Les fonctions

Cours

v1

IUT d'Annecy, 9 rue de l'Arc en Ciel, 74940 Annecy

Exercices complémentaires

Après un rapide rappel sur les notions de base de la programmation orientée objet, ce cours s'intéresse à la problématique suivante :

Problématique

Comment mettre en place une programmation orientée objet sous CodeSys ?

1 Rappels sur la programmation orientée objet

1.1 Quelques définitions

La programmation orientée objet est une technique de programmation qui consiste à découper un programme en plusieurs objets qui interagissent entre eux. Chaque objet est une instance d'une classe. Une classe est un ensemble d'attributs et de méthodes. Les attributs sont des variables propres à chaque objet. Les méthodes sont des fonctions propres à chaque objet.

Il s'agit d'une façon de penser la programmation. Si elle peut paraître plus complexe au début, elle permet de structurer le code, de le rendre plus lisible et plus facilement modifiable.

La définition des objets (création de classes) peut être complexe, mais l'utilisation des objets est largement simplifiée. C'est d'ailleurs un avantage majeur de la programmation orientée objet : l'utilisateur d'un objet n'a pas besoin de connaître sa définition pour l'utiliser. Il lui suffit de connaître les méthodes et les attributs de l'objet.

Un object est une instance d'une classe. Cela veut simplement dire qu'une classe définit une structure (attributs et méthodes) et qu'un objet est un cas particulier de cette structure. Par analogie, on peut parler des plans d'une maison qui définissent ce qu'est une maison (nombre de pièces, surface, etc.) et des maisons qui sont des instances de ces plans.

Chaque maison pourra avoir un nombre de pièces différentes, une couleur de toit différente, mais toutes les maisons auront les mêmes caractéristiques de base (elles auront toutes un toit, des pièces, des murs, des fenêtres, etc.).

D'un point de vue extérieur, un objet est une "boite noire" dont les détails d'implémentation sont cachés. On ne peut pas savoir comment est codé un objet, on intéragit avec lui par l'intermédiaires des fonctions d'interfaces. C'est le principe d'encapsulation.

Principe d'encapsulation

'encapsulation est un principe de la programmation orientée objet qui consiste à cacher les détails d'implémentation d'un objet. L'accès aux données de l'objet ne peut alors se faire que par l'intermédiare des services proposées.

Cela a pour objectif d'en simplifier l'utilisation mais également de **garantir l'intégrité des données** contenues dans l'objet. En effet, les fonctions d'interfaces mises en places contrôlent la bonne utilisation des données.

Exemple : Etape d'un grafcet

On peut définir une classe *CStep* qui représente une étape d'un grafcet. Cette classe peut contenir, entre autres, les attributs et méthodes suivants :

Attributs

- **m_name** : nom de l'étape
- **m_xActivityBit** : bit d'activité (actif ou non)
- **m_tActivationDuration** : temps d'activation de l'étape

Méthodes

- **MInit** : Initialise l'étape
- **MActivate** : active l'étape
- **MDeactivate** : désactive l'étape

Chaque étape du grafcet sera alors un objet de la classe *CStep*.

La notion d'Héritage

'héritage est un principe de la programmation orientée objet qui consiste à créer une nouvelle classe à partir d'une classe existante. La nouvelle classe **hérite** alors des attributs et méthodes de la classe existante.

Par exemple, on peut créer une classe *Forme* qui contient les attributs et méthodes communs à toutes les formes géométriques fermées (comme son aire). On peut ensuite créer une classe *Rectangle* qui hérite de la classe *Forme* et qui contient, en plus, les attributs et méthodes spécifiques aux rectangles. On peut ensuite créer une classe *Cercle* qui hérite de la classe *Forme* et qui contient les attributs et méthodes spécifiques aux cercles.

Le Polymorphisme

e polymorphisme est un principe de la programmation orientée objet qui consiste à utiliser un objet de manière différente en fonction du contexte.

Plus spécifiquement, le polymorphisme autorise à utiliser un même nom pour des méthodes différentes selon si elle s'applique sur un contexte ou un autre.

En reprenant l'exemple précédent, on peut créer une méthode *MComputeArea* dans la classe *Forme* qui calcule l'aire de la forme. Cette méthode sera différente dans la classe *Rectangle* et dans la classe *Cercle* car l'aire d'un rectangle et l'aire d'un cercle ne se calculent pas de la même manière. En fonction de l'objet utilisé, la méthode *MComputeArea* sera différente.

