A visual perception account of programming languages : finding the natural science in the art

Stéphane Conversy

A Visual Perception Account of Programming Languages: Finding the Natural Science in the Art

Stéphane Conversy
Université de Toulouse - ENAC - IRIT
stephane.conversy@enac.fr

Abstract

There is no agreed set of grounded principles on which to rely to analyze and discuss code representations. I propose a combination of Semiotic of Graphics and ScanVis. I discovered that this unifying framework brings together many aspects of visual layout and appearance of programming languages. We describe how the framework applies to programming languages, which is not obvious and has never been done before. We show how to use the framework to compare representation of code by relying on sound arguments. Finally, we use the framework to devise design principles to help generate new representations. Relying on such a framework can help researchers and designers invent better languages with respect to this concern. This work also suggests that the gap between textual and graphical languages is narrow, and that true visual languages should rely on the capability of the human visual system.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

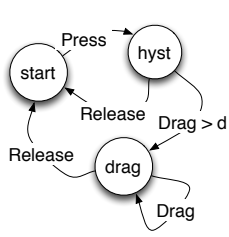
General Terms term1, term2

Keywords Code representation, Visual perception, Semiotic of graphics, Textual and Visual Programming Languages

1. Introduction

An implicit but important aspect of programming languages is that they must support the production of readable programs [1]: "Programs must be written for people to read, and only incidentally for machines to execute. [2]" Programmers read a program by looking at its 'code' i.e. the representation of the program on the screen, perceivable by the eyes of a human. Both textual and so-called 'visual' representations of programs on the screen employ various graphical 'features': texts, shapes, alignements, colors, arrows etc. (fig. 1). Those visual features are often considered as 'aesthetic sugar' that does not map to semantics (e.g. a colored representation of textual C program), but they can also be part of the syntax (e.g. indented textual Python code, arrows in state machines, colored Petri-nets).

As with any visual scene, the performances of programmers at reading textual or visual programs depend on their performances at perceiving the visual features presented on the screen. However, few works exist that help justify the choices made by language designers with respect to those features and performances. Instead, programmers and designers have to rely on 'aesthetics' or 'personal preferences' [1, 3] and specialists warn about possible 'danger of religious wars' when dealing with the topic [4]. The use of such terms is a sign that there may be a lack of foundation to manage the phenomenon of code perception, and how this may help or hinder programmers' performance.




```
CStateMachine sm = new CStateMachine(canvas) {
    CElement toMove = null;
    Point2D lastPoint = null;

    public State start = new State() {
        Transition press = new PressOnShape(BUTTON1, ">> hyst") {
            public void action() {
                toMove = getShape();
                lastPoint = getPoint(); }};
    public State hyst = new State() {
        Transition drag = new Drag(BUTTON1, ">> drag") {
            public boolean guard() {
                Point2D newPoint = getPoint();
                return !(newPoint.distance(lastPoint) < 25); }
            public void action() {
                toMove.translateBy(getPoint().difference(lastPoint));
                lastPoint = getPoint(); }};
        Transition release = new Release(BUTTON1, ">> start"); };
    public State drag = new State() {
        Transition stop = new Release(BUTTON1, ">> start") {
        Transition move = new Drag(BUTTON1) {
            public void action() {
                toMove.translateBy(getPoint().difference(lastPoint));
                lastPoint = getPoint(); }};};
```

Figure 1. Two representations of the same program using various graphical features.

This paper investigates the principles of programming languages that underpin the practice of code representation: we aim at finding the science in the art, instead of finding the art in the science as advocated in [3]. We propose a framework to describe, compare and generate visual representations of programming languages with respect to human perception. The expected benefits of this work would be to better understand the phenomenon of code perception, unify the concepts used in the literature, give accurate definitions to these concepts and help researchers and designers of programming languages invent more readable languages. For example, this work suggests that the gap between textual and graphical languages is narrow, and that true visual languages should rely on the capability of the human visual perceptual system.

This work focuses on the representation of 'a single page' of code. Though current trends in this area focus on the management and representation of large-scale programs, representation at the level of the page is overlooked. Even if visualisation of large-scale programs is an important problem, a programmer's understanding of a single page of code is still required since the very act of programming (i.e. editing code) is often done at this level. In addition, even if interaction with code is important [5, 6], this paper focuses on visual perception of code only.

2. The Proposed Framework

I propose to use a combination of Semiotics of Graphics and the ScanVis model as a framework to analyze various code representations and assess their perceivability. This section presents them briefly.

2.1 Semiotic of graphics

Semiotic of graphics is a theory of abstract drawings (i.e. drawings that do not imitate a natural scene) such as maps and bar charts [7]. A part of this theory describes and explains the perceptual phenomena and properties underlying the act of visualizing 2D abstract drawings. Semiotic of graphics relies on the characterization of data to be represented (the data type: nominal, ordered, and quantitative), and the perceptual properties of the visual variables used in a drawing, such as color or shape.

Drawings are a set of 2D 'marks' (a point, a line or a zone) lying over a background. Marks vary according to visual variables such as X and Y position ('planar' variables), shape, color, luminosity, size, orientation [7], enclosures and lines that link two marks [8]. Visual variables are characterized by their perceptual properties, and can be: Selective - enables a viewer to assimilate and differentiate marks instantaneously (e.g. all red marks) (fig. 2); Ordered - enables a viewer to rank marks perceptually (e.g. from light to dark) (fig. 3); Quantitative - enables a viewer to quantify differences between marks perceptually (e.g. twice as large) (fig. 3).

