Université d'Aix-Marseille III - Licence de math-info 2ème année

I4 : Programmation objet - TP n°5 : classes abstraites

Simulation d'un microprocesseur simplifié (examen de mai 2005)

L'objectif de ce problème est de modéliser et de simuler l'exécution par un microprocesseur d'un programme écrit dans le langage machine qu'il reconnaît. Le microprocesseur et le langage machine que nous allons décrire sont très simplifiés.

La mémoire est composée d'une suite de cases numérotées de 0 à 255. Chaque case peut contenir une donnée ou une instruction. Alors qu'en réalité chaque case de la mémoire devrait contenir un octet et que c'est le contexte qui permettrait de l'interpréter comme une donnée ou une instruction (ou le début d'une instruction), nous avons décidé de simplifier ainsi : chaque case de la mémoire simulée peut contenir soit un objet de type `Octet`, qui représente un entier compris entre 0 et 255, soit un objet de type `Instruction`, qui représente une instruction complète. Pour que ceci soit possible, nous définissons la classe abstraite `Contenu` ainsi :

```
abstract class Contenu {}
```

Chaque case de la mémoire sera de type `Contenu` et les classes `Octet` et `Instruction` hériteront de la classe `Contenu`.

1. *Ecrire la classe* `Octet` *qui représente des entiers compris entre 0 et 255 et contient les méthodes suivantes : un constructeur qui prend un entier en paramètre et mémorise cet entier modulo 256, la méthode* `boolean nul()` *qui indique si la valeur de l'octet est nulle et la méthode* `int toInt()` *qui retourne la valeur de type* `int` *qui est celle de l'octet.*

2. Le microprocesseur contient un certain nombre de registres qui mémorisent chacun un octet. Les registres D0 et D1 sont des registres de données qui sont notamment utilisés pour effectuer des opérations arithmétiques. Le registre d'état SR mémorise le résultat de la dernière instruction exécutée. Contrairement aux registres d'état des microprocesseurs habituels, il ne mémorise pas des informations dans des bits identifiés chacun par un nom, il se contente de mémoriser tout l'octet qui est le résultat de la dernière opération effectuée (par exemple, la valeur qui a été transférée par une instruction MOVE, le résultat de l'addition effectuée par l'instruction ADD, la valeur qui a été empilée par l'instruction PUSH, etc). Il y a aussi des registres d'adresse, c'est-à-dire des registres dont le contenu est un octet qui représente une adresse de la mémoire. Le registre PC mémorise l'adresse de la prochaine instruction à exécuter. Le registre SP contient l'adresse du sommet de la pile. De plus, il existe deux registres d'adresse A0 et A1 qui ne sont pas réservés à un usage particulier.

Ecrire la classe `Registre` *qui représente un registre mémorisant un octet et qui contient les méthodes suivantes : un constructeur qui prend une instance d'*`Octet` *en paramètre et les méthodes d'accès* `Octet get()`, `void set(Octet)` *et* `void set(int)`. *De plus, cette classe définit les attributs de classe D0, D1 et SR qui sont publics et initialisés à 0.*

Ecrire la classe `RegistreAdresse` *qui hérite de* `Registre` *et qui définit les attributs de classe PC, SP, A0 et A1. Ces attributs de classe sont publics et sont tous initialisés à 0 sauf SP, qui est initialisé à 255.* Cette classe existe pour des motifs liés à l'adressage indirect que nous expliquerons plus loin.

3. *Ecrire la classe* `Memoire` *qui représente une mémoire de 256 cases et qui contient notamment les* **méthodes de classe** *suivantes : quatre méthodes d'accès à un octet de la mémoire* `Contenu get(Octet adresse)`, `Contenu get(int adresse)`, `void set(Octet adresse, Contenu c)` *et* `void set(int adresse, Contenu c)`. *On fera en sorte que cette classe ne puisse pas avoir d'instance et qu'on ne puisse pas non plus en hériter. Cette classe contient aussi la méthode de classe* `void execute()` *qui exécute un programme à partir de l'adresse contenue dans le registre PC.* Un programme s'exécute instruction après instruction jusqu'à ce que l'instruction soit de type `Halt` (qui hérite de la classe `Instruction`) ou que le registre PC

contienne l'adresse d'une case qui contient un objet de type `Octet` (et non plus de type `Instruction`). Le cycle d'exécution d'une instruction est le suivant : PC est incrémenté d'une unité, l'instruction pointée par PC est exécutée et on recommence. Toutes les instructions sont des instances de classes qui héritent indirectement de la classe `Instruction` définie ainsi :

```
abstract class Instruction extends Contenu
{
   abstract void execute();
}
```

