Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 7

Dynamic Method Binding and Multiple Inheritance

November 6, 2020

Task 1

(from a previous exam)

Consider the following C++ program:

```
class X {
   public:
      X(int p) : fx(p) {}
      int fx;
};
class Y {
   public:
      Y(int p) : fy(p) {}
      int fy;
};
class B : public virtual X,public Y {
   public:
      B(int p) : X(p-1),Y(p-2){}
};
class C : public virtual X,public Y {
   public:
   C(int p) : X(p+1),Y(p+1){}
};
class D : public B, public C {
   public:
      D(int p) : X(p-1), B(p-2), C(p+1){}
};

int main() {
   D* d = new D(5);
   B* b = d;
   C* c = d;
   std::cout << b->fx << b->fy
             << c->fx << c->fy;
   return 0;
}
```

What is the output of running the program?

(a) `5555`

(b) `2177`

(c) **CORRECT:** `4147`

(d) `7177`

(e) `7777`

(f) None of the above

Task 2 (from a previous exam)

Consider the following C++ code (recall that default constructors, i.e., constructors without arguments, do not need to be called explicitly in C++):

```
class A {
    public:
        A(int i) { std::cout << "A" << i; }
        A() { std::cout << "A1"; }
        virtual int get() { ... }
};

class B: MODIFIER A {
    public:
        B(int i) : A(i) { std::cout << "B" << i; }
};

class C: MODIFIER A {
    public:
        C(int i) : A(i) { std::cout << "C" << i; }
};
class D: public B, public C {
    public:
        D(int i) : B(i + 10), C(i + 20) { std::cout << "D" << i; }
};
```

Now assume that `MODIFIER` is replaced by `public`.

A) Why does the following client code not compile?

```
void client()
{
    D* d = new D(5);
    std::cout << d->get();
}
```

solution

The call `d->get()` is ambiguous because class `D` inherits two versions of `A` (and therefore of `get()`), one from `B` and one from `C`.

B) Add a method to one of the classes so that `client` compiles.

solution

We can resolve the ambiguity by overriding `get` in class `D`, for example to return `B::get()` or any other integer value. The resulting code looks as follows:

```
class D: public B, public C {
    public:
        ...
        virtual int get() { return B::get(); }
};
```

C) What is the output resulting from the call `new D(5)` in method `client`?

solution

The code outputs “A15 B15 A25 C25 D5” (without whitespace).

D) Now, assume that `MODIFIER` is replaced by `public virtual`.

What is the new output resulting from the call `new D(5)` in method `client`?

solution

The code outputs “A1 B15 C25 D5” (without whitespace).

Task 3

Consider the following C++ code:

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

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

   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 him-/herself. 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 what happens.

solution

The following C++ code breaks the invariant:

```
class B : public Person
{ public: B (string n) : Person (n) {} };
class C : public Person
{ public: C (string n) : Person (n) {} };
class D : public B, public C
{ public: D (string n) : B(n), C(n) {} };

void marryMyself ()
{
   D me = D("Me");
   B *b = &me;
   C *c = &me;
   b->marry(c);
```

```
    if (b->getSpouse()) cout << b->getSpouse()->getName();
}
```

The object `me` contains an object of class `B` and an object of class `C`. The addresses of these objects are different and they are obtained using the assignments to `b` and `c` respectively. During the call `b->marry(c)`, the condition `p == this` compares these two addresses and finds them not equal.

Task 4 (from a previous exam)

Consider the following Java classes:

```
class A {
   public void foo (Object o) { System.out.println("A"); }
}

class B {
   public void foo (String o) { System.out.println("B"); }
}

class C extends A {
   public void foo (String s) { System.out.println("C"); }
}

class D extends B {
   public void foo (Object o) { System.out.println("D"); }
}

class Main {
   public static void main(String[] args) {
      A a = new C(); a.foo("Java");
      C c = new C(); c.foo("Java");
      B b = new D(); b.foo("Java");
      D d = new D(); d.foo("Java");
   }
}
```

What is the output of the execution of the method `main` in class `Main`?

