ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 5

Inheritance

October 24, 2014

Task 1

Consider the following Java class, representing a singly linked list:

```
package sll;

public class SLL{
   public SLL() {}

   public void prepend(int x)
   {
      SLLNode n = new SLLNode(x);
      n.next = head;
      head = n;
   }
   public SLLNode getFirst(){ return head; }
   public SLLNode findFirstGreaterThan(int x)
   {
      SLLNode n = head;
      while (n!=null)
         if (n.getVal()>x)
            return n;
         else
            n=n.next;
      return null;
   }
   protected SLLNode head = null;
}

public class SLLNode{
   protected SLLNode(int x){value = x;}
   public int getVal(){ return value; }
   public SLLNode getNext(){ return next; }

   private int value;
   protected SLLNode next = null;
}
```

And the following subclass representing a doubly linked list:

```
package dll;
class DLL extends SLL{
   public DLL() {...}
   public DLLNode getFirst(){ ... }
   public DLLNode findFirstGreaterThan(int x){...}
   ...
}
class DLLNode extends SLLNode{
   protected DLLNode(int x){...}
```

```
    public DLLNode getNext(){...}
    public DLLNode getPrev(){...}
    private DLLNode prev;
}
```

- If we wanted to implement the `DLL` class as a subclass of the `SLL` class, with the purpose of reusing code and without modifying anything in the `sll` package, which method we could definitely not reuse and have to duplicate? Which language construct is the reason for that?
- How can we modify the implementation of the `sll` package in order to allow reuse of this method, without changing the public interface and client visible behaviour for either the `SLL` or `SLLNode` classes (the protected and private parts can be changed as long as these changes are not visible to clients of the classes that are outside the `sll` package). The `sll` package is not allowed to refer to the `dll` package or its contents in any way, and must not implement a doubly linked list itself. Write the code for all the changes that are necessary to make this method reusable.

Solution

- The method we cannot reuse is `prepend` as it includes a direct constructor call to `SLLNode` and the doubly linked list is made up of `DLLNodes`.
- We would remove the node creation into another method, which we can then override:

```
package sll;

public class SLL{
    ...
    protected SLLNode makeNode(int x){ return new SLLNode(x); }
    public void prepend(int x)
    {
        SLLNode n = makeNode(x);
        n.setNext(head);
        head = n;
    }
    ...
}
public class SLLNode{
    ...
    protected void setNext(SLLNode n){ next = n; }
}
....

package dll;
class DLL extends SLL{
    ...
    protected DLLNode makeNode(int x){ return new DLLNode(x); }
}
public class DLLNode extends SLLNode{
    ...
    protected void setNext(SLLNode n)
    {
        super.setNext(n);
        if (n!=null)
            ((DLLNode)n).prev=this;
    }
}
```

Task 2

Consider the following classes in a Java-like language:

```
public class TotalMap {
   ///ensures forall z : lookup(z)==z
   public TotalMap(){...}

   ///ensures result!=null
   public Integer lookup(Integer x){...}

   ///requires x!=null ∧ y!=null
   ///ensures lookup(x)==y
   ///ensures forall z : x!=z => lookup(z)==old(lookup(z))
   public void set(Integer x,Integer y){...}

   ///ensures result==true
   public boolean has(Integer x){...}

   ...
}

public class PartialMap {
   ///ensures forall z : lookup(z)==null
   ///ensures forall z : !has(z)
   public PartialMap(){...}

   ///ensures has(x) <==> result!=null
   public Integer lookup(Integer x){...}

   ///requires x!=null ∧ y!=null
   ///ensures lookup(x)==y
   ///ensures forall z : x!=z => lookup(z)==old(lookup(z))
   ///ensures forall z : has(z) <==> old(has(z)) || z==x
   public void set(Integer x,Integer y){...}

   public boolean has(Integer x){...}

   ...
}

public class Set{
   ///ensures forall z : !has(z)
   public Set(){...}
   public boolean has(Integer x){...}

   ///ensures forall z : has(z) <==> (old(has(z)) || z==x)
   public void include(Integer x){...}

   ///ensures forall z : has(z) <==> (old(has(z)) && !z==x)
   public void exclude(Integer x){...}

   ...
}
```

The complete public interfaces of the classes are shown, but they may have private fields and methods which are not shown.

- List all the behavioural subtyping relationships that exist between the given classes, and explain why.

In the following sections assume our language includes the keyword `inherits` which functions like `extends` except it does not create a subtype relation but just an inheritance relation. Each of the sections is independent of the others:

- Implement `TotalMap` using inheritance or subclassing from `PartialMap` with maximal

code reuse and without adding any new fields.

- Implement `PartialMap` using inheritance or subclassing from `TotalMap` with maximal code reuse and without adding any new fields. Assume that `Integer` is unbounded.
- Implement `Set` using inheritance or subclassing from `TotalMap` with maximal code reuse and without adding any new fields.
- Implement `Set` using inheritance or subclassing from `PartialMap` with maximal code reuse and without adding any new fields.

