

1. **Visual and natural programming for non-programmers - a false hope?**

Given the pace of technological development over the last 30 years, it is a bold statement to suggest that that visual/natural programming is a pipe dream.

Visual and natural programming is one of the holy grails of programming and computer science. However, 30 years of trying to develop a visual or natural programming language hasn't delivered a tenable solution.

A visual/natural programming language may never exist. At a fundamental level, programming is more about the logic and mathematics behind organising the flow of operations than it is using a programming language. A programmer will always need to learn programming in order to express program requirements in a precise and unambiguous manner. This article will demonstrate how it is not possible for a person without a computer science background to successfully program, in text or visually, using a simple task: how to sort a list of unordered items in descending order.

This problem is the same for other disciplines. You can't realistically build a car from scratch if you're not a mechanical engineer. Even if you were given all the parts, you'd still have very little chance of assembling them into a working car.

This paper will provide a crash course of the basic concepts behind programming in order to show that it is a non-trivial activity and, that in order to programme, you need to think in a certain way to successfully describe what you want to do so the computer can carry out your instructions.

If any of the past attempts at visual and natural programming actually delivered on the promise, then they would be in widespread use. Instead, developers are still using standard development tools such as [VisualStudio](#) and [Eclipse](#).

Computer science's inability to create a visual/natural programming is not cause for despair. It's not that the end goal is impossible; it's that trying to make computer programming natural or visual for non-programmers is simply not possible. Instead, we need to attack the problem from a very different perspective.

1.1. A simple task for people: sorting items in a list

Here's a simple test to prove the point of this paper: consider the following list of items:

Banana	Hotel
Apple	Car
Orange	Fox
Aardvark	Consequence
Xylophone	Decision

Can you sort them alphabetically the order A to Z? Of course you can – it should take you less than a minute.

Now, can you describe exactly what you did to sort the items? For example:

- ◀ What procedures are you performing on each item in the list?
- ◀ How do you choose which item goes at the top of the list?
- ◀ How do you choose the next item?
- ◀ How do you know when you have finished sorting the items?

Chances are you'll find this very difficult. Try getting someone else to do the same thing. Did you both describe it the same way? Could s/he describe it at all?

Now take your description of your procedure and give it to someone to do it on a new list. Did it work? Does it work on other lists of items, such as a list of unsorted numbers, or a list of sentences?

1.2. Programming a sort procedure on a computer

Making the computer perform the sort routine is not a trivial exercise, despite how easily we can do it in our minds. I'll show you why.

It turns out there are several different ways to perform a sort that generally differ in terms of efficiency and speed.

The most basic way is called a [Bubble Sort](#). But this is slow and inefficient for long lists. Instead, people use routines like the [Insertion sort](#) or the [Selection sort](#), with the best performing general purpose sorts being the [Heapsort](#).

Who would have thought there are so many different ways to sort a list? Could you have thought them up? Unless you are a computer scientist or a mathematician, I'd imagine probably not.

The different sorting methods were developed to improve efficiency and make sorting faster. If the person creating the sort instructions does not have a good background in mathematics as well as the concepts involved in testing the efficiency of algorithms and improving them, it simply is not possible for that person to work it out from first principles.

Let's take a look at the bubble sort in natural language, followed by pseudo code and then in an actual programming language.

1.2.1. Using natural language to describe the sort procedure

The Bubble sort procedure can be described as follows:

1. In a list, compare the first item with the second item
2. If the second item is smaller than the first, swap the order of those two items
3. If the second item is bigger than the first, do not swap
4. Compare the second item with the third, third with fourth, and so on, performing steps 2 and 3 for each successive comparison
5. Repeat the entire process, starting with the first item in the list until no more swaps are needed.

Who would have thought a sort routine is actually quite difficult to describe? The Bubble sort gets its name because of the way items in the list 'bubble' their way up to the top of the list after repetitions of the process. The items do not go straight into their correct positions.

An interesting aspect of the Bubble sort is that you only need to compare the last two items in the list once (i.e. the first time the bubble sort is performed). See if you can work it out, or, in the irritating words of many text books 'the proof is left as an exercise for the reader.'

1.2.2. Using pseudocode to describe the sort procedure

In order to make an application do a sort, we (as programmers) need to describe how to sort using the same kind of exercise we just went through. We need to describe the procedures and operations to perform on the items in order to end up with a sorted list.

Programmers sometimes use [pseudocode](#) as a way of describing how the program is likely to work using the general approaches of a programming language. This means using natural language, but constraining it using the common reserved words and control structures.

