Une technique d'implémentation objet de systèmes

Le modèle RdP

Le comportement de tout système, ou sous-système, sera décrit au travers d'un Réseau de Pétri (RdP). De ce fait, dans le système chaque objet représentant un sous-système devra être capable de gérer l'évolution d'un réseau de Pétri : (gestion d'un ensemble de places, une méthode gérant les tirs de transition, …).

Puisque cette propriété est commune à tout objet du système, nous proposons de réaliser une classe abstraite *systeme* qui s'occupe de cette gestion. Elle définit :

- un ensemble de places :

```
int * place_; // Les places du RdP
int nb_place_;  // Le nombre de places du RdP
```

- une méthode d'évolution du RdP

(cette méthode est abstraite, aux classes dérivées de l'implémenter)

```
virtual int evolue(void)=0; // Evolution du RdP
```

- un ensemble énuméré optionnel pour un accès plus intuitif aux places

ex : `enum nom_place{p_initial, p_op, p_fin_op}; //noms associes aux places`

utilisation : `place_[p_initial]=0 ;`

ou au travers des fonctions de marquage : `marquer(p_initial); //recommandé`

- des méthodes permettant par exemple l'affichage de l'état du système

Toute classe implémentant un système devra :

1. hériter de la classe *systeme*
2. (optionnel) nommer les places par la définition du type énuméré *nom_place*
3. redéfinir la méthode *evolue* qui définie le comportement du RdP. Technique classique du switch.

Les paradigmes de communication

Si le système est décomposé en sous-systèmes, une communication est souvent nécessaire entre les différents objets. Cette communication est dissymétrique : il existe un appelant et un appelé. Nous allons voir comment implémenter ces moyens de communications entre objets.

Hormis la dissymétrie des rôles, il existe deux modes de communication : Asynchrone (communication par place) et Synchrone (communication par transitions). Ces deux modes nécessiteront deux types d'implémentation différente.

La communication entre objets est naturellement réalisée par appel de méthode. Deux objets souhaitant communiquer devront donc échanger par l'intermédiaire de méthodes.
Une technique consiste à :

- Définir une méthode (un point de communication) pour l'objet appelé
  Exemple : `bool machine::op2(void)`
- Appeler la méthode d'un objet avec lequel on souhaite communiquer pour l'objet appelant.
  Exemple : `if (place_[e_op2] && m1_->op2())`

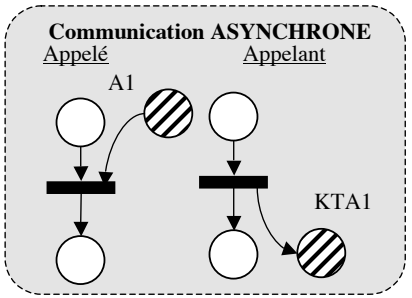
Remarque :
Afin de pouvoir appeler une méthode d'un objet avec lequel il communique, l'appelant devra posséder *une référence* sur le (les) objet(s) avec le(s)quelle(s) il communique.

```
protected :
     machine *m1_, *m2_; // Les machines qu'il synchronise
     kambanKA2 *k_;      // Le kamban avec lequel il est synchronise
```

Attention, le programmeur doit s'assurer que toute référence pointe bien sur un objet existant, par exemple en les initialisant dans le constructeur :

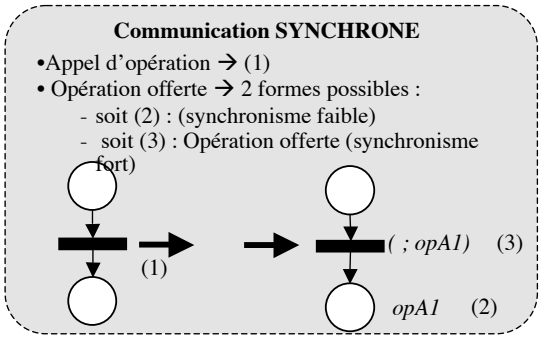
```
kambanKA1::kambanKA1(machine *m1, machine *m2, kambanKA2 *k) : systeme(4),
m1_(m1), m2_(m2), k_(k)
{}
```

Le contenu de la méthode réalisant le point de communication dépendra du mode de communication requis.



Dans le cas d'une communication asynchrone, l'objet appelé possèdera une place supplémentaire destinée à la communication. La méthode offerte par l'appelé réalise le marquage de la place offerte pour la communication.
L'appel de cette méthode ne sera pas bloquant pour l'appelant (dans le sens ou la transition sera tirée).
Exemple : Partie de la définition de l'appelé

```
enum nom_place{p_1, p_message, p_2} ;
…
void machine::message(void)
{
     place_[p_message]++ ; // marquage de la place de communication
}
…
int machine::evolue(void)
{
   if (place_[p_1] && place_[p_message])
   {
     place_[p_1]--; place_[p_message]--;
     place_[p_2]++;
   }
… }
```



Dans le cas d'une communication synchrone, il est nécessaire de posséder un « retour » de l'appelé (indiquant si la transition est tirable chez l'appelé) afin de déterminer chez l'appelant s'il faut ou non tirer la transition locale. Ce paradigme de communication peut se modéliser par un mécanisme de rendez-vous (ajout de deux places), mais il peut s'implémenter plus simplement.
Puisque l'appelant doit attendre le tir de la transition de l'appelé, la méthode (le point de communication) peut retourner une valeur booléenne indiquant si la transition est tirable (chez l'appelé). Dans le cas positif, la transition est tirée (réalisée par la méthode) chez l'appelé, et la valeur booléenne renvoyée permet le tir de la transition chez l'appelant. Dans le cas négatif, aucun tir n'est réalisé et le système n'évolue pas.

Exemple :
Coté appelé :

```
// Definition d'un point de synchronisation
// Le tir est réalisé dans le corps de la méthode
bool machine::op1(void)
{
  int nb_tr=0;
  if (place_[e_libre])                // Condition du tir
    {      place_[e_libre]--; place_[e_op1]++; // Marquage des places
           nb_tr++;                   // inc. du nbr de trans tirees
    }
  return(nb_tr);
}
```

Coté appelant

```
int kambanKA1::evolue(void)
{
  …
  if (place_[e_op1] && m2_->fin_op1()) //condit° tir (locales et distantes)
    {
      place_[e_op1]--; place_[e_av_op2]++;  // evolution du marquage
      nb_tr++;
    }
…}
```

Astuce pour l'implémentation d'un échange symétrique

Problème :

Les communications symétriques posent un problème si l'on utilise uniquement les passages de référence par constructeur. En effet si l'objet *A* doit appeler *B* et *B* répondre à *A*, chacun doit posséder une référence sur l'autre.
Ceci est impossible à réaliser uniquement avec les constructeurs puisque cela imposerait la déclaration suivante :

```
A A1(&B1) ;
B B1(&A1) ;
```

Une solution consiste à *ajouter une méthode* dans la classe pour initialiser le pointeur qui référencera l'objet à appeler.

L'instanciation des objets pourrait alors être de ce type :

```
A A1() ;
B B1(&A1);
A1.connect_B1(&B1) ;
```

Et les modifications à faire dans la classe A sont les suivantes :

```
class B; // Déclaration anticipée de la classe B

class A : public systeme
{...
// exemple de code à ajouter dans la classe A.
private :
      B *ref_B1; // élément avec lequel se synchroniser
public :
      void connect_B1(B *obj)
      { ref_B1=obj; }
...} ;
```

Exemple de communications symétriques