1

OF/DE

1

PM-1 3 1/2"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Exsulat - Université de la Colombie

Chapitre - Notre Université

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

The Visual Object Editing Kit

by

Duong B.T. Nguyen, B.C.S.

thesis submitted to the

Faculty of Graduate Studies and Research

in partial fulfillment of

the requirements for the degree of

Master of Computer Science

Ottawa - Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

March 15, 1993

copyright

1993, D.B.T. Nguyen

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your lib. / Votre bibliothèque

Our lib. / Notre bibliothèque

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-84098-6

Canada

Name DUONG NGUYEN

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

0984

U·M·I

SUBJECT TERM

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern 0401
Literature 0294
General 0295
Classical 0297
Comparative 0298
Medieval 0316
Modern 0591
African 0305
American 0352
Asian 0355
Canadian (English) 0593
Canadian (French) 0311
English 0312
Germanic 0315
Latin American 0313
Middle Eastern 0314
Romance 0370
Slavic and East European 0372

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology 0323

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
Archaeology 0326
Cultural 0327
Physical 0310
Business Administration 0272
General 0770
Accounting 0454
Banking 0338
Management 0385
Marketing 0501
Canadian Studies 0503
Economics 0505
General 0508
Agricultural 0509
Commerce Business 0510
Finance 0511
History 0358
Labor 0366
Theory 0351
Folklore 0578
Geography 0578
Gerontology 0578
History 0578
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science 0615
General 0616
International Law and Relations 0617
Public Administration 0814
Recreation 0452
Social Work 0626
Sociology 0627
General 0938
Criminology and Penology 0631
Demography 0628
Ethnic and Racial Studies 0629
Individual and Family Studies 0630
Industrial and Labor Relations 0700
Public and Social Welfare 0344
Social Structure and Development 0709
Theory and Methods 0999
Transportation 0453
Urban and Regional Planning 0453
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and Nutrition 0476
Animal Pathology 0359
Food Science and Technology 0476
Forestry and Wildlife 0479
Plant Culture 0480
Plant Pathology 0817
Plant Physiology 0777
Range Management 0746
Wood Technology 0306

Biology

General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0379
Ecology 0329
Entomology 0353
Genetics 0369
Immunology 0793
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oceanography 0416
Physiology 0433
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics 0786
General 0760
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 0566
General 0300
Audiology 0992
Chemotherapy 0567
Dentistry 0350
Education 0769
Hospital Management 0758
Human Development 0982
Immunology 0564
Medicine and Surgery 0347
Mental Health 0569
Nursing 0570
Nutrition 0380
Obstetrics and Gynecology 0354
Occupational Health and Therapy 0381
Ophthalmology 0571
Pathology 0419
Pharmacology 0572
Pharmacy 0382
Physical Therapy 0573
Public Health 0574
Radiology 0575
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry 0485
General 0749
Agricultural 0486
Analytical 0487
Biochemistry 0488
Inorganic 0738
Nuclear 0490
Organic 0491
Pharmaceutical 0494
Physical 0495
Polymer 0754
Radiation 0405
Mathematics 0605
Physics 0986
General 0606
Acoustics 0606
Astronomy and Astrophysics 0608
Atmospheric Science 0748
Atomic 0607
Electronics and Electricity 0798
Elementary Particles and High Energy 0759
Fluid and Plasma 0609
Molecular 0610
Nuclear 0752
Optics 0756
Radiation 0611
Solid State 0463
Statistics 0463

Applied Sciences

Applied Mechanics 0346
Computer Science 0984

Engineering 0537
General 0538
Aerospace 0539
Agricultural 0540
Automotive 0541
Biomedical 0542
Chemical 0543
Civil 0544
Electronics and Electrical 0348
Heat and Thermodynamics 0545
Hydraulic 0546
Industrial 0547
Marine 0794
Materials Science 0548
Mechanical 0743
Metallurgy 0551
Mining 0552
Nuclear 0549
Packaging 0765
Petroleum 0554
Sanitary and Municipal 0790
System Science 0428
Geotechnology 0796
Operations Research 0795
Plastics Technology 0994
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis

The Visual Object Editing Kit

submitted by **Duong B. T. Nguyen, B.C.S.**,
in partial fulfillment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

Chairman, School of Computer Science
Carleton University

Date

Abstract

This thesis focuses on the development of an object visualization tool kit called the Visual Object Editing Kit. The tool kit provides a facility to visually inspect and edit Smalltalk objects using direct manipulation. An object can be decomposed into smaller components using the visual editor. The decomposition of an object is represented by a graph of interconnecting icons. Novice users can use the editor to examine the various parts of an object. The editor provides a simple mechanism for a user to access the components of an object without having to know the syntax of Smalltalk. Advanced users can also use the visual editor in place of the Smalltalk Inspector to examine objects.

Acknowledgments

I would like to thank my thesis advisor, Professor Wilf Lalonde, for his comments and support to make the completion of this thesis possible

To my family, I would like to express my deepest gratitude. They have always been a source of encouragement and support

Special thanks go to my friends who spent many long hours to proofread the drafts of this thesis. I would like to especial thank Greg Adams for his suggestions during the writing of this thesis.

Table Of Contents

	Page
Acceptance Sheet	ii
Abstract	iii
Acknowledgments	iv
Table Of Contents	v
Chapter 1: Introduction	1
1.1 Overview	3
1.2 Intended User	5
1.3 Overview of the Thesis	6
Chapter 2: Visual Programming	7
2.1 What is Visual Programming?	7
2.2 Classification of Visual Programming Techniques	9
2.2.1 Visualization of Data Structure	10
2.2.2 Visualization of Programs	12
2.2.3 Visual Program Specifications	12
2.2.4 Visual Program Execution Flow	15
2.3 Related Work	17
2.4 Limitations of Visual Programming	20
2.5 Summary	20
Chapter 3: The Visual Object Editing Kit	22
3.1 The Facilities In Detail	23
3.2 Icons	24
3.3 The Icon Library	25
3.4 The Object Library	27
3.5 The Object Editor	28
3.6 The Structure Editor	30
3.7 Examples of Object Inspection	32
Chapter 4: Design and Implementation	36
4.1 Icons	36
4.1.1 Accessing An Icon	39
4.1.2 Collection Icons	39
4.2 Editors	39
4.3 Icon-Editor Interaction	40
4.4 Object Cloning	41
4.5 Object Mutation	42
4.6 Summary	42
Chapter 5: Conclusions	44
5.1 Assessment of Work Done	44
5.2 Limitation of the Editors and Icons	45
5.3 Future work	46
References	48

Chapter 1: Introduction

Visual programming has been gaining popularity in recent years because it provides an intuitive and straightforward way of programming. With the decreasing cost of high-resolution graphics workstations, visual programming has become available to a larger population of computer users. Although the number of computer users is growing rapidly, few users can program using conventional programming languages. Since conventional languages are difficult to learn and use for the majority of the population, visual programming systems attempt to make the task of programming simpler and more intuitive [Myers 89].

There are two main categories of visual programming [Shu 85]. The first category of visual programming provides a visual environment for presenting information or programs. In the second category, programming languages are designed to support visual interaction so that programs can be created using visual expressions (see Figure 1.1). This thesis focuses on the visualization of data structures.

