Existe-t-il une différence entre langages visuels et textuels en termes de perception ?

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

RÉSUMÉ
Comme pour toute scène visuelle, la performance des programmeurs lisant du code dépend de leur performance à percevoir les éléments graphiques représentant le code. On oppose souvent les langages visuels et textuels, en arguant du fait que les langages visuels seraient plus "faciles" à lire et plus "compréhensibles". Nous montrons que la Sémiologie Graphique et le modèle d'analyse ScanVis remettent en question cette opposition et unifient les langages visuels et textuels en termes de perception visuelle. Ce travail est la première étape d'une démarche de construction de connaissances sur la conception de représentation de langages de programmation.

Mots Clés
Langages visuels et textuels ; Sémiologie Graphique ; ScanVis.

ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (*i.e.* HCI): Miscellaneous.

INTRODUCTION
Comme pour toute scène visuelle, la performance des programmeurs lisant des programmes textuels ou visuels dépend de leur performance à percevoir les éléments graphiques représentant les programmes. Cependant, peu de travaux existants permettent d'analyser ces éléments et leur impact sur leur performance (une exception est [10]). Ainsi, les spécialistes opposent souvent les langages visuels et textuels, en arguant du fait que les langages visuels seraient plus "faciles" à lire [14] et plus "compréhensibles" [2]. Pourtant, il est aussi facile avec des langages visuels de créer du code spaghetti désordonné conduisant à des structures cryptiques ou obscures [14]. De même, Mohet et al. ont observé que "les performances étaient fortement dépendantes de la disposition" des représentations de réseau de Petri [9]. Green *et al.* ont trouvé que les représentations textuelles surpassaient LabView et son langage graphique G [7] : "la structure des graphiques dans les programmes visuels est 'paradoxalement' plus difficile à balayer visuellement que les programmes textuels". Dans [12], Petre pense notamment que "les notations secondaires (p. ex., la disposition) sont sujettes à des compétences individuelles (c.-à-d. celles qui ont été apprises) et constituent la différence entre novices et experts. Ce qui est nécessaire est un bon usage de la notation secondaire, qui est comme le 'good design' sujet au style personnel et à la compétence individuelle".

Dans cet article, nous prenons un point de vue très différent de Petre. Nous pensons que le fonctionnement de la perception visuelle (façonné par l'évolution et partagé par la plupart des humains), et non la compétence individuelle (apprise), suffit à caractériser certaines difficultés évoquées plus haut. L'application de modèles de perception aux langages de programmation remet en question l'opposition supposée entre langages visuels et textuels.

MODÈLES DE PERCEPTION VISUELLE
Nous nous appuyons sur les modèles de perception visuelle que sont la Sémiologie Graphique et ScanVis.

La Sémiologie Graphique
La sémiologie graphique est une théorie des dessins 2D abstraits comme les cartes ou les *bar charts* [3]. Une partie de cette théorie explique et décrit les phénomènes perceptifs et les propriétés qui sous-tendent l'acte de visualiser des graphiques 2D abstraits. La sémiologie graphique repose notamment sur les propriétés perceptives des éléments graphiques utilisés dans les représentations. Ces dernières sont des ensembles de *marques* 2D (*points, lignes et zones*) superposés à un fond. Les marques varient selon des *variables visuelles* comme la position (*Xpos et Ypos*), la *forme*, la *couleur*, la *luminosité*, la *taille*, l'*orientation* [3], la *contenance* (p. ex. un cercle contenant d'autres formes) et les *liens qui connectent* deux marques [4]. Les variables visuelles sont caractérisées par leurs propriétés perceptives et peuvent être : *sélectives* – permettent à un lecteur d'*assimiler* ou *différencier* des marques de façon instantanée (p. ex. toutes les marques rouges) (fig. 1) ; *ordonnées* – permettent à un lecteur de ranger des marques perceptiblement (p. ex., du plus clair au plus foncé) (fig. 2) ; et *quantitatives* – permettent à un lecteur de quantifier perceptiblement les différences entre marques (p. ex. deux fois plus large) (fig. 2).