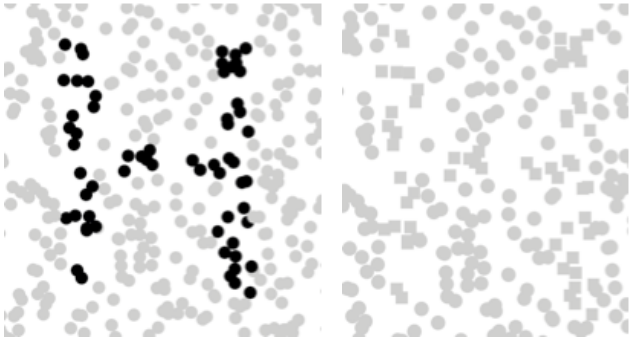


Figure 2. by default all marks are circles and light. (left) Some marks are dark to produce an H. Luminosity is selective: the H letter emerges because the eye discriminates two groups of marks (light and dark) instantaneously. (right) Some marks are square to produce the same H. Shape is not selective: the H letter does not emerge because the eye does not discriminate groups instantanously when marks differ by their shape only.

All visual variables but shape and link are selective (fig. 2). All visual variables but shape, link and color are ordered (colors are ordered in a limited spectrum only). X and Y position, angle, length, size are quantitative, to various degrees as experimentally evidenced [9]. The performance of readers at selecting, ordering or quantifying depends on the number of values differentiable by them (e.g. 5 levels of luminosity for selection, 20 levels of luminosity for order), the difference between each value (the less the worse) and the spatial distance between marks (the more the worse).

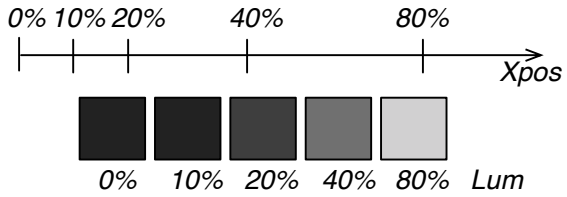


Figure 3. (top) Position is quantitative: one can perceive the ratio and the difference between various X positions (pos. 80% is 2x pos. 40% and 4x pos. 20%); (bottom) Luminosity is ordered but cannot be perceived quantitatively.

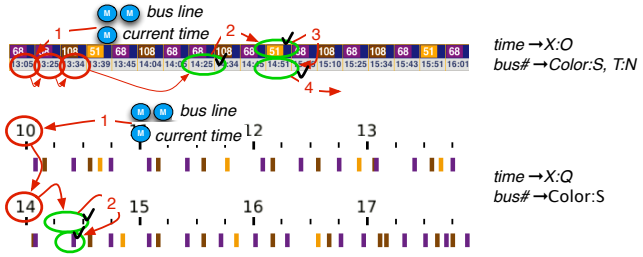


Figure 4. Two visualization and visual operations for the task "find how long I have to wait for the next bus." [6]

Fig. 4 shows two representations of a bus schedule, overlaid with the arrows and circles depicting a visual scanning required to fulfill a task (see below). The figure also contains the mapping between data and visual variables (e.g. '$time \rightarrow X : O$' means that the data 'time' is mapped to the X visual variable in an Ordered manner). An efficient representation (with respect to a task, see below) enables the reader to instantaneously perceive patterns, trends, correlations or outliers inside the totality of marks (global reading e.g "when are the occurrences of busses?" Fig. 4, bottom), or in a subset of marks (partial reading e.g. "when are the occurrences of yellow busses between 4 and 5pm?" in fig. 4, top). A bad representation only allows the reader to perform elementary reading: the reader is required to focus on marks one by one, perceive the visual variables and retrieve the data. In this case, the representation is not better than a mere display of texts representing data.

2.2 ScanVis

ScanVis is a descriptive model that helps designers analyze how visualizations are read [6] and assess their effectiveness with respect to a reading task. ScanVis relies on the decomposition of visualization tasks into elementary visual operations: memorizing information, entering into and exiting from the representation, seeking a subset of marks, seeking and navigating among a subset of marks, unpacking a mark and verifying a predicate. Given a task, one can infer the required sequence of visual operations to accomplish this task. Fig. 4 shows the required elementary visual operations with circles and arrows for a particular task (more information in [6]). Operations are facilitated by the use of adequate visual cues, such as selective visual variables (color, size or alignment).

3. Application to Programming Languages

Semiotics of Graphics and ScanVis have already been applied to graphs, charts or visualizations of information. The remainder of this paper is devoted to their application to programming languages. Though this seems a rather obvious task (a mere application of an existing framework), it is not: reducing visual features of programming languages to the concepts of the framework proved to be difficult and required maturing ideas for a long time.

In order to help the reader assess the significance of the proposed framework, we will describe its descriptive power (what are the phenomena that the framework capture?), its comparative power (how can it help assess or compare particular code representations?) and its generative power (how can it help explore the design space of code representation?).

As seen above, assessing a particular visual representation of a program requires identifying the set of reading tasks performed by the programmer. In the following, I present a number of visual representations together with tasks. Since reading tasks have seldom been clearly stated in the literature, I had to devise them by trying to justify interesting aspects of the representations. I assume that the tasks are the right ones, but I do not claim perfect validity

on this area, and the reader of this article may disagree with them. Actually, an outcome of this paper is to initiate a work on the elicitation of reading tasks. The tasks presented here are a first step in this direction.

4. Describing Visual Features Of Languages

The goal of this section is to explore how popular sayings about code can be appropriately deconstructed and translated to the concepts and vocabulary (in italic) of the framework. If correct and comprehensive, such translation is the first step toward the validation of the significance of the framework.

4.1 Visually structuring the code

"Lots of Irritating Superfluous Parenthesis." Lisp uses parenthesis to structure code. Lists are designated with spaced expressions surrounded by an opening and a closing parenthesis. Function composition uses compound parenthesizing.

```
(defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))
(defun fac ◁n▷ ◁if ◇<= n 1◇ 1 ◇* n ♡fac ☞- n 1☜♡◇▷)
```

Figure 5. Delimiters varying in shape. Top: parenthesis, bottom: other shapes.

