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?

Solution

1. (a) Typecase:

```
class GeneralMatrix{
   Matrix add(Matrix m) {
      if(m instanceof SparseMatrix) {
         return addGeneralSparse(m);
```

```
        } else if (m instanceof GeneralMatrix{
            return addGeneralGeneral(m);
        }else
            crash();
    }
    ...
}

class SparseMatrix{
    Matrix add(Matrix m) {
        if(m instanceof SparseMatrix) {
            return addSparseSparse(m);
        } else if (m instanceof GeneralMatrix){
            return addSparseGeneral(m);
        }else
            crash();
    }
    ...
}
```

(b) Visitor: In the `Matrix` interface:

```
interface Matrix{
    Matrix add(Matrix m);
    Matrix addGeneral(GeneralMatrix m);
    Matrix addSparse(SparseMatrix m);
    ...
}

class GeneralMatrix{
    Matrix add(Matrix m) {
        return m.addGeneral(this);
    }
    GeneralMatrix addGeneral(GeneralMatrix m) {
        return addGeneralGeneral(m);
    }
    GeneralMatrix addSparse(SparseMatrix m) {
        return addGeneralSparse(m);
    }
    ...
}

class SparseMatrix{
    Matrix add(Matrix m) {
        return m.addSparse(this);
    }
    GeneralMatrix addGeneral(GeneralMatrix m) {
        return m.addSparse(this); //for non commutative multiply
            this would be different
    }
    SparseMatrix addSparse(SparseMatrix m) {
        return addSparseSparse(this);
    }
    ...
}
```

Note that we have some freedom in choosing the return types of the specialized operations.

(c) For the multiple dispatch approach:

```
class GeneralMatrix{
    ...
```

```
    GeneralMatrix add(GeneralMatrix m) {
        return addGeneralGeneral(m);
    }
    GeneralMatrix add(SparseMatrix m) {
        return addGeneralSparse(m);
    }
}
class SparseMatrix
{
    ...
    GeneralMatrix add(Matrix m) {
        return m.addSparse(this); //different for multiply
    }
    SparseMatrix add(SparseMatrix m) {
        return addSparseSparse(m);
    }
}
```

2. The last approach is probably the simplest and most intuitive, but requires language support.

3. The changes would be as follows:

   (a) For the downcast approach:

```
class GeneralMatrix{
    Matrix add(Matrix m) {
        ...
        else if (m instanceof ZeroMatrix){
            return this;
        ...
    }
    ...
}
class SparseMatrix{
    Matrix add(Matrix m) {
        ...
        else if (m instanceof ZeroMatrix){
            return this;
        ...
    }
}

class ZeroMatrix{
    Matrix add(Matrix m) {
        return m;
    }
}
```

   (b) For the visitor approach

```
interface Matrix{
    Matrix addZero(ZeroMatrix m);
    ...
}

class GeneralMatrix{
    GeneralMatrix addZero(ZeroMatrix m) {
        return this;
    }
    ...
}
```

```
class SparseMatrix{
   SparseMatrix addZero(ZeroMatrix m) {
      return this;
   }
   ...
}

class ZeroMatrix{
   Matrix add(Matrix m) {
      return m;
   }
   ...
}
```

(c) For the multiple dispatch approach:

```
class GeneralMatrix{
   GeneralMatrix add(ZeroMatrix m) {
      return this;
   }
   ...
}

class SparseMatrix{
   SparseMatrix add(ZeroMatrix m) {
      return this;
   }
   ...
}

class ZeroMatrix{
   Matrix add(Matrix m) {
      return m;
   }
   ZeroMatrix add(ZeroMatrix m) {
      return m;
   }
   SparseMatrix add(SparseMatrix m) {
      return m;
   }
   GeneralMatrix add(GeneralMatrix m) {
      return m;
   }
   ...
}
```

The downcast approach has the problem that the compiler can not check that we have implemented all the necessary options, so the program may fail at runtime if we have forgotten one of the additions. The multiple dispatch approach suffers because we have to add much more code that is essentially redundant.

4. The first and third approaches are not affected by this change. The second approach is not feasible without modifying the interface, as it requires adding virtual methods at the top level. One might consider a workaround where we create an additional interface `MatrixVisitor` that extends the `Matrix` interface and includes the methods necessary for the visitor pattern. The classes `GeneralMatrix`, `SparseMatrix`, and `ZeroMatrix` could all use this second interface and implement the required methods for the visitor pattern. However there is still an issue: client code that only knows about the `Matrix` interface, might try to call the `add` or `multiply` methods with some new kind of matrix that implements the `Matrix` interface, but not the `MatrixVisitor` one, since the client

didn't know about it. This will break the suggested workaround.

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?

Solution

1. We would need to add the equivalent of a get method to the abstract class, even if it is only protected. We would also need a way to construct a matrix, here we have added a constructor for `GeneralMatrix` that accepts a 2 dimensional array:

```
abstract class Matrix
{
   int size();

   Matrix add(Matrix m) {
      int[][] m = new int[size()][size()];
      for (var i=0; i<size; i++)
         for (var j=0; j<size; j++)
            m[i][j] = this.get(i,j)+m.get(i,j));
      return new GeneralMatrix(m);
   }

   public abstract int get(int i, int j);
}
class GeneralMatrix
{
   GeneralMatrix(int[][] m){ ... }
}
```

   The abstract class would have to define the `get` method. The subclasses would each have to implement the `get` method (although reasonably they would have it anyway) We must have a way of constructing a general matrix with given values - although that would usually be the case anyway.