Figure 1.1: A classification of visual programming

In either categories, a visual programming system makes use of graphics and direct manipulation to provide a simple and comprehensible representation of data and algorithms. Instead of using textual commands to retrieve information and to perform tasks, pictures are used as the means of communication. Pictures need not be used solely

for displaying information. They can also be used for editing and creating structured information. It should be possible for a user to interact with pictures on the screen in a manner that is similar to the way the information might be graphically drawn, changed, or more generally manipulated on paper. Users can express their requirements in terms of pictures and diagrams, a mode that is much more natural to unsophisticated computer users. When carefully designed, such visual programming systems would permit them to develop simple applications even without knowing the details of conventional programming languages.

This thesis describes the design and implementation of an iconic data visualization tool kit for Smalltalk called the Visual Object Editing Kit. This kit can serve as an important component of a visual programming environment for Smalltalk. The editing tools do not offer support for creating programs (i.e., methods) using visual interactions. The tools in the kit provide the ability to visually create and edit objects using direct manipulation without changing the fundamentals of the Smalltalk language. These tools fall into the visualization of data structures category that was mentioned earlier. Objects are represented by small pictures known as icons. Using visual editors, a user can define or modify a class definition by interacting with icons. An instance of an object can be decomposed into smaller components and is displayed as a graph of interconnecting icons. A visual inspector for objects permits a user to graphically view, change, and create new components by manipulating the graphical representation of the object.

A Smalltalk Inspector is a tool used for debugging and provides a method to examine Smalltalk objects. With this text based interactive tool, a user can examine and modify the contents of an object by using a few mouse motions and key strokes. A text based presentation of objects is enhanced with the addition of graphics. When text is combined with pictures to describe objects, the resulting displays provide a more understandable

description of the objects. Instead of using a text-based inspector, the Visual Object Editing Kit provides a graphical (or visual) inspector on objects which themselves are displayed as icons. The graphical inspector can be used as a generic object editor in place of the text-based inspector. Similar to the Inspector, the visual inspector can be used to edit any objects in the Smalltalk environment. Specialized editors can be added to the editing kit to facilitate editing specific types of objects. For example, a graphical editor to examine binary trees can be developed by overriding a subset of protocols provided by the framework to specifically display and access the trees.

1.1 Overview

The Visual Object Editing Kit consists of a collection of libraries and editors which provide a visual mechanism for creating and reusing objects. The editing kit contains an object browser for examining the library of objects. Objects in the library are displayed as icons. To prevent novice users from being overwhelmed from an otherwise large library of objects [Skublic 91], they can initially be provided with a limited set of objects thereby reducing the quantity of objects that they need to understand. As the user becomes more familiar with the objects in the library, additional Smalltalk objects could be added, e.g., by the instructor. Advanced users can use the editor to edit any object in Smalltalk by sending a message to the desired object.

A visual object editor is invoked from the library by double-clicking on the desired object. The object is displayed as an icon in the editor. A user can explore the object by pointing at the parts (sub-components) of the object and clicking on them with a mouse. The user traverses the object by expanding components until he reaches the desired level of details. For example, the decomposition tree of a person object can be displayed to be similar to

the diagram shown on Figure 1.2. Each node in the tree represents a component, and each edge represents the relationship between a parent object and its component.

Figure 1.2: The components hierarchy of a person object.

Unlike the diagram above, the components of an object in the Visual Object Editing Kit are displayed as icons. Directed lines point from the parent icon to each of the component icons. The parent object can be modified by reconnecting a component to a new object. To do this, the user selects another object from the library and drops it into the object editor. A line is drawn connecting the parent to the new object (see Figure 1.3). The user can edit the appearance of an icon by manipulating the icon's attributes; e.g., by invoking a picture editor to change the icon's picture.

Figure 1.3: A value of a linked-list is changed by connecting it from object A to object B.

To create a new type of object, a user selects an object that resembles the desired object and adds new variables to the existing object until the desired object is created. The new variables can be instantiated by connecting them to other objects. After the new object is created, it can be saved in a library. New objects are created by combining several existing components together. These objects, in turn, can be used to form other objects thus fostering the notion of re-usable components.

1.2 Intended User

The system is designed so that novice and experienced users will be able to make use of the visual inspectors (editors). Most of the tools are currently geared towards novice users so that it is easier for them to examine and modify the large library of Smalltalk's objects. One way for a Smalltalk novice to learn about objects is to play with them. By actually manipulating the components of an object directly, a great amount of information about the structure and behaviour of an object can be discovered. In particular, objects containing circular structures can be easily noticed. For example, a linked-list structure may have a link that points back to one of its previous links. Such circular references are difficult to observe using standard inspectors but easy to see with an iconic inspector (see Figure 1.4).

Figure 1.4 A graphical display of a circular linked list

Novice users who are learning Smalltalk can visualize and describe an object as an instance. Such instances can then be used as a base (or prototype) for new objects by copying them. The prototype can then be extended by adding new parts (as instance variables). Consequently the Visual Object Editing Kit is instance-based rather than class-based, i.e., classes are never manipulated, only instances are. The Visual Object Editing Kit can be used by experienced users for examining objects while debugging complex algorithms.

The advantage of using a visual editor is that the user can immediately see the changes made to a data structure. More importantly, a novice user can change the value of an object without having to know the syntax of Smalltalk.

1.3 Overview of the Thesis

Chapter 2 provides background information on visual programming. In the survey, we discuss and categorize visual programming techniques. Chapter 3 introduces the Visual Object Editing Kit and the tools it provides. Examples are used to illustrate how an object is visually constructed. Chapter 4 describes the design and implementation details of the kit. Finally, chapter 5 provides conclusions and suggests future work.

Chapter 2: Visual Programming

This chapter introduces the basic concepts of visual programming by providing an overview of the programming techniques that are available in visual programming systems. The Related Work section discusses the visual techniques that are used in visual programming systems. Although the goal here is to create a tool kit for iconic (or visual) inspectors, it is important to have an understanding of visual programming systems in general since they serve as a model for the capabilities that we wish to design and implement in the Visual Object Editing Kit. In particular, a major goal is to incorporate this tool kit in a language that is not considered to be an iconic programming language, i.e., in Smalltalk.

2.1 What is Visual Programming?

Visual programming is defined by Shu as "the use of meaningful graphic representations in the process of programming. Programming can be defined as specifying a method for doing something the computer can do in terms the computer can interpret." [Shu 88] A visual program can be displayed as a collection of meaningful pictures or icons interconnected in a graph-like structure with clearly defined underlying semantics. Since the human brain is adapted for multi-dimensional perception, visual programming systems present information or algorithms in two (or more) dimensions to take advantage of this spatial sensing capability [Chakravarty 90 et al.] A visual programming system is an environment satisfying at least one of the following [Chang 90]

- high-level graphical entities are available as atoms that can be manipulated as programs.
- high-level graphical entities are available as atoms that can be manipulated by programs

- high-level graphical entities form an integral and essential part of the display generated by the system during the programming phase or at execution time

The graphical capabilities and direct manipulation facilities of visual programming systems make it enticing and feasible for non-programmers to learn how to program simple but useful applications [Shu 88]. Using direct manipulation, graphical information is immediately displayed to inform the user of changes that have been performed on important data [Chang 90]. An example of this is the programming language LOGO. With the addition of a graphical object (a Turtle) to LISP, LOGO became very popular among children [Tanimoto 86]. The quick feedback provided by LOGO and the notion of a bi-directional conversation between the user and the computer emerged as one of the major factors in popularizing such systems. Up to that point, communicating with the computer was a one-way conversation. Rather than simply extracting information from a machine or feeding it with data, the user could now control the process by interacting directly with the information displayed on the screen [Chang 90].

