A survey of external representations employed in Object-Oriented programming environments

Pablo Romero
Richard Cox
Benedict du Boulay
Rudi Lutz
School of Cognitive and Computing Sciences
Sussex University, U.K.

June 25, 2001

Abstract

This document presents an overview of the program visualisations additional to the program code provided by some of the most popular object-oriented programming environments to support tasks involving program comprehension. These representations were compared in terms of the programming aspects they highlight and of their information modality. Those with common characteristics according to these criteria were identified. Finally, a brief analysis of these common representations in terms of Green's (1989) Cognitive Dimensions is presented.

Two questions arising from this survey are a) whether representations additional to the code should be redundant and highlight similar information to the main notation or be complementary and highlight different programming aspects and b) which factors might increase the cognitive difficulty of co-ordinating these additional representations and the program code. More theoretical knowledge about the way these additional representations influence the comprehension of computer programs seems to be needed.

Keywords: external representations, program comprehension, program visualisation

1 Introduction

Program comprehension is a skill that is central to programming. System documentation, maintenance and debugging are important programming activities for which program comprehension is an essential subtask. Even code generation, with its current emphasis in re-use, normally comprises frequent program comprehension episodes. Comprehension of computer programs is a complex cognitive activity which could benefit from the appropriate tools, both at the novice and experienced levels. One common approach to support

the program comprehension process is by complementing the program code with other abstractions or visualisations of it. Novice and professional software development environments often offer multiple visualisations of the program code, each one normally highlighting a specific aspect of it. The use of graphical interfaces has allowed these additional representations to increase their graphical complexity. However, there is little theoretical understanding about the way these multiple representations influence the comprehension of computer programs.

Graphical abstractions seem to be a central feature of Object-Oriented programming. The very name of this programming paradigm evokes abstractions of a graphical nature, and indeed almost any Object-Oriented programming environment provide the programmer with multiple abstractions of the code. However, there does not seem to be a standard for the kinds of representations employed as an aid to program comprehension related tasks. The Unified Modelling Language (UML) (Booch, Rumbaugh, & Jacobson, 1998), for example, proposes several types of views of the program, but these form part of a methodology for software design.

The purpose of the survey reported in this paper is to identify the types of representations commonly used by Object-Oriented programming environments to aid tasks involving program comprehension, to characterise them both in terms of their graphical properties and of the programming aspects highlighted by them and to offer an analysis of their notational properties. This investigation is being undertaken within a project with the more global aim of exploring the way multiple representations influence computer programming. In particular, this project focuses on the issue of how different factors affect the way users co-ordinate representations when performing programming tasks. Some of the factors under consideration are: the different programming aspects afforded by the representations, information modality and individual differences such as cognitive style.

This paper is divided into four parts. The following section gives a overview of external representations in general and in the programming area in particular, and presents a summary of a framework to analyse their notational properties. The next section offers a brief account of some of the most popular Object-Oriented programming environments in terms of the types of representations they use to support tasks involving program comprehension. The subsequent sections analyse these types of representations in terms of the programming aspects highlighted by them and their information modality and also present a brief analysis of the notational properties of some of these representations. The final part presents a summary and conclusions of this survey.

2 External representations in programming

When trying to perform a programming activity in everyday settings, programmers normally work with other external representations as well as the program code. Some of these external representations occur in debugging

packages, prototyping and visualisation tools in software development environments, or are included as part of internal and external documentation. Therefore, programming normally requires working with multiple representations.

Two important aspects to consider when using multiple representations are the issue of sentential versus graphical representations and the different programming aspects implicit in programs. The first aspect, *information modality*, refers to the characteristics, advantages and disadvantages of representations which are basically propositional and those that are mainly diagrammatic. It is not clear whether, for example, including a high degree of graphicality in the code visualisations has potential benefits for performing the programming task.

The second aspect refers to the different *programming perspectives* highlighted by the visualisations. Computer programs are information structures that comprise different types of information, and programming notations usually highlight some of these perspectives at the cost of obscuring others. It is an open issue whether, for example, abstractions that highlight information types different to the program code will be more beneficial to programmers than those that highlight the same ones.

A descriptive account of these two aspects would not be very useful on its own. However, because of the exploratory nature of the survey reported here, a formal analysis of their merits would be outside the scope of this investigation. The best alternative seems to be to perform an analysis of the notational properties of the program visualisations in terms of a framework like Cognitive Dimensions (Green, 1989).

The next subsections offer a brief account of the relevant characteristics of information modality and programming perspective and of the Cognitive Dimensions framework.

2.1 Information modality

Information modality refers to the particular form of expression that is used to present information. A typical distinction here is between propositional and diagrammatic representations, although these two terms can be thought of as situated at the extremes of a continuum containing representations with different degrees of 'graphicality' (Cheng, Lowe, & Scaife, 1999). Program code, for example, cannot be regarded as fully propositional, because it uses location conventions to enhance its comprehension (there is normally a line-per-instruction format and it makes extensive use of indentation). In the taxonomy of graphic languages developed by Twyman (1979), program code would not be an example of pure linear text but a hybrid category of text between a list and a linear branching configuration.

Representations of differing modality are a common case of multiple external representations that support complementary processes (Ainsworth, 1999). For example, diagrams, unlike propositional representations, exploit perceptual

processes by grouping relevant information together and therefore make the search and recognition of information easier. Another difference between propositional and diagrammatic representations is that the former permit the expression of abstraction or indeterminacy while the latter compel the representation of specific information (Stenning & Oberlander, 1995). This specificity of graphical representations allows some inferences to be more tractable.

Although programmers normally have to coordinate representations of different modalities, there has not been much research on these issues in the area of programming. One of the few examples here is the GIL system (Merrill, Reiser, Beekelaar, & Hamid, 1992), which attempts to provide reasoning-congruent visual representations in the form of control-flow diagrams to aid the generation and comprehension of LISP, a textual programming language. Merrill et al. claim that this system is successful in teaching novices to program in this language; however, their work did not relate their results to issues like information modality and programming perspective.

Other studies in the area have been concerned with issues related to the format of the output of debugging packages (Patel, du Boulay, & Taylor, 1997; Mulholland, 1997). Those studies have offered conflicting results about the co-ordination of representations of different modalities. Patel et al. (1997) found similar performance for subjects working with representations of the same and different modalities while Mulholland (1997) reported poor performance for those working with different modalities. In both cases, participants worked with the program code and with the debugger's output. The debugger notations used by both of these studies were mostly textual. The only predominantly graphical debugging tool used by these studies was TPM (Eisenstadt, Brayshaw, & Paine, 1991). While the performance of Patel et al.'s participants was similar for the textual debuggers and TPM, the subjects of Mulholland's study found working with TPM more difficult. One important difference between these two studies is that while the former used static representations, the later employed a visualisation package and therefore dynamic representations. However, other factors might be playing a role in these conflicting results. First, it is not clear that the graphical debugger (TPM) displayed a similar amount of information to its textual counterparts. Other important factors that Patel et al. (1997) identified are the choice of experimental materials (the programs employed in the experiments) and the degree to which the information required for the experimental task is hidden or implicit in the additional representation. An important issue is the degree to which the above work on LISP and Prolog generalises to Object-Oriented languages.

2.2 Programming perspective

Programming perspective refers to the different 'views' that can be adopted to look at a domain. In programming it has been established that programs can be looked at from different perspectives (Pennington, 1987b); and experienced programmers, when comprehending code, are able to develop a mental

representation that comprises these different perspectives or information types, as well as rich mappings between them (Pennington, 1987a; Ormerod & Ball, 1993). This characteristic is similar to what has been found in general problem solving, where the ability to build several perspectives of a domain and to switch between them during problem solving as a means to cope with complexity is directly related to domain expertise (Nix & Spiro, 1990).

Some of the different programming perspectives are: function, structure, operations, data-flow and control-flow. Function refers to what the program does, structure to the programming language objects that are used in order to implement a solution to the programming problem and operations concern the individual statements in a program which carry out low-level actions. Data-flow concerns the transformations which data elements undergo as they are processed and control-flow refers to the sequence of actions that will occur when the program is executed.

Although programs comprise several information types, according to the match-mismatch hypothesis (Gilmore & Green, 1984), notations (in this case programs) highlight some of them and obscure some others. In procedural languages, for example, it is relatively easy to detect control-flow information, and relatively difficult to access function relations (Pennington, 1987b; Corritore & Wiedenbeck, 1991). For different programming languages the information types highlighted and obscured are different. For Prolog, a declarative programming language, it is data structure information which is accessible (Bergantz & Hassell, 1991; Romero, 2001), while according to Wiedenbeck and Ramalingam (1999), novices of C++ working with small programs tend to develop a mental representation strong in function-related knowledge, but have problems detecting operations and control-flow information.

From this point of view, it makes sense to reinforce or complement the information comprised in the program code with additional representations. Two issues to consider here are the number of these additional representations and whether they are redundant or complementary to the program code. There has to be a balance between a representation highlighting as few information types as possible, so as to keep it simple for easy information extraction, and presenting the programmer with only a few additional representations, so that problems of representation co-ordination do not over complicate the task. Regarding the issue of information redundancy, it is not clear whether representations that highlight different information types to the program code have any advantage over those that highlight similar ones.

2.3 The Cognitive Dimensions framework

According to Green (1999), Cognitive Dimensions is a framework that can be applied to the analysis of the notational design of all information artefacts, from computer languages to mobile telephones. It contains three important components: dimensions, activity types and environment. Cognitive Dimensions are descriptors that refer to usability properties of an information structure. For example, five relevant dimensions are *closeness of mapping*,

role-expressiveness, *hard mental operations*, *diffuseness/terseness* and *hidden dependencies*. Closeness of mapping refers to how 'far' an external representation is from its internal counterpart, or how direct the mapping is from what is in the head of the user to what is in the system. Of course the difficulty here is to know what is in the head of the user. In programming, for example, some proponents of the Plan model (Soloway & Ehrlich, 1984; Rist, 1986, 1989; Détienne, 1990) consider that languages in which Plan information can be easily expressed would be superior because programmers would find it easier to map their solutions to a program. In this case, it could be said, at least following the Plans-in-the-head assumption, that such a programming language would have a good closeness of mapping.