Lisp is reputed difficult to read ([10] p65). This difficulty comes partly from the fact that list boundaries are depicted with two *shapes* (opening and closing parentheses, see fig. 5, top) and prevents fulfilling the task "figure out the [lisp] expressions" efficiently. This can be explained with the *selectivity* concept: since shape is *non-selective*, this prevents from seeing Lisp expression boundaries at a single glance, and forces the programmer to *scan* the code linearly to find them out. The bottom line of fig. 5 uses a unique boundary shape according to the level of depth of enclosure. This doesn't work better and also illustrates *non-selectiveness*, especially with multiple symbols: for example, it is difficult to find the diamond closing the multiply expression.

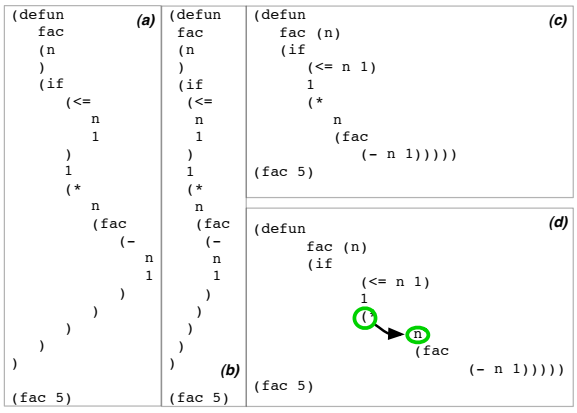


Figure 6. Delimiters varying in using X, a selective visual variable (a.1). A smaller difference between X values hinders selection (a.2); improving X selectiveness by shortening positionnal distances (b.1) or with larger indentation, at the expense of visual scanning, depicted with circles and arrows .(b.2).

"Indentation makes structure obvious." In fig. 6 (a) the level of depth is mapped to the *X visual variable*. Matching parentheses are "vertically aligned", which is another way of expressing an *assimilation of X values*. Since X is *selective*, the perception of expression boundaries is better than using a shape. Selectivity depends on the *amount of difference* between values: shrinking the size of the indentation lowers the selective ability of X (b). Reserving a line for a closing parenthesis alone lengthens the Y *spatial distance* with the matching opening parenthesis and weakens the selective property of the X visual variable (i.e. it is difficult to perceive vertical alignment) (a and b); ignoring the parenthesis matching problem altogether shortens the distances and improves X selectivity (c); more indentation improves selective perception of X (d). However, the assumed improvement is supposedly done at the expense of a longer *scanning* to go from the beginning of a block to its first instruction, as experimentally assessed in [20]. Note that since X is ordered, such a representation also facilitates the task "figure out the hierarchy of expressions".

"LabView's G language is intuitive." G mixes large boxes that *enclose* other objects to specify a hierarchical structure, and *links* that connect the components inside and outside boxes (see [24] for more details). Enclosures may be "intuitive", but a more appropriate qualification is that they are *selective*: one can grasp in a single glance the elements that are part of a parent. They are also *ordered* and help perceive a containment hierarchy.

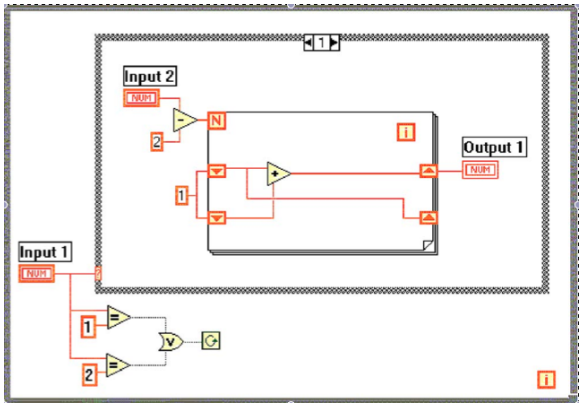


Figure 7. G language from LabView.

```
extern void printf(char*,...);

int fact(int n);

int main(int argc, char* argv[]) {
  int res = fact(argv[1]);
  printf("fact %s: %s\n",argv[1],res);
  return 0;
}

int fact(int n) {
  int res=1;
  while (n) {
    res *= n;
    n-=1;
  }
  return res;
}
```

function decl → Y:O
function def → Y:N
instruction flow → Y:O
loop/branch/jump → Text:N

call function → Text:N
block → X:O
block → shape(X,Y)'{''}':N

Figure 8. C language classification.

"Syntax highlighting improves readability." Fig. 9 shows an existing 'syntax-colored' textual representation of Java code with the NetBeans editor. Blue glyphs correspond to reserved keyword of the Java language, and gray ones to comments. A yellow background corresponds to a variable at which the mouse pointer points.

Coloring all appearances of a variable the mouse is pointing at does make sense with a programming task point of view: this enables the programmer to efficiently grasp all occurrences of this variable thanks to selectivity. Similarly, adding a colored background to a brace enables the programmer to quickly see where the corresponding one is and assess the scope of a block. The gray color is lighter than the other ones: since luminosity is selective, this enables user to rapidly assimilate and differentiate code from comments, and rapidly navigate between sections of the code. Hence

```
// replicable pseudo random generator
Random rpos = new Random(456);
Random r = new Random(321);
double[] sizes = new double[6];
double a = 10, b = 5;
for (int i = 0; i < sizes.length; ++i) {
   sizes[i] = a * i + b;
}
float[] tricol = new float[3], rgb, lch;
lch = tricol;
lch[0] = 40;
lch[1] = 100;
lch[2] = 45;
Color c1 = srgb.fromLCHtoColor(lch);
[…]
System.out.println("[debug] color is"+c1);
// compute each symbol hue
for (int i=0; i<hue_symbol.length(); ++i) {
   lch[2] = (float)(i*360./hue_symbol.length());
   colors[i] = srgb.fromLCHtoColor(lch);
   hue shapes.add(buildShape(g, hue symbol.substring(i,i+1), w, h));
}
```

Figure 9. Colored editor.

this removes the need to scan the first letters of a line to check if it begins with two slashes, a much more demanding visual task since shape ('//') is not selective. In addition, the order of luminosity indicates an order of importance between code, comments, and background.