(a) The code will print A C B D

(b) **CORRECT:** The code will print A C B B

(c) The code will print C C B B

(d) The code will print C C B D

(e) None of the above

Task 5

Java 8 allows interface methods to have a default implementation directly in the interface.

A) What are some advantages of this feature?

solution

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.

B) What could be some problems with this feature? How can they be resolved?

solution

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.

C) What problems of C++ multiple inheritance are avoided by this new design for Java interfaces?

solution

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.

D) 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.

solution

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

- What are some advantages of this feature?

solution

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.

- 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

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 6

Consider the following C# classes:

```
public class Matrix {
   public virtual Matrix add(Matrix other) {
      Console.WriteLine("Matrix/Matrix");
      return null;
   }
}

public class SparseMatrix : Matrix {
   public virtual SparseMatrix add(SparseMatrix other) {
      Console.WriteLine("SparseMatrix/SparseMatrix");
      return null;
   }
}

public class MainClass {
   public static void Main(string[] args) {
      Matrix m = new Matrix();
      Matrix s = new SparseMatrix();
      add(m,m);
      add(m,s);
      add(s,m);
      add(s,s);
   }

   public static Matrix add(Matrix m1, Matrix m2) {
      return m1.add(m2);
   }
}
```

A) What is the output of this program? Please explain.

solution

The output is:

```
Matrix/Matrix
```

```
Matrix/Matrix
Matrix/Matrix
Matrix/Matrix
```

The compiler chooses a method based on the static type of the receiver and the static type of the argument. It thus chooses `add(Matrix other)` in all four cases. At runtime, either this statically chosen method will be executed or its most-derived override. However, `add(SparseMatrix other)` is not an override of `add(Matrix other)`, because overriding methods in C# should have invariant arguments and they should be declared with the `override` modifier. Therefore, we always execute the method from `Matrix`.

B) Without breaking modularity, change only the body of `MainClass.add` to make it possible to always call the most specific `add` method from the matrix hierarchy.

solution

We could change `MainClass` to the following:

```
public static Matrix add(Matrix m1, Matrix m2)
{
   return (m1 as dynamic).add(m2 as dynamic);
}
```

Now, the initial method lookup is also done at runtime, based not on the static, but on the dynamic type of the receiver. Thus in the third and fourth case there will be a choice between the two different `add` methods in class `SparseMatrix`. To also enable a dynamic lookup of the most-specific method based on the argument type, we additionally cast the argument as `dynamic`.

Task 7 (from a previous exam)

Consider the following C# code, which compiles and executes without raising exceptions:

```
class Ingredient {
  public void mix(Ingredient i1, Ingredient i2) {
    Console.WriteLine("Ingredient.mix");
  }
}

class Milk: Ingredient {
  public void mix(Egg e, Flour f) {
    Console.WriteLine("Milk.mix");
  }
}

class PowderedMilk: Milk {
  public void mix(Ingredient i, Flour f) {
    Console.WriteLine("PowderedMilk.mix");
  }
}

class Egg: Ingredient {}

class Flour: Ingredient {}

class Program {
  static void mix(Ingredient i1, Ingredient i2, Ingredient i3) {
```

```
25      (i1 as dynamic).mix(i2 as dynamic, i3 as dynamic);
26    }
27
28    static void Main() {
29      Ingredient i1 = new PowderedMilk();
30      Ingredient i2 = new Egg();
31      Ingredient i3 = new Flour();
32      mix(i1, i2, i3);
33    }
34  }
```

A) Which is the output of the execution of the method `Program.Main()`?

solution

`PowderedMilk.mix`
Overloading resolution in C# chooses the most specific method declaration in the class of the receiver. If there is no applicable method, then the methods of the super class are checked. This process is repeated until an applicable method is found. The program therefore executes method `PowderedMilk.mix(Ingredient i, Flour f)`, even though method `Milk.mix(Egg e, Flour f)` has more specific parameter types. Note: The overloading resolution of Java would pick method `Milk.mix(Egg e, Flour f)`.