4. Les instructions du langage machine se différencient en premier lieu par le nombre d'opérandes qu'elles mettent en jeu : 0, 1 ou 2 opérandes. Par exemple, l'instruction `HALT` n'a pas d'opérande, l'instruction `PUSH D0` possède un opérande (D0) qui désigne la valeur à empiler, et l'instruction `ADD D0,D1` (qui ajoute à D1 la valeur de D0) possède deux opérandes.

Les classes de base de la hiérarchie des classes qui définissent les opérandes possibles sont définies ainsi :

```
abstract class Operande
{
   abstract Octet getValeur();
}
abstract class OperandeModifiable extends Operande
{
   abstract void setValeur(Octet x);
}
```

Les instructions binaires sont des instructions à deux opérandes. L'instruction binaire récupère les valeurs des deux opérandes, effectue une certaine opération avec ces deux valeurs, puis stocke le résultat dans le deuxième opérande et dans le registre SR.

Ecrire la classe abstraite `InstructionBinaire` *qui hérite de la classe* `Instruction` *et définit les méthodes suivantes : le constructeur qui prend une instance d'*`Operande` *et une instance d'*`OperandeModifiable` *en paramètre, la méthode abstraite* `Octet operation(Octet, Octet)` *qui est destinée à être définie par les sous-classes d'*`InstructionBinaire` *qui définissent des instructions binaires du langage machine, et la méthode* `void execute()` *qui exécute l'instruction binaire de la manière dont on l'a décrite ci-dessus.*

Ecrire la classe `Add` *qui hérite d'*`InstructionBinaire` *et définit la méthode* `Octet operation(Octet, Octet)` *de telle manière qu'elle fasse la somme des valeurs des deux octets passés en paramètre et retourne l'octet contenant le résultat de cette somme.*

Ecrire la classe `Move` *qui hérite d'*`InstructionBinaire` *et définit la méthode* `Octet operation(Octet, Octet)` *de telle manière qu'elle retourne le deuxième paramètre.* Les instances de cette classe sont des instructions de déplacement d'octet : la valeur désignée par le premier opérande est copié à l'endroit désigné par le deuxième opérande.

5. Les instructions unaires sont des instructions à un seul opérande. L'instruction récupère la valeur de l'opérande, effectue une certaine opération avec cette valeur, puis stocke le résultat dans l'opérande (uniquement s'il est modifiable) et dans le registre SR.

Ecrire la classe abstraite `InstructionUnaire` *qui hérite de la classe* `Instruction` *et définit les méthodes suivantes : le constructeur qui prend une instance d'*`Operande` *en paramètre, la méthode abstraite* `Octet operation(Octet)` *qui est destinée à être définie par les sous-classes d'*`InstructionUnaire` *qui définissent des instructions unaires du langage machine, et la méthode* `void execute()` *qui exécute l'instruction unaire de la manière dont on l'a décrite ci-dessus.*

Ecrire la classe `Dec` *qui hérite d'*`InstructionUnaire` *et définit la méthode* `Octet operation(Octet)` *qui retourne la valeur passée en paramètre décrémenté de 1.* Cette classe simule l'instruction DEC qui sert à décrémenter son opérande.

Ecrire la classe `Push` *qui hérite d'*`InstructionUnaire` *et définit la méthode* `Octet operation(Octet)` *qui place la valeur passée en paramètre à l'adresse contenue dans SP, décrémente le contenu de SP et retourne la valeur passée en paramètre.* Cette classe simule l'instruction PUSH qui sert à empiler son opérande.

Ecrire la classe `Pop` *qui hérite d'*`InstructionUnaire` *et définit la méthode* `Octet operation(Octet)` *qui incrémente le contenu de SP puis retourne la valeur située à l'adresse contenue dans SP.* Cette classe simule l'instruction POP qui sert à dépiler une valeur et à la stocker à l'emplacement désignée par son opérande.

