

# The Visula programming language and environment

Calum Grant  
calum@visula.org

## Abstract

*Visula is a general-purpose object-oriented visual programming language (VPL). The language uses a new approach, by basing its notation upon UML sequence diagrams, instead of traditional control-flow or data-flow. This paper describes the notation, and presents an editing and debugging environment for the language. A usability analysis shows several potential advantages to the notation.*

## 1. Introduction

Visual programming languages (VPLs) have not become as widespread as it was once imagined they would be. This is somewhat surprising, since work done on Cognitive Dimensions [1] suggests that from a cognitive and usability perspective, graphical notations and editing environments are better in many ways to text, and that text is actually quite poor in many aspects. It is therefore worthwhile to try to improve on the usability characteristics of textual programming.

The main problem with visual programming (VP) is that there appear to be trade-offs in Cognitive Dimensions, so while VPLs improve some aspects of usability, they worsen others. Common problems are diffuseness (how efficiently is space used) [2], visibility (how much you can see), scalability [3] and viscosity (how easy is it to make changes).

Visula (<http://visula.org>) is a general-purpose VPL that mitigates some of these issues. Unlike other VPLs, Visula is based on UML sequence diagrams [4], which are effective and widely used in software engineering. Visula's contribution is to extend sequence-diagram notation to make it executable instead of being used only for documentation. Visula's other contribution is to add structural elements to sequence diagrams, and to combine sequence diagrams into one unified notation.

The notation is scalable and navigable by folding (collapsing) parts of the diagram that are not of relevance, and programs equivalent to several thousand lines can be edited without difficulty.

We also look at some of the cognitive dimensions of this language, which shows Visula's strengths in comparison to other textual and visual languages.

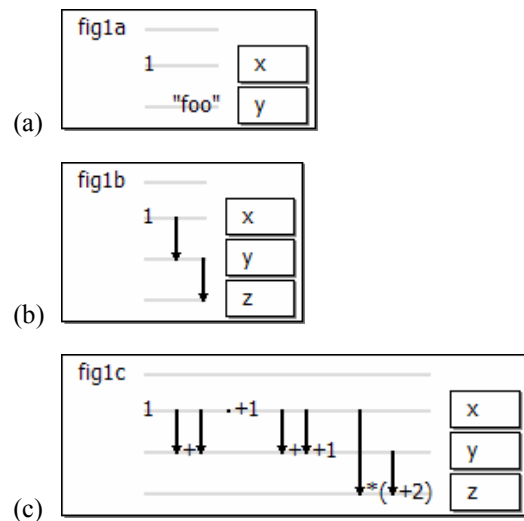
## 2. The Language

### 2.1 Sequences

Visula's notation is based on sequence diagrams. Each object in the program has a "life-line" which represents the object in time. Anything that happens to the object interacts with its life-line.

Unlike UML sequence diagrams [4] which use the vertical direction for time, Visula shows time going horizontally from left to right.

A simple value on a life-line assigns the value to the object. In Figure 1(a), the program assigns the integer 1 to the variable  $x$ , and the string "foo" to  $y$ .



**Figure 1: Sequences.** (a)  $x=1$ ,  $y="foo"$ , (b)  $x=1$ ,  $y=x$ ,  $z=y$ , (c)  $x=1$ ,  $y=x+x$ ,  $x=x+1$ ,  $y=x+x+1$ ,  $z=x*(y+2)$

An arrow between two life-lines copies one object to another. Figure 1(b) assigns  $x=1$ ,  $y=x$  and  $z=y$ .

Compound expressions can also appear on a life-line. In this case, the result of the expression is assigned to the object on the life-line. An expression can include other objects, in which case arrows lead from other life-lines into the expression. Literal values can also be used in expressions.

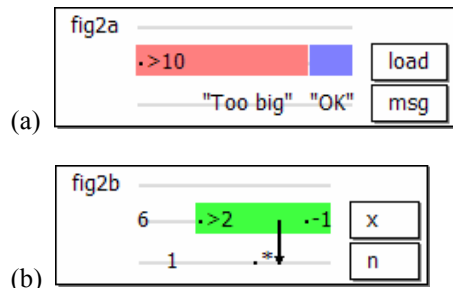
Figure 1(c) shows a sequence of expressions. When an expression refers to itself, a dot appears on the life-line, which is a self-reference.