The pseudocode for the bubble sort is as follows:

Line	Statement
1	function bubble_sort(list L, number listsize)
2	loop
3	has_swapped := 0 //reset flag
4	for number i from 1 to (listsize - 1)
5	if L[i] > L[i + 1] //if they are in the wrong order
6	swap(L[i], L[i + 1]) //exchange them
7	has_swapped := 1 //we have swapped at least once, list may not be sorted yet
8	endif
9	endfor
10	if has_swapped = 0 //if no swaps were made during this pass, the list has been sorted
11	exit
12	endif
13	endloop
14	endfunction

(From http://en.wikipedia.org/wiki/Bubble_sort_algorithm)

Let's break this down into what's happening in the pseudocode:

First and last line

The first and last line (1 and 14) act as a group structure for the procedure to the sort. This designates the procedure (function) name as 'bubble_sort' so we can call it whenever we need to do a sort. We only write it once per application and any part of the application can use this function. This also states that the function must be given a list, called 'L', and that we must know the list size (number of items in the list) in 'number'.

Line 2

The second line is an instruction to loop continuously until a condition is met that causes the loop to stop. This corresponds to statement 5, earlier in our natural language version, where we specified that the procedure continues until no more swaps are needed.

Line 3

Line 3 contains a state called 'has_swapped' that is set to '1' or '0', depending on whether a swap has taken place, or not, respectively. We initially set it to '0' so if the list is already ordered (a possibility) then the function will not loop infinitely. If the loop is still 0 after the first attempt at the sort procedure, then the loop exits.

Line 4

In line 4, the statement 'for number *i* from 1 to (listsize-1) sets *i* initially to '1' and is another looping structure that runs for as many times as the list is long minus 1. That is, if the list has 50 items, then the loop will run 49 times. As discussed earlier, we do not need to test the final pair, therefore, a final loop is redundant.

Line 5

In line 5, the statement 'if $L[i] > L[i + 1]$ ' tests whether the first item in a pair is bigger than the second item in the pair.

Line 6

In line 6, the two values are swapped if, and only if, the first is bigger than the second. Otherwise, it does not swap the numbers.

Line 7

In line 7, we set the state of 'has_swapped' to '1', indicating that a swap took place. This means the main loop will not exit (started in Line 2).

If no swap took place, then the value of 'has_swapped' would be '0', indicating that no swaps were needed, and the loop would end. This means that all the items in the list are sorted.

Line 8

In line 8, we make the boundary of the if statement in line 5.

Line 9

In line 9, we make the boundary of the 'for' statement in line 4.

Line 10

In line 10, we test whether a swap has taken place. If the value of 'has_swapped' is '0', then no swap has taken place.

Line 11

In line 11, the statement 'exit' terminates the loop (from line 2) if no swaps have taken place. This means that the list is sorted.

Line 12

In line 12, we make the boundary for the if statement in line 10.

Line 13

In line 13, we make the boundary for the loop statement in line 2.

Line 14

In line 14, we make the boundary for the function, started in line 1.

You can clearly see that describing the bubble sort, even in pseudocode, routine is not trivial.

For interest, an alternative (but not as clever) version of the pseudocode is:

```
function bubblesort (A : list[1..n]) {
    var int i, j;
    for i from n downto 1 {
        for j from 1 to i-1 {
            if (A[j] > A[j+1])
                swap(A[j], A[j+1])
        }
    }
}
```

(From http://www.codecadex.com/wiki/index.php?title=Bubble_sort#Pseudocode)

1.2.3. Using a programming language to program the sort procedure

Now that we have prepared our pseudocode version, we now need to adapt it to the particulars of the programming language in which you want to implement the sort procedure. We need to know about the reserved words and control structures to implement the procedure. Using [Java to program the sort procedure](#), your actual code would look like this:

```
public static void bubbleSort(int[] data)
{
    boolean isSorted;
    int tempVariable;
    int numberOfTimesLooped = 0;

    do
    {
        isSorted = true;

        for (int i = 1; i < data.length - numberOfTimesLooped; i++)
        {
            if (data[i] < data[i - 1])
            {
                tempVariable = data[i];
                data[i] = data[i - 1];
                data[i - 1] = tempVariable;

                isSorted = false;
            }
        }

        numberOfTimesLooped++;
    }
    while (!isSorted);
}
```

(From http://www.codecodex.com/wiki/index.php?title=Bubble_sort.)

Out of interest, if you wanted it [program the bubble sort in assembly](#), one of the lowest level languages that the CPU in your computer understands, it would look like this:

```
bs proc array:DWORD,len:DWORD
    mov ecx,len
    mov edx,array
    bs_o:
    xor ebp,ebp
    bs_i:
    mov eax,DWORD PTR [edx+ebp*4+4]
    cmp DWORD PTR [edx+ebp*4],eax
    jb @F
    xchg eax,DWORD PTR [edx+ebp*4]
    mov DWORD PTR [edx+ebp*4+4],eax
    @@:
    add ebp,1
    cmp ebp,ecx
    jb bs_i
    loop bs_o
    pop ebp
    retn 8
bs endp
```

(From http://www.codecodex.com/wiki/index.php?title=Bubble_sort.)