Solution

- `TotalMap` is a behavioural subtype of `PartialMap` as it has equivalent or stronger post-conditions for all methods. No other behavioural subtypes exist as the methods of the `Set` class are neither a subset nor a superset of the methods of any of the other classes.
- Here we have a subclassing relation:

```
public class TotalMap extends PartialMap {
   public TotalMap(){}

   public Integer lookup(Integer x)
   {
      if (super.has(x))
         return super.lookup(x);
      else
         return x;
   }

   public boolean has(Integer x){return true;}
}
```

- Here we only have an inheritance relation:

```
public class PartialMap inherits TotalMap {
   public PartialMap(){}

   //We encode the mapping x |-> x+2y+1
   // which ensures super.lookup(x)!=x
   // and enables the recovery of y
   public void set(Integer x,Integer y){super.set(x,x+(y*2+1));}

   public Integer lookup(Integer x)
   {
      if (has(x))
         return (super.lookup(x)-1-x)/2;  //Recover y
      else
         return null;
   }

   //x has been mapped iff super.lookup(x)!=x
   public boolean has(Integer x){return super.lookup(x)!=x;}
}
```

- `Set` does not subtype but could inherit `PartialMap`:

```
public class Set inherits PartialMap{
   public Set(){}

   public boolean has(Integer x)
   {
      return super.has(x) && super.lookup(x)==1;
   }
```

```
    public void include(Integer x)
    {
        super.set(x,1);
    }

    public void exclude(Integer x)
    {
        super.set(x,0);
    }
}
```

- and the same for `TotalMap`:

```
public class Set inherits TotalMap{
    public Set(){}

    public boolean has(Integer x)
    {
        return super.lookup(x)!=x;
    }

    public void include(Integer x)
    {
        super.set(x,x+1);
    }

    public void exclude(Integer x)
    {
        super.set(x,x);
    }
}
```

Task 3

Some research languages have symmetric multiple dispatch - methods are defined outside classes, and dispatch dynamically on all arguments regardless of order (no overloading at all). There is no designated receiver for a method but rather all arguments are of the same priority - this is intended to handle binary methods better which are often naturally symmetric. The runtime selects the most specific method to dispatch according to all arguments, and so there must be a single best implementation for each possible invocation of a method. The return type is not considered in the implementation selection. When compiling a package the compiler analyzes all types used in the package and all methods and makes sure that for each method and argument types combination there is a single best method to be called - or issues an error if that is not the case. Assume the following three classes in such a language:

```
package integer
class Integer
{
    ...
}
Integer add(Integer x,Integer y){...}

package natural
import integer.Integer
class Natural extends Integer
{
    ...
}
Integer add(Natural x,Integer y){...}
Integer add(Integer x,Natural y){...}
```

```
Natural add(Natural x,Natural y){...}
```

```
package even
import integer.Integer
class Even extends Integer
{
   ...
}
```

```
Integer add(Even    x,Integer y){...}
Integer add(Integer x,Even    y){...}
Even    add(Even    x,Even    y){...}
```

The elipsis in each class body represents (possibly) private data but no other methods.

Each package compiles successfully on its own.

A user has now written the following client:

```
package client
import even.Even
import natural.Natural

void f(Integer x,Integer y)
{
   Integer z = add(x,y);
}
```

- What would be the problem in allowing this client to compile in a type safe multiple dispatch language? Show code that would expose the problem.
- Which requirement could we relax so that this call is valid? Dispatch must remain completely symmetric.
- What could we do in the client package, in order to resolve the problem, without modifying other packages and without relaxing the requirement mentioned above? What is the conceptual problem with this resolution?

Solution

- The problem would be that the call `add(x,y)` could be ambiguous between the methods `add(Even,Integer)` and `add(Integer,Natural)` in the call:

  ```
  {
     Even e;
     Natural n;
     f(e,n);
  }
  ```

  Both are the most specific implementations but neither is more specific than the other.
- We could allow the runtime to choose any of the viable methods that is not worse than another method - thus we would lose the ability to predict which method gets called, but functionality should conform to at least that of `add(Integer,Integer)`.
- The client could define a method `add(Even,Natural)` (and any other missing methods) that would resolve the ambiguity. This solution would break modularity, as verifying that we have a complete set of method implementations is not modular - we would need to analyze all of the types and methods in the program in order to enusre that a best implementation always exists.

Task 4

Consider the following C++ classes, implementing part of an expression tree:

```
class Expression {
public:
   virtual int evaluate() = 0;
}

class BinaryExpression : Expression {
public:
   virtual int evaluate() override {
      int l = evalLeft();
      int r = evalRight();
      return op(l,r);
   }

   virtual int op(int l, int r) = 0;

private:
   Expression* left,*right;

protected:
   virtual int evalLeft() {
      return left->evaluate();
   }
   virtual int evalRight() {
      return right->evaluate();
   }
}

class MultiplicationExpression : BinaryExpression {
protected:
   virtual int op(int l, int r) override {
      return r*l;
   }
}
```

The `BinaryExpression` class is an abstract class for binary nodes in an expression tree (such as addition, multiplication etc) and the `MultiplicationExpression` class represents an integer multiplication expression.