Role expressiveness means how easy it is to assemble and de-assemble cognitive structures into a notation. For program comprehension, this descriptor means how easy it is to detect meaningful structures or chunks and also how easy it is to establish relations among them; or in other words, how 'hard' it is to break down a program into its primitive components. Sometimes, the role expressiveness of a system can be improved by adopting an informal, secondary notation. An example of this would be the convention of writing all the syntactic key words in a programming language in upper-case letters.

Hard mental operations refers to how much cognitive load the information artifact imposes. If, for example, users need to resort to fingers and pencil annotations at some points to keep track of what is happening then it is possible that the information artifact imposes hard mental operations.

Some notations use a lot of symbols to achieve the results that others can accomplish more compactly. The diffuseness/terseness dimension refers to this. This dimension has been a common point of comparison for computer languages. Cobol is well known for its verboseness, while some other languages like Basic or Prolog are more compact. Green and Petre (1996) argue that a high degree of diffuseness means more cognitive effort on users, but over-terseness could, on the other hand, make different instances of artifacts more similar and therefore increase the difficulty to discriminate between them.

Finally, hidden dependencies refers to the visibility of the dependencies between components in a notational system. Green uses the example of spreadsheets to illustrate this point. According to him, spreadsheets exhibit a high degree of hidden dependencies. For example, when changing the contents of a cell in a spreadsheet application it is not obvious which other cells will be affected.

The second element of the cognitive dimension framework, activity types, refers to a classification of the types of tasks that can be performed with an information artefact. Activity types are an important element of the framework because the descriptors cannot be analysed in the abstract. It might be the case that a notational system has a good evaluation for a descriptor for a specific activity but not for another. There are four activity types: incrementation, transcription, modification and exploratory design. Incrementation refers to tasks that have to do with adding a new element in the notational system, for example, adding a new address to an address book.

Transcription means transferring a piece of information from another notational system to our target information artefact. For example, converting a formula into spreadsheet terms. Modification means changing something in the notational system's application, like modifying a program to solve a different problem or changing the phone area codes in an address book. Finally, exploratory design activities are characterised by an uncertainty about the final outcome of the task, this final outcome has to be discovered while performing the activity.

Very often information artefacts are abstract notational systems that need a working environment. For example, programming languages are an abstract notion; people need editors or some other kind of working environment to program. In this way, it has to be remembered that claims about a system are made assuming a particular environment. Using a different environment very often changes the notational properties of the system.

3 External representations in some of the most popular programming environments

This section presents a brief review of some of the most popular object-oriented programming environments and the types of representations they use to support tasks involving program comprehension. The section is divided into those environments whose purpose is to support the whole programming activity (proper development environments), those that are mainly code visualisation tools, learning environments and visual languages. Development environments provide representations in addition to the program code mainly for debugging purposes, while visualisation tools and learning environments provide them mainly for general program comprehension.

The development environments considered are: Microsoft's Visual J++, Borland's JBuilder, IBM's Visual Age and Code Warrior. The visualisation tools considered are: IBM's JaViz and Jinsight, VisiComp and CodeVizor. The learning environments taken into account are BlueJ and Cocoa. Finally, the only example of a visual programming language considered is Prograph. As the motivation of this survey was to identify the types of representations commonly used by Object-Oriented programming environments to aid program comprehension related tasks, the main criteria to select these particular programming environments is that they were widely used. The only cases for which it is not clear that this criteria is met was for the learning environments; however, this is normally a difficult point to assess for this type of systems.

Table 1 presents a summary of some of the characteristics considered for this review. This table presents the four development packages, four visualisation tools, two learning environments and one example of a visual language referred to above. Some of these programming environments can display more than one representation window (e.g. Jinsight). All the development packages can display information in several other windows, however, this review focuses only on their debugger browsers. The modality of the representation refers to

Type of tool	System	Type of representation	Information modality	Information type highlighted	Programming concepts represented
Development package	Visual J++	Debugger browser	Propositional	Function, Control-flow, Data structure	Threads, methods, objects, variables
	Visual Age	Debugger browser	Propositional	Function, Control-flow, Data structure	Threads, methods, objects, variables
	JBuilder	Debugger browser	Propositional	Function, Control-flow, Data structure	Threads, methods, objects, variables
	Code Warrior	Debugger browser	Propositional	Function, Control-flow, Data structure	Threads, methods, objects, variables
Visualisation tool	JaViz	Tree and node information windows	Diagrammatic, propositional	Control-flow	Methods
	Jinsight	Execution, table, call tree, object histogram and method histogram view	Table, diagrammatic, histogram	Data structure, control-flow	Methods, objects
	Code Vizor	Class hierarchy view	Diagrammatic	Data structure	Classes
	VisiComp	Objects view	Diagrammatic	Data structure	Objects
Learning environment	BlueJ	Objects and classes view	Diagrammatic	Data structure	Classes, objects
	Cocoa	Simulation view	Diagrammatic	Data structure, condition-action	Objects
Visual language	Prograph	Visual code editor	Diagrammatic	Data-flow	Variables, methods

Table 1: Summary table of the reviewed representations

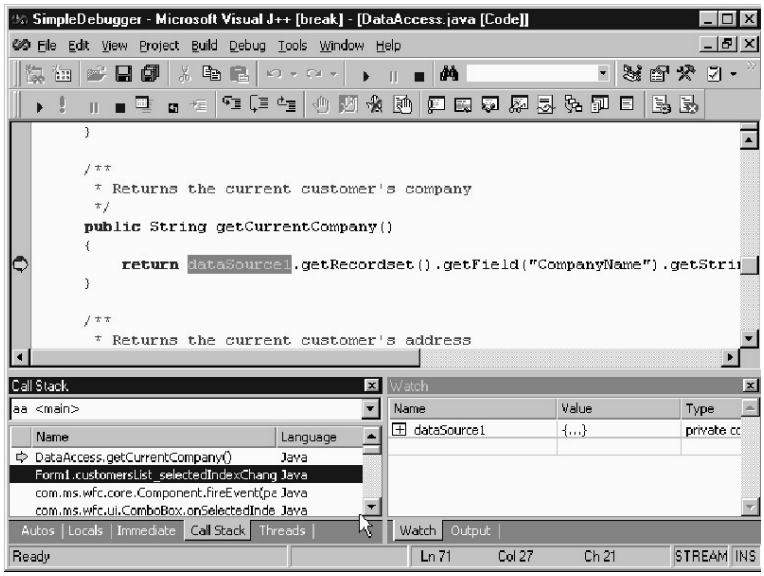

Figure 1: Sample debugging interaction window in Visual J++.

whether the information is presented in a textual, graphical or tabular form. Information type highlighted refers to the different information types comprised and highlighted by the representation (Pennington, 1987b). The information types taken into account here are function, data-flow, control-flow, data-structure and condition-action. Finally, programming concepts represented refers to the object-oriented concepts conveyed in the representations.

This is not a review of programming environments, but of the representations they provide. Therefore, this survey focuses on the representations that these environments use in order to support program understanding rather than on the whole system. The functionality of these systems is described and discussed only in relation to these representations.

3.1 Development environments

Development environments provide external representations mostly to support the debugging process. Although some of their names could imply that they support visual programming, this is mostly true only for building the layout of the user interface. Also, the representations used for debugging purposes are mostly textual rather than graphical.

3.1.1 Visual J++

Microsoft's Visual J++ is a Windows-hosted development tool for Java programming (http://msdn.microsoft.com/visualj/). Its debugger comprises

Debugger window	Information type highlighted	Programming concepts represented
Autos window	Data structure	Variables
Locals window	Data structure	Variables
Immediate window	Function, Data structure	Methods, objects, variables
Call stack window	Control-flow	Methods
Threads window	Control-flow	Threads
Watch window	Data structure	Variables
Output window		Compiler error messages

Table 2: Summary table for Visual J++

two windows (apart from the code window). They display information about local variables, the call stack, the threads, the value of specific variables and the program's output (errors and exceptions for example). Figure 1 shows an example of these windows in a debugging session. The bottom left window shows the call stack, while the bottom right window shows the variables that have been selected for watching. The code window shows the line currently being executed with an arrow which appears at the right hand side of it. At the bottom of the window showing the call stack there is a selection bar in which users can click to select different views. The *autos* window displays the values of all variables within the scope of currently executing methods. The *locals* window displays the local variables and their values for each method in the current stack frame. The difference between these two windows is that the former displays only the variables of the current method, but for all the live threads. This allows the programmer to be aware of the impact that code executing in different threads might have upon variables. In the *immediate* window users can evaluate any expression, variable, or object and see the value that is returned. The *call stack* window displays a list of all active procedures or stack frames for the current thread of execution. Active procedures are the uncompleted procedures in a process. Finally, the *threads* window enables users to change the current thread of execution or view the threads in any attached process.