4.2 Understanding instructions flow

"The instruction flow in C is implicit." In a block of C instructions, a sequence of texts separated by semi-columns denotes a sequence of instructions. Often, instructions are put together with one line for each instruction. In this case, the Y visual variable is used in an *ordered* manner, and maps the ordered sequence of the program counter. This makes the programmer visualize the evolution of the program counter path by *scanning* the textual instructions vertically. Thus, the task "given an particular instruction, what is the next instruction to be executed?" is efficiently supported by the representation since it uses a *selective*, *ordered* variable. As such, the instruction flow is depicted explicitly (and not implicitly) with the X visual variable. Calls to function or gotos are a different story since they are done by name (a *shape*), a visual variable that is not selective. In order to fulfill the same task from a call instruction ("next instruction?"), the user has to *scan* the representation and *seek* the answer.

"Arrows make the instruction flow explicit" Fig. 10 shows a circle-and-arrow description of a Drag'n'Drop interaction with hysteresis [11]. There are three states ('start', 'hyst' and 'drag'), one transition from state 'start', and two from states 'hyst' and 'drag'. The instruction flow is entirely depicted with arrows. To fulfill the task "figure out the flow", a reader must *seek a subset of marks* (*links*) and *navigate* visually by following arrows, which may be slow especially when arrows are numerous.

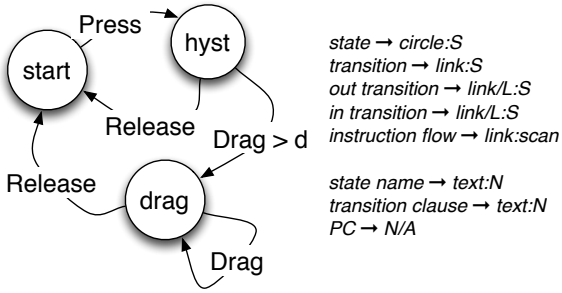


Figure 10. Box-and-arrow representation of a state machine.

Code Bubble is an IDE that presents code with function snippets inside individual window resembling 'bubbles' [12]. Users can juxtapose bubbles that contain related code. One use is to display the code of a callee into a bubble at the right of a bubble containing the caller. Hovering over a bubble highlights the connections and code lines that lead to it by changing the color or luminosity of the links. This turns a *non-selective* variable (link) to *selective* (colored link) and helps users figure out the flow and navigate between instructions that belong to related functions.

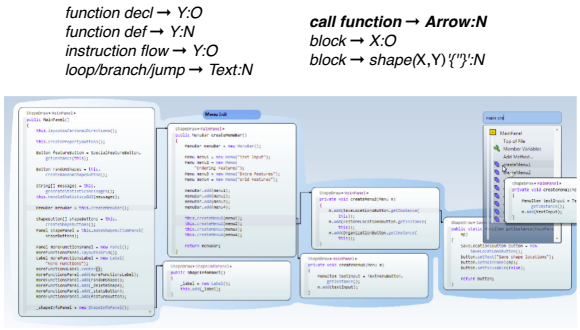


Figure 11. Code Bubble classification.

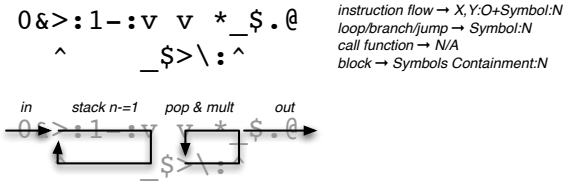


Figure 12. Factorial in befunge (top); explanation of the flow (bottom).

"Befunge is an esoteric language" Befunge is a 2D textual language that can be considered a missing link between textual and visual languages, instead of esoteric. In Befunge, the flow is indicated by the four *shapes* <, >, ^, v, which resemble to pointing arrows in the four cardinal directions. Branching is specified by - (equivalent to < if the condition is true and to > otherwise) and | (equivalent to ^ if the condition is true and to v otherwise). However, not only the flow is not graspable at once (only real arrows and links help a little in the bottom part of the picture), but also directional shapes are not *selective* and cannot be perceived instantaneously.

4.3 Understanding functionality: "Icons are easier to use than texts"

Icons are often considered easier to interpret than text. Fig. 8 in [26] illustrates the use of 'analog' [26] iconic vocabulary in LabView's control-flow structures. In this case, icons are differentiated with *shapes* (arrowheads, page corners, spirals, 'N', 'I'). When icons vary according to shape only (a non-selective variable), one can only perform a slow *elementary reading* on a scene. In other visual languages, icons vary in shape but also along other visual variables, which may turn them selective. For example, the third line of fig. 13 uses a set of shapes with which *selection* and *ordering* seem to 'work': this is because they do not contain the same number of pixels and exhibit different levels of *luminosity*, a selective variable.

5. Comparing Code Representations

This section shows how the framework helps compare code representations.

5.1 Luminosity, color and position of enclosing symbols

In fig. 19, the first line maps level of depth with unique colors. Since color is selective, this enables the reader to *assimilate* at one glance

all parentheses with the same level of depth. However, one has to wonder if the task "assimilate level of depth" is worth facilitating: even if a reader correctly detects each opening and closing parenthesis, one has to remember the discovered structure to make sense of it. If an appropriate visual variable was used instead, the programmer could use it as an externalization of memory to recall the structure by accessing it immediately, in one glance. For example, the fourth line uses luminosity alone. Luminosity is *selective* (and helps match parentheses), and *ordered* (helps perceive the relative depth). One cannot perceive the exact depth since as opposed to X luminosity is not *quantitative*. But ordering may be sufficient for the task at hand.

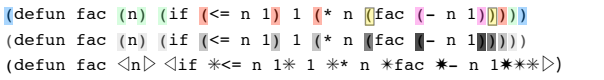


Figure 13. Delimiters varying in hue, luminosity, and shape+luminosity.

