Programming assignment block 1

Isak Karlsson & Beatrice kerblom
{isak-kar, beatrice}@dsv.su.se

1 The Assignment

Implement inheritance and delegation in C or Java for dynamically typed methods. In the implementation of inheritance an object should be represented by a struct with pointer to its class. A class can have a pointer to its superclass. A class with no superclass pointer (i.e., it points to NULL) is a root class, e.g., Object in Java.

In the implementation of delegation, an object should be represented by a struct. A object can have a pointer to its prototype (another object). A object with no prototype pointer (i.e., it points to NULL) is a "root" prototype.

Binding should be controlled by the name of the method and the number of arguments (i.e. the arity of the method) the method takes. Carefully comment your code with a detailed description of what goes on, esp. in the functions that control binding.

This implementation need not be flawless; you are not implementing an entire language, only inheritance and delegation.

Parts of a possible test program for inheritance and how it might look in C (many important parts missing – this program does not compile):

```
#include <stdio.h>
#include <string.h>

typedef struct _Object {
    Class* class;
    char* msg;
} Object;

void printMsg(Object* self){
    puts(self->msg);
}
```

```
void setMsg(Object* self, char* msg){
    self->msg = msg;
}

void* invoke(void* receiver, char* methodname,
             int argnum, void* argvalue){
    /* ... */
}

int main(void){
    Method method2;
    method2.method = (METHOD) setMsg;
    method2.name = "setMsg";
    method2.argnum = 1;
    method2.next = NULL;
    Method method;
    method.method = (METHOD) printMsg;
    method.name = "printMsg";
    method.argnum = 0;
    method.next = &method2;
    Class class;
    class.super = NULL;
    class.firstMethod = &method;
    Object obj;
    obj.class = &class;

    /* Now the important part */
    invoke(&obj, "setMsg", 1, "Hello, world");
    invoke(&obj, "printMsg", 0, NULL);
    invoke(&obj, "setMsg", 1, "This is another message");
    invoke(&obj, "printMsg", 0, NULL);
    return 0;
}
```

It creates two methods that it links into the class, creates an object and determines it to be of the created class. The important part of the program show four invocations of methods on the object obj. The first invocation should bind to the method setMsg which sets the object's msg field to Hello, world. The next invocation should bind to the printMsg function that print

the current message in obj, etc.
When run, this program prints the following output to the terminal:
Hello, world
This is another message

Parts of a possible test program for inheritance and how it might look in Java (many important parts missing – this program does not compile):

```
abstract class Function { }

class Class { }

class Instance { }

Instance o = new Instance();
o.klass =  new Class();
o.klass.dict.put("test", new Function() {
    public void execute(Instance self, Instance... args) {
        System.out.println("test invoked");
    }
});
public void invoke(Instance obj, String method,
                   Instance... args) { }
```

You need not complete these programs. The programs shown here is just to give you some idea of how this might look. Note that we are not parsing any code etc. We are simply implementing a lookup algorithm that binds a certain call to the invoke function to a certain other function in the program.

If you understand how inheritance and delegation works, completing these programs should be fairly easy. Convince yourself that it also supports overriding. In a short, written essay (approximately 500 words) describe what you would need to add support for considering the types of the argument values when binding an invocation to a certain method; for example, it should be possible to have two setMsg method, one that takes an int (that it converts to a string before assigning it into msg) and one that takes a string and the type of the argument (or argument value?) controls the binding. Also reflect on the difference in your implementations of inheritance and delegation. Which do you think is simpler? Why? Which do you prefer? Why? What are the differences? Why? How?

You are allowed to simplify your program so that only one or zero parameters are allowed (as in the example above).

The goal here is not to implement a flawless C/Java program with bells and whistles. Implementing this program requires you to think about (and know something about) inheritance and delegation. That's what we want.

2 Grading

In order to receive the grade E you have to implement inheritance and reflect on your implementation. To receive a grade better than C, you also have to implement delegation, and reflect on the differences.

Grading programming is more an art than a science. In the general case, it is extremely difficult to impossible to say that one program is better than the other. For the obvious reasons, it is impossible to cover the grading criteria completely. In any case, the points below are not totally black/white. This is why you should also should reflect on you implementation.

For the programming assignment for block 2 the following table will be used when grading the assignments.

Grade	Inheritance	Delegation	Reflections
A	Good	Good	Good
B	Good	Good	OK
C	Good	Missing/OK	OK
D	OK	Missing/OK-	OK-
E	OK-	Missing/OK-	OK-
Fx	Missing/OK-	Missing	Missing/OK-
F	Missing	Missing	Missing

Table 1: Grading scheme

To get a Good, your program/answer must:

1. Completely meet the specification,e.g., produce correct output from example inputs (not too little or too much), or the like;

2. Be readable without overly commenting, be correctly indented, and use reasonable names (for variables, method, classes etc.);

3. Be well designed for the problem at hand

For every point above that is not satisfied, the program/answer takes one step down the grade ladder from Good down to Missing/Fail.