The boundary between expressions is clear because a gap separates adjacent expressions. Expressions can be arbitrarily complicated.

Like UML sequence diagrams, the notation is designed to make it clearer how objects are used and interact in a program.

## 2.2 Control flow

Loops and conditions are represented using horizontal bars over the diagram. The horizontal bar covers the expressions in the loop or condition.



**Figure 2: Control flow. (a) An if-then-else control, (b) a while loop.**

Figure 2(a) shows a condition (an if-then-else). The first expression in the red bar is the condition. The remainder of the red bar covers the expressions that are executed when the condition is true, and a blue bar covers the expressions that are executed when the condition is false.

Figure 2(b) shows a loop. The first expression in the green bar is the condition, and the body of the loop is the remaining expressions in the bar.

The display of loops and conditions is very plain, but decorating them further would not actually add any information to the diagram.

Loops and conditions can be nested using several bars.

## 2.3 Functions

Figure 3(a) shows a function *sum* that performs a calculation on its inputs, *a* and *b*. The entire function is surrounded by a box, with its name displayed in top-left. The result of the function is assigned to the *return* value. Inputs to the function are written on the left of the function, while the output (*return*) is written on the right, which is consistent with time flowing from left-to-right.

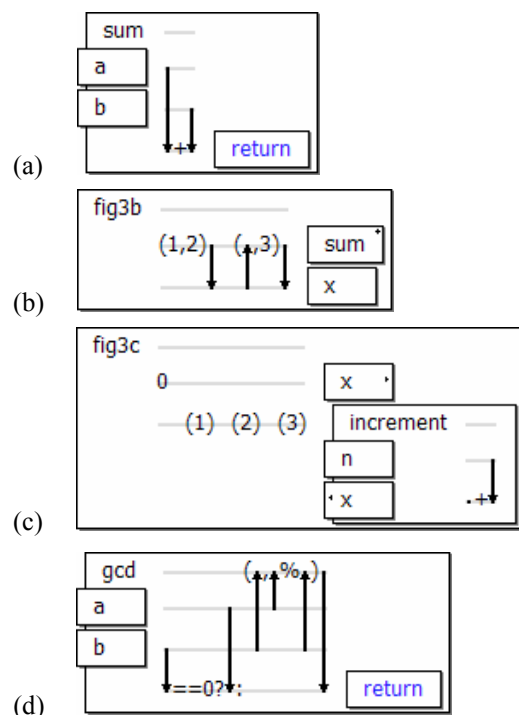
Functions can be defined inside other functions, in the scope where they are used, for example in Figure 3(b). The *sum* function has been folded.

The notation for calling a function is a pair of round brackets on the life-line of the function being called. The arguments are within the brackets, and the return value is optionally passed by an arrow to the object that stores the result.

Figure 3(b) calls the *sum* function twice, with the end result that *x=9*.

Nested functions can access variables in the outer function, by placing the variable as an input on the left. Figure 3(c) shows a nested function *increment* that does this, and end up with the value *x=9*. The marks (◀ and ▶) indicate nested variables.

Each function has a life-line on its top-row, which refers to the function itself. A function can call itself recursively, as shown in Figure 3(d).



**Figure 3: Functions. (a) A function to add two numbers, (b) calling a function, (c) access to nested variables, (d) a recursive function.**