At the bottom of the window showing the variables selected to be watched there is another selection bar which contains two display options. The watch option is the one currently being displayed in Figure 1. The *output* window displays messages generated by the compiler and the output resulting from debugging instructions that users place in their code. Table 2 shows a summary of the properties of the different windows comprised in this debugger

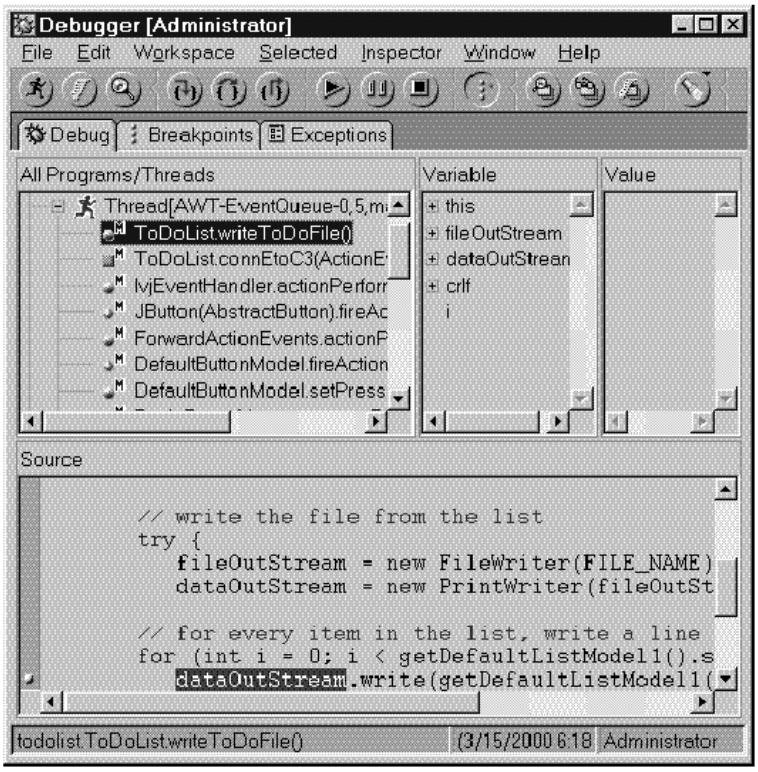

Figure 2: Sample debugging interaction window in Visual Age.

browser. This table does not show the information modality dimension because all of the debugger windows present propositional representations.

The main way of interacting with these windows is selecting the desired view by clicking on their selection bars. Also, by selecting the 'immediate' view in the bottom left window the user can enter code and have it interpreted according to the current state of the program.

3.1.2 Visual Age

Visual Age is IBM's software development environment (http://www-4.ibm.com/software/ad/vajava/). Its debugger displays four windows: the *all programs/threads* window, the *variables* window, the *value* window and the *code* window. The *all programs/threads* window displays the thread being debugged, and below this thread there is a list of methods representing the current stack. The *variables* window shows the variables local to the method selected in the *all programs/threads* window. The *value* window displays the value of the variable selected in the *variables* window. The *code* window shows the line of source currently executing by highlighting it. Figure 2 shows an example of a debugging session in Visual Age.

Debugger window	Information type highlighted	Programming concepts represented
All programs/threads window	Control-flow	Threads, methods
Variables window	Data structure	Objects, variables
Values window	Data structure	variable values

Table 3: Summary table for Visual Age debugger browser windows

By clicking on a method of a current thread (in the *all programs/threads* window), the *variables* window displays the variables associated with that method. The variables' elements can be expanded/contracted by clicking at the left hand side of them. Also, by clicking on a specific variable of this window, its value is displayed in the *value* window. Table 3 shows a summary of the properties of the different windows comprised in this debugger browser. As in the table in the previous section, this table does not show the information modality dimension because all of the debugger windows employ propositional representations.

3.1.3 JBuilder

The debugger in Borland's JBuilder provides a wide range of information (http://www.borland.com/jbuilder/). When debugging, the display is divided into two parts: the application browser and the debugger browser. In Figure 4, the application browser is displayed at the top of the screen, while the debugger browser appears at the bottom. The application browser comprises three windows: the project, the class hierarchy and the code windows. The project window shows the files associated with the current project, the class hierarchy window displays the static class relationships of the current object and the code window shows the code of the method currently being executed. The highlighted line in the code window is the line of the source code being executed at that moment.

The debugger browser has four parts: the views window, the toolbar, the status bar and the session tabs. The views are the main part of the debugger browser. There are six views and they can be displayed all at once as floating windows: the *console*, the *classes with tracing disabled*, the *breakpoints*, the *threads*, the *data watch* and the *loaded classes* view. The *console* view displays output and errors from the program being executed and also allows users to input data to it. The *classes with tracing disabled* view lists classes and packages the debugger will not step into. The *breakpoints* view gives a list of all the breakpoints set and provides commands for updating them. The *threads* view displays the current status of all threads in the program. Each thread group expands to show a list of its threads and current method call

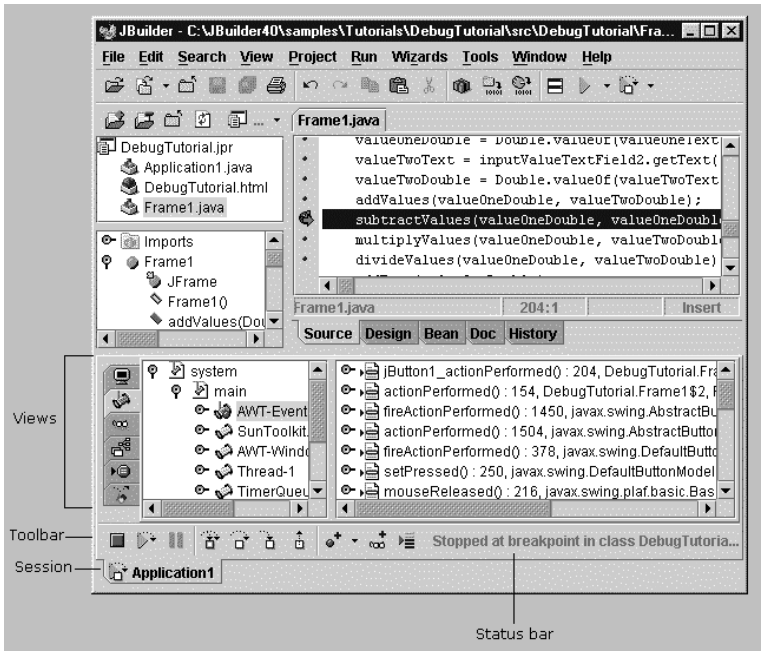

Figure 3: Sample debugging interaction window in JBuilder.

Debugger view	Information type highlighted	Programming concepts represented
Console output, input and errors	Function	Program input and output
Classes with tracing disabled	Data structure	Classes
Data and code breakpoints	Data structure, control-flow	Methods, objects, variables
Threads, call stack and data	Data structure, control-flow	Methods, objects, variables
Data watches	Data structure	Objects, variables
Loaded classes and static data	Data structure	Classes, packages

Table 4: Summary table for JBuilder

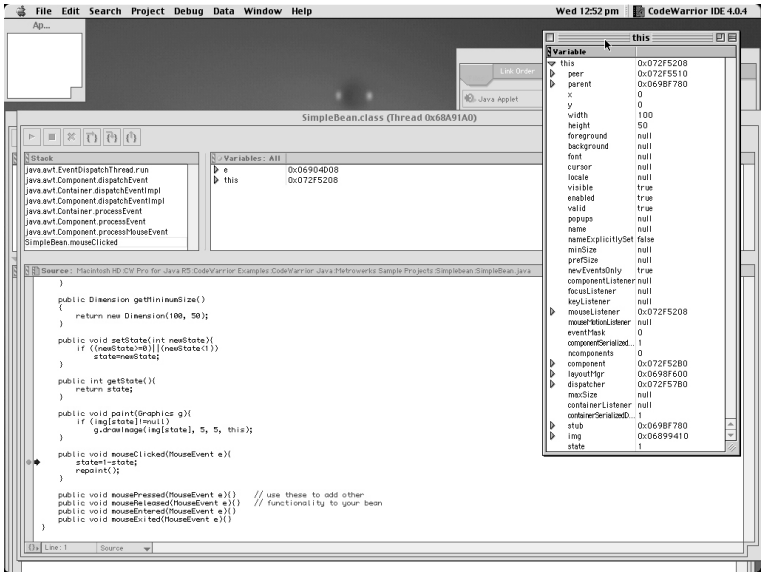

Figure 4: Sample debugging interaction window in Code Warrior.

Debugger window	Information type highlighted	Programming concepts represented
Call stack window	Control-flow	Threads, methods
Variables window	Data structure	Objects, variables
Watch window	Data structure	Objects, variables

Table 5: Summary table for Code Warrior debugger browser windows

sequence. Data associated with the current thread is shown at the right side of this window. The *data watch* view displays the current values of data elements that have been selected. Finally, the *loaded classes* view shows the classes currently loaded by the program. Each class can be expanded to show static data belonging to it. Table 4 shows a summary of the properties of the different views comprised in this debugger browser. As in the summary tables for the other two debugger browsers presented, this table does not show the information modality dimension because all the debugger windows employ propositional representations.