I'd like to see even a computer scientist put the assembly language version together! Most new graduates don't often get to do assembly as they mostly work with Java, C++ and other [third generation languages](#).

I would think that now the question you're probably asking yourself is 'can't we just tell the application to sort the data for us?' The short answer is maybe. The question is a mix of [visual programming](#) and natural language programming.

I used the sort routine as a very basic example to illustrate how programming even an apparently simple procedure is complicated. Many databases already contain a sort procedure and you call it from your program. You don't need to go to the trouble of writing it from scratch yourself.

1.2.4. Visually programming the sort procedure

Given the complexities of programming, your question might be: 'Can't I just draw what I want?' Here, the answer is 'Yes.' There are various development tools that let you visually program what you want, for example, [Andescotia's Marten](#), [Labview](#) and [others](#). In fact, your Excel spreadsheet is a form of visual programming where you layout the various numbers of interest on a grid and use various formulae to perform your desired calculations.

However, if you look closely at the visual programming tools, you'll find that what's really happening is that the same programming example, earlier, is just being represented in a visual form. You still need to know how to describe the sort procedure in a mathematically defensible way and use various statements and control structures of the development environment to implement what you want.

1.2.5. Natural language programming the sort procedure

One of the holy grails of computer science is natural language programming. That is, you merely type, in English or your native language, what you want and the code magically writes itself and your program runs. Artificial intelligence (AI) is a critical component of natural language programming. Despite the advances, English, like other languages, contains ambiguities. That is, there is more than one meaning for a given phrase. It is very difficult for AI and [natural language processing](#) to infer what you meant and create the right result.

Needless to say, there is no natural language programming environment on the horizon.

1.3. To program, you need to think like a computer scientist

Why is it so hard to program for a non programmer? Simply, you don't [think like a computer scientist](#). In order to successfully program a solution, you need:

- ◀ To get some input (e.g. from the keyboard, mouse or some other device).
- ◀ Perform a mathematical or other operation (e.g. add two numbers together'.
- ◀ Test for certain conditions to be true or false.
- ◀ Repeat some operation a number of times until a condition occurs.
- ◀ Display the output (e.g. on screen or a speech system)

To program (in Java) a procedure to display the words 'Hello, World' on the screen is complex (especially in Java). For example:

```
class HelloWorld
{
    public static void main(String args[])
    {
        System.out.println("Hello, World.");
    }
}
```

However, in BASIC, the same procedure is simpler:

```
print "Hello World!"
```

You can see how the ['Hello, World' example is done in other programming here](#).

Without learning the programming language, its syntax, type system and execution semantics (control structures), you'll never be able to program, let alone visually program. Natural language programming will not happen until AI is strong enough to work out what we mean when we speak (or type it).

1.4. Conclusion

By now you can see that programming can range from being simple (relatively speaking) to complex. The complexity, on the whole, cannot be underestimated.

[Object oriented programming](#), a very complex and abstract approach to programming, has principles such as [inheritance](#), [encapsulation](#) and [polymorphism](#). Just reading about these concepts confuses the novice computer user, let alone programming using these concepts.

The problem of visual and natural language programming is simply too great a problem to solve. In order to 'program', you really need to learn a computer language and how to use it to make do what you want. Just think how long it takes to learn a foreign language, let alone when you first learnt your native language.

Is this the end of the story as far as visual application development goes? Absolutely not! We're working on how to solve that problem, but from a very different perspective.

2. About the Author

Craig is the founder and Managing Director of The Performance Technologies Group (PTG Global), with over 15 years in user experience, user interface design and change management.

Craig runs the R&D function at PTG, having produced a number of world firsts including XPDesign – the first systematic methodology for user interface design and Certified Usable – the first guarantee for usability and user experience.

Craig has been the primary architect behind many of Australia's most popular websites including CBA, Virgin Blue and ASIC and works on cutting edge technologies such as touch, medical and special-purpose applications.

Craig holds a Masters qualification in organisational psychology, is a member of the APS and the APS College of Organisational Psychologists and is a Registered Psychologist in NSW. He is also an Associate of the University of NSW and Macquarie University.



Contact Craig on:

Email: craige@ptg-global.com

Phone: +61 (0)2 9251 4200

Mobile: +61 (0) 416 266 216

Address: Level 16, 207 Kent St, Sydney, NSW, 2000, Australia