Exercise 4

Inheritance, and more Inheritance

October 22nd, 2010

Unless otherwise stated, assume we are working with a language (such as Java) in which method dispatch is dynamic for the type of the receiver and static for the type of the arguments.

1. Consider a class `Matrix` to implement matrices with integer values. A simple implementation would be to store a (private) 2-dimensional array of integers, and provide methods such as:

```
void set(int i, int j, int value);
int get(int i, int j);
Matrix add(Matrix m);
Matrix multiply(Matrix m);
```

   A *sparse matrix* is a matrix which contains mainly zeros. When such matrices are large it can be that an alternative representation of the matrix, which only stores the locations and values of non-zero entries, can provide much more efficient implementations for common expensive operations such as addition and multiplication with other sparse matrices. If a sparse matrix is to be added or multiplied with a standard matrix, it also is possible to define an implementation which is more efficient that the standard one (but not as good as for two sparse matrices).

   Consider writing a new class `SparseMatrix` to implement sparse matrices, with the similar methods available to those for `Matrix`.

   - Is it likely that there will be scope for reusing code from the class `Matrix`?
   - Does it seem that `SparseMatrix` can (and should?) be a behavioural subtype of `Matrix`?
   - What would be the implications of making `SparseMatrix` a subclass of `Matrix`?
   - What alternative ways are there of expressing the relationship between the classes?

2. Suppose from now on that `SparseMatrix` is to be implemented as a *subclass* of `Matrix`. Assume (reasonably!) that the two classes will use different internal representations (fields). If you sketch a possible implementation, it might help.

   - What would happen if client code could access the fields? e.g., suppose `entries` is the 2-d array field of `Matrix`, and `m` is a local `Matrix` variable, and consider:

```
m.entries[i][j] = 4;
if(m.get(i,j)!= 4) { // crash }
```

     What can go wrong here? To what extent are these problems avoided by making the fields private?
   - What might go wrong (or at least give unexpected behavior) if we do not override all of the methods of `Matrix` when writing `SparseMatrix`?

- What difficulties might occur if we wanted to add extra methods to `Matrix` later?

3. Consider the `add` and `multiply` methods. These operations should be implemented differently depending on the (runtime) types of both the receiver and the argument the methods are applied to, i.e., we need binary methods to handle this situation.
   - Sketch how to implement the `add` method (the details of how to perform the actual addition are not essential) in both `Matrix` and `SparseMatrix` based on each of the following approaches to binary methods:
     - i. Explicit type tests to check the runtime type of the argument
     - ii. Double invocation (Visitor pattern)
     - iii. Multiple dispatch
   - Which approach seems most elegant/appropriate for this example?
   - Suppose that, for reasons of compatibility with existing code, we are not allowed to change the existing definition of the `Matrix` class. For each of the three approaches above, consider how feasible it is to adapt to this constraint. Does your answer depend on how the existing `Matrix` class is actually defined?
4. Suppose we introduce a further class `ZeroMatrix` which is a subclass of `SparseMatrix`, representing the zero matrix (in particular, all instances of this class should be indistinguishable in behaviour). We observe that we can improve efficiency still further by implementing simplified versions of `add` and `multiply` when zero matrices are involved.

   We observe that we can overload the definition of `add` in `Matrix` to treat the special case of a `ZeroMatrix` argument with a simplified implementation.
   - What should the result of a call to `add()` be, when the argument is a `ZeroMatrix`? What happens if we simply overload the definition in `Matrix`?
   - Symmetrically, when the receiver of a call to `add` is a `ZeroMatrix` we can use a more efficient implementation. Sketch how to extend each of the three approaches from the previous question for implementing `add` as a binary method.
   - In the case of multiple dispatch, there is an additional requirement – what is it? Is this extra requirement reasonable?
   - Which of the approaches seems most elegant/appropriate for this example now?
   - Suppose that, for reasons of compatibility with existing code, we are not allowed to change the existing definitions of either the `Matrix` or `SparseMatrix` classes. By comparing with your sketches for the previous question, consider how feasible it is to adapt to this constraint.