Visual programming systems utilize tools such as a mouse, menus, windows, forms, diagrams and icons to enhance the communication of information between the user and the computer. By using simple mouse motions to express instructions, the syntax of the language is reduced. These visual tools guide a user to perform operations that are syntactically correct [Chakravarty 90 et al.]. Consequently, users no longer need to remember detailed command line instructions and syntax to perform basic tasks. When the syntax of the language is removed, programming becomes easier, particularly for inexperienced programmers. Compare the point and click approach of Microsoft Windows to the command line interface of DOS - performing file commands (such as dir, copy, delete) is a matter of performing a few mouse operations (point and click). These tools become more effective when they are applied to applications that require modifications to

forms or changes to layouts. Such changes are tedious to make when they must be done using textual programming commands [Chakravarty 90 et al], e.g., compare constructing and configuring a window via code versus doing it using a window builder.

Textual notations may not be effective mechanisms to explain information, especially to novices who are in the process of learning [Chakravarty 90 et al]. Charts and diagrams accompanied with text can express information in a clearer manner because they provide a higher level of abstraction; i.e., a more understandable view of the information [Myers 89]. Trees and linked-lists are frequently drawn as diagrams to simplify explanations. The relevant notions are easier to understand if they are visually described. Algorithms are explained easier with pictorial displays such as data flow diagrams, finite state machine diagrams or flow charts. These visual representations are well-known for their ability to assist in the understanding of programs.

Visual programming takes these visual representations one step further by allowing a user to program in a similar manner. Instead of acting only as visual aids, a user is able to use visual programming tools to create data structures in a format that closely mirrors the corresponding "mental" pictures. Programs are created by connecting together simple pictures that represent small tasks, or by performing certain gestures to illustrate the operation of an algorithm. Data structures (in addition to code) can be manipulated and described on the screen in a manner similar to the way they would appear on paper.

2.2 Classification of Visual Programming Techniques

The following sections describe the most relevant tools and the important aspects of visual programming systems. In particular, we focus on how the tools are used to display or

create data structures and programs. We also provide basic background information where relevant.

2.2.1 Visualization of Data Structure

Visualizing data structures is one of the most effective ways in which visual programming can be applied [Myers 89]. A user can understand information better when the associated data structures are displayed as active pictures on a screen. A user can select and examine different components of the data structure using direct manipulation. For more details, the sub-components in turn can be inspected until ultimately a set of basic or atomic structures is reached. The result is a decomposition tree (or graph) that illuminates the important structure of the data. The data visualization tool is more than just an inspection facility. It also permits the user to change the structure by deleting or inserting components at any inspection point.

Some algorithms can be explained to a user more easily if successive alterations to the example data structures are displayed. For example, the insertion of a value into a linked-list can be easily understood if the inserted link is visually displayed as shown in Figure 2.1a and Figure 2.1b.

Figure 2.1a: A new link is to be inserted between A and C

Figure 2.1b: The result of the list after B is inserted.

Similarly, differences between the results of two algorithms are observable when the results are compared graphically. For example, a user can observe the differences between destructively and non-destructively appending two lists (see Figure 2.2a and Figure 2.2b). But such tools are not just for novices. Data structure visualization tools can also be used by experienced users during debugging sessions to enhance their understanding of their programs.

Figure 2.2a. The "appended" list is created as a result of a non-destructive append

Figure 2.2b The result of destructively appending "list2" to "list1"

2.2.2 Visualization of Programs

Program visualization [Myers 89] tools are used to graphically display previously written programs or to illustrate the run-time execution of existing programs. This term is not to be confused with visual programming because the program need not have been created visually. Program visualization can be as simple as pretty printing a piece of code or as complex as animating the execution of a running program. These tools can be used to provide multiple views of code at different levels of abstraction, thus improving the readability and comprehensibility of the code. Novice users can use these tools to learn about the execution of an algorithm that is difficult to describe but easy to follow when the steps can be observed [Shu 88]. For example, understanding is enhanced if the user can see how the divide-and-conquer technique works in a quick sort algorithm. Program visualization tools are also useful to experienced users when testing and debugging complex code. By watching the execution flow of a program when it is running and by examining the data, a user can gain useful insights about the program.

2.2.3 Visual Program Specifications

There are several ways of specifying programs or information graphically in a visual programming system. Programs can be specified as inter-connecting icons or as traditional flow-charts and data-flow diagrams. This section describes the specification tools that are typically used in a visual programming environment.

Diagram-Based Specification

Diagram-based specification refers to traditional diagrams such as flow-charts, data-flow diagrams or state-transition diagrams. A user creates a program by selecting a program structure template such as an IF-THEN-ELSE or a REPEAT-UNTIL and filling in the prompts with values to complete the program. Structures can be nested in one another to build more complex expressions. This technique removes syntax errors by providing a template with the correct language syntax. PIGS (A System for Programming with Interactive Graphical Support) is a visual programming environment that uses structured flow-charts for programming [Pong 84 et al.]. PIGS uses Nassi-Shneiderman diagrams (NSD) to describe and construct programs (see Figure 2.3). It is intended to help in the process of developing and maintaining programs. The system provides a template that represents a desired NSD, and the user fills in the template. The user can watch the execution of his program, make modifications to the NSD, and then continue execution. The system is capable of reducing syntax errors in Pascal by providing a template for Pascal constructs.

Figure 2.3: An example of an NSD module that can be created with PIGS

Form-Based Specification

Form-based specification is also known as table-based specification and is usually built on top of database systems. A query to a database is created by providing values for a selected number of columns in a relational table. The database system searches the table

for all entries that matches the given query. FORMAL [Shu 85] is a form-based system. FORMAL simplifies queries by providing a visual representation of data structures through the use of forms. A user specifies a command by providing a few relevant properties within the fields of a form.

Icon-based Specification

In an icon-based specification, icons; i.e., simple pictures (possibly containing words), are used to represent actions or data [Chang 90]. The icon's picture is intended to "suggest" the logical meaning of the data or action. In a well-designed iconic system, pictures help a user recognize familiar ideas quickly even though they lack the details that numbers and words could provide. The object represented by an icon can be modified through direct manipulation. Complex icons are constructed through composition of other icons.

Icon-based systems are typically used in conjunction with diagram-based specification facilities to support iconic programming. Programming is performed by selecting and connecting icons together. The resulting program is a graph where the nodes represent data or actions and the arcs represent the relationship between the nodes. The main problem with icons is that their meanings are often ambiguous [Chang 90]. Although they typically take up less screen space than a word explanation, they are not as clear or precise. For this reason, icons are often accompanied by words to make them more understandable. As users become familiar with the meaning of the icons, the words often become unnecessary.