2 La programmation orientée objet sous CodeSys

Application : Etape d'un grafcet

fin de faciliter la compréhension de cette partie, nous allons construire, tout au long de ce cours, une classe *CStep* qui représente une étape d'un grafcet. Cette classe contiendra les attributs et méthodes suivants :

Attributs

- **m_xActivityBit** : bit d'activité (actif ou non)
  - ◇ Accessible en lecture uniquement
- **m_tActivationDuration** : Stocke le temps d'activation de l'étape
  - ◇ Accessible en lecture uniquement
- **m_tTaskPeriod** : période de la tâche
  - ◇ Accessible en lecture et en écriture

Méthodes

- **MInit** : Méthode appelée à l'initialisation du programme
- **MActivate** : active l'étape
- **MDeactivate** : désactive l'étape

2.1 Les classes

On l'a vu, la notion de classe nous demande de pouvoir créer des attributs et des méthodes. Les attributs sont des variables internes aux classes qui devront être conservées entre deux utilisations d'un objet de cette classe. L'utilisation de fonctions (POU FUNCTION) ne serait pas satisfaisante car la durée de vie d'une variable interne à une fonction est limitée à l'exécution de cette fonction.

A l'inverse, un bloc fonctionnel s'utilise après en avoir créé une instance et ses variables internes sont conservées entre d'une utilisation à l'autre. Les POU FB semblent donc adaptées à la création de classes.

À retenir

Nous utiliserons les blocs fonctions pour implémenter la notion de classe sous CodeSys.

Déclaration d'une classe sous CodeSys

éfinir une classe sous CodeSys revient à définir un bloc fonctionnel. Afin de différencier les variables privées (inaccessibles depuis l'extérieur) des attributs publics (accessibles depuis l'extérieur), on utilisera la convention suivante :

- Les variables publiques commenceront par *m_*
- les variables privées n'ont pas de préfixe particulier

Ainsi, la déclaration d'une classe `<CName>` se fera de la manière suivante :

```
1        FUNCTION_BLOCK <CName> (* définition de la classe <CName> *)
2            VAR
3                <type><Name> : <type>; (* variable locale non accessible de l'extérieur *)
4                m_<type><Name> : <type>; (* membre de l'objet accessible de l'extérieur sous certaines conditions*)
5            END_VAR
6
```

uisqu'elle ne possède que des attributs accessibles, la déclaration de notre classe *CStep* se fera de la manière suivante :

```
1 FUNCTION_BLOCK FB_CStep
2     VAR
3         m_xActivityBit : BOOL; (* bit d'activité (actif ou non) *)
4         m_tActivationDuration : TIME; (* Stocke le temps d'activation de l'étape *)
5         m_tTaskPeriod : TIME; (* période de la tche *)
6     END_VAR
```

L'instanciation d'un objet de cette classe pourra alors s'écrire `Etape1 : FB_CStep;`

Pour le moment, ces attributs ont été déclarés dans notre classe *CStep* mais ils ne sont pas accessibles depuis l'extérieur (Il est impossible d'évaluer l'expression `Etape1.m_xActivityBit`). Ces variables sont "coincées" à l'intérieur de notre bloc fonction. Pour les rendre accessibles, nous allons utiliser l'objet `PROPERTY` proposé par CodeSys.

2.2 L'objet PROPERTY (propriété)

L'objet PROPERTY est un outil proposé par CodeSys pour la programmation orientée objet. Il permet de définir des attributs (variables) propres à un contexte avec un niveau d'accessibilité paramétrable (publique, privée, protégée ou interne).

Cela signifie que l'on peut définir des variables à l'intérieur d'une classe qui pourront être accessibles depuis l'extérieur de la classe par l'intermédiaire de méthodes appelées *accesseurs*.

Les accesseurs

l existe deux types d'accesseurs : GET et SET.
GET : Permet de lire la valeur d'une variable
SET : Permet d'écrire la valeur d'une variable

Une propriété est donc un objet que l'on associe à une POU. On décrit ensuite ses accesseur pour définir comment lire ou écrire la valeur de la propriété.

2.3 Méthodes

L'objet METHOD est un outil proposé par CodeSys pour la programmation orientée objet. Il permet de définir des méthodes (fonctions) propres à un contexte avec un niveau d'accessibilité paramétrable (publique, privée, protégée ou interne).

Comme une fonction, on peut définir :
- Des variables d'Entrées, de sorties ou d'entrées-sorties
- Des variables locales à la méthode
  - ◇ La durée de vie de ces variables locales est alors limitée à l'exécution de la méthode

Par ailleurs, **une méthode a accès au contexte de l'instance à laquelle elle est associée**. Cela signifie qu'elle a accès aux variables et méthodes de l'objet dont elle dépend.

Enfin, CodeSys autorise la déclaration de METHODS dans une interface. Elle devra alors être implémentée dans les classes qui implémentent cette interface.

À retenir

Les méthodes des classes que nous définirons seront des méthodes assocées au bloc fonction concerné.

Les méthodes de la classe CStep

es méthodes de la classe *CStep* seront définies dans le bloc fonctionnel *FB_CStep*. Elles seront donc des méthodes associées à ce bloc fonctionnel.

Méthodes

- **MInit** : Méthode appelée à l'initialisation du programme
- **MActivate** : active l'étape
- **MDeactivate** : désactive l'étape

```
METHOD MInit : BOOL
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR
// Code de la méthode
// ...
// Fin du code de la méthode
END_METHOD

```