- In an attempt to optimize expression tree evaluation, the following variant for `MultiplicationExpression` was written:

  ```
  class MultiplicationExpression : BinaryExpression {
  protected:
     virtual int evalLeft() override {
        int r = BinaryExpression::evalLeft();
        leftIsNonZero = r!=0;
        return r;
     }

     virtual int evalRight() override {
        if (leftIsNonZero)
           return BinaryExpression::evalRight();
        else
           return 0;
     }
     virtual int op(int l, int r) override {
        return r*l;
     }

   private:
  ```

```
    bool leftIsNonZero = false;
}
```

This version tries to avoid evaluating the right hand side expression tree if the left hand side had evaluated to 0. Would this version work as the above version? What does this version assume about the `BinaryExpression` class that the former does not? What change in the `BinaryExpression` class would break the optimized version but not the original version?

- How would you modify the optimized version of `MultiplicationExpression` to fix this problem, while still avoiding the unnecessary evaluation of subtrees when possible?

Solution

- The optimized version would work as the original version, however it further assumes that the subexpressions are evaluated in left to right order - to which the original version is oblivious. The following, seemingly innocent, change to the `BinaryExpression` class could break the optimized version:

```
class BinaryExpression : Expression {
public:
    virtual int evaluate() override {
        return op(evalLeft(),evalRight());
    }
    ...
```

As the order of evaluation for function arguments is not specified (but usually right to left), this version might not work on some compilers/architectures (check it!) where the arguments are evaluated in right to left order.

- If we drop the assumption about the order of evaluation, we need to account for both orders:

```
class MultiplicationExpression : BinaryExpression{
protected:
    virtual int evalLeft() override {
        if (!first && !firstNonZero)
            return 0;
        return evalCommon(BinaryExpression::evalLeft());
    }

    virtual int evalRight() override {
        if (!first && !firstNonZero)
            return 0;
        return evalCommon(BinaryExpression::evalRight());
    }

    virtual int op(int l, int r) override {
        return r*l;
    }

private:
    int evalCommon(int r) {
        if (first)
            firstNonZero = r!=0;
        first=!first;
        return r;
    }

    bool firstNonZero = false;
    bool first = true;
}
```

Now if the first subexpression is evaluated to zero, the second one would not be evaluated, regardless of the order of evaluation

Task 5

Consider the following C++ classes, representing 2 and 3 dimensional vectors:

```
class Vector2D{
public:
   Vector2D(double _x, double _y)
   : x(_x), y(_y), lengthCache(-1)
   {}

   double length() {
      if (lengthCache<0)
         lengthCache = calculateLength();
      return lengthCache;
   }

private:
   double x,y;
   double lengthCache;

protected:
   virtual double calculateLength() {
      return sqrt(sqr(x)+sqr(y));
   }
}

class Vector3D : public Vector2D{
public:
   Vector3D(double _x, double _y, double _z)
   : Vector2D(_x,_y), z(_z)
   {}

private:
   double z;

protected:
   virtual double calculateLength() override {
      return sqrt(sqr(Vector2D::calculateLength())+sqr(z));
   }
}
```

- Assuming that the `sqrt` function is significantly more expensive than other floating point operations, can you suggest a way to make the implementation more efficient without causing code duplication and without exposing private fields? If you have to modify the superclass - is that a reasonable modification if we had no knowledge of subclasses?
- Normalizing a vector means scaling it uniformly so that it points in the same direction but its length is 1 (dividing each coordinate by the length) - implement a `normalize()` method for both classes, avoiding code duplication
- Would your solution work if the implementation of the Vector2D class were to be changed so as not to cache the vector length but recalculated it every time? Is that significant?

Solution

- We would have to modify the superclass specifically for this - it would not be a sensible implementation for the superclass if we did not know the subclass:

  ```
  class Vector2D{
  ```

```
    ...

protected:
    virtual double calculateSquareLength() {
        return sqr(x)+sqr(y);
    }
    virtual double calculateLength() {
        return sqrt(calculateSquareLength());
    }
}

class Vector3D : public Vector2D{
    ...
protected:
    virtual double calculateSquareLength() override {
        return Vector2D::calculateSquareLength() +  sqr(z);
    }
}
```

- We could implement it as follows:

```
class Vector2D{
    ...

public:
    virtual void normalize() {
        double s = length();
        x /= s;
        y /= s;
        lengthCache = 1;
    }
}

class Vector3D : public Vector2D{
    ...

public:
    virtual void normalize() override {
        double s = length();
        z /= s;
        Vector2D::normalize();
    }
}
```

- The solution would fail in that case, because the length calculated for normalizing x and y would be wrong. This is significant because the correctness of the subclass depends on the private fields/implementation of the superclass - so they have to be verified/tested together and not modularly.