3.1.4 Code Warrior

Code Warrior's debugging browser displays standard debugging information (http://www.codewarrior.com/). As shown in Figure 4, Code Warrior's

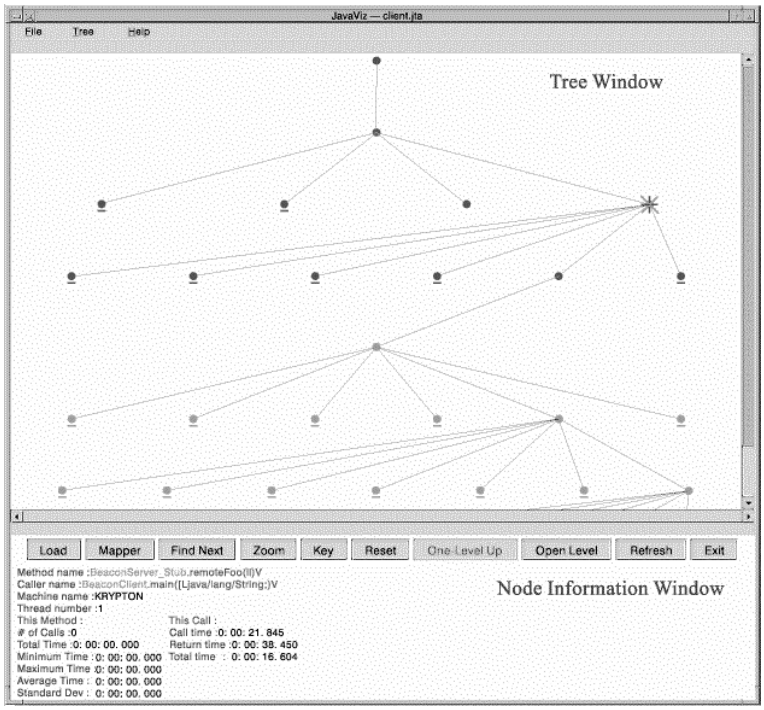

Figure 5: Sample visualisation in the JaViz tool set.

debugger has a window to display the call stack, another for its current variables and yet another for the program's source. The call stack window shows the associated methods. By clicking on a particular methods, its associated variables are displayed in the variables window. Each variable can be expanded by clicking on it into a floating window which shows its structure and values. The line of code currently under execution is marked by an arrow. Table 5 shows a summary of the properties of the different windows comprised in this debugger browser. Again, this table does not show the information modality dimension because all of the debugger windows employ propositional representations only.

3.2 Visualisation tools

Unlike debugging browsers within software development environments, visualisation tools use graphical as well as textual representations to provide support for program understanding. Four visualisation tools will be briefly described here, again in terms of the representations they use: IBM's JaViz and Jinsight, CodeVizor and VisiComp.

Visualisation type	Information modality	Information type highlighted	Programming concepts represented
Execution view	Table, diagrammatic	Control-flow	Methods
Table view	Table	Control-flow	Methods
Call tree view	Table, diagrammatic	Control-flow	Methods
Object histogram view	Table, histogram	Data structure	Objects
Method histogram view	Table, histogram	Control-flow	Methods

Table 6: Summary table for Jinsight

3.2.1 JaViz

IBM's JaViz (Kazi, Jose, Ben-Hamida, Hescott, Kwok, Konstan, Lilja, & Yew, 2000) is a visual analysis tool that supports performance tuning for large scale distributed Java application programs. JaViz displays the execution of a program graphically as a tree with detailed node information (see Figure 5). The nodes of this tree are the methods that the application has to execute. The JaViz display comprises two windows: the *tree* window and the *node information* window. The *tree* window displays the graphic representation of the execution tree. Through the tree menu option, the user can select to 'hide' or 'expand' nodes of this tree so as to obtain the desired level of granularity, to zoom in or out a selected area and to find a node instance that matches specified criteria. The *node information* window displays data about the currently selected node. This information includes its name, the name of its caller, the name of the machine executing it, an identifier for the thread it belongs to and additional statistical information about it. Examples of this statistical information are: the total time the current method took to execute over all its invocations in the program, the minimum, maximum and average time this method took to execute over all its invocations and the total execution time of this method instance. Some of its functionality includes the ability to find a node that matches a set of given parameters, setting specific style features to a group of nodes (like color, shape or size of nodes), zoom and expand tree nodes.

3.2.2 Jinsight

IBM's Jinsight is one of the most complete visualisation tools in the market (http://www.research.ibm.com/jinsight/). It is aimed at professional programmers who want to perform code tuning in terms of performance. It allows programmers to visualise the program according to several views: the *execution*, *table*, *call tree*, *object histogram* and *method histogram* view. Table 6 shows a summary of the properties of the different views comprised in this

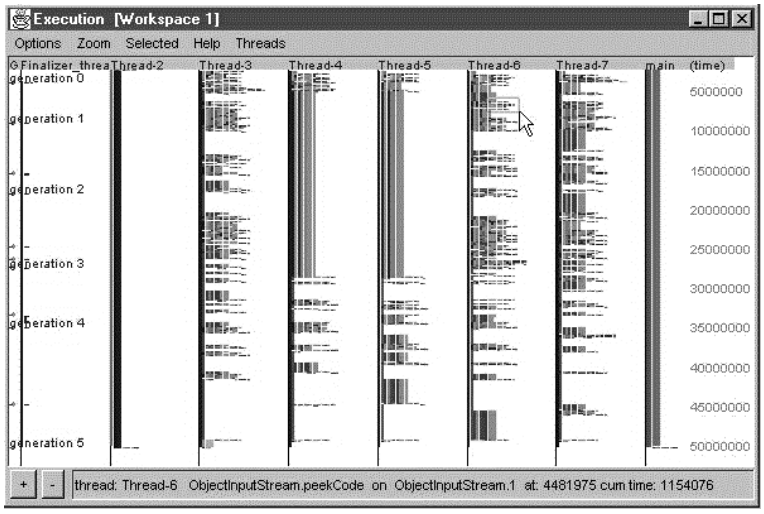

Figure 6: An example of the *execution* view in Jinsight.

visualisation tool.

Figure 6 shows an example of the *execution* view. This view shows an overview and details of communications among objects per thread as a function of time. Within each thread, an object is represented by a vertical stripe coloured according to the object's class. Time progresses downward in the view (time labels are displayed at the right side of the view). The top edge of a stripe coincides with the time when the particular method it represents was called, and its height represents its total execution time. Zooming into this view enables users to see the name of each method as well as its arguments and return type. By default, the leftmost column is reserved for garbage collection information.

Figure 7 shows an example of the *table* view. This view shows measures of execution activity and memory for threads, packages, classes, methods or objects. In Figure 7, the cumulative processing time spent by each method is shown along with their total number of calls. The measures of execution activity and memory that can be displayed vary according to whether the table is showing threads, packages, classes, methods or objects. For classes, for example, other such measures are package name, instance size, number of methods called, number of instances created and number of live instances.

Figure 8 shows an example of the *call tree* view. This view displays method invocations as *call tree* paths. The nesting of method calls is shown graphically as parent nodes 'owning' child nodes. For each method, its contribution to the cumulative execution time as well as the number of times it has been called are shown to the right side of the screen.

Figure 9 shows an example of the *object histogram* view. This view displays object instances grouped by class, indicating their level of activity and dependencies. The name of the object appears in the leftmost column, and to

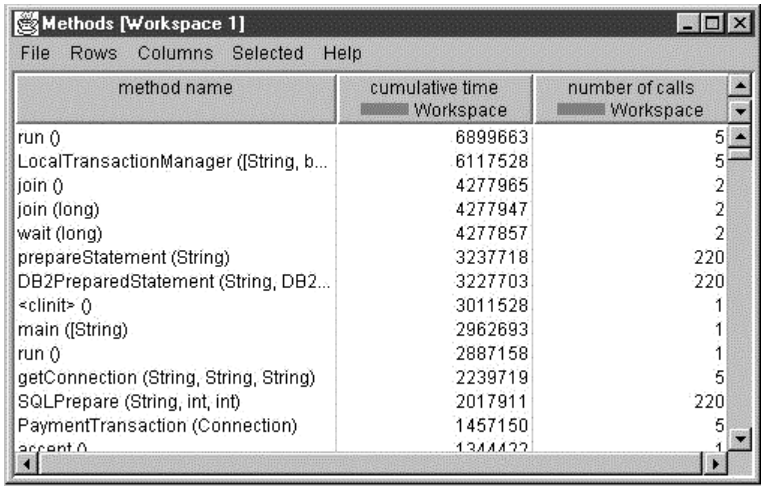

Figure 7: An example of the *table* view in Jinsight.

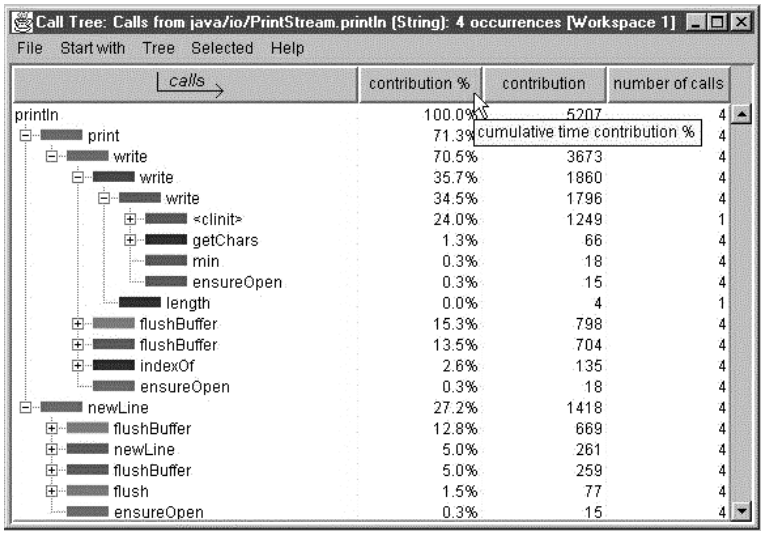

Figure 8: An example of the *call tree* view in Jinsight.

