Réutilisation de classes:

Composition et héritage

8

Dans tout langage de programmation, le programmeur essaie de réutiliser du code existant qui a déjà été testé et débogué en le recopiant dans ses propres programmes. Avec les langages à objets, il est possible de réutiliser du code en créant de nouvelles classes. Mais plutôt que de repartir de zéro et créer des classes en recopiant des morceaux de code, il existe des mécanismes pour réutiliser des classes déjà existantes sans toucher au code de celles-ci.

Il y a deux manières de procéder: (i) la composition et (ii) l'héritage.

1. Composition: réutilisation d'objets

La composition est la pratique la plus simple; elle consiste à définir une classe dans laquelle vous créez des objets à l'intérieur. La classe est *composée* d'objets de classes existantes.

1.1. Exemple

Une séquence au format FASTA est définie par un titre et des données codées par un symbole à une lettre. La classe FASTA_SEQUENCE peut s'écrire:

```
class FASTA_SEQUENCE
feature
    title : STRING
    data : SEQUENCE
end -- end of class FASTA_SEQUENCE
```

En utilisant la composition, on utilise les fonctionnalités d'objets définis par des classes existantes.

2. Héritage: réutilisation de classes

Comme la composition décrite dans le chapitre précédent, l'héritage permet de créer de nouvelles classes à partir de classes existantes mais ici, la nouvelle classe sera du même *type* que la classe existante (dite parente). Ici, on va réutiliser non pas un objet mais la classe elle-même. Une des implications principales de l'héritage est le polymorphisme décrit dans le chapitre suivant.

L'héritage est donc un moyen de réutiliser et mettre en commun des primitives des parents.

2.1. Syntaxe

```
class <NOM_DE_CLASSE>

inherit
    <NOM_DE_CLASSE_PARENTE_1>
    <NOM_DE_CLASSE_PARENTE_2>
    <NOM_DE_CLASSE_PARENTE_3>
          rename
```

```
            ...
        export
            ...
        undefine
            ...
        redefine
            ...
        select
            ...
        end

creation

feature

end -- end of class <NOM_DE_CLASSE>
```

On peut voir l'héritage comme un moyen de réutiliser du code déjà présent dans le(s) classe(s) parent(s). Une classe fille est en fait constituée du code de la classe parent et de ses primitives propres.

Par exemple, on peut définir une classe SEQUENCE_OF_PROTEIN comme un cas particulier de la classe SEQUENCE à laquelle on souhaite ajouter des primitives spécifiques aux protéines (structures_secondaires, weights, is_proteic...). La classe SEQUENCE_OF_PROTEIN est une **spécialisation** de la classe SEQUENCE. On dit aussi que la classe SEQUENCE_OF_PROTEIN hérite de SEQUENCE. On définit ainsi la **superclasse** et la **sous-classe**.

- SEQUENCE est la **superclasse** de SEQUENCE_OF_PROTEIN
- SEQUENCE_OF_PROTEIN est la **sous-classe** de SEQUENCE.

L'héritage suit les règles fondamentales suivantes:

✔ Les primitives sont héritées avec le même niveau de visibilité

✔ Les clauses creation ne sont jamais héritées

✔ La clause invariant

On peut distinguer trois types d'utilisation de l'héritage.

2.2. Spécialisation par enrichissement

Une des premières utilités de l'héritage est de pouvoir enrichir la classe parent de nouvelles propriétés.

Par exemple, SEQUENCE_OF_PROTEIN hérite de SEQUENCE mais en plus, on peut lui ajouter d'autres primitives spécifiques des protéines comme par exemple, le calcul de structures secondaires.

Voici la classe parent....

```
class SEQUENCE

creation
    from_string

feature
    from_string(seq : STRING) is
    do
        -- To do
```

```
    end

end-- end of class SEQUENCE
```

....et voilà la classe fille.

```
class SEQUENCE_OF_PROTEIN

inherit
    SEQUENCE

creation
    from_string
feature -- Ce qui est propre aux proteines
    calc_secondary_struct is
          -- calc secondary structures
    do
          -- Implantation pour calculer les structures secondaires
    end
end -- end of class SEQUENCE_OF_PROTEIN
```

Si on affiche l'interface publique de SEQUENCE_OF_PROTEIN avec l'utilitaire **short**; on obtient ceci:

```
class interface SEQUENCE_OF_PROTEIN

creation
   from_string (sequ: STRING)

feature(s) from SEQUENCE
   from_string (sequ: STRING)

feature(s) from SEQUENCE_OF_PROTEIN
   -- Ce qui est propre aux proteines

   calc_secondary_struct
      -- calc secondary structures

end of SEQUENCE_OF_PROTEIN
```

Il y a bien eu inclusion des primitives de SEQUENCE dans SEQUENCE_OF_PROTEIN.

