João Vítor Fernandes Dias

The goal is to “offer informal definitions on terms and ideas that surrounds Object-Oriented Programming (OOP), such as “data abstraction”, “type hierarchies” and the “OOP” itself. It will be presented using C++ but not limited by that.

1 Introduction

“Not all programming languages can be ‘object oriented’”.

This paper will show what “Object Oriented (OO)” means “in the context of a general-purpose programming language.”

Paragraph 2 will distinguish OOP “and ‘data abstraction’ from each other and from other styles of programming”.

2 Programming Paradigms

For some reason OOP is considered a “good” programming technique. An OOP language is a language that supports and are easy to develop using OOP. Even though some languages are capable of handling OOP it doesn’t mean that it is easy to use. Languages like C and Fortran can handle OOP, but they make it too hard to make it feasible.

Support for a paradigm is made through various ways such as compile-time and run-time checks for deviation from the paradigm, like type checking. Integrated Development Environments (IDEs) are also helpful in supporting the paradigm.

There aren’t languages better than others. They are all different and are used for different purposes. What really matters is its features are enough to support the desired programming style/paradigm. It should also have its features safely integrated into the language, be able to mix different features to achieve new solutions; And the user should only need to know and understand the code that it wrote for a particular problem “what you don’t know won’t hurt you”.

2.1 Procedural Programming

The programming paradigm is:

“Decide which procedures you want; use the best algorithms you can find.”

Description:

Uses functions and returns values from them. The way it is thought is modelling the right way to pass the different kinds of arguments and how to deal with the different kinds of functions. “Functions are used to create order in a maze of algorithms”

Some languages that use this paradigm are:

Fortran, Algol60, Algol68, C, and Pascal

Code Example:

```
Type1 function_name1 (type_received variable_name) {

//code that this function runs

// maybe returns some value

}
```

```
Type2 function_name2 () {

Type1 variable_that_receives_the_function_name1_value = function_name1 (value_passes_of_type_”type_received”)

}
```

2.2 Data Hiding

The Programming Paradigm is:

“Decide which modules you want; partition the program so that data is hidden in modules”.

Description:

The focus changes from organizing procedures to organizing data. This increases the program size. A set of procedures and the data they manipulate is called a module. So, it is also known as “data hiding principle”. A common example is the stack module. Considering that the user can only use the functions push () and pop (), that the user can’t directly access the stack’s elements and that it’s initialized before the first use.

Some languages that use this paradigm are:

(gradually more supportive) Pascal, C, Modula-2.

Code Example:

```
// declaration of the interface of module stack of characters

char pop ();

void push (char);

const stack_size = 100;
```

Assuming that this interface is found in a file called stack.h, the stack itself can be defined like this:

```
#include "stack.h"

char v[stack_size];

char pop () {// check for underflow and pop}

void push (char c) {// check for overflow and push}
```

2.3 Data Abstraction

The programming paradigm is:

“Decide which types you want; provide a full set of operations for each type”.

Description:

It goes in a similar direction of the data hiding paradigm. But, instead of only being able to manage one data structure, now the module manager must define a way for creating more of that same data structure, each one having its own identifier. This new structure created is then called an “abstract data type”. An example of a data type like that are the real-imaginary numbers that is user-defined that will make those as real as int and float.

Some languages that use this paradigm are:

Ada, Clu, C++.

Code Example:

```
void f ()
{
        stack_id s1;
        stack_id s2;
        s1 = create_stack (200);
        // Oops: forgot to create s2

        char c1 = pop (s1, push(s1,'a'));
        if (c1!= 'c') error("impossible");

        char c2 = pop (s2, push(s2,'a'));
        if (c2!= 'c') error("impossible");

        destroy(s2);
        // Oops: forgot to destroy s1
}
```

2.4 Problems with Data Abstraction

Once the Abstract Data Type (ADT) is defined, it doesn't interact at all with the rest of the program. This leads to inflexibility. In a situation where you have a class for manipulating shapes by drawing, moving, rotating, etc. the function that draws must "know" all possible shapes, so the code gets bigger for each shape added.

2.5 Object Oriented Programming

The programming paradigm is:

"Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance".

Description:

Expressing the distinction between a general thing and a specific one and taking advantage of it defines OOP. In the example of the shapes, you should specify a class that represents the general properties of all shapes. Given this definition, general function manipulation shapes now can be written. To define a particular shape, first it is needed to declare that it is a shape and then specify its properties. A "class circle is said to be derived from a class shape, and the class shape is said to be a base of class circle", or also Circle and Shape could be called subclass and superclass. The data abstraction is useful when there are enough commonalities. In areas like interactive graphics, there is a huge scope for OOP, but for areas like classical arithmetic types there are almost no scope.

Some languages that use this paradigm are:

Simula, C++

Code Example:

The Simula inheritance mechanism provides a solution. First, specify a class that defines the general properties of all shapes:

```
class shape {

point center;
```

```
color col;

// ...

public:

point where () {return center;}

void move (point to) {center = to; draw ();}

virtual void draw ();

virtual void rotates(int);

// ...

};
```

The functions for which the calling interface can be defined, but where the implementation cannot be defined except for a specific shape, have been marked ‘‘virtual’’ (the Simula and C++ term for ‘‘may be redefined later in a class derived from this one’’). Given this definition, we can write general functions manipulating shapes:

```
void rotate_all (shape* v, int size, int angle)

// rotate all members of vector "v" of size "size" "angle" degrees

{

for (int i = 0; i < size; i++) v[i].rotate(angle);

}
```

To define a particular shape, we must say that it is a shape and specify its particular properties (including the virtual functions).

```
class circle : public shape {

int radius;

public:

void draw() { /* ... */ };

void rotate(int) {} // yes, the null function

};
```

END