B) List **all** the casts (from line 25) and **all** the methods that can be removed from the given code, such that it still compiles and when executed produces the output from Task A.

solution

the cast for `i2`
the method `Milk.mix()`
the method `Ingredient.mix()`

Task 8 (from a previous exam)

Consider the following C++ class definitions, which compile fine:

```
class Box {
  private:
    int content;
  public:
    void set(int x) { content = x;     }
    int  get()      { return content; }
};

class IncreaseBox: virtual public Box {
  public:
    void increase() { set(1 + get()); }
};

class MultiplyBox: virtual public Box {
  public:
    void multiply(int x) { set(x * get()); }
};

class BestBox: virtual public IncreaseBox, virtual public MultiplyBox {};
```

In the lectures you have seen that in an object-oriented language one can simulate inheritance by a combination of subtyping and aggregation. In this task you will need to adapt this technique to handle multiple virtual inheritance in C++, and use it to translate the provided C++ code to Java, **without** using default methods in Java interfaces.

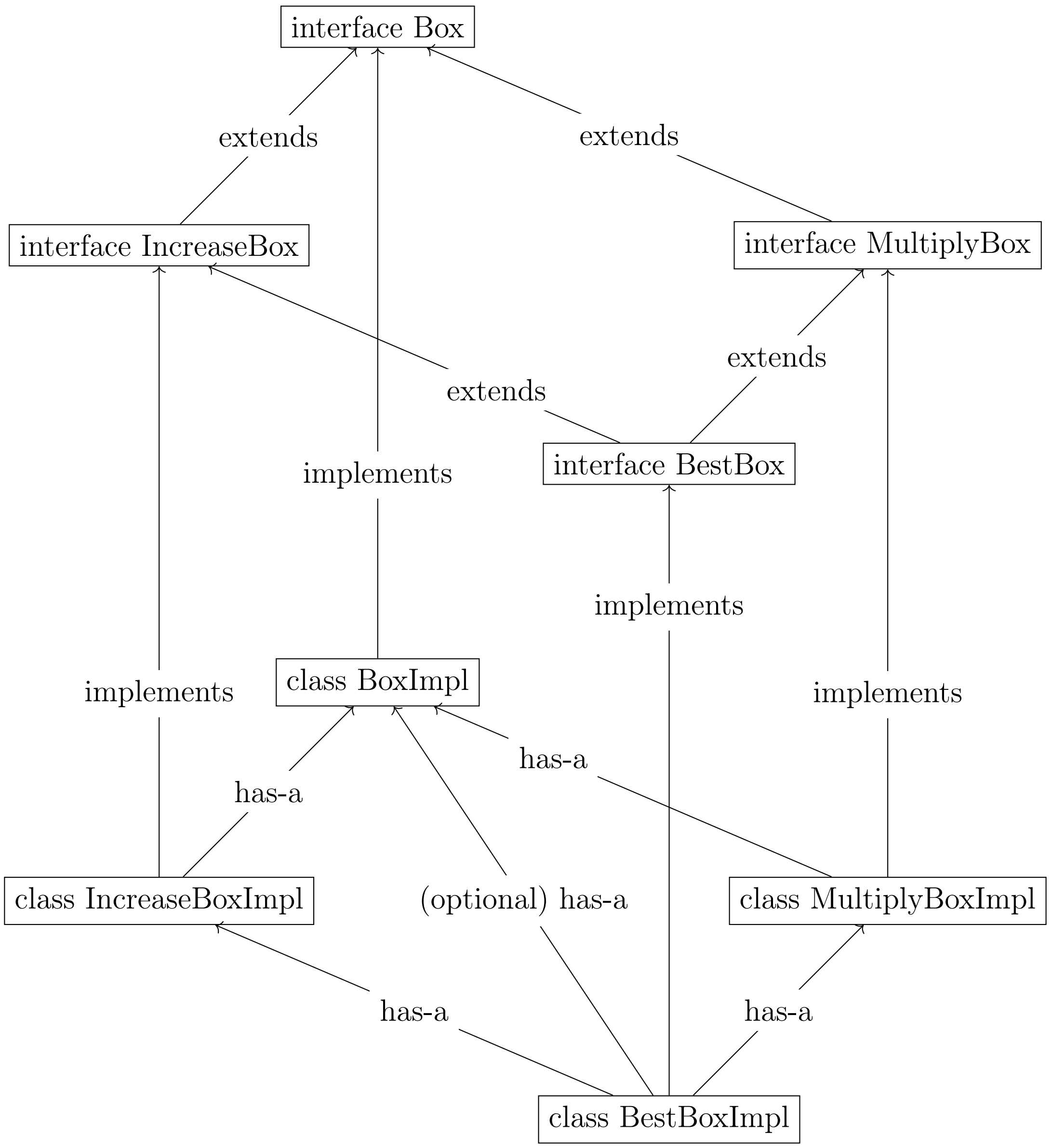
To help you check that the translation preserves the original behavior, we provide on the right an example of client code that (i) should compile fine in Java, and (ii) should compute the same values as the analogous C++ version on the left. Note that in the Java version it should be possible to create instances of `MultiplyBox` that are not instances of `IncreaseBox`, and vice versa.

```cpp
// C++
void client(BestBox* bb) {
    Box* b = bb;
    IncreaseBox* ib = bb;
    MultiplyBox* mb = bb;

    b->set(10);
    ib->increase();
    mb->multiply(2);

    std::cout << b->get();
}
```

