Concepts of Object-Oriented Programming

Exercise 5

Binary Methods and Multiple Inheritance

October 28th, 2011

In-class Assessment: One or more questions from this sheet will be used for the in-class assessment. No notes are allowed during the assessment.

1) Consider a class `Matrix` to implement matrices with integer values, and its subclass `SparseMatrix` that implements sparse matrices. Consider now 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.
   a. 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
   b. Which approach seems most elegant/appropriate for this example?
   c. 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?

2) 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.
   a. 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`?
   b. 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.
   c. In the case of multiple dispatch, there is an additional requirement – what is it? Is this extra requirement reasonable?
   d. Which of the approaches seems most elegant/appropriate for this example now?
   e. 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.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

3) Consider the following C++ code:

```
class Person
{
    Person *spouse;
    string name;

public:
    Person (string n) { name = n; spouse = NULL; }

    bool marry (Person *p)
    {
        if (p == this) return false;
        spouse = p;
        if (p) p->spouse = this;
        return true;
    }

    Person *getSpouse () { return spouse; }
    string getName () { return name; }
 };
```

The method `marry` is supposed to ensure that a person cannot marry itself. Without changing the code above, create a new object that belongs to a subclass of `Person` and marry it with itself. Hint: use multiple inheritance. Explain exactly what happens.

4) Consider the following C++ code:

```
class Person
{
    bool likesDiamonds;

public:
    Person (bool l) { likesDiamonds = l; }
};

class Programmer : virtual public Person
{
public: Programmer () : Person (false) {}
    // diamonds are a programmer's worst enemy
};
```

It is expected that `!likesDiamonds` is an invariant in class `Programmer`. Use virtual inheritance to break this invariant, without altering the above code.

5) Write three classes

- A normal queue class `Queue`
- A subclass of `Queue` that also stores (and allows clients to retrieve) the current sum of all items in the queue, using the `enqueue` and `dequeue` methods

- A subclass of `Queue` that also stores (and allows clients to retrieve) the current product of all items in the queue, using the `enqueue` and `dequeue` methods

We now want a class that supports both functionalities.

- Suppose that we want to use multiple inheritance to do that. We want to override the `enqueue` and `dequeue` methods of the new class, such that the new methods call the `enqueue` and `dequeue` methods of both the old classes. Are there any problems with this approach?
- How do we attack the problem using traits? Does this fix the above-mentioned problems? Are there any new problems with this approach?