PROGRAPH is an object-oriented programming environment which connects icons together to create a program [Cox 88 et al.]. It is a fully integrated programming environment that creates and modifies programs and data structures by manipulating icons. PROGRAPH is based on both LISP and PROLOG with extensions for inheritance.

The original implementation used both data-flow and control-flow structures but control-flow facilities have been removed and replaced by data-flow "CASE" structures in later versions. Because it is based on data flow, there is no need for local or temporary variables. Complex data structures are constructed from simpler components. Similarly, operations called methods (as in Smalltalk), are designed by interconnecting the icons for previously defined methods. In a method definition, inputs are located at the top and outputs are located at the bottom. Data flows from the inputs through a network of method invocations (uses) that transforms the data and produces the result at the bottom of the method definition.

2.2.4 Visual Program Execution Flow

Visual programming environments are generally heavily oriented towards either control-flow, data-flow, or constraints.

Control-Flow Systems

Control-flow systems make use of control structures such as IF-THEN-ELSE and REPEAT-UNTIL. They are commonly extensions of procedural languages where program execution proceeds sequentially from one point in a piece of code (or memory location) to another. PIGS, described earlier, is a control-flow system.

Data-Flow Systems

Data-flow systems are the most common visual programming systems. When a program executes in a data-flow environment, each execution step is displayed from the perspective of the data. Data elements are the active agents, and they flow through a network of nodes (or operations). An operation becomes active only when all of its input data is available. When active, the operation transforms the input data to produce a set of output data and

forwards the output data to successor operations in the network. PROGRAPH, described above, and Fabrik [Ingalls 88 et al] are data-flow systems. Fabrik is an interactive graphical environment that attempts to make programming possible for novice programmers. The system provides a library manager, graphical iconic editors, an interactive run-time environment and debugging tools. Complex programs can be created in Fabrik by connecting simple components to create a data-flow graph. Objects are created by visually modifying existing objects in the library to meet new criteria. Once a new object is created, it is then stored in a library for later reuse. Users can visually browse and edit existing components in the library and get immediate feedback from the construction process.

Constraint-Based Systems

In a constraint-based system, a program is defined by creating objects graphically and then specifying the constraints between the objects. When parts of an object are modified by external means, the related objects must also change to satisfy the constraints between them. This type of programming separates conditions to be satisfied from the actions to be performed. ThingLab [Borning 81] is a constraint-based iconic programming system that was designed to construct "dynamic models of experiments in geometry and physics." To construct a program, a user connects graphical building blocks and specifies the constraints between them. Icons representing objects or actions are dragged from the library and dropped into a workspace where they are interconnected using lines representing bi-directional constraints. Equality constraints can also be specified by placing two or more icons on top of each other until they "lock" together. The result is a constrained set of icons that can move or adjust themselves to satisfy the imposed constraint when an outside influence, like a user, makes changes.

2.3 Related Work

This section examines some related works in the area of data and object visualization systems. The important features of each system are emphasized to show how they relate to the Visual Object Editing Kit.

INCENSE is a visual system that is used for displaying data structures [Myers 83]. Data structures are designed using pictorial representations in a Pascal-like language called Mesa. A user can choose forms from a set of default forms or create new forms to represent new data structures. A user-defined form can display a data structure at a high level of abstraction to hide unnecessary details. During debugging, INCENSE can generate graphical displays of selected data structures providing graphical representations that are similar to those a user could have drawn on paper. An example of INCENSE displays are shown in Figure 2.4.

Figure 2.4: A record structure and an array structure are displayed using INCENSE.

KAESTLE is a graphical editor for Lisp data structures [Shu 88]. It supports the display and the direct manipulation of list structures. It is particularly useful for examining linked-lists that would be difficult to display textually; e.g., circular list structures since these are difficult when displayed only in textual form. A display of a list using KAESTLE is shown in Figure 2.5.

Figure 2.5: A display of a list in KAESTLE.

The Visual Object Editing Kit contains similar ideas to the systems that are mentioned above. It permits objects in Smalltalk to be displayed graphically as they might be drawn on paper. Furthermore, the object and its components can be altered using direct manipulation. Myers stated that, "Other languages, such as CLU and Smalltalk, have the definition of types and their operations as the central programming paradigm. A data display system like INCENSE would be a great asset to programmers using a language of either type." [Myers 83] The Visual Object Editing Kit is an equivalent of INCENSE for Smalltalk.

The Visual Tool Set [Borges 90 et al.] is a multiparadigm visual programming language built on top of Foible. Foible [Jindrich 90] is an object-oriented framework designed for creating visual programming systems. One of the tools that the Visual Tool Set provides is a Data Definition Tool (DDT). The DDT is used to create data types and data objects by combining primitive objects such as numbers, strings and dates into more complex composite objects. The primitive objects are displayed as boxes inside the box for a composite object.

The Visual Object Editing Kit provides a facility that is similar to the DDT in the Visual Tool Set. A new object is defined by adding new variables to a previously defined object in the Visual Object Editing Kit. A value is assigned to the new variable by connecting the variable to a desired value. The new object is stored in a library so that it can be reused as

a component of other objects. The idea of reusing objects by placing them in a library is taken from Fabrik. It is suggested in Fabrik that, "a successful visual programming kit requires only three things: Specification of an effective visual and computational interface for each component, interactive access to an interesting (network) library of existing components, and the ability to use and combine these components interactively to build new library components and finished applications " [Ingalls 88 et al]

Digitalk's PARTS Workbench is a commercial product that is used to develop graphical applications in Smalltalk. The layout of an application window is created by dragging and dropping components from a parts window into a workbench. In the workbench, visual components such as buttons and windows are visually defined by placing smaller components inside them (e.g., a window is composed of panes and a menu is composed of menu items.) Non-visual components such as arrays or files are displayed as icons in the workbench. The behaviour of the application is defined by drawing connections between the parts that have been laid out in the workbench.

Similar to PARTS, the Visual Object Editing Kit supports the drag and drop of components to construct an object using existing parts. In both environments, a copy of an object is created when it is dropped from one window to another. Unlike PARTS, the editors in the Visual Object Editing Kit does not provide a mechanism to define the behaviour of an object.

In summary, the Visual Object Editing Kit provides tools to visually create and manipulate objects (or data) instead of emphasizing tools for program visualization. Using direct manipulation, objects are edited by manipulating a graph of interconnecting icons. A new object is created by altering an existing one, and by re-using objects in a library to create sub-components.

2.4 Limitations of Visual Programming

As with any other programming style, visual programming is not the perfect solution to programming. There are several drawbacks to visual programming. The meaning inferred from icons in a visual programming system can be ambiguous, and it is very difficult to find a set of icons that are universally interpreted the same way by everyone. Even a simple icon sometimes contains too much information and it is up to the person observing it to extract the correct meaning. Icons are designed to be language independent but their intended meanings may not be obvious to everyone. Visual programming systems have not yet proved themselves to be general purpose. Most were designed as experimental systems and have a limited domain and limited functionality. Additionally, pictures as a rule take up more storage space and more importantly screen space than the text they replace. A piece of code that takes half a page of text could take four or five times more space to display iconically. So far, there are few experimental results indicating that iconic languages are better than conventional textual languages for programming complex applications.