Toutes les variables visuelles sauf la forme et les liens (pour tout cardinal de marques supérieur à une dizaine) sont sélectives (fig. 1). Toutes les variables visuelles sauf la forme, les liens et les couleurs sont ordonnées (les couleurs sont ordonnées sur une partie du spectre seulement). Xpos, Ypos, l'angle, la longueur, la taille sont

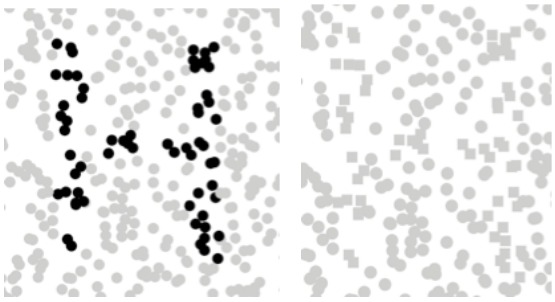

Figure 1. Par défaut toutes les marques sont rondes et claires. à g. : Une partie des marques sont foncées et engendrent la perception d'un H. La luminosité est sélective : la lettre H émerge car l'œil discrimine deux groupes de marques (claires et foncées) instantanément. à d. : Les mêmes marques ont une forme carrée : la forme n'étant pas sélective, la lettre H n'émerge pas.

quantitatives à des degrés de précision divers (prouvés expérimentalement par [5]). La performance des lecteurs à sélectionner, ordonner ou quantifier dépend du *nombre de valeurs différentiables* (5 niveaux de luminosité pour la sélection, 20 niveaux pour l'ordonnancement), de la *différence* entre les valeurs (moins elles diffèrent, plus c'est difficile) et la *distance spatiale* entre les marques (plus elles sont éloignées, plus c'est difficile).

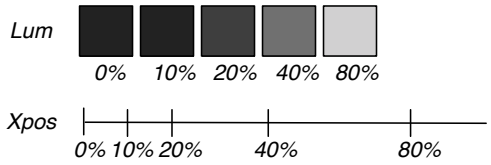

Figure 2. En haut : la luminosité est ordonnée mais ne peut être perçue quantitativement. En bas : la position est quantitative : on peut percevoir le rapport et la différence entre les positions en X (pos. 80% = 2x pos. 40% ou 4x pos. 20%).

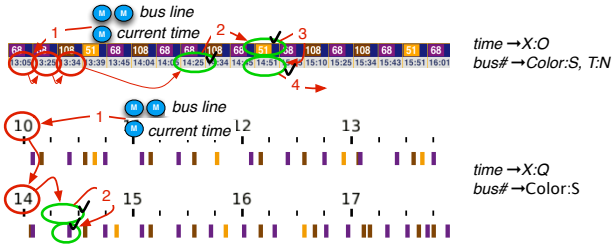

Figure 3. A g. : Deux représentations et les opérations visuelles pour la tâche "trouver le temps d'attente du prochain bus". A d. : correspondance entre données et variables visuelles (p. ex. '$time \rightarrow X : O$' signifie que le temps correspond à Xpos d'une manière ordonnée).

ScanVis

La sémiologie graphique peut aider la conception de représentations qui permettent à des utilisateurs de percevoir des informations en un coup d'œil. Néanmoins, quelle que soit sa qualité, une représentation ne peut pas être absolument efficace (notamment en *temps*) : elle peut être adaptée à une tâche particulière, mais pas à d'autres tâches inhérentes à l'activité de l'utilisateur. Pour ces tâches, l'utilisateur doit balayer la représentation pour découvrir de l'information, au lieu de la percevoir d'un seul coup d'œil.

