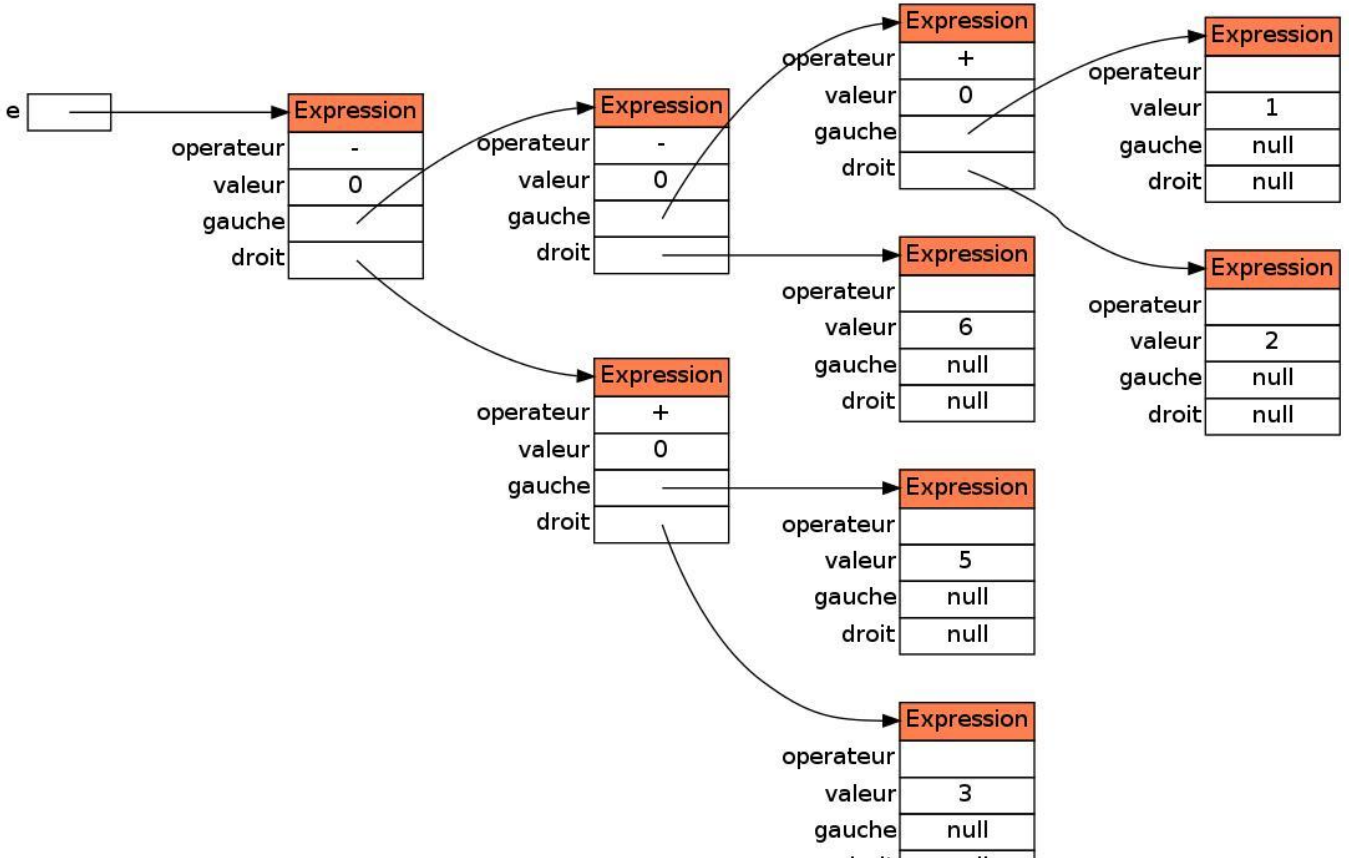
Faire !

Ces fonctions n'ont pas de paramètre ; elles travaillent sur le receveur.

```
package calc ;
public class Expression {
   private char operateur ;
   private int valeur ;
   private Expression gauche ;
   private Expression droit ;
   public int valeur() ;
   public String toString() ;
  ...
}
```



Représentation intermédiaire .....



Représentation intermédiaire .....

Comment calculer la valeur ?

```
int valeur() {
  int resultat ;
  switch (operateur) {
    case ' ' : resultat = valeur ; break ;
    case '+' : resultat = gauche.valeur() + droit.valeur() ; break ;
    case '-' : resultat = gauche.valeur() - droit.valeur() ; break ;
         ...
  }
  return resultat ;
}
```

Bilan

- Les fonctionnalités attendues sont réalisées.
- Mais

→ la représentation mémoire est inadaptée

→ chaque fonction doit tester le type de l'opérateur

→ l'ajout de nouveaux opérateurs oblige à compléter le `switch` dans chaque fonction

→ le temps d'exécution augmente à chaque ajout

Bilan

- Les fonctionnalités attendues sont réalisées.
- Mais

→ la rep

→ cl teur

→ l'a éter
le sw

→ le temps e à chaque ajout

On oublie cette mauvaise solution.

Améliorer l'évolutivité

- POO : Définir des classes pour représenter des entités qui ont des comportements distincts.