2.5 Summary

Visual programming must demonstrate that it is able to solve real-world problems in order to thrive in the future [Myers 89]. At the very least, it must be applied in appropriate domains where its application will be beneficial. Currently, textual programming languages are more appropriate for general programming requirements. There are several outstanding issues that visual programming systems must resolve. Substantial effort will have to be spent on

- developing tools to manage large, complex structures,
- providing automatic layout of graphs, trees or other complex data structures,

-
- specifying or displaying complex algorithms with appropriate icons such that they can be understood better than their equivalent textual counterparts,
 - determining fast and efficient ways to update multiple views of structures when modification are made in one view,
 - providing a universal set of icons to eliminate possible misinterpretations,
 - avoiding reintroduction of the iconic equivalent to spaghetti-like code due to the lack of abstraction facilities at the iconic level, and
 - providing flexible programming environments for diverse types of users (experienced users should not be slowed down and novice users should not be overwhelmed).

On the positive side, visual programming [Myers 89] is proving useful by

- helping novice users to learn how to program,
- allowing non-programmers to construct simple applications graphically, and
- helping with the construction of user interfaces (window builders are visual programming tools).

Chapter 3: The Visual Object Editing Kit

This chapter introduces the facilities supported by the Visual Object Editing Kit. Examples are used to illustrate how objects (and their icons) can be inspected, created, and edited.

Most of the facilities discussed in this chapter have corresponding classes that implement them. Consequently, it is convenient to use the notation “a **MultiWordClass**” to refer to an instance of the specified class where the individual words in the class name are replaced by their lowercase counterparts and separated by spaces. Thus we speak of “an **IconLibrary**” as a short form for an instance of class `IconLibrary`; more specifically, as a short form for “an icon library”.

To begin with, an object can be represented iconically (in **unexpanded** form) as shown in Figure 3.1

Figure 3.1: An unexpanded icon.

When **expanded**, the parts (or components) of the object are indicated by directed edges from the name of the component to the corresponding associated object (see Figure 3.2).

Figure 3.2: An expanded icon with associated components

3.1 The Facilities In Detail

The Visual Object Editing Kit consists of two libraries (for objects and icons) and three editors (two for objects and one for icons)

Libraries

- The **ObjectLibrary** is a repository that is used to store useful reusable objects. These objects are used as components for building new objects
- The **IconLibrary** is a repository that stores reusable icons. The icons are associated with new classes of objects. It is also possible to modify the attributes of an icon; e.g., the picture (a bitmap), the name (a string), and a filter. A **filter** is a constraint that specifies which components should be displayed when an icon is expanded, e.g., instance variables that are uninteresting or private need not be displayed unless explicitly requested by the user.

Note that both libraries display their contents iconically. An icon in the object library is attached to an object; i.e., represents an object. An icon in the icon library is unattached; i.e., represents itself. Moreover, there is only one instance of each library.

Editors

- An **ObjectEditor** is an editor that permits objects to be viewed, edited, and constructed. Existing objects can be obtained from the object library (or moved to the library from this editor). New objects can be obtained via a Smalltalk expression prompter. It also permits the object's parts (or components) to be viewed and specified. A directed edge connects the name of the component (in the object) to the associated value as shown in Figure 3.2.
- A **StructureEditor** is an editor that permits new kinds of objects (and their classes) to be created iconically. These new kinds of objects are created from existing objects by adding new instance variables.
- A **PictureEditor** is a simple bitmap editor that permits the pictures associated with icons to be modified.

We will first consider icons, the icon library, and picture editors. Next we will consider the object library, object editor, and the structure editor.

3.2 Icons

Several kinds of icons are needed to represent objects in Smalltalk. Some objects, for example, have only named instance variables; e.g., fractions and bitmaps. Others have indexed instance variables; e.g., sets and ordered collections. Object with only named instance variables are represented by **SimpleIcon**. A **SimpleIcon** (see Figure 3.3) is displayed as a picture with an attached text description. Recall that an icon can be unexpanded (see Figure 3.1) or expanded to show its components (see Figure 3.2). Container objects are represented by one of four different types of icons. Currently, specialized icons are provided for **IndexedIcon**, **OrderedIcon**, **DictionaryIcon**, and **SetIcon**. An **IndexedIcon** is used to represent collections like arrays with a fixed size and integer indices. An **OrderedIcon** is used to represent collections like **OrderedCollection**.

that can grow in size. Similarly, DictionaryIcon and SetIcon are specialized for dictionaries and sets respectively. Examples are illustrated in the following sections

Figure 3.3. Sample icons in the Visual Object Editing Kit.

3.3 The Icon Library

The IconLibrary stores icons as they appear in unexpanded form (see Figure 3.4). Since an icon can be referenced by name in Smalltalk code, the names of the icons in the IconLibrary must be unique. Icons intended to be associated with instances of a class are named after the class itself. Instances of subclasses will inherit the icon unless they themselves have more specialized icons defined in the IconLibrary.

Figure 3.4: The IconLibrary.

New icons are created in the IconLibrary by duplicating an existing icon or by creating a new one. By double-clicking on the icon, a `PictureEditor` is invoked; e.g., when the `LinkedList` picture is double-clicked, an editor containing the picture shown in Figure 3.5 appears.

Figure 3.5: Editing the picture of an icon in a `PictureEditor`.

Bits (squares) can either be made opaque and coloured or they can be made transparent in the tradition of most bit editors

It is possible to modify the name of the icon from the IconLibrary using the menu item 'Rename' under the 'Utilities' menu. When this menu item is selected, a prompter will ask the user for the new name of the selected icon. An icon is associated with a class by having the name of the icon be the same as the class name that it represents. The Dictionary icon in the IconLibrary (see Figure 3.4) is used to represent dictionary instances. The IconLibrary is also used to specify which instance variables are to be filtered from the expanded form through menu items (not explicitly shown). Recall that filtered variables are instance variables not important enough to display when the icon is expanded. To filter out an instance variable, for example, it is sufficient to add it to a list. It is equally simple to remove it from this list. In essence, the filter is the collection of instance variables whose connections are initially hidden when the icon is expanded.

3.4 The Object Library

The ObjectLibrary (see Figure 3.6) is intended to be a repository for reusable objects with sufficient complexity that it is worth storing them for future use. As of this writing, we have not as yet created sufficiently complex objects to merit storage in the object library. For illustration purposes, we have nevertheless created a number of simple objects.

Figure 3.6: The IconLibrary.

An object can be copied from the ObjectLibrary and placed in an ObjectEditor by simply dragging the object from one place to the other (we refer to this as the **drag and drop** facility). It can be moved just as easily in the other direction; i.e., from an editor to the library. When an object is dropped into another window, a copy of it is created and placed underneath the mouse pointer.

3.5 The Object Editor

An object can be visually inspected by sending it a message **visualInspect**; e.g.,

```
#(a b c) visualInspect
```

When this message is sent to an object, an ObjectEditor appears containing the inspected object (see Figure 3.7). The object is displayed iconically in the editor.

Figure 3.7: Inspecting an array.

From the editor, it is possible to obtain additional objects either from the object library (using the drag and drop facility), or by constructing them via expression prompters (by supplying the Smalltalk code to execute). For example, simply expanding the icon as shown in Figure 3.8 would make the icons for the components visible.