2. We can now implement as follows:

   (a) For the downcast approach: We can now modify the type case statements to be safe In the `GeneralMatrix` class:

```
Matrix add(Matrix m) {
   if(m instanceof SparseMatrix) {
      return addGeneralSparse(m);
   } else if (m instanceof GeneralMatrix{
      return addGeneralGeneral(m);
   }else
      return super.add(m);
}
```

   And analogously in the `SparseMatrix` class. Here omitting any case but the last would just lead to a less efficient implementation.

(b) The visitor approach: Here we may (but do not have to) implement the specific visitors in the base class, and then sub classes do not have to (but can) specialize them - this gives us the flexibility to omit some cases that would not benefit us, but has the downside that if we forget a case in the subclass the compiler will not warn us.

```
class Matrix{
   ...
   Matrix addGeneral(GeneralMatrix m)
   {
      return add(m); //by default do full addition
   }
   Matrix addSparse (SparseMatrix m)
   {
      ... //add sparse matrix to a full matrix
   }

}

class SparseMatrix{
   ...
   Matrix addSparse (SparseMatrix m)
   {
      ... //add two sparse matrices
   }
   Matrix addGeneral (GeneralMatrix m)
   {
      return m.addSparse(this); //Will need to be implemented
         differently for multiply as it is not commutative
   }
}
```

(c) The multiple dispatch approach:

```
class GeneralMatrix{
   Matrix add(SparseMatrix m)
   {
      ... //add a sparse matrix to a full matrix
   }

}

class SparseMatrix{
   ...
   Matrix add(SparseMatrix m)
   {
      ... //add two sparse matrices
   }
   Matrix add(GeneralMatrix m)
   {
      return m.add(this); //Will need to be implemented
         differently for multiply as it is not commutative
   }
}
```

Note that if we were to override `add(Matrix m)` in any subclass then we would have potential ambiguities and need to implement many more cases.

3. If we now wanted to add the `ZeroMatrix` class, we would do as follows:

(a) For the downcast approach: We could add the downcast to `ZeroMatrix` at the top level:

```
abstract class Matrix{
    ...
    Matrix add(Matrix m) {
        if (m instanceof ZeroMatrix) {
            return this;
        }else
        ...
    }
}
class ZeroMatrix{
    ...
    Matrix add(Matrix m) {
        return m;
}
```

However this relies on the subclasses calling super by default - and we have seen that this can cause subtle bugs - we do not have compiler support to tell us if we had missed a case.

(b) The visitor approach: Here we can implement the following visitor at the top level and at `ZeroMatrix`:

```
abstract class Matrix{
    ...
    Matrix addZero(ZeroMatrix m) {
        return this;
    }
}
class ZeroMatrix{
    ...
    Matrix add(Matrix m) {
        return m;
    }
    Matrix addGeneral(Matrix m) {
        return m;
    }
    Matrix addSparse(Matrix m) {
        return m;
    }
}
```

(c) The multiple dispatch approach: Here we could do:

```
abstract class Matrix{
    ...
    Matrix add(ZeroMatrix m) {
        return this;
    }
}
class ZeroMatrix{
    ...
    Matrix add(Matrix m) {
        return m;
    }
    Matrix add(ZeroMatrix m) { //this is needed to disambiguate
        the other two for add(ZeroMatrix,ZeroMatrix)
        return m;
    }
}
```

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.

Solution

(a) Default method implementations of interface methods is an upcoming feature in Java 8.

- An advantage is obviously that default implementations can be reused in multiple classes. Another advantage (and the main reason this feature is added to Java) is that default method implementations will allow interface evolution. Without a default implementation, adding new methods to an interface would break all existing classes that implement that interface, since they do not contain an implementation for the new methods. The new features removes this problem.
- A problem could be inheriting two default implementations of the same method from unrelated interfaces. In that case we will have to either choose which implementation we prefer or write a new implementation that overrides both.

  Another issue is that interfaces can now suffer from the fragile base class problem. Compared to the usual issue with normal Java classes, this is even more dangerous for interfaces with default methods, since these methods will mostly call other methods of the interface which are overriden in implementing classes. A very restrictive solution here could be to prohibit calls to other methods of the interface, within the implementation of default methods. Alternatively we can "deal" with the problem just like Java deals with the issue in classes - do nothing and rely on the programmer to be careful.
- We still avoid problems with correct initialization of fields of super types, since only one super type (the extended class) can have fields, and we can directly call its constructor. Furthermore there are no problems with field duplication as in non-virtual C++ inheritance.

(b) This makes multiple inheritance in Java very similar to C++.