- Définir une classe par opérateur

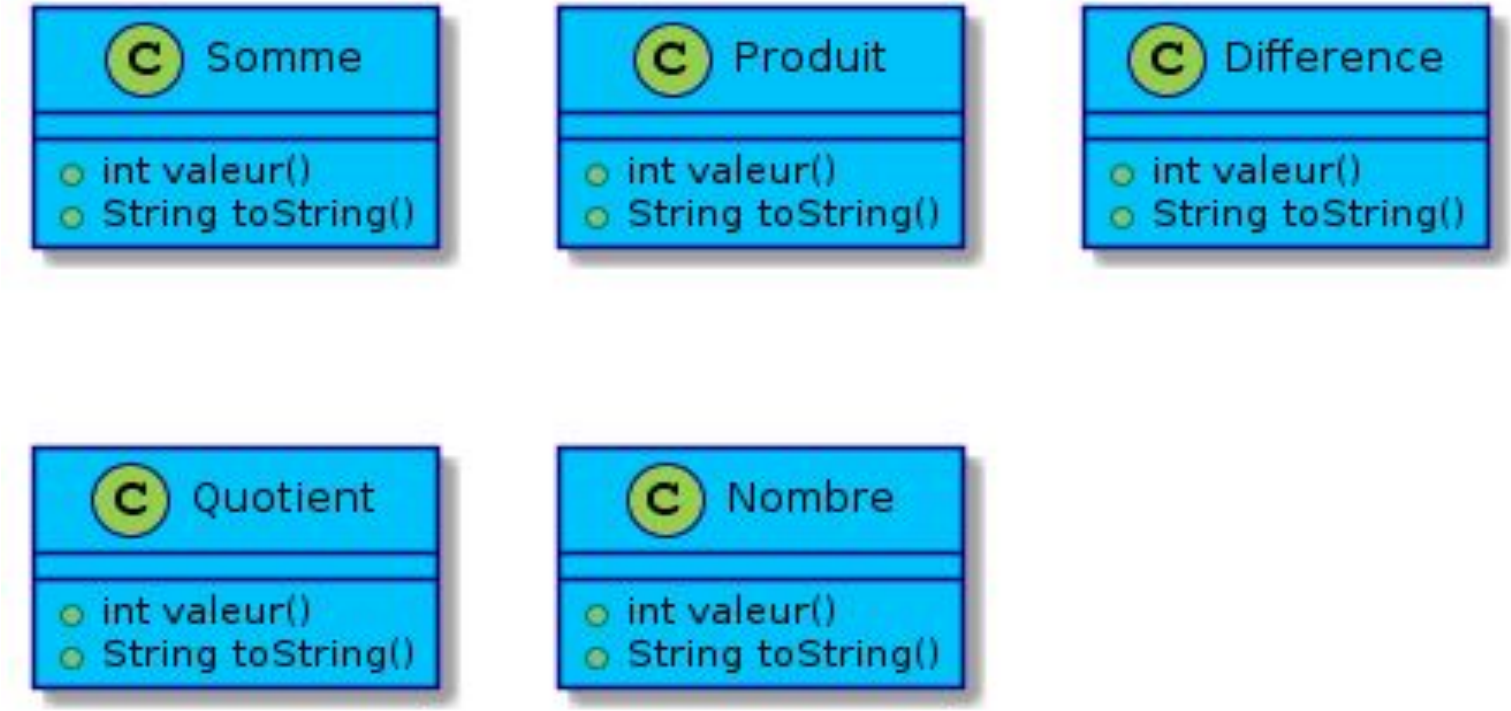
  → `Somme, Produit, Difference, Quotient`
  → les fonctions `valeur` et `toString` sont définies dans chaque classe

- Définir une classe pour les nombres

  → `Nombre`
  → avec les fonctions `valeur` et `toString`

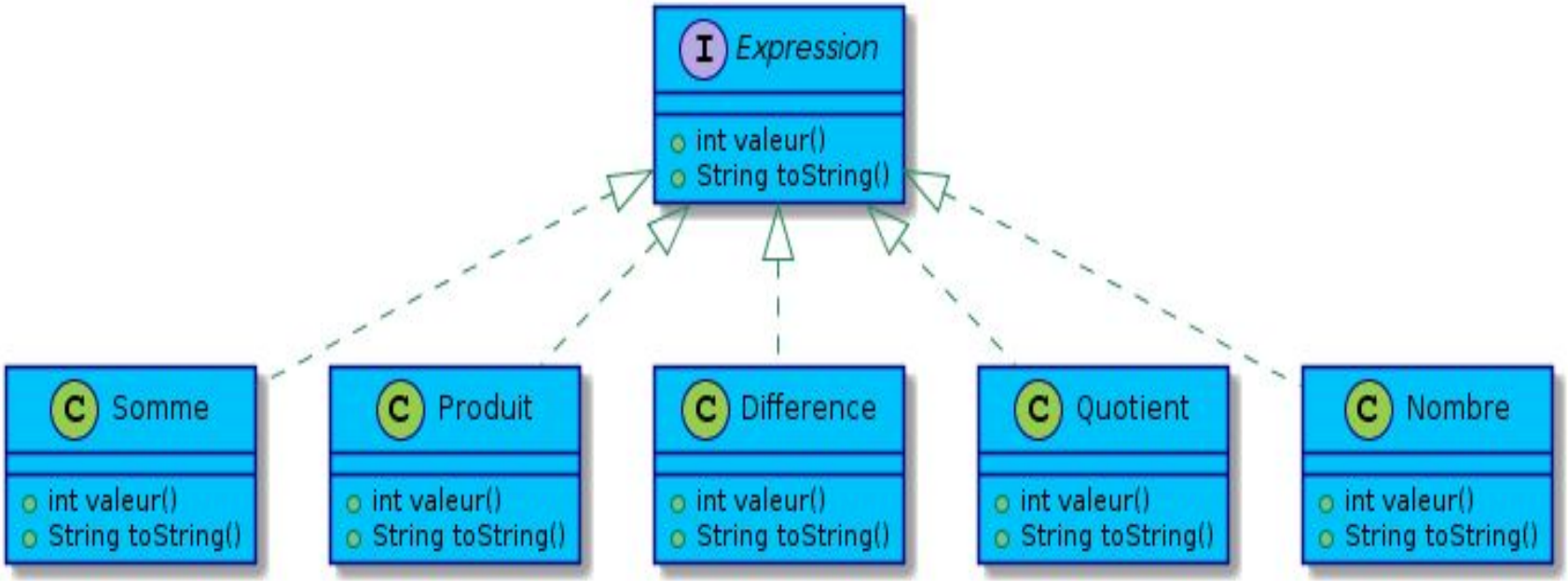
- Ajouter un opérateur = ajouter une classe