5.2 Y versus Arrows

As we have seen above, instruction flow can be depicted using Y or links and arrows. Links and arrows are not selective visual variables: a reader is forced to follow the chain of links to figure out the flow (fig. 14-a). This can be supplemented with alignment cues, i.e. using the selective property of Y. In this case, the visualization is equivalent to indented code in a C program (fig. 14-b). Everything is as if an arrow that would connect two following C instructions is actually not showed because it would be redundant with the Y:O representation (fig. 14-c) Nonetheless, keeping an arrow for the loop would help the reader *scan* up to the beginning of a loop, similarly to box-and-arrow languages. Scratch [13] is a visual language with connectors on blocks suggesting how they should be put together. The connectors are similar to the arrows: they guide a reader following the instructions sequence (fig. 14-d). The start of the loop can be perceived selectively with color and containment. These examples show how different representations can be unified with the same underlying principles.

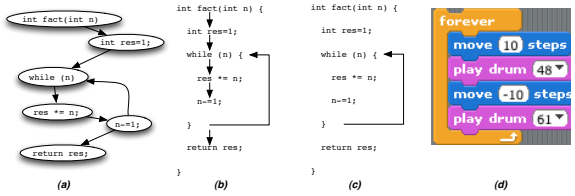


Figure 14. Arrows could have been used in C (b), as in box-and-arrow languages (a). Since arrows are redundant the ordered Y visual variable, they can be removed, except for the loop (c). Scratch uses similar visual variables (d).

As seen above, arrows are often said to be an explicit representation of the instructions sequence. To be more precise, they are an explicit representation of the direction of the sequence. However, they are no more explicit on the order of the sequence than Y, since the selective visual variable Y explicitly shows it already (both in the C version and the scratch version).

5.3 Comparing with multiple, more demanding tasks

Code representations are used to fulfill multiple reading tasks. When comparing them, one should enumerate a realistic set of tasks and assess how each representation rate with respect to each task. Fig. 15 shows the SwingState code describing the same Drag'n'drop interaction as in Figure 10. [11]. SwingStates is a

```
CStateMachine sm = new CStateMachine(canvas) {
    CElement toMove = null;
    Point2D lastPoint = null;

    public State start = new State() {
        Transition press = new PressOnShape(BUTTON1, ">> hyst") {
            public void action() {
                toMove = getShape();
                lastPoint = getPoint(); }}};
    public State hyst = new State() {
        Transition drag = new Drag(BUTTON1, ">> drag") {
            public boolean guard() {
                Point2D newPoint = getPoint();
                return !(newPoint.distance(lastPoint) < 25); }
            public void action() {
                toMove.translateBy(getPoint().difference(lastPoint));
                lastPoint = getPoint(); }};
        Transition release = new Release(BUTTON1, ">> start"); };
    public State drag = new State() {
        Transition stop = new Release(BUTTON1, ">> start") {
        Transition move = new Drag(BUTTON1) {
            public void action() {
                toMove.translateBy(getPoint().difference(lastPoint));
                lastPoint = getPoint(); }};};};
```

state → X:S?
transition → X:S?
out transition → text:N
in transition → text:N
next state → text:N

state name → text:N
transition clause → X:S
instruction flow → Y:O

Figure 15. 1D Text representation of the state machine.

textual language for describing state machines [14]. SwingStates relies on the Java reflexive facility to be embedded seamlessly in regular java code. The code is indented to facilitate perception of the states, the transitions from each state, and the clauses associated to the transitions.

Fig. 16 compares the visual operations required for the task "what are the out transitions for a particular state?" In both representations, readers have to seek and navigate among states until they find the right one, and find and seek all transitions from this state. With circles-and-arrows, one can consider that large white circles are selective compared to other marks (because of their size and luminosity). With SwingStates code, the indentation is also selective. Hence both representations help seek a subset of marks. Finding 'out' transition is more efficient in SwingStates code since all transitions are out transitions. With circle-and-arrows, one has to differentiate between links with and without arrowhead laid around the circles. Links without arrowhead may be more difficult to differentiate than other marks.

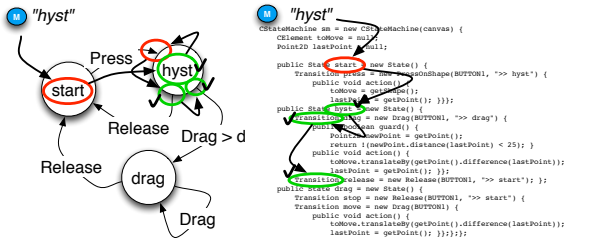


Figure 16. ScanVis for the task: "what are the out transitions for state 'hyst'?" .

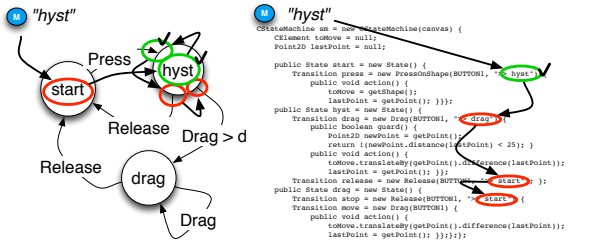


Figure 17. ScanVis for the task: "what are the in transitions of state 'hyst'?"

Fig. 17 illustrates the visual operations required for the task 'what are the in transitions for a particular state?" For the circle-and-arrows representation, the operations are almost similar to the operations required in the previous task. Finding the 'in' transition may be facilitated by the fact that arrowheads are dark and thus

selective. With the SwingState code, the visual operations are very different: one has to find the name of the states inside a transition. Most of those names are on the right part of the code, which helps seek and find them. Still, since they are texts, it may be difficult to navigate without any risk of missing one.

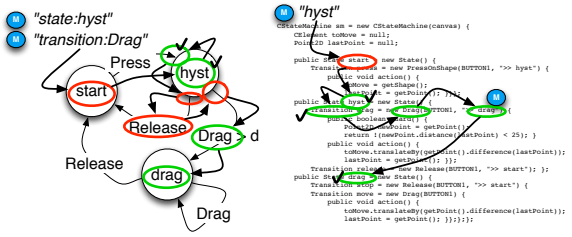


Figure 18. ScanVis for task: "go to the state following a transition 'Drag' on state 'Hyst'."

