INF3110

Solutions OO-II, 2.11 – 6.11, 2009

Problem 1

Sketch of a solution.

It is required that the solution has a rotate for all classes Point, Circle and Rectangle (according to the text), but it is not said if there is an even more general superclass for all of these, or if Point is the most general class.

We therefore sketch two solutions. {...} indicates that here an implementation is required.

a1)

```
class Point {
  int x, y;
  void move(int dx, int dy){x +=dx; // note the error in Mitchell
                            y +=dy };
  void rotate(){};

};

class Circle extends Point {
  // the inherited point can be the center
  int radius;
  // no move() or rotate() are needed
};

class Rectangle extends Point {
  // the inherited point can be the center, although the original
  // did not have that
  Point tl, br;
  void rotate(){...};

};
```

a2)

Note that this will require more definitions of e.g. move and rotate, as the most general class will in fact only provide the signatures of the methods.

```
class Shape {
  void move(int dx, int dy){};
  void rotate(){};
};

class Point extends Shape {
  int x, y;
  void move(int dx, int dy){...};
  void rotate(){...};
};

class Circle extends Shape {
  Point c;
  int radius;
  void move(int dx, int dy){};
  void rotate(){...};
};

class Rectangle extends Shape {
  Point tl, br;
  void move(int dx, int dy){};
  void rotate(){...};
};
```

b)

Adding a triangle class in the Mitchel book solution would amount to

1. Changing shape_tag by adding a new tag value
2. Adding a new class Triangle (with methods like move, rotate,...)
3. Changing procedure rotate (new case)

Adding a triangle in the class/subclass solution above would amount

1. Adding a new class Triangle (with methods like move, rotate,...)

c) Almost no difference: the non-trivial code anyway has to be changed for both rectangles and triangles, but the Mitchel book solution will have a small benefit in that the change is only in one unit of the program (the rotate procedure), while the class/subclass solution will require changes to two units (classes rectangle and triangle).

Problem 2

a) The inherited setUL method would in an object of class ColorRect (type of ‘this’ in ‘this.ul’ would be ColorRect) assign a Point to a ColorPoint, and that is not allowed in Java.

b)

```
class Rect {
  virtual class TypeOfPoint < Point;
  TypeOfPoint ul;       // upper left corner
  TypeOfPoint lr;       // lower right corner

  void setUL(TypeOfPoint newUL){
    this.ul = newUL;
  };
};

class ColorRect extends Rect{
  class TypeOfPoint:: ColorPoint;
}
```

The implication would be a run-time type check.

c)

This would make the run-time type check explicit, but it will not change the type of the variables ul and lr.

Problem 3

a.1) Start out with Dequeue:

```
class Dequeue {
  Object[] r;
  void insert_front(Object o){...};
  void insert_rear(Object o) {...};
  void delete_front(){...};
  void delete_rear(){...};
}

class Stack {
  Dequeue d = new Dequeue();
  void push(Object o){d.insert_front(o);};
  void pop(){d.delete_front();}

};
class Queue {
  Dequeue d = new Dequeue();
  void insert_rear(){d.insert_rear(o);};
  void delete_rear(){d.delete_rear();};
}
```

When Dequeue is a superclass of Stack and Queue, the Dequeue properties become properties of Stack and Queue. The same can be obtained by ensuring that Stack and Queue do not assign a new value to d, and if one wants to be sure, d can be specified to be final.

a.2) Start out with Stack and Queue

and define Dequeue to share the same representation.

```
class Dequeue {
  Stack s = new Stack();
  Queue q = new Queue();
  Object[] r;
  void Dequeue(){s.r = r; q.r = r;}

  void insert_front(Object o){s.push(o);};
  void insert_rear(Object o){q.insert_rear(o)};
  void delete_front(){s.pop()};
  void delete_rear(){q.delete_rear()};
}
```

b)

In order to obtain this we will have to define e.g. interfaces Stack and Queue, together with classes that implement these interfaces StackImpl and QueueImpl.

```
Interface Stack {
  void push(Object o);
  void pop()
};
Interface Queue {
  void insert_rear();
  void delete_rear();
};
```

Then define Dequeue like this:

```
class Dequeue implements Stack, Queue {
  Stack s = new StackImpl();
  Queue q = new QueueImpl();
  Object[] r;
  void Dequeue(){s.r = r; q.r = r;}

  void insert_front(Object o){s.push(o);};
  void insert_rear(Object o){q.insert_rear(o)};
  void delete_front(){s.pop()};
  void delete_rear(){q.delete_rear()};
}
```

Problem 4

There would be no compile time type error, but the assignment in line (2) will cause a run time type error. Line (3) will therefore never be reached. Had Java had static type check in this case, then a type error would have been given at line (1).

Problem 5

```
void print(Collection<? extends Reservation > reservations) { ... }
```

Problem 6

No, this is not an example of structural (sub)typing. In Java it is still so that you have to type references with interface Cowboy or Shape (or Lucky Luke) in order to call draw, while with structural (sub)typing the class name (or interface name) plays no role.

Problem 7

```
class FlightType {
  City source, destination;
  TimeOfDay departureTime, arrivalTime;
  Duration flyingTime;

  class Flight {
    Seat[] seats;
    TimeOfDay actualDepartureTime, actualArrivalTime;
    Duration ActualFlyingTime;

    Duration departureDelay(){
      return actualDepartureTime – departureTime;
    };
    Duration arrivalDelay(){
      return actualArrivalTime – arrivalTime;
    };

  }
};

class TimeTable {
  FlightType SK451 = new FlightType(Oslo, Copenhagen, 8.20);
                     // provided a corresponding constructor
};

TimeTableWithReservations ttwr = new TimeTable {
  SK451.Flight[365] SK451flights;
  ...
}
```

Problem 8

a)

```
c.equals(c)     //1     equals 1       C_equals 1
c.equals(c’)    //2     equals 1       C_equals 1
c.equals(sc)    //3     equals 1       C_equals 1

c’.equals(c)    //4     equals 1       SC_equals 1
c’.equals(c’)   //5     equals 1       SC_equals 1
c’.equals(sc)   //6     equals 1       SC_equals 1

sc.equals(c)    //7     equals 1       SC_equals 1
sc.equals(c’)   //8     equals 1       SC_equals 1
sc.equals(sc)   //9     equals 2
```

// 7 is equals 1, because SC has an inherited (C,C)-typed equals. equals 2 requires a SC typed parameter. Both c and c’ may at runtime denote a SC-object, but it is only safe to assume that they denote C-objects

// 8 is equals 1, with the same reasoning as for //7

b)

```
c.equals(c)     //1     equals 1       C_equals 1
c.equals(c’)    //2     equals 1       C_equals 1
c.equals(sc)    //3     equals 1       C_equals 1

c’.equals(c)    //4     equals 1       C_equals 1
c’.equals(c’)   //5     equals 1       C_equals 1
c’.equals(sc)   //6     equals 1       C_equals 1

sc.equals(c)    //7     equals 1       C_equals 1
sc.equals(c’)   //8     equals 1       C_equals 1
```