Figure 3.8: Expanding the array

After adding a new number, say named 'Number: 0' as shown in Figure 3.9, it is easy to rebind the first element of the array to the number 0 by clicking the mouse on element 1 and dragging to the new 'Number: 0' icon

Figure 3.9: Binding element 1 of the array to number 0.

There are several ways in which the name of an icon is defined. By default, the name of most icons are set to be 'Unnamed' until they are changed by the user. In the above examples, the name of the array #(a b c) is 'Unnamed' when it is inspected in the ObjectEditor (see Figure 3.7). In Figure 3.9, the value 0 was named 'Number' by the user using a menu option. The name of the selected icon is changed by selecting 'Rename' option in the 'Utilities' menu. Icons that represent components of an object obtain their names from the parent object. In Figure 3.9, the names of the strings 'a', 'b' and 'c' are '1', '2' and '3' respectively. These names matches the index in which they belong in the parent object.

3.6 The Structure Editor

A StructureEditor is invoked on an object to construct a new kind of object iconically. Operationally, it is similar to the ObjectEditor since it supports the modification of existing components. However, the StructureEditor also permits new variables to be added to the object that is being edited, e.g., by selecting **Add** in the **Variables** menu. As an example,

Figure 3.10 illustrates the result of adding a new variable 'previous' to a linked list that originally only had two instance variables 'first' and 'rest'.

Figure 3.10: An instance variable called 'previous' is added to a LinkedList

Modifications are temporary until a special **commit** operation is performed. The commit is specified via a '**Mutate As...**' option under the **Variables** menu. Two mutation techniques are supported:

- The new object is made the sole instance of a new class (by providing a new class name).
- All objects of the original class are changed to look like the new one (by providing the same class name).

In either case, the new object (see Figure 3.11) ends up with the additional variables

Figure 3.11: The LinkedList after mutating into a BidirectionalLinkedList

The value for the new variable can be defined by connecting it to some object in the standard manner (as we illustrated above with the ObjectEditor). The icon could also be placed into the IconLibrary for subsequent reuse.

3.7 Examples of Object Inspection

This section provides examples of "typical" objects in the Smalltalk environment when they are inspected using the object editor in the Visual Object Editing Kit. These examples illustrate the usefulness of the visual editor when it is used to inspect circular structures. The first example is the BankAccount.

Figure 3.12: Inspection of a BankAccount.

In this example, the bank account consists of an account number, a balance, an owner and a transaction list (see Figure 3.12). The transaction list keeps track of the history of all the transactions that were performed on the bank account. A transaction simply contains the account and an amount to be added or subtracted from the balance of the account. The

following piece of code was performed to generate the above instance of the bank account.

```
| account |  
(account := BankAccount new)  
  accountNumber: 123.  
  ownedBy: 'Joe Smith'.  
(DepositTransaction on: account amount: 100)  
  postIntoAccount.  
(WithdrawalTransaction on: account amount: 10)  
  postIntoAccount  
account visualInspect
```

After the account was created, two transactions are posted into the account. The first transaction deposits \$100 into the account, and the second transaction withdraws \$10 from the account. Figure 3.12 shows that the account contains a collection with two transactions. Each transaction references back to the same bank account. The circular structure is clearly shown by the visual inspector which would not be shown by a text-based inspector.

To illustrate the point that circular structures are common in Smalltalk, the following examples will use the editor to inspect structures in the Visual Object Editing Kit. The next example shows the structure of the icon representing the bank account (see Figure 3.13).

Figure 3.13: Inspection of a SimpleIcon.

The label of the icon is called 'MyAccount', and the owner (the represented object) is the BankAccount object. The icon contains a list of variable names which are buttons (called SubNode). Each button is associated with exactly one variable in the BankAccount object. The button refers to the SimpleIcon as its parent. The circular structure exists between the SimpleIcon and each of its buttons.

The final example shows the relationship between a window, the pen and the drawing medium (see Figure 3.14).

Figure 3.14. Inspection of a window, a pen and a drawing medium

The circular structure exists between the window, the pen and the graphics medium. The window (called an ObjectConstructor) contains a graphics pen, the graphics pen contains a graphics medium, and the owner of the graphics medium is the ObjectConstructor window. Another circular between the graphics pen and the graphics medium as shown in Figure 3.14

The examples in this section show that it is quite often that circular structures exist between objects in the Smalltalk environment. The visual editor illustrates these kinds of structures better than a text-based inspector.

Chapter 4: Design and Implementation

The Visual Object Editing Kit provides a collection of editors which are used to visually create and edit objects. The current implementation can be extended further by modifying the basic set of visual classes and their methods. This chapter provides an overview of the design and implementation of the Visual Object Editing Kit. The issues that will be discussed here are the design and representation of the classes, the important protocol (at a generic level) that the classes (and their instances) use to communicate with each other, the interaction of icons and objects with their editors, the drag-and-drop mechanism, the cloning and the mutation of objects, and the limitations of the current implementation.

4.1 Icons

An icon is an interactive picture that represents an object. In the Visual Object Editing Kit, an icon contains six main components:

- a **represented object** (any object),
- a **picture** (a foreground bitmap and a mask bitmap: both are needed to support transparency),
- a **name** (a string),
- a **filter** (a collection of instance variable names that are not to be shown in the expanded form),
- **children** icons (one for each instance variable),
- a **position** (a point),
- a number of **buttons** associated with expanded icons.
- a **boolean** equivalent indicating whether or not the icon is expanded, and

A symbol is used rather than a boolean to make it easy to perform the required display operation, e.g., `#displayExpanded` versus `#displayUnexpanded`. Additionally, when an icon is expanded, some portion of the icon is allocated to each instance variable (either a named

variable or an indexed variable) Each area behaves as a button since clicking on the area causes the object referenced by the variable associated with it to appear along with a line connecting the button to the referenced object For example, Figure 4.1 illustrates an icon in its iconic mode and expanded mode In the expanded mode, the top part, 'Unnamed', is the name of the icon; beside it is the text description (obtain from `printString`), the remaining parts contain the index part names '1', '2' and '3' (button '1', button '2' and button '3'). Clicking on button '1' causes the string 'a' to appear along with the connection from the right side of button '1' to the string Clicking a second time will hide the connection and the associated value.

Figure 4.1: An icon in its iconic state and expanded state

Expanded icon buttons are relatively simple to implement. They just need to keep track of a rectangular area in the icon relative to some reference point such as the top left corner, and whether or not the associated value is hidden. The icon itself can keep track of the associated instance variable name and the associated value icon. Consequently, a relatively simple family of icons is needed to represent icon buttons (see Figure 4.2). Since dictionaries are key/value pairs, it is convenient to partition basic buttons into two parts

the left part devoted to a key and the right part to a value. Consequently, there are three minor variations of expanded icon buttons:

Figure 4.2 The ExpandedIconButton hierarchy.

The icon hierarchy is shown in Figure 4.3. Class `AbstractIcon` contains most of the important functionality. All objects can be represented either by a simple icon or an icon devoted specifically to representing a collection. Special collection icons are needed to support the iconic addition and removal of elements because different protocols are used by the different collection classes for adding and removing elements. For example, arrays must be indexed whereas sets must not.