Ecrire la classe `Jne` *qui hérite d'*`InstructionUnaire` *et définit la méthode* `Octet operation(Octet)` *qui teste si le contenu de SR est non nul, place la valeur passée en paramètre dans PC si c'est le cas, et retourne la valeur passée en paramètre.* Cette classe simule l'instruction JNE (Jump if Not Equal) qui effectue un branchement à l'adresse désignée par son opérande si SR contient une valeur différente de zéro.

6. Chaque opérande se caractérise par son type d'adressage : registre, immédiat, direct ou indirect. L'opérande peut donc être un *registre*. Il peut aussi être une valeur à prendre comme telle (adressage *immédiat*). Son adressage peut aussi être *direct* et alors cet opérande est l'adresse (un octet) de la valeur désignée. Enfin, son adressage peut être *indirect* et alors cet opérande est un registre d'adresse, A0 ou A1, contenant l'adresse de la valeur désignée. A l'écriture, pour différencier l'adressage immédiat de l'adressage direct, on place le signe # devant la valeur pour un adressage immédiat (et rien devant pour un adressage direct). Pour différencier l'adressage registre de l'adressage indirect, on place le signe @ devant le registre pour l'adressage indirect. Ex : `MOVE #2,@A0` copie la valeur 2 à l'adresse contenue dans A0 et `MOVE 5,A1` copie la valeur située à l'adresse 5 dans le registre A1. Seul un opérande dont l'adressage est immédiat est non modifiable (car une valeur ne se modifie pas). Les opérandes dont l'adressage est direct ou indirect ou est un registre désignent un registre ou une adresse dont le contenu peut être modifié.

Ecrire les classes concrètes `OperandeRegistre`, `OperandeImmediat`, `OperandeDirect` *et* `OperandeIndirect` *qui héritent des classes* `Operande` *ou* `OperandeModifiable` *(qu'on a explicitée en question 4).*

7. Afin de simplifier la création d'instances de classes héritant d'`Operande`, nous allons définir la classe `OperandeFactory` qui contient un ensemble de méthodes de classe dont le nom est `newInstance` et qui possèdent un ou deux paramètres.

Ecrire la classe `OperandeFactory` *de telle manière qu'elle permette la création d'opérandes grâce aux instructions suivantes* :

```
OperandeDirect di = OperandeFactory.newInstance(100);
OperandeRegistre re = OperandeFactory.newInstance(Registre.D0);
OperandeImmediat im = OperandeFactory.newInstance("#", 1);
OperandeIndirect in = OperandeFactory.newInstance("@", RegistreAdresse.A0);
```

En guise de complément d'aide à la compréhension du sujet, voici un exemple d'une suite d'instructions illustrant l'utilisation des classes que nous avons vues jusqu'à présent :

```
// MOVE #10,D0 (place la valeur 10 dans D0) :
Memoire.set(0, new Move(OperandeFactory.newInstance("#",10), OperandeFactory.newInstance(Registre.D0)));
// MOVE #0,100 (place la valeur 0 à l'adresse 100) :
Memoire.set(1, new Move(OperandeFactory.newInstance("#",0), OperandeFactory.newInstance(100)));
// ADD D0,100 (additionne la valeur de D0 à celle de l'adresse 100) :
Memoire.set(2, new Add(OperandeFactory.newInstance(Registre.D0), OperandeFactory.newInstance(100)));
// DEC D0 (soustrait 1 à D0) :
Memoire.set(3, new Dec(OperandeFactory.newInstance(Registre.D0)));
// JNE #2 (branchement à l'adresse 2 si l'opération précédente a donné une valeur nulle) :
Memoire.set(4, new Jne(OperandeFactory.newInstance("#", 2)));
// MOVE 100,A0 (place la valeur située à l'adresse 100 dans A0) :
Memoire.set(5, new Move(OperandeFactory.newInstance(100), OperandeFactory.newInstance(RegistreAdresse.A0)));
Memoire.execute(); // Execute le programme en mémoire à partir de l'adresse contenue dans PC (zéro)
```