Un premier diagramme de classes UML



Somme, Difference, … sont des expressions

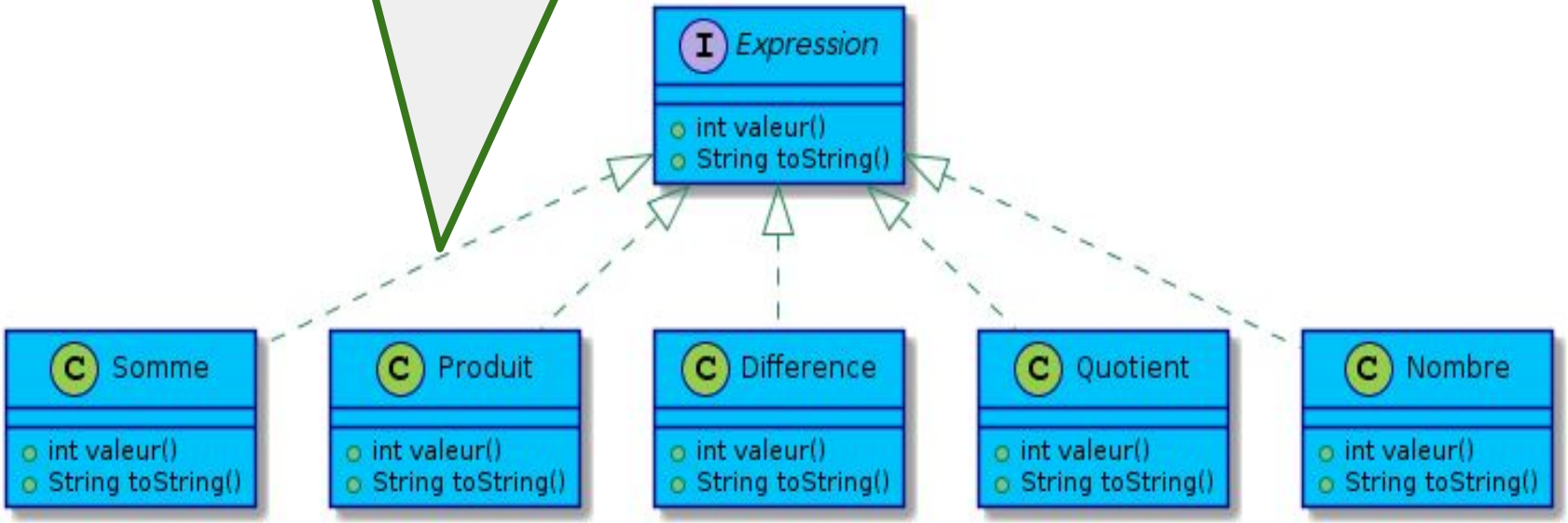
- L'interface `Expression` regroupe tous les cas possibles d'expressions.
  → classification (ex : mammifère)



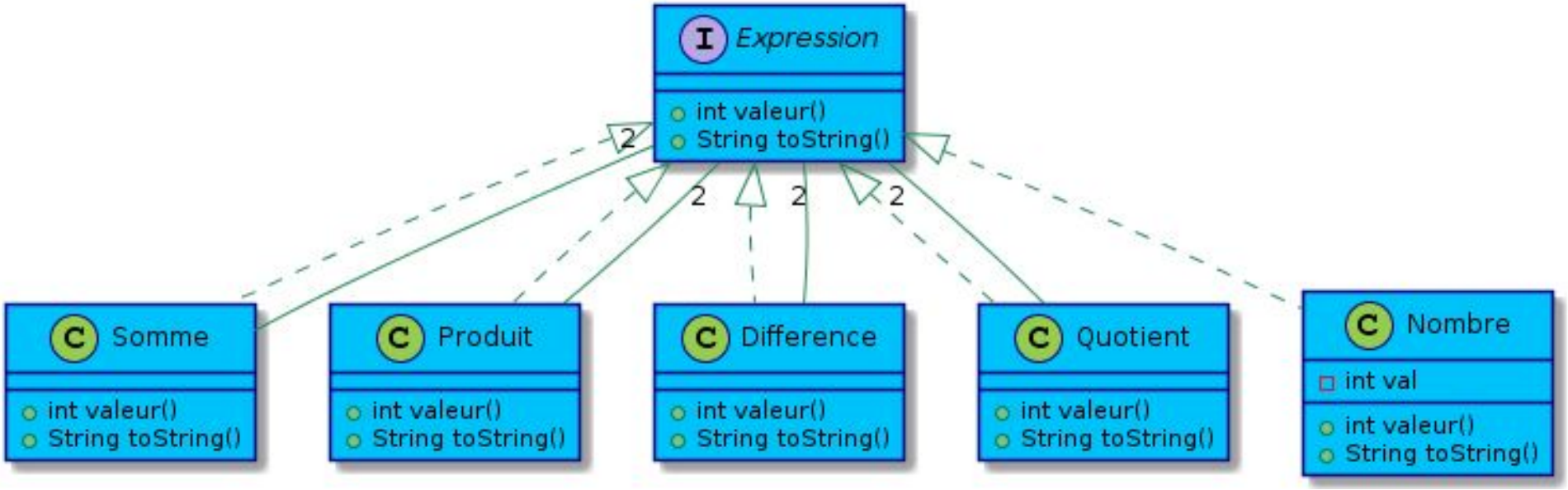
Somme, Difference, … sont des expressions

La classe Somme **implante** l'interface Expression.

ssibles d'expressions.