Special icon classes can easily be added as illustrated by the addition of a folder icon class to represent hierarchies of icons in the object library. An icon in the icon library does not represent another object. It is actually implemented as an instance of a picture icon where the associated object is itself.

Figure 4.3 The icon hierarchy.

4.1.1 Accessing An Icon

Icons are also used to provide a common set of protocols to access and modify the components of an object. The 'get' and 'set' methods are called,

```
valueAt: aButton  
valueAt: aButton put: anIcon
```

The **valueAt:** answers the value of an icon that is currently connected to the given button when it is invoked. Since the button contains the part name (e.g., instance variable) of an object, the value associated to the instance variable can be changed by associating the button with another icon. The **valueAt:put:** method is used to associate a button with an icon. The ObjectEditor use these protocols to change the value of an instance variable when a button is reconnected to another icon

4.1.2 Collection Icons

Collection objects are treated a little bit differently from objects with named instance variables. The part name of a button in a CollectionIcon is the index of the collection. The **valueAt:** and the **valueAt:put:** methods are re-implemented in the collection icons so that access to a button will retrieve or modify the corresponding value at the index in the collection. With the use of these protocols, the components of an object can be modified regardless of the types of part names (named variables or index variable)

4.2 Editors

Editors are used in the Visual Object Editing Kit to modify various aspects of an icon or the object represented. An editor contains a list of icons that are displayed in it. When a user makes changes to an icon via mouse interactions, the mouse gestures are translated to input events by the editor. The editor captures the mouse events and forwards them to the

selected icon. The icon uses these mouse events to determine how to display or change the structure of the represented object. The editor also provides a display area for icons to draw themselves.

Most editors in the Visual Object Editing Kit support drag-and-drop. An icon can be dragged from one editor and dropped into another editor. Each editor must support two major messages: **“supportsDragAndDrop”** and **“addIcons: iconCollection from: sourceWindow”**. The first message is used to determine whether or not a window supports drag-and-drop. Windows that do not belong to the Visual Object Editing Kit return false when they receive the message **“supportsDragAndDrop”**. If a window returns true when it receives this message, it must support the message **“addIcons: iconCollection from: sourceWindow”**. This message must make a copy of the icons supplied from the source window and place them into the destination (or receiving) window.

4.3 Icon-Editor Interaction

An icon frequently receives messages from an editor indicating that a mouse event has occurred or asking the icon to display itself. A standardized display protocol and event handling protocol is supported by all icons. When an editor generates a clicked event, for example, for a specific icon, the icon determines whether or not any of the expanded icon buttons were selected (if the icon was not expanded, no button would be selected). If a specific button is selected, the icon sends a message back to the editor indicating that subsequent mouse events will be used to modify the state of the button. Otherwise, the editor receives a message from the icon indicating that subsequent mouse events are used for dragging the icon. Using the double-dispatching technique, the icon is given control to determine the type of action the editor should perform with a given mouse event.

Double-dispatching is a well-known technique for handling automatic conversion in Smalltalk. Rather than the receiver determining what to do, the receiver tells the parameter to do the work and provides the parameter with additional information about itself. To get a fraction to add an unknown number using operation “+”, for example, it is sufficient for the fraction to ask the unknown to “**addFraction**” itself. A float could respond to the request by converting the fraction to a float before the addition since it knows both its own type and the type of the parameter (a fraction) without having to ask. By contrast, a fraction could respond without type conversion. An integer would convert itself to a fraction before performing a fractional add.

4.4 Object Cloning

When an icon is dragged from one window and dropped into another window, the icon and the object represented must be copied. In general, a **deep copy** is needed, i.e., the kind of copy in which the object is copied, its parts are copied, their parts are copied, etc., until basic objects like small integers, characters, symbols, and classes are reached. Additionally, the copy operation must work properly in the presence of self-referencing objects (circular structures). Although equivalent operations are provided by object filers that are designed to write objects to disk, such a copy operation does not currently exist as a basic operation in the system. A special **clone** operation was provided in class **Object** specifically for this purpose; i.e., a **clone** is a deep copy of an object. The basic technique is to keep track of duplicated objects in an identity dictionary where the key is the original object and the value is the duplicate. Reaching the same object via different paths will, with the use of such a dictionary, result in retrieving the same copy (or duplicate).

4.5 Object Mutation

The mutation of objects in the structure editor uses concepts similar to that used by the **clone** operation. An object is mutated when its class definition is altered by adding (or removing) instance variables. Since Smalltalk/V does not permit a class definition to be altered when there are existing instances of the class, a facility is needed in the Visual Object Editing Kit to provide this capability. Before mutating the collection of objects (instances of the same type), each instance is converted into a dictionary. The keys of the dictionary are the variable names (named or indexed) and the values of the dictionary are objects that are referenced by the variable names. The instances are first mutated into these special dictionaries by using **become:**. Each dictionary acts as a place holder for the mutated object (i.e., references to a mutated object will now be referring to the corresponding dictionary). Now, the class definition can be changed to the new definition because there are no references to any of its instances. After the class is redefined, new instances are created using information that were stored in the special dictionaries, and finally the dictionaries are mutated back into the new instances.

4.6 Summary

Several key design issues were mentioned in this chapter. An icon contains information about the object represented and the state of the icon as it is currently being displayed. To support the facility whereby links can be hidden or visible, buttons must be part of the area relegated to icons. Different types of icons are needed, particularly for different kinds of collections, because they use different protocol for adding and removing elements.

Drag and drop, a relatively recent though by now well-known and intuitive facility for copying objects between windows is easily implemented with a few special messages. The

double-dispatching technique can be used to manage the potentially complex interactions between an editor, its icons, and an expanded icon's buttons. The cloning and mutating facilities provide a straightforward way of copying and altering objects.

Chapter 5: Conclusions

5.1 Assessment of Work Done

The Visual Object Editing Kit provides graphical facilities for inspecting, creating, and editing objects. Pictures and text are used simultaneously in icons to represent arbitrary objects thus taking advantage of both modalities. The objects are displayed and modified on the screen in a manner that is similar to the way they are drawn on paper.

Facilities for browsing and storing generic icons (the icon library) and arbitrary objects (the object library) are provided. Pictures for icons are created using the PictureEditor. Corresponding icon editors and object editors are provided to support both icon and object inspection, modification, and creation. Additionally, a structure editor is provided that permits new kinds of instances to be created; i.e., it supports the creation of prototypes, sometimes known as exemplars.

All editors are visual editors supporting direct manipulation facilities for connecting object parts (or components) with their values. Modifications can be made to objects without having to know the details of Smalltalk syntax (although some knowledge is needed if objects must be created from expressions). After an object is modified, it can optionally be saved in the object library where it can be reused. Icons can similarly be created for new classes of objects and reused whenever instances of the original class or subclasses are created.

Novices can browse and uncover the structure of an object by using the visual editors. Such editors can also be used to advantage by advanced users for debugging complex circular structures. These structures are encountered frequently. For example, banks can

keep track of bank accounts while bank accounts keep track of their banks. Elevator controllers keep track of the elevators they control while elevators keep track of their controllers to inform them of changes.