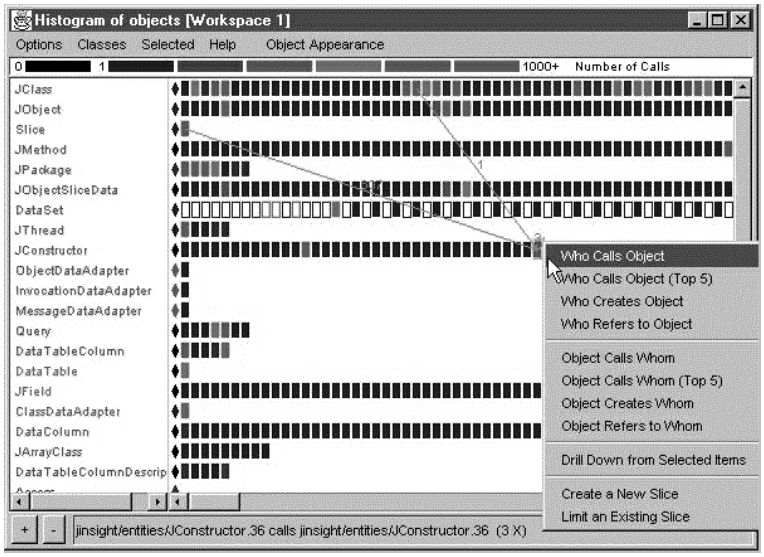

Figure 9: An example of the *object histogram* view in Jinsight.

its right there is a diamond shape and several rectangles. The diamond shape represents the class object for a given class, while each of the rectangles to its right denotes an instance of that object. The color of the rectangle denotes the level of activity of the particular instance, light coloured rectangles are those with a low number of calls, while dark ones denote high number of calls. White rectangles represent objects that have been garbage collected. On clicking on an object additional information about it can be accessed (who calls it, who has created it, etc).

Figure 10 shows an example of the *method histogram* view. This view shows methods grouped by class indicating their level of activity. Each rectangle to the right of a class name represents a method. Similarly, in the *object histogram* case, the colour of this rectangle is related to the method's level of activity. Light colours represent a low amount of processing time dedicated to this method, while dark colours mean the method has been assigned high amounts of processing time. Methods associated with darker colours can be considered as candidates for optimisation, as they consume the most time. Additional information about a specific method (such as its callers or the methods which in turn it calls) can be accessed by clicking on it.

There is a wide range of functions for manipulating the representations presented by this visualisation tool. A summary of these follows.

Within the execution, *object histogram* and *method histogram* views, selecting a rectangular area of the display (by dragging the mouse) causes the tool to zoom and display only this rectangular area on the whole display. Also, when passing the mouse pointer over an entity of these views, a tooltip window will display more detailed information about this entity. In the case of the *object histogram* view, for example, the additional information will be about the

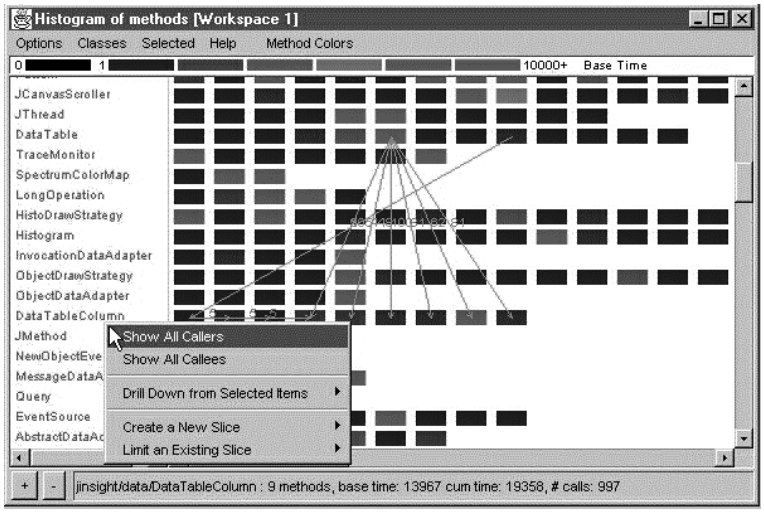

Figure 10: An example of the *method histogram* view in Jinsight.

object's class and instance identifier, its time of creation, cumulative time spent in its methods and so forth.

Within any view, selecting an area of it and using the options of the *selected* menu allows users to switch to other views but focusing on this selected part of the data.

3.2.3 CodeVizor

CodeVizor is a tool for printing class hierarchies (http://www.codevizor.com/). The main feature of this tool is its ability to produce static information about the class hierarchies of an application (see Figure 11), even from .zip or .jar class files. CodeVizor's editor divides the screen into two areas. The window to the left (the global window) shows all the classes involved in a project. By clicking on any of the classes in this window, a window to the right (the local window) displays only the selected class and those related to it. Each of these windows can present different views of the project. By clicking on the bottom bar icons, the user can switch between several views of the data. In the case of the global window, these views are the derivation, hierarchy, package and file view. In the local window these views are the diagram and source view. The derivation view provides a bottom-up (child-to-parent) view showing what classes a particular class is derived from. The hierarchy view provides a top-down (parent-to-child) view showing what classes are derived from a particular class. The package view depicts a project's packages and the classes they contain. The file view shows the files in the project and the classes they contain. The diagram view renders diagrams of a project's class hierarchy (top-down) or class derivations (bottom-up). Finally, the source view provides a read-only view of the source code using colour syntax highlighting.

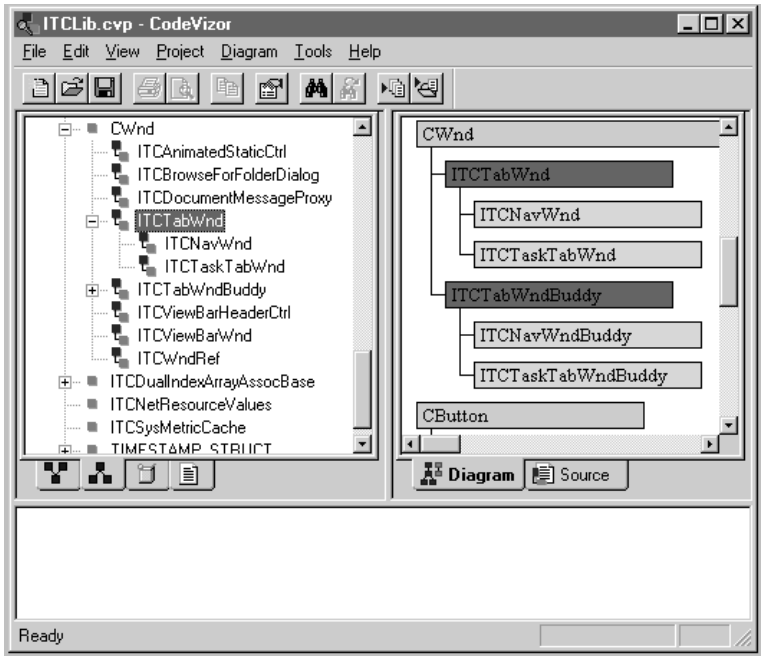

Figure 11: Sample visualisation in CodeVizor.

Note that this tool does not allow users to execute the application program, it only visualises static information about its class hierarchies.

3.2.4 VisiComp

VisiComp is a visualisation tool that dynamically displays the data structures built by the program (http://www.visicomp.com/product/index.html). As seen in Figure 12, its visualisation window is divided into two areas; the area to the right displays the overview of the data structure, while the left side shows a portion of it in greater detail. The portion to show is selected by clicking on the desired part of the overview area. The toolbar at the top of these two areas enables users to zoom in or out of the selected area, to customise the visualisation window to show these two areas as top and bottom, or to show only one on the whole screen. This toolbar also controls the execution of the program, allowing the user to step trace, stop or execute it in 'slow motion'. The main limitation of this tool is the fact that it only supports one view of the program.

3.3 Learning environments

Tutoring environments employ external representations mostly as a way to graphically show the behaviour of the program. The assumption here seems to be that graphical external representations, by allowing some inferences to be

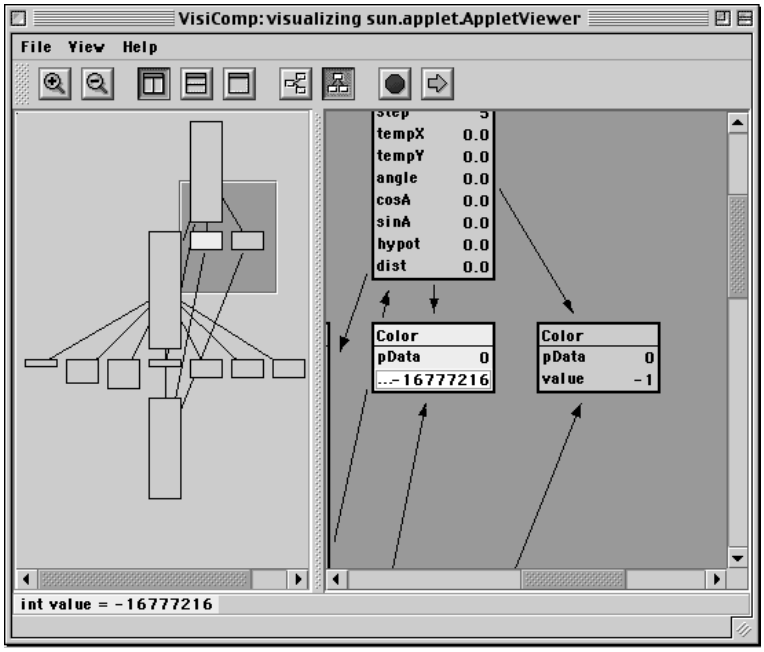

Figure 12: Sample visualisation in VisiComp.

Environment window	Information modality	Information type highlighted	Programming concepts represented
Classes window	Diagrammatic	Data structure	Classes
Objects window	Diagrammatic	Data structure	Objects

Table 7: Summary table for BlueJ windows

more tractable (Stenning & Oberlander, 1995), are more suitable for teaching purposes and therefore the term 'tutoring environment' normally implies a graphical environment.