Les opérateurs sont binaires



Le texte de l'interface Expression

```
package calc ;
public interface Expression {
    public int valeur() ;
    public String toString();
}
```

Dans le fichier calc/Expression.java

Que des profils de fonctions publiques

Le texte de Nombre

```
package calc ;
public class Nombre implements Expression {

    private int val ;

    public Nombre (int v) {
        this.val = v ;
    }

    public int valeur() {
        return this.val ;
    }

    public String toString() {
        return “”+this.val ;
    }
}
```

La classe implémente l'interface Expression.

Un seul champ, pour la valeur

La définition des fonctions valeur et toString est obligatoire.

Les fonctions valeur et toString ne gèrent que le cas d'un nombre.

Le texte de Somme

```
package calc ;
public class Somme implements Expression {

   private Expression gauche ;
   private Expression droit ;

   public Somme (Expression g, Expression d) {
      this.gauche = g ;
     this.droit = d ;
   }
   public int valeur() {
      return gauche.valeur() + droit.valeur() ;
   }
   public String toString() {
      return “(”+gauche+”+”+droit+”)” ;
   }
}
```

La classe implémente l'interface Expression.

Deux champs pour les opérandes

La définition des fonctions valeur et toString est obligatoire.

Les fonctions valeur et toString ne gèrent que le cas de la somme.

Une classe de test ?

```
package calc.tests ;
public class Test   {

   public static void main(String[] a) {

      Nombre n1 = new Nombre(10) ;
      Nombre n2 = new Nombre(88) ;
      Somme s1 = new Somme(n1, n2) ;

      Nombre n3 = new Nombre(543) ;
      Somme s2 = new Somme(s1, n3) ;
  }
}
```

On instancie les classes Nombre ou Somme, mais jamais Expression.

Le constructeur de Somme attend deux paramètres de type Expression ; Nombre est un cas particulier de Expression ; donc l'appel est correct.

Le texte de Difference

```
package calc ;
public class Difference implements Expression {

   private Expression gauche ;
   private Expression droit ;

   public Difference (Expression g, Expression d) {
      this.gauche = g ;
      this.droit = d ;
   }
   public int valeur() {
      return gauche.valeur() - droit.valeur() ;
   }
   public String toString() {
      return "("+gauche+"-"+droit+")" ;
   }
}
```

Copier-coller-adapter

Factoriser le code au lieu de le dupliquer

- Duplication de code

  → pas facile à maintenir

  → à bannir

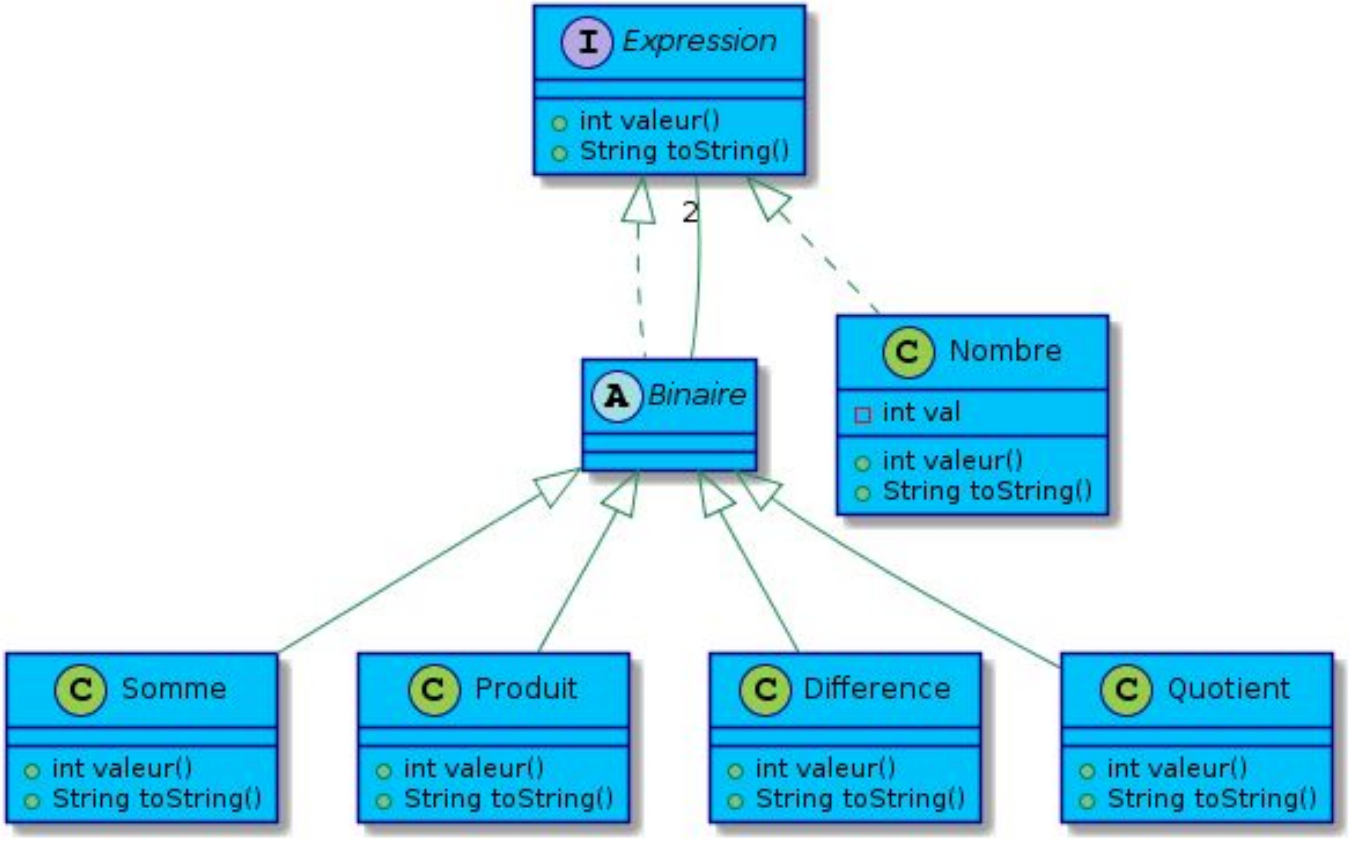
- Où factoriser ?

