Maitrise d'Informatique
Module POD

Programmation Objet et Distribution
Documents autorisés

Université Pierre et Marie Curie
Année 1997–1998
Durée 3 heures

Examen du 21 janvier 1998

Exercice 1 : Des points en programmation objet

Soient les déclarations suivantes de classes pour les points et les points colorés (ces exemples sont tirés des notes de cours) :

O'Caml

```
class point (x_init,y_init) =

  val mutable x = x_init
  val mutable y = y_init

  method get_x = x

  method get_y = y

  method moveto (a,b) =
    begin x <- a; y <- b end

  method rmoveto (dx,dy) =
    begin x <- x + dx; y <- y + dy end

  method affiche () =
    begin
      print_string "( ";
      print_int x;  print_string " , ";
      print_int y;  print_string ")";
     end

  method distance () =
    sqrt(float(x*x + y*y))

end;;

(* ---------------------------------- *)

class point_colore p c as self =
  inherit point p as super

  val c = c

  method get_c = c
  method affiche () =
  begin
      super#affiche();
      print_string (" de couleur "^
                      self#get_c);
  end
end;;
```

Java

```
class Point {

  int x, y;

  Point(int a, int b){x=a;y=b;}

  Point(){x=0;y=0;}

  void moveto (int a, int b){x=a;y=b;}

  void rmoveto (int dx, int dy){x+=dx;y+=dy;}

  void affiche(){
    System.out.println("(" + x + "," + y + ")");
  }

  double  distance(){
    return Math.sqrt(x*x+y*y);
  }

}

// ------------------------------------------

class PointColore extends Point {

  String c;

  PointColore(){c="blanc";}

  PointColore(int a, int b, String x){
    moveto(a,b);c=x;
  }

  void affiche(){
    System.out.println("(" + x + "," + y +
                       ") de couleur " + c);
  }

}
```

et soient les programmes suivants :

O'Caml

```
let p = new point(0,0);;
p#rmoveto(3,4);;
p#affiche();;

let pc = new point_colore (0,0) "vert";;
(pc#rmoveto(1,2))#affiche();;

let np = (pc : point_colore :> point);;

np#affiche();;

let npc = (np : point :> point_colore);;

npc#affiche();;
```

Java

```
Class Test {
  public static void main(String args[]) {
    Point p = new Point(0,0);
    p.rmoveto(3,4);
    p.affiche();

    PointColore pc = new PointColore();
    pc.rmoveto(1,2).affiche();

    Point np = (Point)pc;
    np.affiche();

    Point npc = (PointColore)np;
    npc.affiche();
}
```

1. Décrivez pour le programme O'Caml, entré au niveau de la boucle d'interaction, les réponses du système après chaque phrase O'Caml en les justifiant brièvement.

2. Indiquer pour le programme Java le résultat de l'exécution en le justifiant brièvement.

3. Ajouter aux deux programmes précédents une classe `ensemblePoint` correspondant à un ensemble de points qui pourra être représenté soit par une liste, soit par un vecteur.

4. Ecrire une méthode recherchant le point le plus proche de zéro. Que se passe-t-il si l'ensemble est vide?

Exercice 2 : Télédiscussion

Cet exercice consiste en la réalisation d'un client/serveur de discussion sur le réseau. Différents clients se connectent sur le serveur pour échanger des messages. Le serveur sera développé en O'caml et le client en Java.
Le serveur accepte des connexions de différents clients, se met en écoute de ceux-ci et dès qu'il reçoit un message d'un client, le renvoie à tous les autres y compris l'émetteur.
Les clients sont initialement en mode texte, effectuent une connexion, affichent à l'écran tout ce qui arrive, et dans le même temps attendent au clavier un message d'une ligne qui sera transmis au serveur. Pour différencier les différents auteurs de la discussion, le client envoie en début de ligne l'identifiant (par un pseudonyme) de l'auteur.