ScanVis est un modèle descriptif de ce type de parcours visuel [6]. ScanVis repose sur la décomposition du parcours visuel en opérations visuelles élémentaires. Par exemple, la fig. 3 montre deux représentations des temps de passage de bus. En fonction de la représentation, le nombre et la nature des opérations visuelles diffèrent. Les flèches et les cercles superposés à chaque représentation dépeignent le balayage visuel nécessaire pour répondre à la même tâche : "trouver le temps d'attente du prochain bus". Un petit cercle bleu avec la lettre M dépeint le besoin de mémoriser une donnée, p. ex. le temps courant et les lignes de bus compatibles avec la destination voulue. Les cercles rouges [resp. verts] dépeignent les *prédicats* évalués à faux [resp. vrai]. Ainsi, la vue ordonnée (au-dessus) requiert que l'utilisateur *mémorise* le temps courant ainsi que le numéro et la couleur des lignes de bus, *navigue* de gauche à droite au sein de la liste ordonnée de temps textuels (donc n'utilisant pas de variable sélective), atteigne le premier temps supérieur au temps courant (*vérifier un prédicat*), *cherche et navigue* parmi les cellules colorées de gauche à droite jusqu'à ce qu'un bus compatible soit trouvé, lise (ou *décompose*) le temps correspondant, et *sorte de la représentation* en soustrayant du temps trouvé le temps courant afin d'obtenir le temps d'attente.

Les opérations élémentaires de ScanVis peuvent être facilitées par l'usage de propriétés visuelles adéquates décrites par la Sémiologie Graphique p. ex. des variables visuelles *sélectives* pour faciliter *chercher et naviguer parmi un sous-ensemble de marques* : si l'on veut trouver le prochain bus 51 dans la représentation en bas de la fig. 3, on peut *sélectionner* visuellement le sous-ensemble de marques "jaunes" (couleur, une variable sélective) dont la position (une variable sélective) correspond grossièrement au temps courant. On peut donc balayer les marques de ce sous-ensemble puis sauter de marque en marque jusqu'à trouver le prochain bus.

APPLICATION AUX LANGAGES

ScanVis et la Sémiologie Graphique (nommés ci-après le 'framework') ont déjà été utilisées pour analyser les graphiques statistiques ou les visualisations d'information. La Physique des Notations suggère que la Sémiologie Graphique pouvait s'appliquer aux langages [10]. Nous offrons ici une analyse effective, détaillée, et complétée par l'utilisation de ScanVis et par l'identification de tâches de lecture. Pour illustrer le caractère explicatif du framework, nous montrons comment des adages populaires à propos du code peuvent être analysés et traduits en concepts (en italique) du framework. Nous avons choisi ces adages pour leur variété. Nous ne prétendons pas qu'ils sont exhaustifs, mais leur nombre suggère que le framework a une portée significative, et leurs différences suggèrent le caractère unificateur du framework.

"Lots of Irritating Superfluous Parenthesis[1]*"* Lisp utilise les parenthèses pour structurer le code. Les listes de Lisp sont désignées avec des expressions espacées entourées de parenthèses ouvrantes et fermantes (fig. 4, haut). La composition de fonctions utilise les parenthèses imbriquées.

1. en.wikipedia.org/wiki/Lisp_(programming_language)

Lisp est réputé difficile à lire ([13] p65). Ceci peut être expliqué par le concept de *sélectivité* : comme la forme est *non-sélective*, l'usage de parenthèses empêche la perception des limites d'expression Lisp en un coup d'œil et force le programmeur à *chercher un sous-ensemble de marques* de façon linéaire pour les découvrir (ligne 1' de la fig. 4). La ligne 2 utilise une forme de délimitation unique pour chaque niveau de profondeur d'imbrication. On pourrait penser qu'une telle représentation pourrait aider le lecteur à trouver les limites d'expression car l'utilisation de symbole unique pourrait aider le lecteur à ne pas prendre la parenthèse fermante d'une expression pour celle d'une autre expression. Cependant, cette représentation n'est pas meilleure que celle avec parenthèses en raison du même phénomène : la *non-sélectivité* des symboles (ou formes) gène la mise en correspondance en un coup d'œil (p. ex. trouver le losange qui ferme l'expression 'multiplier' est difficile et requiert un *balayage* horizontal attentif).

```
1 (defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))
1'(defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))
2 (defun fac ◁n▷ ◁if ◇<= n 1◇ 1 ◇* n ♡fac ☞- n 1☜♡◇▷)
3 (defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1))))))
4 (defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1))))))
5 (defun fac ┼n┼ ┼if ✳<= n 1✳ 1 ✳* n ✳fac ✳- n 1✳✳✳┼)
```