Fig. 18 illustrates the visual operations required for the task 'go to the state following a transition 'Drag' on state 'Hyst' ". One has to find the first state, find the transition, and go to the state following this transition. After having found the transition, it may be easier to follow a link in the case of circle-and-arrows than find a text (the name of the next state) in the case of the SwingState code.

6. Generating New Code Representations

This section introduces a number of design principles to make new code representations emerge and illustrates them with a number of examples. I devised the design principles by examining how existing representations improve over former ones.

Identify the task and seek selectivity only if needed. In the colored code editor fig. 9, using color for all keywords may not be related to any task the programmer should accomplish (e.g. find all 'for' loops). Of course, one can argue that it helps assess that a keyword has been recognized as the user types it, and that no lexical error has been made. Nonetheless, fulfilling this task does not require a selective variable such as color. One should fall down to elementary reading instead, by using a non-selective variable such as a shape, or a typeface e.g. 'unrecognized' in 'italic', 'recognized' in 'regular'. This would reserve color, a scarce resource, for a more efficient use.

Try swapping visual variables. One way to generate representations is to explore the design space by swapping visual variables for more efficient ones. Fig. 19 illustrates alternative representations using size and Y position as visual variables instead of color. Since these visual variables are selective and ordered, they help the reader visualize the structure of the code, similarly to the more traditional use of the X visual variable (indentation).

```
(defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))
(defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))
```

Figure 19. Using size and Y as visual variables.

Shorten spatial distance. As said previously, reducing spatial distance may improve selectivity. Fig. 6 b.1 shows a representation that shortens spatial distance, but one cannot match parentheses anymore, which can be annoying when trying to add a missing parenthesis. Fig. 20 is a representation that shortens the spatial distance while keeping easy to match parentheses. Remarkably, if the parentheses match perceptually according to the X visual variable, they do not match conceptually: for example, the opening parenthesis at the beginning of the 'defun' function conceptually matches with the rightmost closing parenthesis on the penultimate line of the code, but is aligned with the closing parenthesis of the call to factorial. This illustrates that perception can prevail over the conceptual model, as long as semantic is preserved.

```
(defun
   factorial (n)
   (if
      (<= n 1)
      1
      (*
         n
         (factorial
            (- n 1)
)  )  )  )

(factorial 5)
```

Figure 20. Shortening spatial distance: parentheses match perceptually, but do not match conceptually.

Another representation that reduces distance is shown in Fig. 21. With a debugger, the user can step inside the call of a function. This can be made with a toggle arrow: when toggled, the code of the function unfolds under the call of the function. A similar feature could be used for static code: this would help understand how functions compose without the need to memorize the code surrounding the call of a function and switch visually between the distant representations of the two functions.

```
(defun                          (defun
   factorial (n)                   factorial (n)
   (if                             (if
      (<= n 1)                        (<= n 1)
      1                               1
      (*                              (*
         n                               n
       ▶(factorial (- n 1)))))         ▼(factorial (- n 1)))))
                                        (if
(factorial 5)                              (<= n 1)
                                           1
                                           (*
                                              n
                                            ▶(factorial (- n 1)))))

                                (factorial 5)
```

Figure 21. 'Debugger view' of code.

An asset of such a 'tree-view' is that it shortens the distance between instructions before the call and instructions at the beginning of the function being called. However, this also expands the distance between the instructions before the call and after the call, especially when multiple functions are deployed. A 'browser' view ala SmallTalk can help see details and contexts of the call i.e. shorten both the distances (Fig. 22). In fact, Code Bubble can be seen as a tentative to shorten distance between calling and called code. The previous examples are variations on this idea that do not need arrows, and may thus be more efficient.

```
                                  ((eq (car e) 'quote) (cadr   e))
                        ((atom e) (assoc. e a)) ((eq (car e) 'atom)  (atom   (eval. (cadr e) a)))
(defun eval. (e a) ▶(cond▶((atom (car e)) (cond ▶((eq (car e) 'eq)    (eq     (eval. (cadr e) a)   (eval. (caddr e) a)))
                                                ((eq (car e) 'car)   (car    (eval. (cadr e) a)))
                                                ((eq (car e) 'cdr)   (cdr    (eval. (cadr e) a)))
                                                ((eq (car e) 'cons)  (cons   (eval. (cadr e) a)   (eval. (caddr e) a)))
                                                ((eq (car e) 'cond)  (evcon. (cdr    e) a))
                                                ('t                  (eval.  (cons  (assoc. (car e) a) (cdr e)) a
```

Figure 22. Browser view' of the lisp 'eval' function.

Explore and leverage off properties of visual variables. In a typical imperative language, Y is used in an ordered manner only. Since the distance between instructions has no meaning, a representation could vary distances to misalign statements and align synchronization statement only. In Fig. 23-left, selectivity of the Y variable helps see at a glance the synchronization points, and removes false information conveyed by perfectly aligned statements. Distance can also be used to convey quantity. Fig. 23-right illustrates

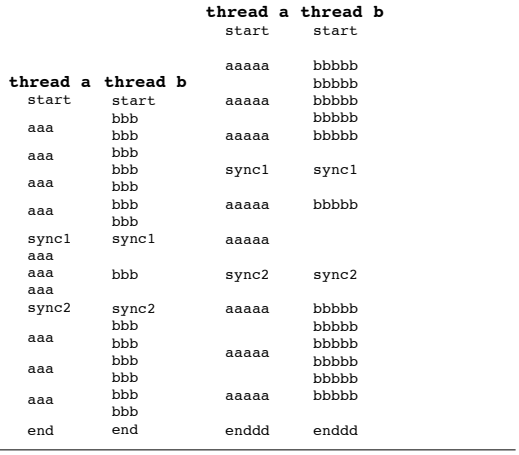


Figure 23. Left: Y used as a selective variable: instructions are aligned when synchronized, and misalign when not synchronized. Right: Y used as a quantitative variable: the number of cycles is mapped to the distance between instructions (aaaaa: 2cycles, bbbbb: 1 cycle).

a representation that uses Y as a quantitative variable to depict two concurrent sequences of instructions. The number of cycles taken by each instruction is mapped to the Y dimension. The larger the space after an instruction, the larger the number of cycles it takes to execute it. This gives a sense of the time spent on some parts of code, and can help balance the instructions in order to minimize wasted cycles while waiting for the concurrent process when synchronization is needed.

