Concepts of Object-Oriented Programming
AS 2013

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 6

Multiple Inheritance, Multiple Dispatch and Traits

November 1, 2013

Task 1

Consider an interface `Matrix` to represent integer valued matrices, which has two implementations- `GeneralMatrix` that represents matrices as 2 dimensional arrays, and `SparseMatrix` that represents only non-zero elements with their position. The idea is that matrices with few non-zero elements can implement operations more efficiently. 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 - they are binary methods. In this question assume that matrices are immutable.

1. Sketch how to implement the add method (the details of how to perform the actual addition are not essential), in both `GeneraMatrix` and `SparseMatrix` based on each of the following approaches to binary methods:

   (a) Explicit type tests to check the runtime type of the argument

   (b) Double invocation (Visitor pattern)

   (c) Multiple dispatch

   Detail how implementing `multiply` differs from implementing `add`.

2. Which approach seems most elegant/appropriate for this example?

3. If we were to add a third class, `ZeroMatrix`, which represents a matrix where all values are zero, how much would we have to change in each approach? Does any of them excel or suffer in particular?

4. We are given the following `Matrix` interface:

```
interface Matrix{
   int get(int i, int j);
   Matrix add(Matrix m);
   Matrix multiply(Matrix m);
}
```

   For reasons of compatibility with existing code, we are not allowed to change the existing definition of the `Matrix` interface. Which of the three approaches above can be adapted to this constraint?

Task 2

Now assume that we want to reduce code duplication by pooling all the common implementation of matrices into an abstract `Matrix` class instead of an interface. This class would be a superclass for `GeneralMatrix` and `SparseMatrix`

1. Sketch how to implement the add method in the `Matrix` class. Does this require any other methods or changes in the other classes?
2. In each of the three approaches detailed above, can we now omit some code? What would be the consequences?
3. How would you add the `ZeroMatrix` now? Is it easier or harder?

Task 3

(a) Suppose that Java allowed interface methods to have a default implementation directly in the interface.

- What are some advantages of this feature?
- What could be some problems with this feature? How can they be resolved?
- What problems of C++ virtual inheritance are avoided by this new design for Java interfaces?

(b) Now suppose that, in addition to method implementations, Java also allowed interfaces to define fields. Interfaces would not have constructors and interface fields would always be initialized with a default value.

- What are some advantages of this feature?
- Given the restrictions above, are there any problems left with such an implementation of multiple inheritance? If so what are they? Propose a solution for each problem you have identified.

Task 4

Consider the following Scala code:

```
class Cell
{
   private var x:int = 0
   def get() = { x }
   def set(i:int) = { x=i }
}

trait Doubling extends Cell
{
   override def set(i:int) = { super.set(2*i) }
}

trait Incrementing extends Cell
{
      override def set(i:int) = { super.set(i+1) }
}
```

- What is the difference between the following objects?

```
val a = new Cell
val b = new Cell with Incrementing
val c = new Cell with Incrementing with Doubling
val d = new Cell with Doubling with Incrementing
```

- We use the following code to implement a cell that stores the argument of the set method multiplied by four:

```
val e = new Cell with Doubling with Doubling
```

Why doesn't it work? What does it do? How can we make it work?

- Find a modularity problem in the above, or a similar, situation. Hint: a client that gets given a class `C` does not necessarily know if a trait `T` has been mixed in that class.

Task 5

Assume all the definitions of the previous exercise. Assume that `Cell` has the invariant that `x` is always even. Furthermore, consider a Scala method

```
foo(x: Cell with Doubling with Incrementing) {...}
```

- During the execution of `foo`, if we assume that all subclasses of `Cell` respect behavioral subtyping, then are we allowed to conclude that `x.get()` always returns an even number?
- We propose the following solution to support traits together with behavioral subtyping: Assume `C` is a class with specification `S`. Each time we create a new trait `T` that extends `C`, we must ensure that `C with T` also satisfies `S`.
  Show a counterexample that demonstrates that this approach does not work.

Task 6

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?