  → Impossible dans l'interface `Expression` qui ne peut contenir que des profils de fonctions (et pas de corps)

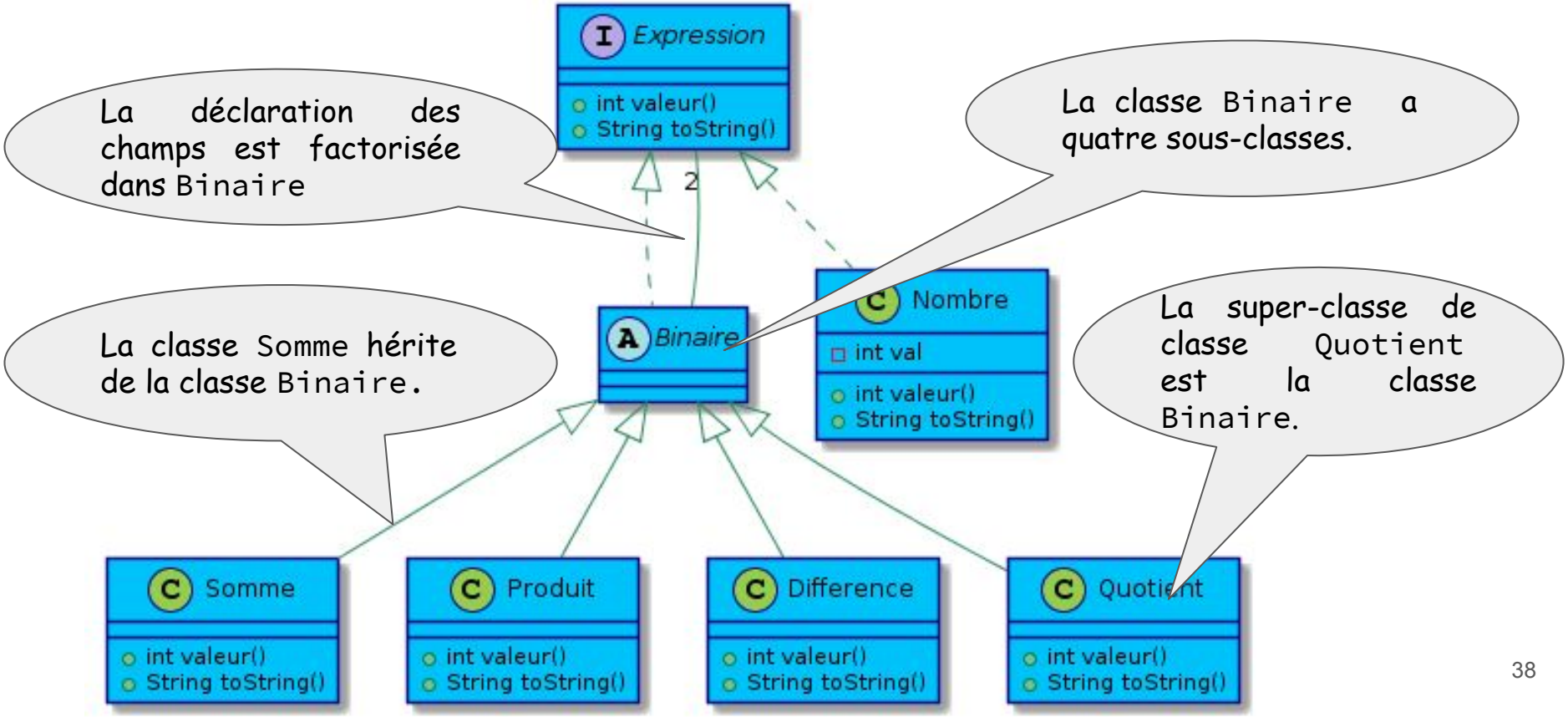
- Définir une nouvelle classe `Binaire` pour factoriser le code commun à `Somme`, `Produit`, `Différence` et `Quotient`

  → `Nombre` n'est pas concernée

Le diagramme de classes UML évolue



Le diagramme de classes UML évolue



Le texte de Binaire

```
package calc ;
public abstract class Binaire
                          implements Expression {

   protected Expression gauche ;
   protected Expression droit ;

   public Binaire (Expression g, Expression d) {
      this.gauche = g ;
     this.droit = d ;
   }

}
```

La classe implémente l'interface Expression.

La classe est abstraite car les deux fonctions valeur et toString ne sont pas définies.

Une classe abstraite ne peut pas être instanciée.

Les champs sont déclarés protected, c'est-à-dire accessibles dans les sous-classes comme Somme.

Le texte de Somme

```
package calc ;
public class Somme extends Binaire {

   public Somme (Expression g, Expression d) {
      super(g, d) ;
   }

   public int valeur() {
      return gauche.valeur() + droit.valeur() ;
   }

   public String toString() {
      return “(”+gauche+”+”+droit+”)” ;
   }
}
```

La classe hérite de la classe Binaire

Plus besoin de champs

L'appel à super dans le constructeur permet d'exécuter les instructions du constructeur de la super-classe.

La définition des fonctions valeur et toString est encore obligatoire.