Figure 4. Délimiteurs variant selon la forme (parenthèse et symboles divers), la teinte, la luminosité, et forme+luminosité

La ligne 3 de la fig. 4 fait correspondre la profondeur d'imbrication avec des couleurs uniques. Comme la couleur est sélective, cela permet au programmeur d'*assimiler* d'un coup d'œil toutes les parenthèses d'un même niveau. Cependant, on peut se demander si la tâche "assimilier le niveau d'imbrication" mérite d'être facilitée : même si un programmeur discerne correctement les parenthèses ouvrantes et fermantes, il doit s'en souvenir pour inférer la structure du code. Si une variable visuelle plus appropriée est utilisée, le programmeur pourrait l'utiliser comme une externalisation de la mémoire pour se rappeler de la structure en y accédant d'un seul coup d'œil. Par exemple, la ligne 4 utilise la luminosité. La luminosité est *sélective* (et aide à faire correspondre les parenthèses), et *ordonnée* (ce qui aide à percevoir la profondeur relative). On ne peut pas percevoir rapidement la profondeur exacte car la luminosité n'est pas *quantitative*. Cependant, l'ordonnancement peut être suffisant pour la tâche à accomplir.

"L'indentation rend la structure évidente [8]" Dans la fig. 5 (a) le niveau d'imbrication correspond à la *variable visuelle Xpos*. Les parenthèses correspondantes sont "alignées verticalement", ce qui est une autre façon d'exprimer une *assimilation* de valeurs de *Xpos*. Comme Xpos est *sélective*, la perception des limites d'expression est meilleure qu'avec une *forme*. La sélectivité dépend de la *quantité de différence* entre valeurs : diminuer la taille de l'indentation affaiblit le caractère sélectif de Xpos (b). Réserver une ligne par parenthèse fermante agrandit la *distance spatiale* Ypos avec la parenthèse ouvrante correspondante et affaiblit le caractère sélectif de Xpos (c.-à-d. qu'il devient difficile de percevoir des alignements) (a et b) ; ignorer le problème de la correspondance réduit les distances et améliore la sélectivité de Xpos (c) ; plus d'indentation améliore la sélectivité de Xpos (d). Cependant, l'amélioration présumée est accomplie au prix d'un balayage plus long du début du bloc vers la première instruction, comme cela a été démontré expérimentalement dans [8]. Notez que puisque Xpos est ordonné, une telle représentation facilite aussi la tâche "discerner la hiérarchie des expressions".

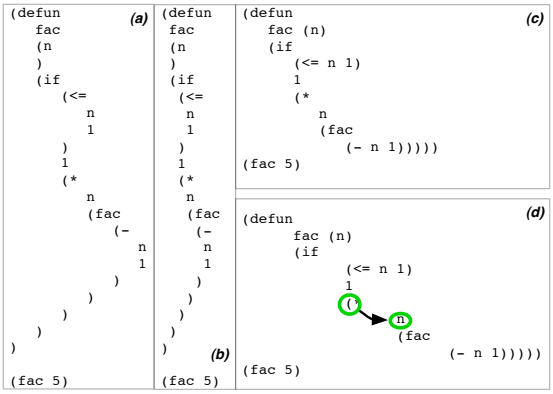

Figure 5. Délimiteurs variant selon Xpos et Ypos, qui sont des variables visuelles sélectives. (a)(b) Un différence plus petite entre les valeurs de Xpos détériore la sélection. (c) Améliorer la sélectivité de Xpos en réduisant la distance spatiale en Ypos ou (d) avec une indentation plus grande, au prix d'un balayage visuel pour atteindre un sous-bloc.

"Le langage G de LabView est intuitif[2]*"* G combine de grandes boîtes qui *contiennent* d'autres objets pour spécifier une structure hiérarchique (fig. 6), et *des liens* qui connectent des composants dans les boîtes. La contenance est peut-être "intuitive", mais une meilleure qualification en termes de perception est qu'elle est *sélective* : on peut discerner en un seul coup d'œil les éléments qui appartiennent à un parent. La contenance est aussi *ordonnée* et favorise la perception de hiérarchie de contenance.