3.3.1 BlueJ

BlueJ is a Java tutoring environment built by the School of Network Computing at Monash University, Australia (Kolling, 2000). Like CodeVizor, BlueJ graphically displays the static class hierarchy built by the program. However, through BlueJ the user can interact with the static data structure representation to create associated objects and execute their methods. By clicking on one of the non-abstract class icons, BlueJ presents a menu that allows the user to perform several functions, such as to create objects of that

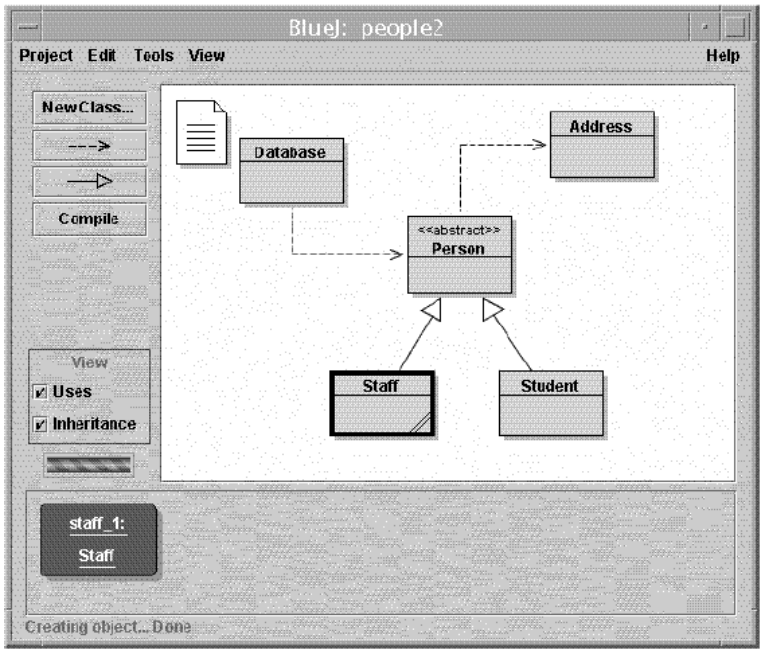

Figure 13: Sample execution in the BlueJ learning environment.

class, edit the class implementation or compile it even before the application program has been created. The generated objects are displayed in the project's bottom window (see Figure 13). As with class icons, by clicking on object icons the user can execute their public operations or to inspect their variables' current values. In this way, the tool supports students by showing them how their created classes build dynamic data structures and how these change when their methods are executed. In particular, by displaying the static data structure hierarchy separated from the objects dynamically created, it stresses the difference between these two concepts. The editing of the classes' implementation is done by conventional methods (with a text editor) and the debugging module of this tool is similar to those described earlier. In this way, BlueJ is a visualisation tool in which students can evaluate expressions involving classes or objects in a graphical, interactive way. Table 7 shows a summary of the properties of the two views comprised in this learning environment.

3.3.2 Cocoa

Cocoa is a programming tutoring environment for children (http://cocoa.apple.com/cocoa/). Cocoa can run simulations on worlds designed by the programmer. A world is basically a set of board pieces and rules that dictate the behaviour of the pieces. When the world is run, all the pieces which have rules and are on the board get a chance to try their rules at every clock tick. Cocoa allows the programmer to draw pieces and then to

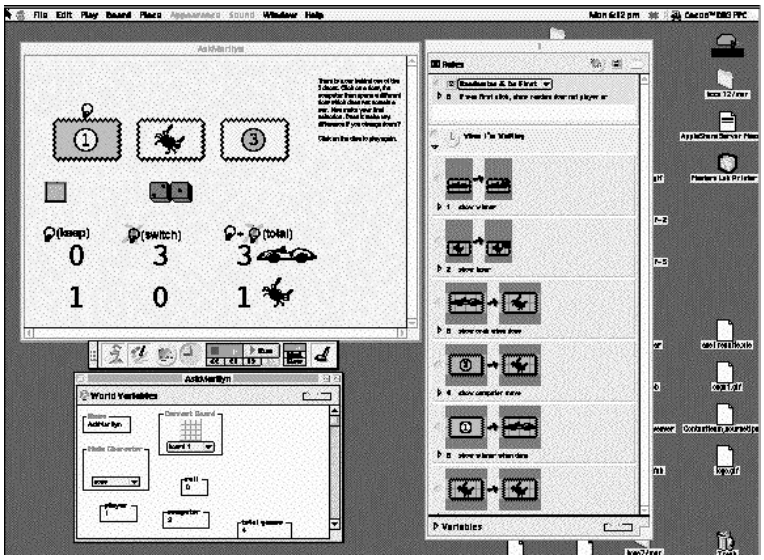

Figure 14: Sample execution in the Cocoa learning environment.

Environment window	Information modality	Information type highlighted	Programming concepts represented
World window	Diagrammatic	Data structure	Objects
World variables window	Diagrammatic	Data structure	Variables
Rules window	Diagrammatic	Condition-action	Code of rules

Table 8: Summary table for the windows in Cocoa

define their rules of behaviour in a graphical way. As shown in Figure 14, the Cocoa interface comprises several windows. The main window is the world window. It is in this area where the simulation or execution of a world takes place (the upper left window in Figure 14). Below this window there is a tool bar that enables users to draw board pieces, specify their behaviour rules, execute, execute in slow motion, step trace or stop the simulation. Below the main window and its tool bar there is the world variables window. This window contains the simulation's variables with their current values. When the simulation is not in execution, by clicking on any of the board pieces a window which enables users to edit its rules appears (the long window at the right in Figure 14). Each rule is represented as a condition action statement in which the antecedent depicts the initial state of the piece on the board and the consequent shows the state of this piece once the rule is applied.

Table 8 shows a summary of the properties of the different views comprised in this learning environment. Cocoa is similar to BlueJ in that users only have to define the behaviour of the board pieces (classes in BlueJ), and then they can create instances of these pieces (objects) and execute the world (their methods).

3.4 Visual languages

Visual programming languages do not employ graphical representations as an alternative to propositional code but use diagrammatic representations as the program source. As in the case of learning environments, the assumption is that graphical representations are better to encode programs. The empirical evidence, however, does not always support this assumption (Green, Petre, & Bellamy, 1991).

3.4.1 Prograph

Prograph is an object-oriented, visual, data-flow programming language for electronics design which has been commercially successful (http://www.pictorius.com/prograph.html). Its visual code editor, illustrated in Figure 15, supports the definition of methods. Each method can be defined in a separate window, and within each definition, its parameters appear at the top of the method's window, while its outputs are the variables at the bottom of it. Lines join these variables with the built in operators or with the nested methods (shown as icons) that process them. These nested methods' icons accept parameters at the top and produce output variables at the bottom. A double click in a non-primitive icon opens their method window in turn. In this way, data objects are carried from top to bottom of the screen, highlighting a data-flow perspective of the program.

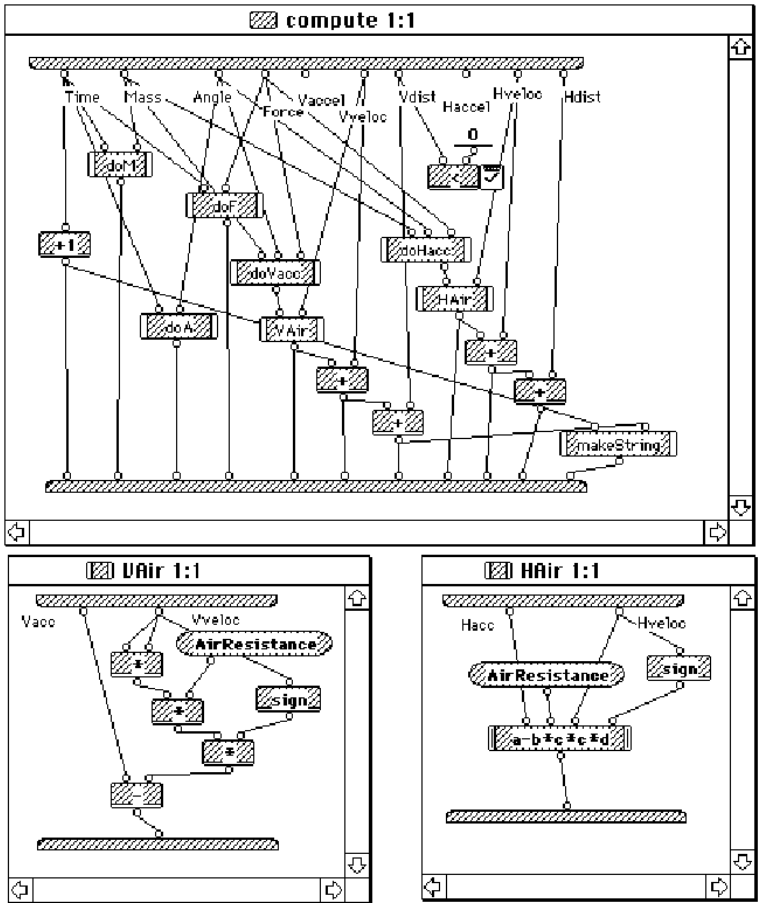

Figure 15: Visual editing window in Prograph.

From Green et al. (1991)

4 An analysis of the reviewed representations

This section offers an analysis of the code visualisations reviewed in the previous one. First, these representations are analysed in terms of their programming perspective and information modality. Next, the common characteristics and formalisms within these representations are identified and described. Finally, an analysis of these common formalisms in terms of Green's (1989) Cognitive Dimensions is presented.

4.1 Information types and modality

This section analyses the representations of the programming environments described above in terms of the information types highlighted by them and their information modality. The information types considered will be similar to those defined by Pennington (1987b): function, data-flow, control-flow, condition action and data structure. Function refers to what the program does. Data-flow concerns the transformations which data elements undergo as they are processed. Control-flow refers to the sequence of actions that will occur when the program is executed. Condition-action is information about the events that will result when a set of conditions is true. Finally, data structure relations are those that concern the type and number of program objects that are transformed during the course of program execution. In the Object-Oriented paradigm a distinction has to be made between static and dynamic data structure relations. Descriptions at the level of classes and subclasses are of the 'static' type. Normally, these relationships will not change during the execution of the program. On the other hand, descriptions at the level of objects and their variables are considered as 'dynamic' because they are dictated by the execution of the program.