5.2 Limitation of the Editors and Icons

There are limitations to the implementation of the Visual Object Editing Kit. The pictures that are used to display objects are currently of a fixed size. As a result, a particular picture might be too small on some machines but too large on others. The system must provide a mechanism to permit icons of any size to be created.

The system provides a mechanism for associating an icon with a class but most classes do not have any icons defined for them. The default icon that is used to display an object without an associated icon does not convey much information about the object. This implies that the user must create an icon in the IconLibrary for every class in Smalltalk so that the icons can reveal more information about the objects they represent.

An expanded icon is currently rectangular in the Visual Object Editing Kit. New facilities are required to define the shapes of icons. The locations of buttons for instance variables have to be made more adjustable and flexible if more interesting shapes are to be handled. Additionally, there should be a mechanism for hiding selected instance variables separate from hiding the associated value.

For objects with many components, the display of an expanded icon can sometimes overrun the size of the screen. Currently, the display is clipped by the screen and the screen must be scrolled in order to see the remaining parts of the icon. Icons should have a scroll button attached to them. The icon should only display a partial structure of an object

(as much as will fit on the screen). To see the remaining parts of the icon, the scroll button could be pressed

There is a need to incorporate more direct manipulation into the icons. For example, to change the name of an icon, the system currently asks the user for a string using a prompter. The name of the icon should be changed directly on the screen at the display point. Also, an icon could have explicit buttons within as a substitute for often used menu options; e.g., a button in an icon could open the picture editor when it is pressed.

A complex object with all of its components fully expanded can clutter up the screen. Since each icon displays both textual and graphical information, screen resources become scarce. There is a need for a facility to better manage the display of icons. In particular, heuristics are needed for layout. Additionally, lines between icons in an editor need not be straight. A user should be able to draw arbitrary splines or curves.

5.3 Future work

The icons provided by the Visual Object Editing Kit are sufficient for displaying relatively complex structures but there is a need for a wide variety of icons. Extensions providing the features mentioned in the previous section could be implemented. One facility that we experimented with but did not completely implement is the notion of a dynamic icon. A **dynamic** icon is an icon whose picture changes as parts are provided. We used it to implement a bicycle object built from parts like handle bars, a frame, and wheels. As the parts are added, the icon for the bicycle automatically incorporates the new parts. We can actually see a bicycle being constructed. The only difficulty in adding this feature is to display the pictures of the children at the correct location with respect to each other so that the resulting picture resembles the parent object.

There is also a need for more specialized editing facilities. The generic visual editor does a reasonable job whenever a user wants to examine the details of the object. But, flexible editing demands facilities that depend on the kind of object being edited. Editing a string is different from editing a bank account, which is different again from editing a fraction. It is impossible to provide all possible types of editors that will suit everyone's needs. The development of specialized editors evolves as developers find the needs for them.

Another beneficial capability for a novice Smalltalk user would be one that permits them to simultaneously view an object in a visual editor while sending Smalltalk messages to the inspected object. When the object changes structure (for example, it is a collection object with elements being added or removed), the editor reacts to show the new structure. This would be particularly interesting if circular structures and destructive operations were being experimented with. This feature is quite easy to add to the Visual Object Editing Kit by prompting a user for a message to send to the selected object and telling the selected object to evaluate the expression. The visual editor can be used to inspect the result of the message send.

References

- [Blake 87 et al] Blake, Edwin, et al , "On Including Part Hierarchies in the Object-Oriented Languages, with an Implementation in Smalltalk " *EC'OOP '87*, June 1987, pp 41-50
- [Borges 90 et al] Borges, Jose A , et al , "Multiparadigm Visual Programming Languages " *IEEE Workshop on Visual Languages*, Oct 1990, pp 109-120
- [Borning 81] Borning, A H , "The Programming Language Aspects of ThingLab, a Constraint Oriented Simulation Laboratory " *ACM Trans. on Programming Languages and Systems*, Oct 1981, pp 353-387
- [Chakravarty 90 et al] Chakravarty, Indranil, et al , "Visualisms for Describing Interactive Systems." In *Engineering for Human-Computer Interaction* (G Cockton editor) Elsevier Science Publishers B V , North-Holland, 1990, pp 333-357
- [Chang 90] Chang, Shi-Kuo , *Principles of Visual Programming Systems*. Prentice Hall, Englewood Cliffs, New Jersey (1990).
- [Cox 88 et al] Cox, P T , et al , "Using a Pictorial Representation to Combine Dataflow and Object-Orientation in a Language Independent Programming Mechanism " In *Visual Programming Environments: Paradigm and Systems* (E P. Glinert editor). IEEE Computer Society Press, Los Alamitos (1990), pp. 313-322
- [Glinert 84 et al] Glinert, P. et al , "PICT: An Interactive Graphical Programming Environment " In *Visual Programming Environments: Paradigm and Systems* (E P Glinert editor) IEEE Computer Society Press, Los Alamitos (1990). pp 265-283

-
- [Finzer 84 et al] Finzer, William, et al . "Programming By Rehearsal " In *Visual Programming Environments: Paradigm and Systems* (E P Glinert editor) IEEE Computer Society Press, Los Alamitos (1990), pp 356-366
- [Gold 91 et al] Gold, Eric, et al . "Portia: An Instance-Centered Environment for Smalltalk " *OOPSLA '91*, Nov 1991, pp 62-74
- [Ingalls 88 et al] Ingalls, Dan, et al . "Fabrik: A Visual Programming Environment " *OOPSLA '88*, Sept 1988, pp 176-190
- [Jindrich 90] Jindrich, William A . *FOIBLE: A Framework for Visual Programming Languages* M.S. Thesis, University of Illinois at urbana-Champaign, Department of Computer Science (1990)
- [Myers 83] Myers, Brad A . "INCENSE: A System for Displaying Data Structures " *ACM Computer Graphics*, Vol 17, No 3, July 1983, pp 115-125
- [Myers 89] Myers, Brad A . "The State of the Art in Visual Programming and Program Visualization " In *Graphics Tools for Software Engineers* (A Kilgour editor) Cambridge University Press, New York, 1989, pp 3-25
- [Pong 84 et al.] Pong, M C , et al . "PIGS: A System for Programming with Interactive Graphical Support " In *Visual Programming Environments: Paradigm and Systems* (E P Glinert editor) IEEE Computer Society Press, Los Alamitos (1990), pp 259-264
- [Raeder 85] Raeder, Georg , "A Survey of Current Graphical Programming Techniques " *IEEE Computer*, Aug 1985, pp 11-25
- [Shu 85] Shu, Nan C . "FORMAL: A Forms Oriented Visual Directed Application Development System " *IEEE Computer*, Vol 18 No 8, Aug 1985, pp 38-49

-
- [Shu 88] Shu, Nan C , *Visual Programming*. Van Nostrand Reinhold Company, New York (1988).
- [Skublies 91] Skublies, S. *Sherpa: Guiding Students up the Smalltalk Mountain*, MCS Masters Thesis, School of Computer Science, Carleton University, 1991.
- [Tanimoto 86 et al] Tanimoto, Steven L , et al , "PLAY An Iconic Programming System for Children " In *Visual Programming Environments: Paradigm and Systems* (E.P. Glinert editor). IEEE Computer Society Press, Los Alamitos (1990), pp 367-377.

END

14-12-93

FIN