Le texte de Somme

```
package calc ;
public class Somme extends Binaire {

   public Somme (Expression g, Expression d) {
      super(g, d) ;
   }

   public int valeur() {
      return gauche.valeur() + droit.valeur() ;
   }

   public String toString() {
      return “(”+gauche+”+”+droit+”)” ;
   }
}
```

La classe hérite de la classe Binaire

Plus besoin de champs

L'appel à super dans le constructeur permet d'exécuter les instructions du constructeur de la super-classe.

La définition des fonctions valeur et toString est encore obligatoire.

Cela ne change rien à l'exécution du test. Mais le code est mieux structuré.

Et le texte de Difference ?

```
package calc ;
public class Difference extends Binaire {


   public Difference (Expression g, Expression d) {
      super(g, d) ;
   }

   public int valeur() {
      return gauche.valeur() - droit.valeur() ;
   }

   public String toString() {
      return “(”+gauche+”–”+droit+”)” ;
   }
}
```

La définition des fonctions valeur et toString est encore partiellement dupliquée.

Et le texte de `Difference` ?

```
package calc ;
public class Difference extends Binaire {


   public Difference (Expression g, Expression d) {
      super(g, d) ;
   }

   public int valeur() {
      return gauche.valeur() - droit.valeur() ;
   }

   public String toString() {
      return “(”+gauche+”-”+droit+”)” ;
   }
```

Chercher à factoriser au maximum dans `Binaire`

La définition des fonctions valeur et toString est encore partiellement dupliquée.

Le texte de Binaire, factorisé au maximum

```
package calc ;
public abstract class Binaire
                          implements Expression {

 protected Expression gauche ;
 protected Expression droit ;

 public Binaire (Expression g, Expression d) {
   this.gauche = g ;
   this.droit = d ;
 }

 public String toString () {
  return “(”+gauche+getOperateur()+droit+”)” ;
 }

 public abstract String getOperateur() ;
}
```

La fonction toString est définie en utilisant une fonction abstraite getOperateur.

La fonction getOperateur est abstraite, c'est-à-dire qu'elle est définie dans les sous-classes.

Le texte de Somme est plus court

```
package calc ;
public class Somme extends Binaire {

   public Somme (Expression g, Expression d) {
      super(g, d) ;
   }

   public int valeur() {
      return gauche.valeur() + droit.valeur() ;
   }

   public String getOperateur() {
      return “+” ;
   }
}
```

Difficile de factoriser valeur

Le texte de `Difference` est similaire

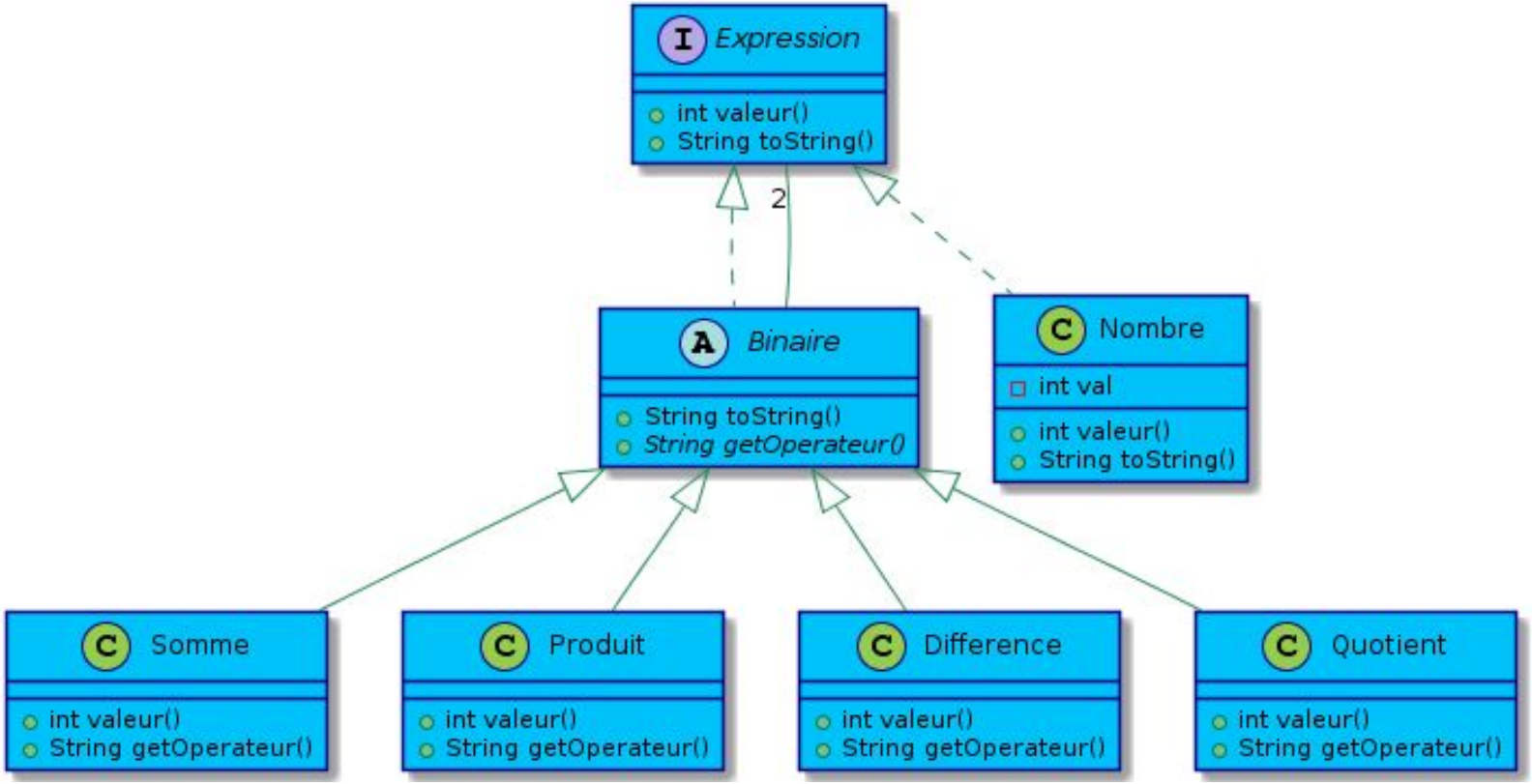
```
package calc ;
public class Difference extends Binaire {

   public Difference (Expression g, Expression d) {
      super(g, d) ;
   }

   public int valeur() {
      return gauche.valeur() - droit.valeur() ;
   }

   public String getOperateur() {
      return “-” ;
   }
}
```

Le diagramme de classes UML définitif



Que se passe-t-il lors de l'exécution ?