- An advantage is that we can also reuse fields. This will enable more methods with default implementations in interfaces which could increase code reuse and reduce the effort required to create new classes.
- These restrictions are somewhat similar to Scala traits, which also do not have specialized constructors (only a default constructor). In this way we manage to avoid problems with initialization order. However a problem that still remains is:

how many copies of a field exist? In particular:

- A class might implement the same interface multiple times (for example by implementing two different interfaces that are a subtype of the same interface). A solution here might be to only have a single copy of the field (as in C++ virtual inheritance).
- A class might implement two different interfaces that both declare the same field. Here we could either restrict interfaces to defining only private fields (which are invisible to the implementor), or we could require some disambiguation syntax when accessing fields, similar to C++ or the proposed syntax for disambiguating conflicting default methods in Java 8.

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.

Solution

- Object `a` behaves like a normal cell. Object `b` is also a cell, but it increases the stored value by 1. The interesting difference is between `c` and `d`. They are both cells. They have mixed in exactly the same traits. However, calling `set(i)` has a different effect on them: it stores `2i+1` to the first one and `2(i+1)` to the second one.
- Trait `Doubling` will not get mixed in twice, as perhaps the programmer would expect. Scala rejects this statically.

The problem can be bypassed in an ugly way, by creating a new trait `Doubling2` that behaves exactly like `Doubling` and then introducing `e = new Cell with Doubling with Doubling2`. Here is our first try:

```
trait Doubling2 extends Doubling
val e = new Cell with Doubling with Doubling2
```

The code passes through, but dynamically `e` behaves as if it were a `Cell with Doubling` . Scala lets the code go through, because `Doubling2` may introduce new functionalities, but refuses to include `Doubling` twice in the linearization.

Our last try, the ugliest of all, but the one which will finally work, is to create a whole new trait from scratch, reusing nothing:

```
trait Doubling3 extends Cell
{
   override def set(i:int) = { super.set(2*i) }
}
val e = new Cell with Doubling with Doubling3
```

And now `e.set` quadruples its argument as expected.

- A problem is that a method that accepts `Cell with Doubling with Incrementing` as an argument could also be passed a class of the type `Cell with Incrementing with Doubling` - so what it can actually assume about its inputs is less than would be expected.

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.

Solution

- No. The dynamic type of `x` can be mixed with traits that break the invariant. Even if we suppose that the only traits that can extend `Cell` are `Incrementing` and `Doubling`, this is not enough to enforce behavioral subtyping. In particular, an object of type `Cell with Incrementing with Doubling` can be still passed as argument to method `foo` in this restricted context, and this would break the invariant.
- Consider the following example:

```
class C
{
   var x:int;
   def foo() = {} //ensures true
}
trait T1 extends C
{
   override foo() = { x=x+1 } //ensures x>old(x)
}
```

```
trait T2 extends C
{
   override foo() = { x=x-1 } //ensures x<old(x)
}
```

Both `C with T1` and `C with T2` are behavioral subtypes of `C`. But `C with T1 with T2` is not a subtype of `C with T1`.

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?

Solution

Here are the three requested classes:

```
class Queue
{
   int[] contents;
   int size;

public:
   Queue() { contents = new int[100]; size = 0; }
   void enqueue(int x) {...}
   int dequeue() {...}
   int getSize() { return size; }
};

class SumQueue : virtual public Queue
{
   int sum;

public:
   SumQueue() : Queue() { sum = 0; }

   void enqueue(int x)
   {
      sum+=x;
      Queue::enqueue(x);
   }

   int dequeue()
   {
      int r = Queue::dequeue();
      sum-=r;
```

```
        return r;
    }

    int getSum() { return sum; }
};

class ProductQueue : virtual public Queue {...};

class SuperQueue : public ProductQueue, SumQueue
{
public:
    SuperQueue()
        : public Queue(), ProductQueue(), SumQueue() {}

    void enqueue(int x)
    {
        ProductQueue::enqueue(x);
        SumQueue::enqueue(x);
    }

    int dequeue()
    {
        int r = ProductQueue::dequeue();
        SumQueue::dequeue();
        return r;
    }
};
```

One obvious problem is that the `enqueue` and `dequeue` methods of the superclass are called twice. An item is enqueued and dequeued twice. Interestingly, this behaves exactly like a queue, but the capacity is half of the capacity of the original and the `getSize` method reports the correct size multiplied by 2.

We can use traits and linearization to ensure that the `enqueue`/`dequeue` methods are called only once. Here is the relevant Scala code:

```
class Queue
{
    ...
    def enqueue(x:int) = {...}
    def dequeue():int = {...}
}

trait Sum extends Queue
{
    var sum:int = 0
    override def enqueue(x:int) =
        { sum+=x; super.enqueue(x) }
    override def dequeue():int =
        { var x = super.dequeue; sum=sum-x; x }
}

trait Prod extends Queue
{
    var count:int = 1
    override def enqueue(x:int) =
        { prod*=x; super.enqueue(x) }
    override def dequeue():int =
        { var x = super.dequeue; prod=prod/x; x }
        // side remark: this assumes no zeros in the queue!
}
```

Now, an object of `Queue with Sum with Prod` has both functionalities, but calls each under-

lying `enqueue`/`dequeue` method only once. The problems of the multiple inheritance solution do not appear here.