Object Oriented Programming (Sheet #2)

Marius Gavrilescu

1. These variables are used to enable Ewoks to remember the column we started a sequence of UP and DOWN motions from. The first UP or DOWN move in such a sequence will target the column we started from, and all other moves in the sequence will share the same goal column. As an example, if we're at (5,5), the next line has 3 characters, and the one after has 10 characters, then typing DOWN twice will get us to (6,3) then (7,5). Using two variables instead of a single goal variable makes the code simpler – with a single variable the `obey` method would have to inspect the passed command to decide whether to reset the goal.

2. Simplest change is to call `display.chooseOrigin()` at the end of the `obey` method in `Editor`.

   The second change can be implemented by modifying the `checkScroll` method as follows:

```
private def checkScroll() {
    if (row < origin)
        origin = row;
    else if(row >= origin + LINES)
        origin = row - LINES + 1;

    origin = Math.max(Math.min(origin, ed.numLines - 1), 0);
}
```

3. When `x` is typed, `perform` is called, which calls `obey`. `obey` calls the command (`insertCommand`) which inserts the character. Finally, the change is appended to the history stack.

   When `^Z` is typed, `perform` is called, which calls `obey`. `obey` calls the command (`undo`), which pops the history stack and executes the `undo` method of the popped history item. Finally, the `undo` change is appended to the history stack.

   The changes returned by commands are wrapped using the `wrapChange` method in EdBuffer to make them remember the position of the point and restore it when the command is undone. Hence after an undo the cursor will be moved to where it was before the command was typed.

4. The abstract state of `PlaneText` is a superset of the abstract state of `Text`.

```
STATE: text : seq char

charAt(pos : Int) = c
PRE: 0 <= pos < (length pos) - 1
POST: c = text !! pos

clear
POST: text = []

...

numLines = n : Int
POST: length(lines text)

getLineLength(n : Int) = len
PRE:  n < numLines
POST: len = length((lines text) !! n)

getRow(pos : Int) = row
PRE:  pos < length text
POST: sum (map length (take (row - 1) (lines text)) <=
      pos < sum (map length (take row (lines text)))
```

5.
```
class MaxIntSet extends IntSet {
    private var max_ = 0;

    override def insert(x : Int) = {
        super(x);
        if(x > max_)
            max_ = x;
    }

    def max : Int = max_
}
```

We could add a method `insertArray(x : Array[Int])` to `IntSet`. Here are two implementations of this method:

```
def insertArrayBad(xs : Array[Int]) = {
    for(x <- xs)
        if(find(x) < 0) {
            assert(size < MAX);
            elems(size) = x;
            size = size + 1;
        }
}

def insertArrayGood(xs : Array[Int]) = {
    xs.map(insert);
}
```

The first method would need to be overriden in the subclass, while the second one need not be overriden.

6. (a) `PlaneText` could wrap a `Text` object and provide (alongside its own methods) some methods with the same signature as `Text` that simply call the methods of the wrapped object.

   (b) We could make a trait `CharSequence` whose signature is the intersection of the signatures of `Text` and `PlaneText`, and then make `doSomething` take a `CharSequence` parameter.

   (c) This change makes the fragile base class problem better, because the calls to `Text` methods in `PlaneText` are explicit. However, this makes the code uglier and longer, because methods common to both classes need to be explicitly forwarded, and inheriting from `PlaneText` becomes more difficult.

7. (a)
```
class ArrayPointSet extends PointSet {
    private val MAX = 10000;
    private var elems = new Array[Point](MAX);
    private var size = 0;

    def isEmpty : Boolean = size == 0

    def contains(goal : Point) : Boolean = {
        for (x <- elems) {
            if(goal.equals(x))
                return true;
        }
        return false;
    }

    def insert(x : Point) = {
        if(!contains(x)) {
            assume (size < MAX);
            elems(size) = x;
            size = size + 1;
        }
    }

    def delete(x : Point) = {
        if(contains(x)) {
            elems = elems.filterNot(x.equals(_));
            size = size - 1;
        }
    }
}
```

(b)
```
class Point(val x : Int, val y : Int) {
    override def equals(other : Any) : Boolean = {
        other match {
            case o : Point => o.x == x && o.y == y
            case _ => false
        }
    }

    override def hashCode() : Int = x * 1000000 + y
}
```