1. Que fait l'appel de la fonction suivante :

```
 let fd_serveur adresse port max =
  let fd = Unix.socket Unix.PF_INET Unix.SOCK_STREAM 0
  in begin
    Unix.bind fd (Unix.ADDR_INET (Unix.inet_addr_of_string adresse,port));
    Unix.listen fd max;
    fd end;;

fd_serveur "132.227.83.1" 5000 10;;
```

2. Décrire l'organisation d'une classe `serveur` pour implanter le serveur. Une instance de cette classe sera initialisée par une `socket` (retour de l'appel de la fonction précédente). La classe contiendra une liste des clients, possédera une méthode `accepte_connexion`, une méthode `lancement` intégrant une boucle sans fin acceptant de nouvelles connexions et traitant les messages en réception et en les envoyant à tous les clients.

3. Implantez cette classe et écrire le programme principal. Vous pouvez utiliser soit la lecture multiplexée (appel Unix `select`), soit des processus légers (*threads*), soit des `fork`.

4. Ecrire un client en Java se décomposant en deux processus légers (threads) : le premier attend une entrée au clavier et le deuxième écoute sur la socket et affiche à l'écran ce qu'il lit.

5. Modifier le client pour obtenir une *applet*. Celle-ci aura principalement deux composants graphiques : une `TextAera` pour l'affichage des messages reçus sans avoir à gérer le défilement vertical et un `TextField` pour la ligne en entrée. L'appel du constructeur `TextArea(10,40)`construit une zone de 10 lignes de 40 colonnes. L'envoi de la méthode `appendtext(String)` ajoute une chaîne au texte visible. Le texte de l'entrée sera envoyé au serveur suite à un retour chariot.

6. Décrivez le comportement du serveur (que vous avez écrit) quand un client disparaît.

Exercice 3 : Les tours de Hanoi en objets distribués

Cet exercice consiste à implanter le problème des tours de Hanoi sous deux formes : la première avec des objets locaux et la deuxième avec des objets distribués selon le RMI de Java. On rappelle brièvement le problème des tours de Hanoi.

On dispose de 3 piquets et d'un nombre fixe de disques de taille croissante. Ces disques sont sur le premier piquet. Le problème est de les faire passer sur le 3 ième piquet en respectant les règles suivantes :

- On ne peut déplacer qu'un seul disque à la fois;
- on ne peut pas poser un disque sur un disque plus petit;
- on ne peut déplacer que le disque du dessus.

Soit le programme principal suivant qui crée les trois piquets dont le premier contient 8 disques de taille décroissante.

```
class Go {
  public static void main (String args[]) {
    Piquet p0 = new Piquet();    // creation des 3 piquets
    Piquet p1 = new Piquet();
    Piquet p2 = new Piquet();
    int n = 8;
    for (int i=n; i>0; i--) {
      Disque d = new Disque(i);  // nouveau disque de taille i
      p0.recoit(d);              // envoi de ce disque sur le piquet P0
    }
    p0.deplaceVers(P1,1);        // deplacement d'un disque de P0 sur P1
    p0.deplaceVers(P2,n-1);      // deplacement de n-1 disque de P0 vers P2
    p0.deplaceVers(P2,1);        // deplacement d'un disque de P0 vers P2
  }
}
```

Les deux premières questions consistent à implanter les classes `Disque`, `PiquetVide` et `Piquet` pour que ce programme principal fonctionne.

1. Ecrire une classe `Disque` vérifiant l'interface suivante :

```
  interface DisqueInterface {

    void aller(Piquet ori, Piquet dest);
    int taille();
}
```

2. Ecrire une classe `Piquet` vérifiant l'interface suivante et déclare la classe `PiquetVide` :

```
  interface PiquetInterface {

    void recoit(Disque d);
    void deplaceVers(Piquet dest, int nb_disque);
    Disque enleveSommet() throws PiquetVide;
  }
```

3. Quelles modifications doit-on apporter au programme pour que le disque se déplaçant vérifie qu'il ne se pose pas sur un disque plus petit? Implanter les.
4. Définir l'interface PiquetRMI dans le but de créer des objets `Piquet` distribués.
5. Ecrire la classe PiquetDistribue implantant l'interface précédente.
6. Ecrire un programme Java enregistrant 3 piquets vides sur un serveur RMI.
7. Quelles sont les modifications à apporter à l'ensemble du programme local pour pouvoir écrire l'application en utilisant les piquets distants enregistrés sur un serveur RMI? Implanter les.