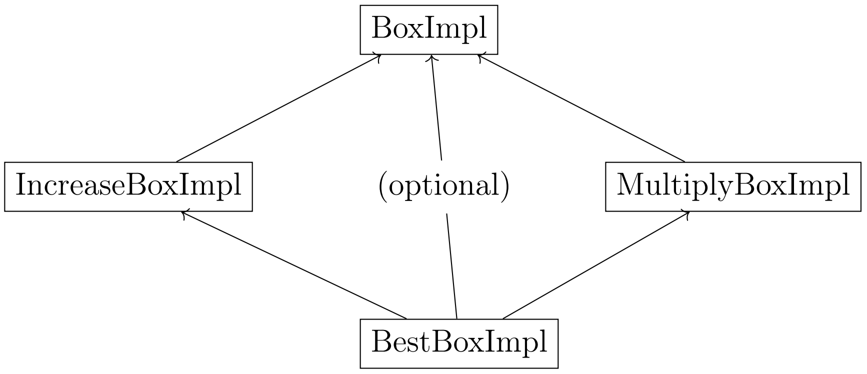
```java
// Java
void client(BestBox bb) {
    Box b = bb;
    IncreaseBox ib = bb;
    MultiplyBox mb = bb;

    b.set(10);
    ib.increase();
    mb.multiply(2);

    System.out.print(bb.get());
}
```

Draw a graph of the translated Java code, expressing classes, interfaces and the relations between them. Each node must be a class or interface (write which one) with an intuitive name; the directed edges must be labeled with “implements”, “extends” or “has-a”, according to the relation that they express. You do not have to specify fields, methods nor constructors.

solution



Note that this graph cannot express if the inheritance is virtual or non-virtual. To show that `IncreaseBox` and `MultiplyBox` inherit virtually from `Box`, we can draw the graph of all the objects on the Java heap that are reachable from an instance of the Java version of the `BestBox` C++ class (i.e., `BestBoxImpl`):



In this second graph, each node represents a Java object and the directed edges represent references between objects. To further express that `BestBox` inherits virtually from both `IncreaseBox` and `MultiplyBox`, we would need to create instances of additional subtypes of `IncreaseBox` and `MultiplyBox`, respectively.