The difference between data-flow and data structure some times is not clear. They both refer to data, but data structure refers to the hierarchical relations that a piece of data establishes within and outside itself, while data-flow stresses the path or 'threading' that a piece of data travels as the program executes.

Generally speaking, a representation highlights some information types, but this does not mean that other information types are not present or cannot be derived from it. The rest of this section discusses each of the information types considered in terms of the particular representations that highlight them.

It is interesting to note that all the debugging browsers within the development environments considered here employ representations which are mainly propositional. The reasons for this are not clear, although execution efficiency and consideration of the display area as a limited resource might explain this decision. On the other hand, the visualisation and tutoring environments considered employ mostly graphical representations. An example of a system that combines representations of different modality is Jinsight. This system employs representations with different degrees of 'graphicality' which also sometimes include tabular components.

4.1.1 Function

Program execution is the main source of information about function. The problem is that sometimes this is not fine grained enough for programmers to understand/debug programs. A fine degree of granularity is offered by debugging tools that allow programmers to run the program in a line by line fashion. In graphical applications, for example, if there is a window showing the execution of the program, the programmer can see the impact of every line of code on the behaviour of the application. Another source of functional information is the 'output' window that debugging tools sometimes offer, for example Visual J++ and JBuilder which display error messages and exceptions. This information is displayed in a textual form.

In Object-Oriented languages, and especially for certain applications, functional information can be easily accessible from data structure accounts. Systems like VisiComp and BlueJ, for example, assume that by showing data structure information (classes, objects and their variables executing their methods), users will be able to infer the program's function. Although this might be possible for the examples presented (small programs dealing with database information about staff and students in a university, for example), there might be other kinds of applications for which this is not so.

Simulation systems like Cocoa seem to be situated at the extreme where data structure and function information are almost equivalent. In Cocoa, the simulation presented when the world is being executed can be considered either as a functional or as a data structure representation.

4.1.2 Data-flow

Data-flow focuses on the dynamical aspect of the 'threading' of data objects through the execution of the program. The only programming environment with representations that highlight data-flow is Prograph (see Section 3.4.1). The 'path' that data objects traverse as the program (method) executes is shown by the lines joining variables with nested methods. Data objects 'travel' from top to bottom of the window, and in this way, nested methods receive their input variables at the top and produce their output data at the bottom.

4.1.3 Control-flow

The main source of propositional control-flow information are the threads and call stack windows in all the debugging browsers presented. These windows present a list of threads/methods that, similarly to the complex variables in the watch window, can be expanded to show their associated methods (the methods they call as part of the execution of the program). The tree window in JaViz and the *call tree* view in Jinsight (Figures 5 and 8 respectively) are examples of graphical representations highlighting this information type. In both cases control-flow information is presented as a tree whose nodes are the methods executed and its hierarchical relation is determined by the program's

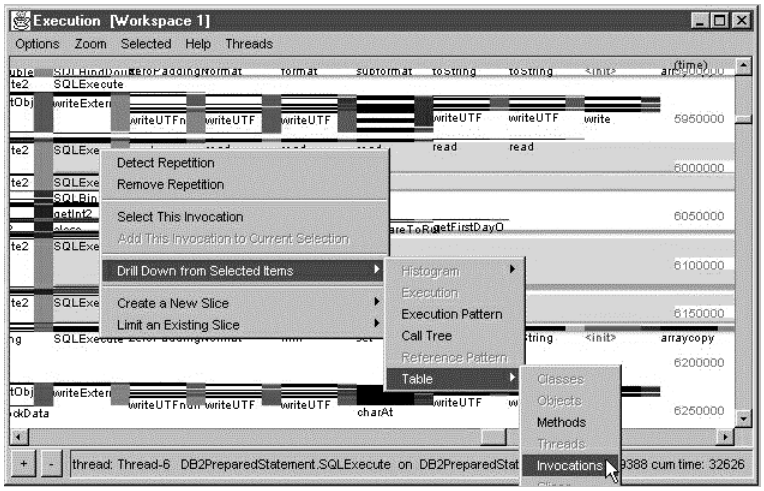

Figure 16: Zoomed view of control-flow information in the execution view in Jinsight.

calling sequence. A good summary of this information, which displays the concurrent nature of threads is given in the execution view in Jinsight (Figure 6). By clicking on a certain area of this view, a more granular account of the control-flow information of this area can be shown (see Figure 16). This view represents an alternative to the threads and call stack windows in the debugging browsers or the *call tree* diagram in JaViz and Jinsight. However, Jinsight's execution view highlights control-flow information but as a function of time and stressing the concurrent nature of thread execution.

4.1.4 Condition-action

This information type does not seem to be as important for Java as it would be for, say Prolog or for an expert system programming shell. The clearest example of condition action information is given in the behaviour rules in Cocoa.

4.1.5 Data structure

There are two sides to data structure information, a static and a dynamic aspect. Class hierarchies of an application, like those displayed by CodeVizor, represent the static aspect of data structure information, while accounts of the way objects execute their methods, like those presented by VisiComp or BlueJ, represent its dynamic side. Data structure information, especially when displayed in a representation with a strong graphical component, seems to be the preferred way to visualise code for tutoring purposes in an Object Oriented language like Java. More propositional representations of this information type are employed by debugging browsers in the watch variables window.

The predominant graphical convention to represent data structure information seems to be to use boxes for data entities and lines or arrows to specify their relationships. VisiComp shows dynamic data structures only, while BlueJ shows both dynamic and static data structures. However, in the latter system the emphasis is on the relations between static data structures; the dynamic ones are only shown as a product of their associated classes.

Almost all of the development environments attempt to show data structure information by textual means through the *watch variables* window. Users select the variables they want to watch and this window will show these variables throughout the execution of the program. If these variables are complex structures, they can be 'expanded' to show their elements. Probably the main difference between the propositional and graphical approach to show data structure relationships is that the textual watch window does not specify static relations (class hierarchy). However, it seems that specifying both kinds of relations in graphical data structure representations could potentially confuse novices.

Finally, the table and histogram views in Jinsight (Figures 7, 10 and 9) do not seem to highlight any of the information types referred to here. These views seem to stress levels of execution activity that are important for code tuning.

4.2 The common representations

The most common representations in the environments described are those used to represent control-flow (the threads/call stack window of debugging environments, the tree window in JaViz and *call tree* view in Jinsight) and data structure (the watch window of debugging environments, VisiComp and BlueJ).

As mentioned before, the propositional version of both control-flow and data structure representations presents a set of elements representing threads or methods that help to describe this information type. Every element is displayed on a different line, and if they are complex structures they can be expanded to show their components. When they expand, their components are shown with an indentation to denote this hierarchical relation. These components, if complex, can in turn be expanded in a recursive fashion. Therefore, what is being displayed is a hierarchical tree.

The *call tree* view in Jinsight follows almost the same format. The only difference is that every method entry displays additional information in a table-like form.

Graphical control-flow and data structure representations comprise a network of boxes joined by lines. The boxes represent program entities (methods, classes and objects) and the lines represent relations between them. In the case of control-flow, boxes represent methods and lines show the methods' invocation path. Graphical control-flow representations belong to the language T of trees, which according to Marriot and Meyer (1997), belong to the type of context-free constraint multiset grammars.

In the data structure case, boxes stand for either classes or objects and lines can represent either hierarchical relations (the class of an object, a class that was defined as having objects as variables) or data structure relations (a list of objects). This type of representations belong to the language G of graphs, which according to Marriot and Meyer (1997), belong to the type of context-sensitive constraint multiset grammars.

The discussion about notational properties in the next section focuses on these two information types and associated representations.

4.3 A notational analysis of the common external representations provided by programming environments

The notational analysis presented here is developed in terms of the activity types the common representations support and the five dimensions described in Section 2.3 (closeness of mapping, role-expressiveness, hard mental operations, diffuseness/terseness and hidden dependencies).

The activity these representations support is program comprehension. They can be considered as comprehension aids and as such, to be useful, programmers have to perform a kind of translation from the program code to these additional representations and back. Therefore, the activity type that can be performed over these representations can loosely be described as transcription. However, it has to be noted that translation is more a means than an end. The objective of the programmer is not to translate between representations but to understand a piece of code. This program understanding may, most of the time, also be a supporting activity to perform program debugging or re-use, for example. Another difference is that this translation is more of a recognition rather than of a production task. Programmers do not produce these representations, rather, they recognise them as related (and as an alternative) to the program code.

These representations seem to have a good closeness of mapping. According to Pennington (1987b), a program comprises several kinds of information and some of these information types dominate the mental representations that programmers build when comprehending a program. Research with Object-Oriented programming languages indicates that function is an important information type in this paradigm (Wiedenbeck & Ramalingam, 1999; Wiedenbeck, Ramalingam, Sarasamma, & Corritore, 1999; Corritore & Wiedenbeck, 1999). It has been mentioned that, for certain kinds of applications, data structure representations also seem to highlight function. In these cases, it could be said that data structure representations have a good closeness of mapping. However, one important aspect that has to be taken into account is that these representations represent a kind of secondary notation to the program code, and as such, it is not clear how they could support this main representation.

According to Green and Petre (1996), role-expressiveness can be enhanced by a secondary notation. Representations additional to the code, when used in

program understanding tools can be considered as a kind of secondary notation, not so much to the code, but to the programming environment. However, it is not clear whether these representations should highlight the same or different information types to the code. Wiedenbeck and Ramalingam (1999) and Wiedenbeck et al. (1999) have found that Object-Oriented programming code seems to highlight function at the cost of obscuring other information types such as data-flow and control-flow. Therefore, one of the common representations identified seems to be redundant to the code (data structure), while the other seems to be complementary to it (control-flow). An interesting research question is whether representations additional to the code, and secondary notations in general, should be redundant and highlight similar information to the main notation or be complementary and highlight different information types. The answer to this question is likely to depend upon factors such as program size and complexity and the programmer's level of skill among others.