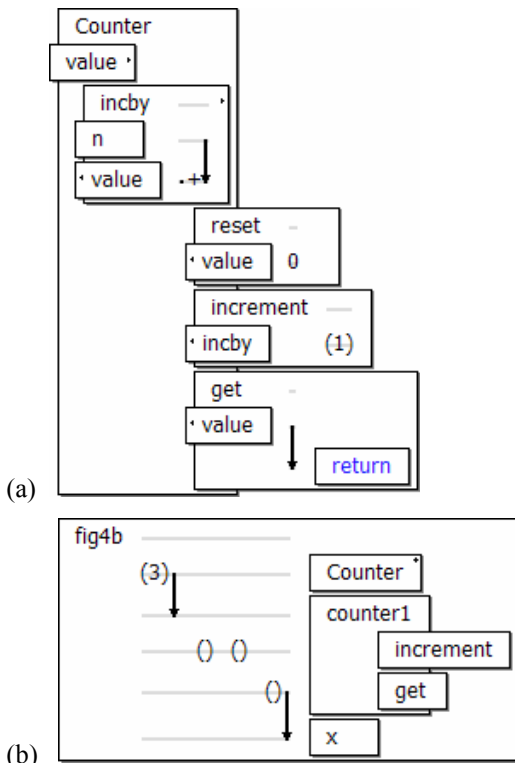
## 2.5 Classes

There is no explicit notation for classes. Classes are written in the same way as functions, except that the class has members and methods protruding from the right of the box. Figure 4 shows a *Counter* class.

The notation for methods is the same as for nested functions, and they behave exactly the same. Class members and methods may be made private by drawing them completely enclosed in the class, for example the *inby* method shown in Figure 4(a).

A class is instantiated by calling it as a function. The class can contain expressions to initialize the object. Initialisation values can be passed into the class as function arguments, for example *value*. Inheritance is possible via delegation.

Figure 4(b) shows a use of the *Counter* class. The *Counter* class is drawn “folded”. *Counter* is the class, and *counter1* is an instance of that class. *Counter* is instantiated, with *value* initially 3. Methods (and members) are drawn protruding from the object, and have their own life-lines. To call a method, a function-call is made on the life-line of the method. In Figure 4(b), the *increment* and *get* methods are called.



**Figure 4: Classes. (a) Defining a Counter class, (b) using the class.**

### 2.6 Modules and Libraries

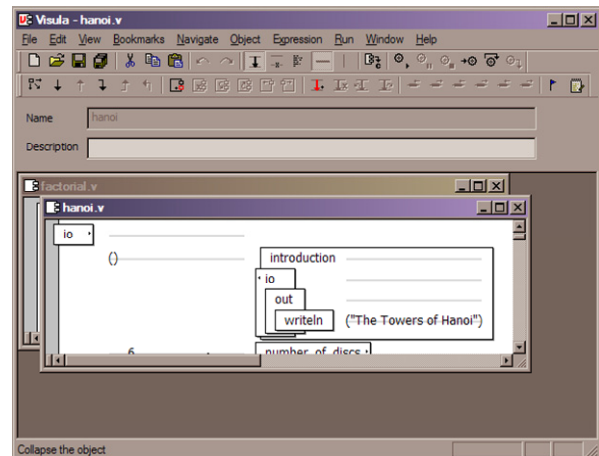
Visula programs can be split into modules. A module is imported into a program (or other module) by placing it on the left hand side of a program., like an input to a function.

Visula has a small core set of libraries, which can be readily extended using Visula's C interface. Data structures (including sets, maps, linked lists and arrays) are all provided in Visula's *std* library. There is no language support for these data structures, but they are provided via “containers” and “iterators”, much like the C++ standard library.

### 3. The Environment

Programs can be edited and debugged within an integrated development environment (IDE). This provides a multiple document interface, toolbars, an integrated visual debugger, a documentation generator and a C generator. The IDE is shown in Figure 9.

There are generally four ways of achieving an action: using a button on the toolbar, using a keyboard shortcut, using a pull-down menu, and using a pop-up (*context*) menu. The mouse cursor changes as it is moved over the diagram, indicating the mouse action.



**Figure 9: The IDE.**

Each file is edited in a different window. It is possible to split windows, and to have two windows open of the same file. The program can be dragged using the middle mouse button, scrolled using the mouse wheel or scrolled using the scroll bars.

In each window there is a currently selected item which is highlighted. The selected item can be the name of an object, or an expression. Details of the currently selected object appear in the toolbar. Each item has an annotation that can be viewed and edited in the toolbar. The annotation appears in "pop-up" text over the diagram when the mouse hovers over it.

The entire program can be navigated using the keyboard. The PgUp/PgDown/up/down keys select a different object (each horizontal line represents a unique object). Object names are edited directly in the diagram.

Boxes can be folded or unfolded to expand or collapse them. When a box is folded, a cross is drawn in its top-right. A single mouse-click folds or unfolds an object, so the entire program can be navigated as a tree.