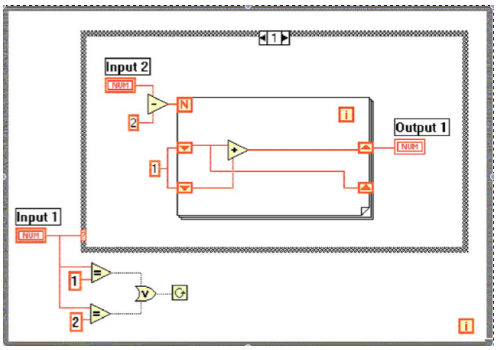

Figure 6. Le langage G de LabView.

"La coloration syntaxique améliore la lisibilité[3]*"* La fig. 7 montre une représentation textuelle à 'syntaxe colorée' de code Java dans l'éditeur NetBeans. Les caractères bleus correspondent aux mots clefs réservés du langage Java et les gris aux commentaires. Un fond jaune correspond à une variable du programme sur laquelle le pointeur de la souris pointe.

2. selon l'éditeur http ://www.ni.com/white-paper/14556/en
3. en.wikipedia.org/wiki/Syntax_highlighting

```
// 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.subst
}
```

Figure 7. Editeur de code coloré et navigation parmi les commentaires

Colorier toutes les occurrences d'une variable pointée par la souris a du sens du point de vue du programmeur : cela permet au programmeur d'identifier efficacement toutes les utilisations grâce à la *sélectivité*. De façon similaire, ajouter un fond coloré à une accolade permet au programmeur de vérifier rapidement où se trouve l'accolade correspondante et évaluer la portée d'un bloc. La couleur grise est plus claire que les autres : comme la luminosité est *sélective*, cela permet à l'utilisateur d'*assimiler* et *différentier* le code des commentaires, et *naviguer* rapidement entre les sections de code. Cela supprime la nécessité de *balayer* le début d'une ligne pour vérifier si elle commence avec deux slashs, une opération visuelle plus exigeante car la forme ('//') n'est pas *sélective*. De plus, l'*ordre* de luminosité indique un ordre d'importance entre code, commentaires, et fond.

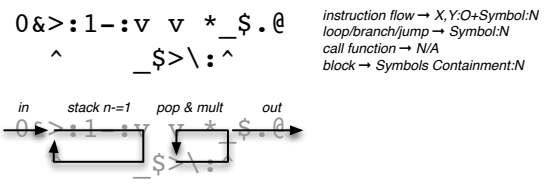

Figure 8. Factorielle en befunge (haut) ; explication du flot (bas).

"Befunge est un language ésotérique[4]*"* Befunge est un langage 2D textuel dans lequel le flot est indiqué par quatre *formes* <, >, ˆ et v, qui ressemblent à des flèches pointant dans les quatre directions cardinales. Le branchement est spécifié par − (équivalent à < si la condition est vraie et à > sinon) et — (équivalent à ˆ si la condition est vraie et v sinon). Non seulement le flot n'est pas discernable en un coup d'œil (de vraies flèches et des liens aident un peu dans le bas de la figure), mais de plus les formes directionnelles ne sont pas *sélectives* et donc ne permettent pas à l'utilisateur de voir d'un seul coup d'œil les changements de direction. Ainsi, Befunge n'est peut-être pas si ésotérique puisqu'il peut être considéré comme un chaînon manquant entre code textuel et visuel (ce qui illustre l'aspect unificateur du framework). En revanche, sa difficulté à être lu est due aux raisons évoquées plus haut.