7. Threats To Validity

The proposed framework relies on models, and as such it is also a simplification of the reality. Even if the framework allows us to describe a number of the perceptual phenomena underlying the perception of code, some of the phenomena may not have been identified because of the limited capability of the framework, or because their explanation or cause is different (hammer and nail problem). Visual perception is complex, and some of the visual operations may be bypassed because of specific conditions (layout, number of items involved). Furthermore, code representation is not the only factor that contributes to program understanding. Other cognitive factors, such as learning, expertise, API usability and documentation [19] contribute to program understanding, and may influence the way the user perceives or scans the code.

The aim of the examples is not to convince the reader of this paper that the analyses are right or wrong. Nonetheless, most of the visual features of programming language implicitly assume the kind of phenomena described here. Again, the framework serves as a rationale tool for language design. By using common concepts and vocabulary, designers of language can argue precisely why they think that a particular feature is or is not appropriate. This is similar to the objectives of the cognitive dimension of notations [5].

8. Related Work

Reading code is a complex process that involves many aspects. I have selected a number of works that address formatting, the performances at reading, the difference between textual and visual languages, and the frameworks to analyze them.

8.1 Formatting and pretty-printing

In order to facilitate reading, 'formatting well' is often advised and discussed in early fundamental papers about programming languages even if not visual (see the discussion in [15]): "Code formatting is about communication, and communication is the professional developer's first order of business" [4]. In a recent work, formatting is still associated to an 'art' [3]. Actually, the problem of program representation is well beyond mere code formatting, too narrow a concept that refers to the more general problem of the visual perception of the code presented to the programmer through language editors.

8.2 Performances at reading programming languages

A number of visual designs were proposed to improve reading performances [16–19]. The indentation length has been experimentally shown to have an impact on the comprehension of code: 2- and 4-spaces indentation makes readers better at understand the code than 6-spaces indentation, be they readers novice or expert [20]. Eye tracking has been used to observe programmers but only to measure switching between a view on the code and a view presenting an animated algorithm [21].

Moher et al. observed that "performance was strongly dependent to the layout of the Petri nets. In general, the results indicate that the efficiency of a graphical program representation is not only task-specific, but also highly sensitive to seemingly ancillary issues such as layout and the degree of factoring" [22]. Green et al. found that textual representations outperformed LabView for each and every subject [23]. Their explanation is that 'the structure of the graphics in the visual programs is, 'paradoxically', harder to scan than in the text version". LabView and its G language have been studied "in the wild" [25]. Respondents declared that G is easier to read than textual programming languages. G is supposed to provide an overview (a gestalt view) and clarifies the structure. However, they also say that it is very easy to create messy, cluttered, hard to read spaghetti code and that sequence structures tend to be cryptic or obscure.

8.3 Differences between textual and visual languages

Researchers have already wondered where the actual differences between textual and visual languages lie. In [26] Petre argues that the differences in effectiveness between TL and VL "lie not so much in the textual-visual distinction as in the degree to which specific representations support the conventions experts expect". As Petre noticed, programmers can find gestalt patterns in textual representation [26].

Much of 'what contributes to comprehensibility of a graphical representation is not part of the formal notation but a 'secondary notation': layout, typographical cues and graphical enhancements."[26]. Petre adds that "the secondary notations [e.g. layout] are subject to individual skills (i.e. learned ones) and make the difference between novices and experts. What is required in addition is good use of secondary notation, which like 'good design' is subject to personal style and individual skill" [26]. I take an alternative point of view: I argue here that even if skills can be learned, the basic visual capability of humans is enough to explain much of the easiness or difficulty of programmers to decipher a program.

8.4 Analysis frameworks

There have been attempts at building a metric for software readability. In the metric from [27], a few features can be considered perceptual (comma, spaces, indentation), but most are based on the semantic of the texts. The 'cognitive dimensions of notation (CDN)' is a framework that helps designers analyze interactive tools, including programming environments and languages [5]. CDN dimensions targets the cognitive and interactive aspects as opposed to

the perceptive aspects: the graphics and perception concerns are addressed partly in the secondary notation and visibility dimensions. Gestalt is a well-known framework that explains the phenomena underlying pattern perception. Gestalt can be used to explain how programmers may perceive patterns in their code, but I found that Gestalt could not report about all perception phenomena. So-called pre-attentive features also have a role in the perception of code [28]. Semiotic of Graphics subsumes pre-attentive features, and addresses other levels of perception than Gestalt.

Physics of Notations focus on the perceptual properties of notations [29]. Though using similar frameworks than mines, the level of perception addressed is higher and oriented towards guidelines. My work offers a finer grain analysis of code perception together with operational design principles.

9. Conclusion

Pretending that a framework is not only descriptive and comparative, but also generative is a strong claim. In particular, a precise and detailed method to describe, compare and generate code representations is still missing. Still, I exposed some elements that can be reused by designers, in particular a way to compare representations and operational principles to explore the design space. One can wonder why a particular account of a phenomenon would be of any interest if it's not fully operational. However, a framework often takes years to be appropriated and to be developed (e.g. cognitive dimensions). Even a this stage, a unifying framework that relies on a set of consistent concepts may lead to important discoveries and insights.