2.3. Spécialisation par masquage

Parfois, l'héritage est utilisé pour « réajuster » le comportement de routines héritées du(des) parent(s).

2.3.1. Redéfinition d'une routine

```
class SEQUENCE

creation
    from_string

feature
    from_string(seq : STRING) is
    do
          codes := seq
    end

    calc_weight is
```

```
    do
          io.put_string("I'm computing weights ...approximately%N")
    end

end-- end of class SEQUENCE
```

La routine *calc_weight* calcule le poids moléculaire de la séquence de façon approximative. Si cette implantation ne me satisfait pas pour SEQUENCE_OF_PROTEIN, je peux la redéfinir en ajoutant dans la clause **inherit** une rubrique **redefine**

```
class SEQUENCE_OF_PROTEIN

inherit
    SEQUENCE
          redefine
                calc_weight
          end

creation
    from_string

    calc_weight is
    do
          io.put_string("I'm computing weight for protein%N")
    end
end -- end of class SEQUENCE_OF_PROTEIN
```

2.3.2. Renommage d'une routine

On peut renommer une routine avec **rename** dans la clause **inherit**. Ceci peut être intéressant si le nom de la routine de la classe parent n'est pas assez « parlant » dans la classe fille.

```
class SEQUENCE_OF_PROTEIN

inherit
    SEQUENCE
          rename
                from_string as with
          end

creation
    with


end -- end of class SEQUENCE_OF_PROTEIN
```

Réutilisation d'une routine de la classe parente: Precursor

Lorsqu'on redéfinit une méthode avec **redefine**, il arrive qu'on ne souhaite pas récrire complètement la routine mais simplement ajouter du code à la routine de la classe parente. Plutôt que de faire un copier/coller du code de la routine parente, on utilise le mot **Precursor**. Le **Precursor** correspond à l'appel de la routine parente. Pour l'utiliser vous devez respecter la signature de la routine appelée (c'est à dire le nombre et types d'arguments). Par exemple:

Dans une classe PARENT, il y a la routine

```
do_complex_things(a,b : REAL; c : SEQUENCE)
```

La classe FILLE héritant de PARENT pourra compléter la routine parentale en la redéfinissant de la manière suivante:

```
class FILLE

inherit
  PARENT
    redefine
      do_complex_things
    end
.....
.....
feature
  do_complex_things(a,b : REAL; c : SEQUENCE) is
    do
      Precursor(a,b,c)
      -- add new computations
    end
```

```
class SEQUENCE_OF_PROTEIN

inherit
    SEQUENCE
          redefine
                calc_weight
          end

creation
    from_string

    calc_weight is
    do
          Precursor
          io.put_string("I'm computing weight for protein%N")
    end
end -- end of class SEQUENCE_OF_PROTEIN
```

Dans cette exemple, la routine *calc_weight* appelle la routine *calc_weight* de la classe parent SEQUENCE par l'intermédiaire de Precursor, l'exécute (par exemple, pour initialiser des attributs), puis exécute le reste du code dans calc_weight de la classe fille SEQUENCE_OF_PROTEIN

2.4. Héritage multiple

Dans la plupart des cas, une classe fille hérite d'une seule classe parent, mais en Eiffel, il est possible d'hériter de plusieurs parents d'où le nom d'héritage multiple. Par exemple, la classe SEQUENCE_OF_PROTEIN pourrait héritée de :

```
class SEQUENCE_OF_PROTEIN
inherit
    SEQUENCE
    DRAWABLE
.......
```

Cette possibilité est très attractive mais doit être utilisée avec grande précaution. En effet, il est impératif de ne pas se retrouver dans une généalogie en « losange ». Par exemple:

HAUT (*draw*)	
GAUCHE(*draw*)	DROITE(*draw*)
BAS(*draw* ?)	

Les classes GAUCHE et DROITE héritent de la classe HAUT et la classe BAS hérite de GAUCHE et DROITE. Si une routine *draw* de la classe HAUT est redéfinie dans GAUCHE et DROITE, il est impossible de savoir quelle version de *draw* va appeler la classe BAS. Bien qu'il existe des moyens de s'en sortir en utilisant toutes les rubriques de inherit, le plus sage est d'éviter de se retrouver dans cette situation.

Remarque: Les classes qui décrivent des capacités comme « dessinable », « comparable », etc. ont un nom qui finit toujours par « ABLE » (de l'anglais *able to*: capable de). On définit ainsi des classes DRAWABLE, COMPARABLE, HASHABLE,etc.

2.5. Héritage non conforme

<A faire>