Modules — Constructeurs de types
Corrigé

Objectifs

- Comprendre les modules (encapsulation et masquage d'information)
- Savoir définir et manipuler les types utilisateurs (énuméré, enregistrement et tableau)
- Spécifier, implanter et tester des sous-programmes... Toujours !

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 Le module *Dates* se présente en Ada sous la forme de deux fichiers `dates.ads` et `dates.adb`. Le fichier `exemple_dates.adb` est un programme utilisant les dates.

1. Expliquer à quoi correspondent les fichiers `.ads` et `.adb`.

Solution : Le fichier `.ads` (Ada Spécification) correspond à la spécification (interface) du module. Il définit les éléments qui seront accessibles aux utilisateurs du module.

Le fichier `.adb` (Ada Body) correspond à l'implantation (corps) du module. Il donne en particulier l'implantation (le corps) des sous-programmes spécifiés dans l'interface du module. Il peut aussi définir d'autres éléments.

Notons que l'interface d'un module peut comporter une partie privée qui est inaccessible à l'utilisateur du module et fait donc partie du corps.

2. Pourquoi n'y a-t-il pas de commentaire devant les sous-programmes `Initialiser`, `Le_Jour`, `Le_Mois` et `L_Annee` de `dates.adb` ?

Solution : Ces sous-programmes font partie de l'interface du module où ils sont déjà été documentés. Inutile de copier cette documentation (sémantique) dans le corps (pas de redondance).

On est obligé de reprendre la signature du sous-programme (sans les contrats) pour lier l'implantation à la bonne spécification (surcharge).

3. Pourquoi la sémantique (le commentaire) du sous-programme `Afficher_Deux_Positions` est donnée dans `dates.adb` ?

Solution : Ce sous-programme est local, privé au module. Sa documentation n'a pas été donnée dans la spécification du module, on la donne donc dans l'implantation du module.

4. Lister ce qu'un utilisateur du module *Dates* peut utiliser de ce module.

Solution : L'utilisateur du module a accès à ce qui est dans la partie publique de la spécification du module :

1. la définition du type `T_Mois`,
2. la déclaration du type `T_Date` (le type étant privé, on peut seulement déclarer des variables de ce type, faire une affectation de entre `T_Date` et utiliser l'opérateur d'égalité sur des dates),
3. la spécification des sous-programmes `Initialiser`, `Le_Jour`, `Le_Mois` et `L_Annee`.

5. Lire le fichier `exemple_dates_erreurs.adb`. Certaines instructions seront refusées par le compilateur. Les identifier et donner la raison du refus dans un commentaire de fin de ligne.
Solution :

1. On n'a pas le droit d'utiliser la procédure `Afficher_Deux_Positions` car elle n'apparaît pas dans la spécification du module. Même si cette procédure est bien définie dans le corps du sous-programme, on ne peut pas l'utiliser à l'extérieur du module. C'est le principe du **masquage d'information**.
2. On n'a pas le droit d'utiliser les champs (attributs) d'une variable de type `T_Date` car le type étant privé, on n'a pas accès à sa définition qui est donnée dans la partie privée du module, toujours le **masquage d'information**. On ne peut dans mettre un point « . » après la variable « `Une_Date` » car ne connaissant pas la définition de son type, on ne peut pas savoir si c'est un enregistrement ou non.

Contrôler ensuite les réponses en compilant le fichier.
Solution : Ceci est bien confirmé par le compilateur.

```
exemple_dates_erreurs.adb:19:05: "Afficher_Deux_Positions" is undefined
exemple_dates_erreurs.adb:30:05: invalid prefix in selected component "Une_Date"
exemple_dates_erreurs.adb:31:05: invalid prefix in selected component "Une_Date"
gnatmake: "exemple_dates_erreurs.adb" compilation error
```

Mettre en commentaire les lignes refusées par le compilateur.
6. Expliquer pourquoi le type `T_Mois` ne peut pas être déclaré privé.
Solution : Car ses valeurs doivent être connues des utilisateurs du module puisqu'elles doivent être passées en paramètre de `Initialiser`. Sinon, il aurait fallu ajouter d'autre opérations qui permettent d'obtenir une donnée de type `T_Mois`, par exemple à partir d'un entier de 1 à 12.
7. Modifier la déclaration du type `T_Date` dans le fichier `dates.ads` pour en faire un type *très privé*. On remplacera **`is private`** par **`is limited private`**.

