Module et Généricité : Exemple de la pile
Corrigé

Objectifs

- Comprendre la notion de module (paquetage en Ada).
- Comprendre la généricité.
- Savoir gérer deux instances d'un même module générique.

Rappel : Comme pour tous les TP, il faut commencer par faire un « `git pull` » depuis votre dossier « pim/tp » pour récupérer les fichiers fournis pour cette séance.

Exercice 1 : Module générique en Ada : exemple de la pile

L'objectif de cet exercice est de comprendre les modules en Ada. On s'appuiera en particulier sur le module `Piles` (fichiers `piles.ads` et `piles.adb`) et le programme `utiliser_piles.adb`.

1. Pourquoi les commentaires de spécification n'apparaissent que dans la spécification du module `Piles` et pas dans son implantation ?

Solution :

Parce que c'est la spécification du module qui doit spécifier le comportement des sous-programmes pour qu'ils soient connus à la fois de ceux qui vont utiliser le module (autres programmes et modules) et ceux qui vont l'implanter (implantation du module).

Ceci évite aussi d'avoir à écrire deux fois la même chose qu'il faudrait mettre à jour deux fois en cas de modification (correction, évolutions).

2. Où sont formalisés les contrats ?

Solution : De nouveau, ils le sont dans la spécification et s'appliquent aussi, bien sûr, dans l'implantation même si ils ne sont pas repris !

L'exécution du programme le montre. À la fin de `Illustrer_Plusieurs_Piles` on empile trop d'éléments. La violation de la précondition est indiquée.

3. Ce module `Piles` est générique. Comment le sait-on ?

Solution :

C'est le mot-clé `generic` dans la spécification du module, avant `package`.

Quels sont ses paramètres de généricité ?

Solution :

1. `Capacite`, une constante entière.

2. `T_Element`, un type.

Que faut-il faire pour pouvoir utiliser ce module `Piles` ?

Solution :

Il faut instancier le module générique en donnant une valeur à chaque paramètre de généricité. Par exemple `10` pour la capacité et `Character` pour le type des éléments.

```
1  package Pile_Caractere is
2          new Piles (Capacite => 10, T_Element => Character);
```

4. Où faudrait-il définir un sous-programme spécifique au module `Piles` ? Et sa documentation (commentaire de spécification) ?

Solution :

Seulement dans l'implantation du module. Et c'est là que sa spécification doit être donnée. C'est le cas de `Consommer_Blancs` de `Integer_IO`.

5. Qu'est ce que la surcharge ? La lecture de la procédure `Illustrer_Surcharge` aidera à répondre à cette question. On suivra en particulier les consignes données en commentaire et on décommentera la ligne indiquée.

Solution :

On parle de surcharge quand des sous-programmes différents portent le même nom. On les distingue par le nombre et le type des arguments. La connaissance des paramètres effectifs permet de sélectionner la signature et donc le sous-programme.

6. La procédure `Afficher_Element` n'est pas un paramètre de généricité du module `Piles` mais de sa procédure Afficher. Pourquoi ? La lecture de `Illustrer_Plusieurs_Afficher_Pour_Meme_Pile` aidera à répondre à cette question.

Solution : Ceci permet pour un même module d'instancier plus plusieurs fois la procédure Afficher pour qu'elle affiche un même objet du type `T_Pile` avec des `Afficher_Element` différents.

Conclusion : Il est important de définir les paramètres (généricité ou pas) au bon niveau !

7. Si dans un même contexte, par exemple un même sous-programme, on a besoin de deux piles avec des caractéristiques différentes (capacités différentes ou types des éléments différents), comment gérer le fait que les deux instances du module `Piles` fournissent les mêmes noms (`T_Pile`, `Empiler`, `Est_Vide`...) ? Ce cas de figure est présent dans `Illustrer_Plusieurs_Piles`.

Solution :

En fait, créer deux modules de `Piles` avec des paramètres différents conduit à deux type `T_Pile` de même nom. Le `use` des deux, provoquera une erreur de compilation car le compilateur ne pourra pas choisir quel type `T_Pile` prendre (il est dans les deux instances de module). Il faudra donc que le programmeur utilise le nom qualifié par le nom du module.

Exercice 2 : Expression bien appairée

On s'intéresse au fichier `parenthesage.adb`.

1. Compléter la fonction `Index` puis la procédure `Verifier_Parenthesage`.

Pour la procédure `Verifier_Parenthesage`, on utilisera deux piles (même si une seule pile pourrait suffire) : la première `Pile_Ouvrants` stockera les symboles ouvrants rencontrés lors de l'analyse de la chaîne et la seconde, `Pile_Indices`, stockera leur indice.

Un programme de test est fourni pour chacun de ces sous-programmes.

Remarque : Pour un paramètre de type `Chaine : in String`, on peut avoir des indices différents pour la chaîne. Aussi, il est conseillé d'utiliser `Chaine'First`, `Chaine'Last` ou `Chaine'Range`.

2. Est-ce que l'on peut utiliser `Verifier_Parenthesage` dans d'autres programmes ? Que faudrait-il faire pour y arriver ?

Solution :

Actuellement, on ne peut pas réutiliser ce sous-programme.

Il faudrait faire un module. On ne met dans la spécification du module que la spécification de ce sous-programme Index. Son implantation et l'autre sous-programme sont dans l'implantation du module.

Le programme de test est mis dans un programme principal qui utilise ce module.