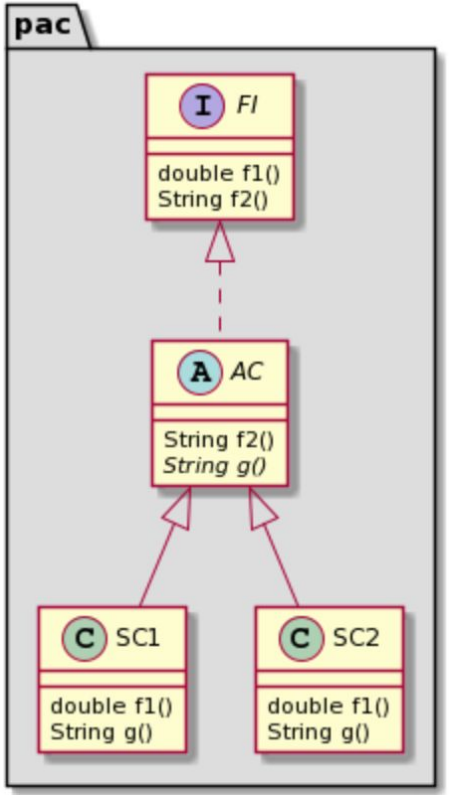
- Schéma mémoire avec artEoz

- Appel de la fonction `valeur`

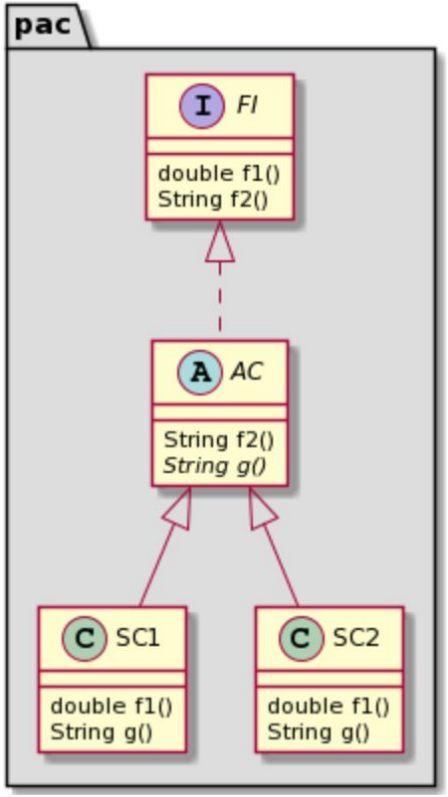
  → laquelle est exécutée ?

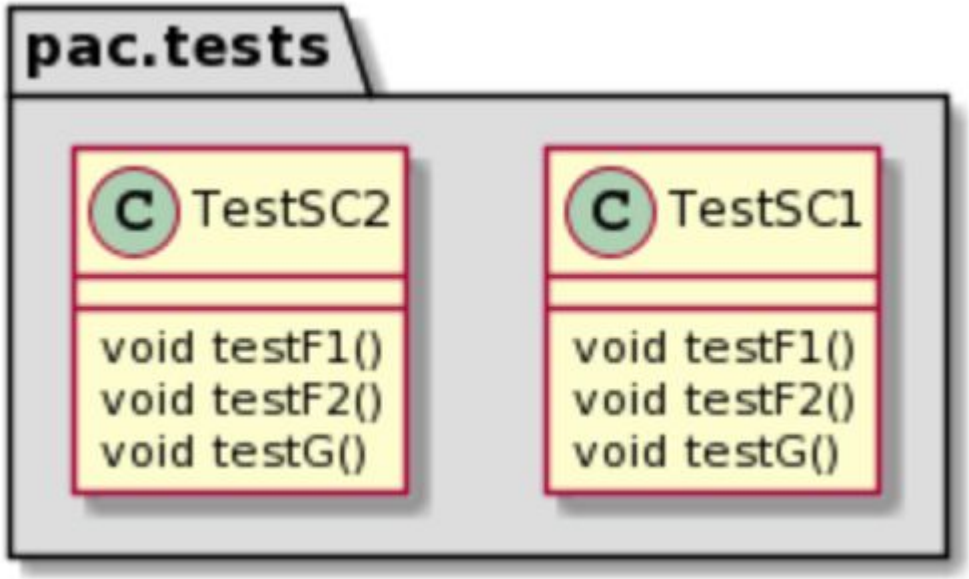
  → la machine virtuelle exécute la fonction `valeur` correspondant au type dynamique du receveur.