4. selon son concepteur catseye.tc/node/Befunge-93.html

"Les icônes sont plus faciles à utiliser que le texte [15]" La Fig. 8 de [12] illustre l'utilisation d'un vocabulaire iconique dit 'analogique' (selon [12]). Dans ce cas, les icônes sont différenciées par des *formes* (flèches, coins de page, spirales, 'N' et 'I'). Avec des icônes ne variant que par la forme (une variable non-sélective), on ne peut effectuer qu'une *lecture élémentaire* et donc lente d'une scène. Si cela peut suffire pour l'identification de structures de contrôle (while, if), cela peut rendre inefficace l'accomplissement des taches évoquées dans la section "coloration syntaxique". Avec d'autres langages visuels, les icônes peuvent varier en forme mais aussi selon d'autres variables visuelles, ce qui peut rendre sélectives les icônes. Par exemple, la dernière ligne de la fig. 4 utilise un ensemble de formes pour lesquelles la *sélection* et l'*ordonnancement* semble 'marcher' : ce phénomène est dû au fait que les icônes ne contiennent pas le même nombre de pixels et ont donc différents niveaux de *luminosité*, une variable *sélective* et *ordonnée*.

"Avec les langages visuels le flot est explicite [2]" Comme vu plus haut, le flot d'instructions peut être dépeint avec Ypos ou des liens et des flèches. Les liens et flèches sont des variables visuelles non-sélectives : le lecteur est forcé de suivre la chaîne de liens pour discerner le flot (fig. 9-a). Cela peut être complété par des indices d'alignement p. ex. utiliser la sélectivité de Ypos. Dans ce cas, la visualisation est équivalente à du code indenté dans un programme C (fig. 9-b). Il n'est pas nécessaire de montrer les flèches entre instructions successives car cela serait redondant avec la représentation alignée en Ypos (fig. 9-c). Cependant, garder la flèche pour la boucle peut aider le lecteur à la suivre jusqu'au début de la boucle, comme avec les langages boite-flèches. Scratch [11] est un langage visuel avec des connecteurs sur des blocs suggérant comment ces derniers peuvent être assemblés. Les connecteurs sont similaires aux flèches : elles guident un lecteur qui suit une séquence d'instructions (fig. 9-d). Le début de la boucle peut être perçu de façon sélective avec la couleur et l'encerclement. Ces exemples montrent comment différentes représentations peuvent être unifiés avec les mêmes principes sous-jacents.

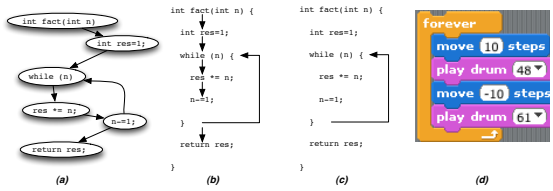

Figure 9. Les flèches auraient pu être utilisées en C (b), comme dans les langages boites-flèches (a). Comme les flèches sont redondantes avec la variable visuelle ordonnée Ypos, elles peuvent être supprimées, sauf pour la boucle (c). Scratch utilise des variables visuelles similaires (d).

Les flèches sont souvent considérées comme une représentation explicite de la séquence d'instruction [2]. Pour être plus précis, elles sont effectivement une représentation explicite de la direction de la séquence. Cependant, elles ne sont pas plus explicites de l'ordre de la séquence que Ypos, puisque la variable visuelle ordonnée Ypos montre cette séquence (à la fois dans la version C et dans la version Scratch).

"Une image vaut mille mots" Les représentations sont utilisées pour accomplir de multiple tâches de lecture. Pour les comparer, il est utile d'énumérer un ensemble réaliste de tâches et évaluer l'adéquation de chaque représentation par rapport à chaque tâche. La fig. 10 montre deux représentations du programme décrivant la même interaction de Drag'n'Drop : à gauche une représentation avec cercles et flèches, à droite du code SwingStates. SwingStates est un langage textuel qui s'appuie sur les classes anonymes afin d'embarquer du code de machine à états dans du code Java classique [1]. Le code est indenté pour faciliter la perception des états, des transitions partant de chaque état et des clauses associées aux transitions.

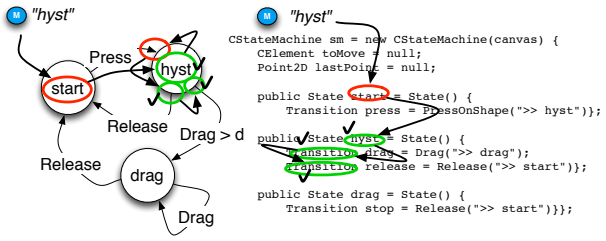

Figure 10. ScanVis pour la tâche : "quelles sont les transitions sortantes pour l'état 'hyst' ?" .