Hard mental operations is a dimension particularly important for representations used as an aid for program understanding. In theory, users should be able to understand and even simulate the execution of their programs using only the source code. One of the main motivations to have representations additional to the code is to make 'life easier' for the programmer, especially when trying to understand dynamic aspects of their code. It seems that the representations analysed here do not impose hard mental operations on the user. Perhaps one difficulty with them is that the 'execution history' is not recorded and therefore the user might sometimes miss the exact moment when, for example, a variable changed its value. Some way of recording the history of execution, like in the Prolog tracer described in Patel et al. (1997), would be appropriate here.

Although each of these additional representations might not impose hard mental operations on the programmer on its own, an issue that has to be considered is the difficulty of co-ordinating several of these secondary representations and the program code. Studies on co-ordination of representations in other fields such as the learning of physics (Sime, 1996), first order logic (Oberlander, Stenning, & Cox, 1999), arithmetic (Ainsworth, Wood, & Bibby, 1996; Ainsworth, Wood, & O'Malley, 1998) and general problem solving (Cox & Brna, 1995) have highlighted the difficulty to co-ordinate several representations. This difficulty has not been considered, to the best of our knowledge, in the design of programming environments and in computer programming in general.

Regarding the diffuseness/terseness dimension, the main difference is between propositional and diagrammatic representations. Propositional representations normally employ an identifier for a program entity, while diagrammatic representations use an identifier, boxes and lines. This seems to consume a considerable amount of screen space and therefore propositional representations are the best choice if the display area is considered as a limited resource. As mentioned before, this might be one of the reasons why commercial debugger browsers prefer propositional to diagrammatic representations.

One of the aims of representations additional to the code is to make hidden or obscure dependencies explicit. Note that in order to achieve this several of these additional representations have to be presented at the same time (as in the debugger browsers described earlier). If only one of them is considered some information might be lost and therefore dependencies will be hidden (for example, it is impossible to obtain control-flow information from the watch variables window). On the other hand, if many additional representations are presented to the user it has already been mentioned that she might experience difficulties in co-ordinating them. Therefore, there is a tension between making hidden dependencies explicit and the difficulty of co-ordinating several representations.

The analysis for these common representations shares many characteristics with that of visual programming environments in Green and Petre (1996). The main difference is that in the present case, the common representations, together with the program code, have to be considered as a system. This is why the issue of co-ordination of representations is of a central relevance here.

5 Conclusions

This paper has presented an overview of the external representations employed in some of the most popular object-oriented programming environments. These representations have also been characterised briefly in terms of their information perspective and modality. Additionally, standard ways to represent control-flow and data structure information in the representations taken into account for this survey were identified. Finally, a brief analysis in terms of Green's (1989) Cognitive Dimensions was presented for these control-flow and data structure representations.

From this analysis it seems that an important potential problem that has to be considered here is the difficulty of co-ordinating these, or indeed any other, additional representations. Also, another important issue has to do with whether representations additional to the code, and secondary notations in general, should be redundant and highlight similar information to the main notation or be complementary and highlight different information types. More theoretical knowledge about about the way these representational systems influence the comprehension of computer programs is needed.

References

Ainsworth, S. (1999). The functions of multiple representations. *Computers & Education*, *33*(2-3), 131–152.

Ainsworth, S., Wood, D., & Bibby, P. (1996). Co-ordinating multiple representations in computer based learning environments. In Brna, P., Paiva, A., & Self, J. (Eds.), *Proceedings of the 1996 European Conference on Artificial Intelligence on Education*, pp. 336–342 Lisbon, Portugal.

Ainsworth, S., Wood, D., & O'Malley, C. (1998). There is more than one way to solve a problem: evaluating a learning environment that supports the development of children's multiplication skills. *Learning and Instruction*, *8*(2), 141–157.

Bergantz, D., & Hassell, J. (1991). Information relationships in PROLOG programs: how do programmers comprehend functionality?. *International Journal of Man-Machine Studies*, *35*, 313–328.

Booch, G., Rumbaugh, J., & Jacobson, I. (1998). *The Unified Modeling Language User Guide*. Addison-Wesley.

Cheng, P., Lowe, R., & Scaife, M. (1999). Cognitive science approaches to understanding diagrammatic representations. *AI Review*, *In press*.

Corritore, C. L., & Wiedenbeck, S. (1991). What do novices learn during program comprehension?. *International Journal of Human-Computer Interaction*, *3*(2), 199–222.

Corritore, C. L., & Wiedenbeck, S. (1999). Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human Computer Studies*, *50*, 61–83.

Cox, R., & Brna, P. (1995). Analytical reasoning with external representations: Supporting the stages of selection, construction and use. *Journal of Artificial Intelligence in Education*, *6*(2/3), 239–302.

Détienne, F. (1990). Expert programming knowledge: a schema-based approach. In Hoc, J., Green, T. R. G., Samurçay, R., & Gilmore, D. J. (Eds.), *Psychology of Programming*, pp. 205–222. Academic Press, London, U.K.

Eisenstadt, M., Brayshaw, M., & Paine, J. (1991). *The Transparent Prolog Machine*. Intellect, Oxford, England.

Gilmore, D. J., & Green, T. R. G. (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, *21*(1), 31–48.

Green, T. R. G. (1989). Cognitive dimensions of notations. In Sutcliffe, A., & Macaulay, L. (Eds.), *People and Computers V*, pp. 443–460. Cambridge University Press, Cambridge.

Green, T. R. G. (1999). Building and manipulating complex information structures: issues in Prolog programming. In Brna, P., du Boulay, B., & Pain, H. (Eds.), *Learning to build and comprehend complex information structures: Prolog as a case study*, pp. 7–28. Ablex, Stamford, Connecticut.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming languages: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, *7*, 131–174.

Green, T. R. G., Petre, M., & Bellamy, R. K. E. (1991). Comprehensibility of visual and textual programs: a test of superlativism against the 'match-mismatch' conjecture. In Koenemann-Belliveau, J., Moher, T. G., & Robertson, S. P. (Eds.), *Empirical Studies of Programmers, fourth workshop*, pp. 121–146 Norwood, NJ. Ablex.

Kazi, I. H., Jose, D. P., Ben-Hamida, B., Hescott, C. J., Kwok, C., Konstan, J. A., Lilja, D. J., & Yew, P. C. (2000). JaViz: a client/server Java profiling tool. *IBM Systems Journal, 39*(1), 96–117.

Kolling, M. (2000). The Bluej tutorial. Tech. rep. 2000-01, School of network computing, Monash University.

Marriot, K., & Meyer, B. (1997). On the classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing, 8*, 375–402.

Merrill, D. C., Reiser, B. J., Beekelaar, R., & Hamid, A. (1992). Making processes visible: scaffolding learning with reasoning-congruent representations. *Lecture Notes in Computer Science, 608*, 103–110.

Mulholland, P. (1997). Using a fine-grained comparative evaluation technique to understand and design software visualization tools. In Wiedenbeck, S., & Scholtz, J. (Eds.), *Empirical Studies of Programmers, seventh workshop*, pp. 91–108 New York. ACM press.

Nix, D., & Spiro, R. J. (1990). Cognitive flexibility and hypertext: Theory and technology for the non-linear and multi-dimensional traversal of complex subject matter. In Spiro, R. J., & Jehng, J.-C. (Eds.), *Cognition, Education and Multi-media: Exploring Ideas in High Technology*, pp. 163–205. Erlbaum, Hillsdale, New Jersey.

Oberlander, J., Stenning, K., & Cox, R. (1999). Hyperproof: Abstraction, visual preference and modality. In Moss, L. S., Ginzburg, J., & de Rijke, M. (Eds.), *Logic, Language, and Computation, Vol. II*, pp. 222–236. CSLI Publications.

Ormerod, T. C., & Ball, L. J. (1993). Does design strategy or programming knowledge determine shift of focus in expert Prolog programming?. In *Empirical Studies of Programmers, fifth workshop*, pp. 162–186 Norwood, New Jersey. Ablex.

Patel, M. J., du Boulay, B., & Taylor, C. (1997). Comparison of contrasting Prolog trace output formats. *International Journal of Human Computer Studies, 47*, 289–322.

Pennington, N. (1987a). Comprehension strategies in programming. In Olson, G. M., Sheppard, S., & Soloway, E. (Eds.), *Empirical Studies of Programmers, second workshop*, pp. 100–113 Norwood, New Jersey. Ablex.

Pennington, N. (1987b). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology, 19*, 295–341.

Rist, R. S. (1986). Plans in programming: definition, demonstration and development. In Soloway, E., & Iyengar, S. (Eds.), *Empirical Studies of Programmers, first workshop*, pp. 28–47 Norwood, New Jersey. Ablex.

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, *13*, 389–414.

Romero, P. (2001). Focal structures and information types in Prolog. *International Journal of Human Computer Studies*, *54*, 211–236.

Sime, J. (1996). An investigation into teaching and assessent of qualitative knowledge in engineering. In Brna, P., Paiva, A., & Self, J. (Eds.), *Procedings of the 1996 European Conference on Artificial Intelligence on Education*, pp. 240–246 Lisbon, Portugal.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *10*(5), 595–609.

Stenning, K., & Oberlander, J. (1995). A cognitive theory of graphical and linguistic reasoning: logic and implementation. *Cognitive Science*, *19*(1), 97–140.

Twyman, M. (1979). A schema for the study of graphic language. In Kolers, P. A., Wrolstad, M. E., & Bouma, H. (Eds.), *Processing of Visible Language*, pp. 117–150. Plenum Press, New York.

Wiedenbeck, S., & Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human Computer Studies*, *51*, 71–87.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, *11*, 255–282.