Tests de hiérarchies de classes

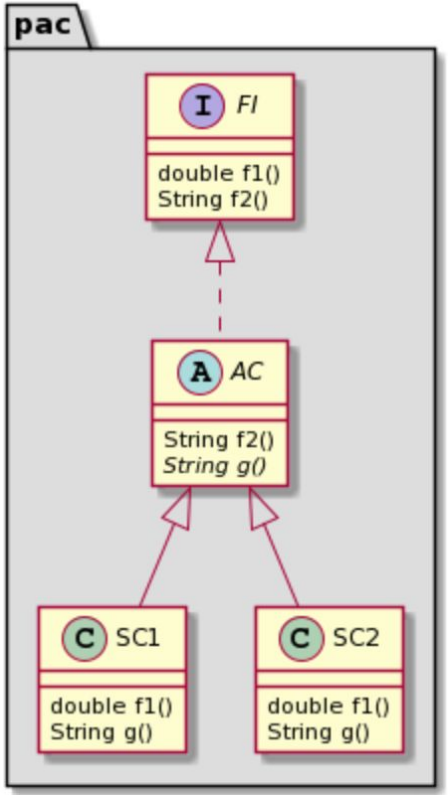


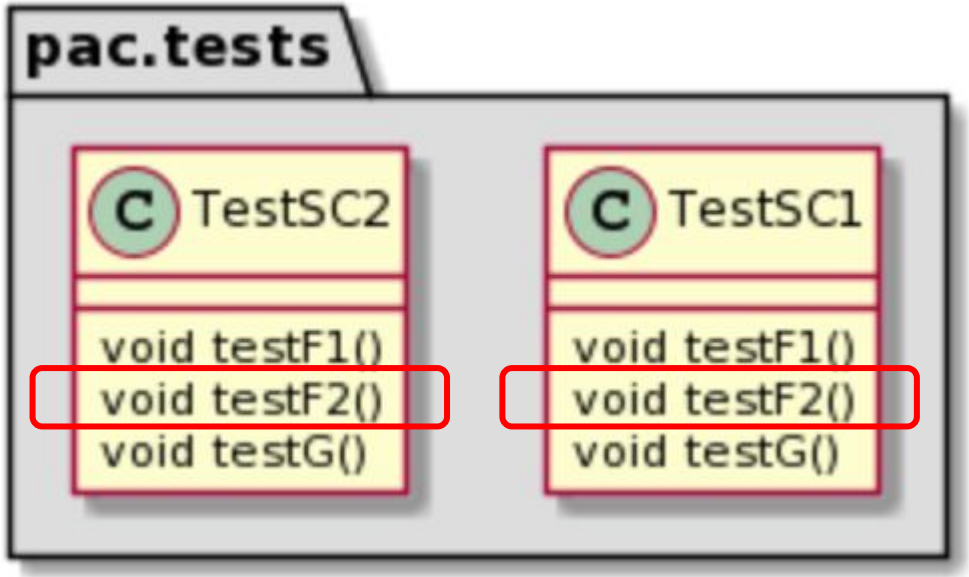
Tests de hiérarchies de classes



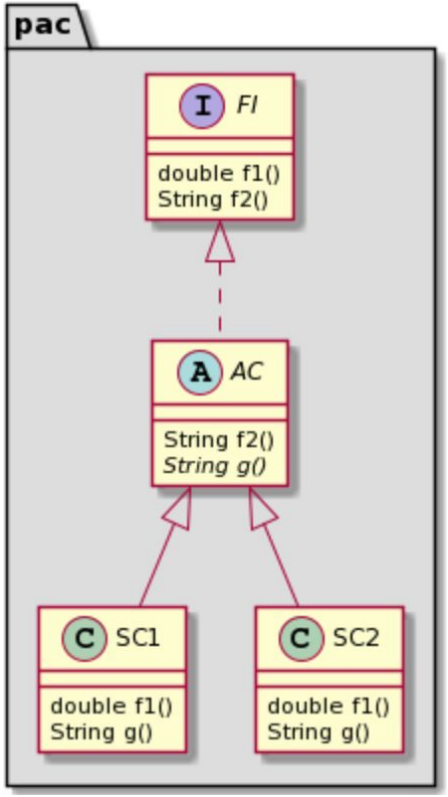


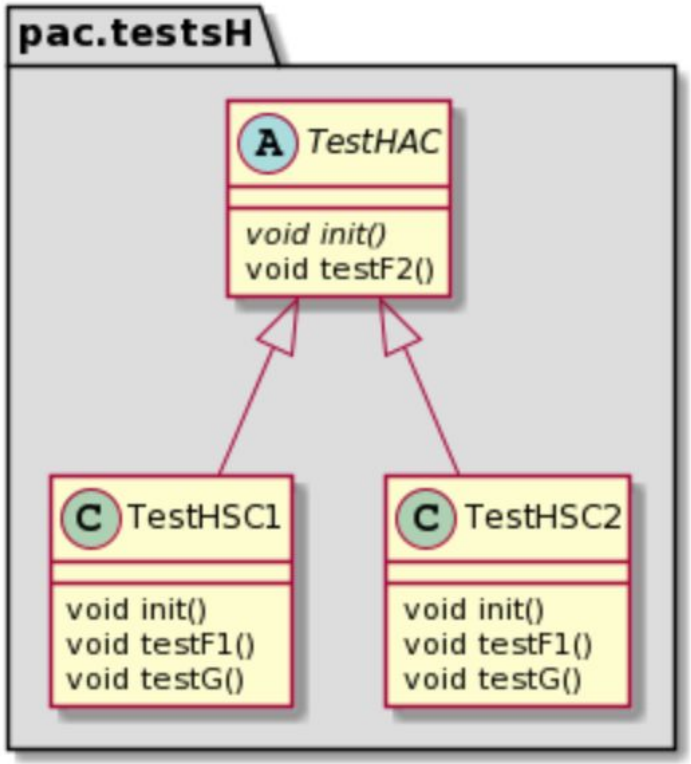
Tests de hiérarchies de classes





Tests de hiérarchies de classes





Bilan

- Les fonctionnalités attendues sont (encore) réalisées.
- La représentation mémoire est adaptée à chaque type d'expression.
- Chaque classe définit les fonctions `valeur` et `toString` selon la nature de l'expression.
- Le code est factorisé autant que possible.
- L'ajout d'un opérateur implique la création d'une nouvelle classe, sans influence sur les autres.
- Le temps d'exécution reste stable en cas d'ajout d'opérateur.