La fig. 10 compare les opérations visuelles requises pour la tâche "quelles sont les transitions sortantes pour un état particulier ?" avec les représentations cercles-flèches et SwingStates. Dans les deux cas, les lecteurs doivent chercher et naviguer parmi les états jusqu'à l'état particulier, puis chercher les transitions partant de cet état. Avec les cercles-flèches, on peut considérer que les gros cercles blancs sont sélectifs par rapport aux autres marques grâce à leut taille et à leur luminosité. Dans le code SwingStates, l'indentation est aussi sélective. Par conséquent, les deux représentations aident à chercher un sous-ensemble de marques (celles représentant les états). Avec les cercles-flèches, on doit différencier les liens qui ont une flèche et ceux qui n'en ont pas parmi les liens autour du cercle. Les liens sans flèches peuvent être plus difficile à discerner car ils ne sont pas différents de façon sélective d'autre marques comme le contour du cercle. Trouver les transitions sortantes est peut-être plus efficace avec SwingStates, car toutes les transitions à cet endroit du code sont sortantes, ce qui contredit l'adage cité plus haut. En revanche, pour la tâche "quelles sont les transitions sortantes pour un état particulier ?", SwingStates est peut-être moins adapté que des cercles-flèches car il faut trouver des textes (non-sélectifs) dans l'ensemble du code.

CONCLUSION ET PERSPECTIVES

Nous avons montré comment les modèles de Sémiologie Graphique et Scanvis peuvent expliquer certains phénomènes relatifs à la perception du code et permettent de requalifier les adages concernant ces phénomènes en utilisant les concepts de ces modèles. Cette analyse permet d'unifier les langages visuels et textuels en termes de perception et montre que l'opposition traditionnelle doit être nuancée : la plupart des langages textuels utilisent des variables visuelles positionnelles qui peuvent utiliser le système perceptif efficacement alors que les langages dits visuels utilisent parfois des variables visuelles (icônes (formes), liens) de façon inefficace.

Un framework qui corrobore, qualifie et explique des observations, et unifie des concepts auparavant présentés comme différents est une première étape d'une démarche scientifique (comme par exemple en physique les tentatives d'unification des forces (électro-magnétisme), ou en informatique la correspondance de Curry-Howard). Une perspective offerte par ce travail est de fournir aux concepteurs des concepts précis pour débattre des mérites de telle ou telle représentation de code. Une autre perspective offerte est d'initier un travail plus complet sur l'explicitation des tâches de lecture de code. Ce travail devrait contribuer à la définition d'implications pour la conception de représentations efficaces de code. Par ailleurs, d'autres modèles comme la Gestlat ou la Cognition Distribuée [16] pourraient offrir des explications complémentaires sur les phénomènes de perception de code.

BIBLIOGRAPHIE

1. 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.

2. Burnett, M. in Encyclopedia of Electrical and Electronics Engineering (John G. Webster, ed.), John Wiley & Sons Inc., New York, 1999.

3. Bertin, J. (1967) Sémiologie Graphique - Les diagrammes - les réseaux - les cartes. Gauthier-Villars et Mouton & Cie, Paris.

4. Card, S.K., Mackinlay, J.D., Shneiderman, B., Readings in Information Visualization : Using Vision to Think. Morgan-Kaufmann, (1999).

5. 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.

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. Green, T. R. G. & Petre, M. (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.

8. Miara, R., Musselman, Navarro, J. and Shneiderman, B. 1983. Program indentation and comprehensibility. CACM 26(11), 861-867.

9. 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.

10. 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.

11. 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. CACM, 52, 11, 60-67.

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

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

14. 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.

15. Wiedenbeck, S. The use of icons and labels in an end user application program : an empirical study of learning and retention. Behaviour & Information Technology, 1999, (18)2, 68-82.

16. Zhang J. and Norman, D.A. Representations in Distributed Cognitive Tasks, in Cognitive Science, 18(1), 87-122, 1994.