For example, such a framework explains why the boundaries between what we used to consider textual and visual languages blur. Indeed, the assets or flaws traditionally associated to a particular mean of presenting code are at stake: most of the textual languages are displayed using planar variables and thus may use the perceptual system efficiently, while so-called visual languages may use visual variables (icons, links) quite inefficiently. Actually, except maybe for lambda-calculus with no spaces, all practical languages are graphical. A true visual language would then be a graphical language that actually leverages the perceptual system. When designing a true visual language, the designer should think with the tasks in mind. This is not a linear process, where the designer would first devise the task, then find a visual representation. Rather, it is an iterative process: devising reading tasks and finding a visual representation contribute to each other, and the process is a refinement of both the problem (task) and the solution (visual representation).

References

[1] Raymond, D.. 1991. Reading source code. In Proc. of the 1991 conference of the Centre for Advanced Studies on Collaborative research (CASCON '91), Ann Gawam, Jan K. Pachl, Jacob Slonim, and Anne Stilman (Eds.). IBM Press 3-16.

[2] Abelson, H. and Sussman, G. 1996. Structure and Interpretation of Computer Programs (2nd ed.). MIT Press.

[3] Green, R. and Ledgard, H. 2011. Coding guidelines: finding the art in the science. Commun. ACM 54, 12 (December 2011), 57-63.

[4] McConnell, S. 2004. Code Complete, 2nd edition. Microsoft Press.

[5] Green, T. R. G. (1989) Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay (Eds.) People and Computers V. Cambridge, UK: Cambridge University Press, pp 443-460.

[6] Conversy, S., Chatty, S., Hurter, C. Visual Scanning as a Reference Framework for Interactive Representation Design. In Information Visualization, 10, pages 196-211. Sage, 2011.

[7] Bertin, J. (1967) Sémiologie Graphique - Les diagrammes - les réseaux - les cartes. Gauthier-Villars et Mouton & Cie, Paris. Rééd. 1997, EHESS.

[8] Card, S.K., Mackinlay, J.D., Shneiderman, B., Readings in Information Visualization: Using Vision to Think. San Francisco, California: Morgan-Kaufmann, (1999).

[9] Cleveland, W., McGill, R., Graphical Perception and Graphical Methods for Analyzing Scientific Data. Science, New Series, Vol. 229, No. 4716 (Aug. 30, 1985), pp. 828-833.

[10] Steele, G. and Gabriel, R. 1996. The evolution of Lisp. InăHistory of programming languages—II, Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. (Eds.). ACM, New York, NY, USA 233-330.

[11] Conversy, S. Improving Usability of Interactive Graphics Specification and Implementation with Picking Views and Inverse Transformations. In Proc. of VL/HCC, pages 153-160. IEEE, 2011.

[12] Bragdon, A., Zeleznik, R., Reiss, S., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F. and LaViola, J. 2010. Code bubbles: a working set-based interface for code understanding and maintenance. In Proc. of CHI '10, ACM, 2503-2512.

[13] Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B. and Kafai, Y. 2009. Scratch: programming for all. Commun. ACM, 52, 11 (Nov), 60-67.

[14] C. Appert and M. Beaudouin-Lafon. (2008). SwingStates: Adding state machines to Java and the Swing toolkit. Journal Software Practice and Experience. 38, 11 (Sep. 2008), 1149-1182.

[15] P. J. Landin. 1966. The next 700 programming languages. Commun. ACM 9, 3 (March 1966), 157-166.

[16] Clifton, M. 1978. A technique for making structured programs more readable. SIGPLAN Not. 13, 4 (April 1978), 58-63.

[17] Crider, J. 1978. Structured formatting of Pascal programs. SIGPLAN Not. 13, 11 (November 1978), 15-22.

[18] Ramsdell, J. 1979. Prettyprinting structured programs with connector lines. SIGPLAN Not. 14, 9 (September 1979), 74-75

[19] T. Tenny. 1988. Program Readability: Procedures Versus Comments. IEEE Trans. Softw. Eng. 14, 9 (September 1988), 1271-1279.

[20] Miara, R., Musselman, Navarro, J. and Shneiderman, B. 1983. Program indentation and comprehensibility. Commun. ACM 26, 11 (November 1983), 861-867.

[21] Bednarik, R. and Tukiainen, M. 2006. An eye-tracking methodology for characterizing program comprehension processes. In Proc. of the 2006 symp. on Eye tracking research & applications (ETRA '06). ACM, New York, NY, USA, 125-132.

[22] Moher, T.G., Mak, D.C., Blumenthal, B., and Leventhal, L.M. Comparing the comprehensibility of textual and graphical programs: The case of Petri nets. In Empirical Studies of Programmers: 5th Workshop. Ablex, 1993, 137-161.

[23] T. R. G. Green & M. Petre (1992) When visual programs are harder to read than textual programs. Proc. of the 6th European Conference on Cognitive Ergonomics (ECCE 6), pp. 167-180.

[24] Whitley, K., Novick, L. and Fisher, D. 2006. Evidence in favor of visual representation for the dataflow paradigm: An experiment testing LabVIEW's comprehensibility. Int. J. Hum.-Comput. Stud. 64, 4 (April 2006), 281-303.

[25] Whitley, K. and Blackwell, A. Visual Programming in the Wild: A Survey of LabVIEW Programmers', Journal of Visual Languages & Computing, 12(4), Aug. 2001, p435-472.

[26] Petre, M. 1995. Why looking isn't always seeing: readership skills and graphical programming. Commun. ACM 38, 6 (June 1995), 33-44.

[27] Buse, R. and Weimer, W. 2008. A metric for software readability. In Proc. of the intern. symp. on Software testing and analysis (ISSTA '08). ACM, 121-130.

[28] Treisman, A. (1982). Perceptual grouping and attention in visual search for features and for objects. Journal of Experimental Psychology Human Perception and Performance, 8(2), 194-214.

[29] Moody, D. 2009. The 'Physics' of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. IEEE Trans. Softw. Eng. 35, 6 (November 2009), 756-779.