Quelles sont les incidences de cette modification ?
Solution : On perd l'affectation entre `T_Date` et l'égalité (et la différence).

Compiler le fichier `exemple_dates_erreurs.adb` pour confirmer les réponses.
Solution :

```
exemple_dates_erreurs.adb:36:05: left hand of assignment must not be limited type
exemple_dates_erreurs.adb:41:19: there is no applicable operator "=" for private type "T_Date"
gnatmake: "exemple_dates_erreurs.adb" compilation error
```

Exercice 2 : Gestion d'un stock de matériel informatique

Afin de connaître l'état de son stock de matériel informatique, une entreprise décide de réaliser un

programme informatique. Un matériel informatique est caractérisé par un numéro de série (que l'on prendra entier mais qui dans une évolution future pourrait être une chaîne de caractères), sa nature (unité centrale, disque, écran, clavier ou imprimante), son année d'achat et son état qui indique s'il est en état de fonctionnement ou pas.

Les opérations que l'entreprise souhaite faire sur ce stock sont les suivantes :

1. Enregistrer un nouveau matériel dans le stock à partir de son numéro de série, sa nature et son année d'achat. On suppose que le nouveau matériel enregistré est en état de fonctionnement.
2. Obtenir le nombre de matériels enregistrés dans le stock.
3. Mettre à jour l'état d'un matériel enregistré dans le stock à partir de son numéro de série.
4. Obtenir le nombre de matériels qui sont hors d'état de fonctionnement.
5. Supprimer du stock un matériel à partir de son numéro de série.
6. Afficher tous les matériels du stock. Ce sous-programme affiche dans le terminal.
7. Supprimer tous les matériels qui ne sont pas en état de fonctionnement.
8. ...

L'analyste a décidé (et impose donc) de définir un seul module (`stocks_materiel`) et de représenter le stock par un unique tableau.

1. Définir le type `Stock` (et les autres types utiles). On précisera les invariants de type.

2. Nous allons implanter maintenant les différentes opérations demandées par l'entreprise en construisant progressivement le module et un scénario [1] d'exemple. Pour chaque opération, on suivra les étapes suivantes :

1. Spécifier le sous-programme dans la spécification du module.
2. Utiliser le sous-programme dans le scénario (avec `pragma Assert` pour en vérifier l'effet).
3. Coder le sous-programme dans l'implantation du module sans donner son code (on mettra `null;` pour une procédure ou on retournera une valeur quelconque pour une fonction).
4. Compiler et exécuter l'application. Le scénario ne devrait pas fonctionner.
5. Implanter le sous-programme dans le corps (implantation) du module.
6. Compiler et exécuter l'application. S'il y a des erreurs, il faut les corriger !
7. Pousser les modifications sur Gitlab, le message indiquera l'opération implantée.

Dans le squelette qui est donné, les étapes précédentes ont été réalisées pour trois sous-programmes : créer le stock (sous-programme qui doit être appelé sur un stock avant de pouvoir l'utiliser), nombre de matériels dans le stock et enregistrer un matériel. Cependant leur implantation correcte n'a pas été donnée dans l'implantation du module.

Attention : Sauf indication contraire, aucun sous-programme n'est interactif : ils ne doivent pas faire d'entrée/sortie (clavier/écran).

Construire progressivement le module et le scénario.

1. On pourrait aussi définir des programmes de test pour les sous-programmes.

Solution : Voir la solution proposée.

Pour aller plus loin...

Exercice 3 : Autre organisation en module

Nous avons fait un seul module pour représenter le stock de matériel informatique. Il aurait été plus judicieux d'en faire davantage.

1. Indiquer les modules qui auraient dû être faits.

Solution : Ici, on a défini un seul module qui regroupe plusieurs types et les sous-programmes associés. Il aurait été préférable de faire plusieurs modules, chacun encapsulant un type et des sous-programme associés.

Ainsi, on pourrait définir un module Etats, un module Stocks et un module Matériels. Notons qu'on pourrait s'appuyer sur le module générique Tableau vu en cours pour définir le module Materiels.

2. Restructurer le code pour les faire apparaître.

Solution : À faire...