Expressions are edited as text in the toolbar. When an expression is selected, it can be edited in the toolbar. Syntax errors are caught immediately.

Controls (if and while) are created by toolbar buttons to extend or retract the horizontal bar.

It is possible to step through program execution which shows a red box around the currently executing expression. A program data explorer allows all program data to be browsed.

## 4. Discussion

### 4.1 Cognitive Dimensions

It is difficult to objectively assess the merits of a programming language, however Cognitive Dimensions [1] offers a way to discuss the cognitive and usability aspects. The goal of VPLs is often to improve usability, so it is important to try to raise the performance of a VPL in as many dimensions as possible, and overcome the tradeoffs. If a notation performs badly in any one of these dimensions, then the product may be unusable overall.

*Abstraction* – Visula offers normal programming abstraction mechanisms, such as functions, modules, libraries, containers and object-orientation. Since Visula is general-purpose, Visula is more abstract than end-user programming languages.

*Hidden dependencies* – Visula has lower hidden dependencies than most programming languages. This is because there are no global names, and nested variables must be explicitly imported into a class or function. It is also clearer how (or whether) a variable is used in a block of code.

*Premature commitment* – lower than some programming languages. Since Visula is untyped, there is no need to define types and complete every method as would be necessary in a statically-typed language. The IDE has an “infinite undo” feature.

*Secondary notation* – less than most languages due to automatic layout. Name capitalization can be used.

*Viscosity* – lower than most textual and visual languages due to automatic layout. It is relatively straightforward to rearrange methods and statements using Ctrl+arrow keys. To rename a variable or function, it only needs to be renamed once on its lifeline, not once in every expression that uses it.

*Visibility* - Visula offers greater visibility compared with text since it is able to fold the diagram to give the view at an appropriate granularity. Visibility is a particular problem when dealing with loops in Prograph [5], since the body of the loop must be placed in a separate window.

*Closeness of mapping* – Visula does not have a close mapping to a particular domain, since it is general-purpose.

*Diffuseness* – Visula is not much more diffuse than text, though this is difficult to quantify. Diffuseness can be a particular problem for VPLs, for example VIPR's [6] concentric rings are very diffuse.

*Error-proneness* – lower than textual languages. Visula catches syntax errors immediately, so there is no picking through error messages.

*Consistency* - "pure" OO languages offer a greater consistency since everything behaves uniformly. For example there is no distinction between "int" and "System.Integer" as there is in Java. Visula is also consistent in its treatment of members of classes and functions. Modules behave as objects - there are no "using" or "import" keywords.

## 5. Conclusions

Visula offers certain cognitive and usability benefits, as can be seen from looking at its cognitive dimensions. Of course there are many other factors in selecting a programming language.

The purpose of new languages is not necessarily to replace C++/C#/Java/Python, but to explore different approaches and trade-offs. Visula's contribution is in adapting the widely-used UML sequence diagrams to make them executable, showing that the notation has usability benefits, and demonstrating how it can be edited effectively.

## 6. References

- [1] T.R. Green and M. Petre, “Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework”, *Journal of Visual Languages and Computing*, (7), 1996, pp. 131-174.
- [2] B.A. Myers, “Taxonomies of visual programming and program visualization”, *Journal of Visual Languages and Computing*, (1), 1990, pp. 97-123.
- [3] M.M. Burnett, M.J. Baker, C. Bohus, P. Carlson, S. Yang, and P. van Zee, “Scaling Up Visual Programming Languages”, *IEEE Computer* 28(3), 1995, pp. 45-54.
- [4] G. Booch, J. Rumbaugh and I. Jacobson, “Unified modeling language user guide”, 2nd Edition, Addison Wesley, 2005.
- [5] P. T. Cox, F. R. Giles and T. Pietrzykowski, “Prograph: a step towards liberating programming from textual conditioning”, *1989 IEEE Workshop on Visual Languages*, Rome, Italy, 1989, pp. 150-156.
- [6] W. Citrin, M. Doherty and B. Zorn, “Design of a completely visual object-oriented programming language.” In “*Visual object-oriented programming*”, Eds. M. Burnett, A. Goldberg and T. Lewis, Prentice Hall New York, 1995.