

Transforming Visual Programs into Java and Vice Versa

by

Lei Dong

**Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science**

at

**Dalhousie University
Halifax, Nova Scotia
May 2002**

**© Copyright by Lei Dong, 2002
Dalhousie University**

DALHOUSIE UNIVERSITY
FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled
_____ Transforming Visual Programs into Java and Vice Versa _____ by
_____ Lei Dong _____ in
partial fulfillment of the requirements for the degree of Master of
_____ Computer Science _____.

Dated: _____

Supervisor: _____

Readers: _____

DALHOUSIE UNIVERSITY

DATE: _____

AUTHOR: _____

TITLE: _____

DEPARTMENT OR SCHOOL: _____

DEGREE: _____ CONVOCATION: _____ YEAR: _____

Permission is herewith granted to Dalhousie University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

Signature of Author

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

The author attests that permission has been obtained for the use of any copyrighted material appearing in the thesis (other than the brief excerpts requiring only proper acknowledgement in scholarly writing), and that all such use is clearly acknowledged.

Table of Contents

| | |
|---|-----|
| Table of Contents | iv |
| Table of Figures | ix |
| List of Abbreviations | xi |
| Acknowledgments | xii |
| 1. Introduction | 1 |
| 2. Visual Languages and Software Engineering | 4 |
| 2.1. The evolution of programming languages | 4 |
| 2.2. Introduction to Visual Programming Languages(VPLs) | 5 |
| 2.3. Some visual programming languages | 7 |
| 2.4. Advantages of visual languages | 11 |
| 2.5. General tools for software engineers | 13 |
| 2.6. Visual software development tools for Java | 15 |
| 3. A formalisation of JGraph | 19 |
| 3.1. Introduction | 19 |
| 3.2. Formal definition of JGraph | 19 |
| 3.2.1. Package | 20 |
| 3.2.2. Class and Interface | 20 |
| 3.2.3. Method and constructor. | 23 |

| | |
|---|----|
| 3.2.4. Operations | 27 |
| 3.2.5. Cases | 32 |
| 3.3. Conclusion | 37 |
| 4. Translating JGraph to Java | 38 |
| 4.1. Introduction | 38 |
| 4.2. Translation to Java | 39 |
| 4.2.1. Package | 40 |
| 4.2.2. Class and Interface | 41 |
| 4.2.2.1. Examples | 42 |
| 4.2.3. Method. | 44 |
| 4.2.3.1. Example | 46 |
| 4.2.4. Cases. | 51 |
| 4.2.4.1. Example. | 54 |
| 4.2.4.2. Example | 56 |
| 4.2.5. Operations | 58 |
| 4.2.5.1. Examples of translating operations | 60 |
| 4.2.6. Controls | 65 |
| 4.2.6.1. Examples of translating controls | 65 |
| 4.2.7. A general example | 66 |

| | |
|--|----|
| 5. Importing Java into JGraph | 70 |
| 5.1. Introduction | 70 |
| 5.2. Class files | 70 |
| 5.3. Class | 71 |
| 5.4. Inheritance | 73 |
| 5.5. Attributes | 74 |
| 5.6. Preprocessing | 74 |
| 5.6.1. Simplifying control structures | 75 |
| 5.6.2. Multiple declarations | 75 |
| 5.6.3. Embedded sequences of statements. | 76 |
| 5.6.4. Method returns | 77 |
| 5.6.5. Unary increment and decrement operators | 79 |
| 5.6.6. The operators +=, -=, /=, *= | 79 |
| 5.6.7. Condition of while loops and conditional statements | 79 |
| 5.6.8. Embedded assignments | 80 |
| 5.6.9. Missing else | 83 |
| 5.6.10. Exceptions | 84 |
| 5.6.11. Isolating variables. | 87 |
| 5.6.12. Removing multiple assignments | 92 |

| | |
|---|-----|
| 5.6.13. Variable to variable assignments - - - - - | 94 |
| 5.7. Methods and Constructors- - - - - | 95 |
| 5.8. Expressions - - - - - | 96 |
| 5.8.1. Expressions which are not assignments - - - - - | 97 |
| 5.8.2. Assignment expressions - - - - - | 101 |
| 5.8.3. Bodies of methods and control structures - - - - - | 103 |
| 5.8.4. Method- - - - - | 104 |
| 5.9. Control structures- - - - - | 108 |
| 5.9.1. If statements - - - - - | 108 |
| 5.9.2. While statements - - - - - | 110 |
| 5.9.3. Try-catch structure - - - - - | 111 |
| 5.10. A comprehensive example- - - - - | 112 |
| 5.11. Conclusion - - - - - | 115 |
| 6. Conclusions and Future Work - - - - - | 116 |
| 6.1. Introduction - - - - - | 116 |
| 6.2. Comparison of Java and JGraph - - - - - | 117 |
| 6.3. Characteristics of Java generated from JGraph and vice versa.- - - - - | 119 |
| 6.4. Assessment of the translations- - - - - | 120 |
| 6.5. Future work- - - - - | 123 |

| | |
|------------|-----|
| References | 124 |
| APPENDIX A | 127 |

Table of Figures

| | |
|---|----|
| Figure 2-1: Von Neumann computer- - - - - | 4 |
| Figure 2-2: A Prograph quicksort method. - - - - - | 8 |
| Figure 2-3: A sample program of LabVIEW - - - - - | 9 |
| Figure 2-4: A sample Forms/3 program.- - - - - | 10 |
| Figure 2-5: A program of KidSim - - - - - | 11 |
| Figure 2-6: A sample Visual Age program- - - - - | 16 |
| Figure 2-7: Java studio main window - - - - - | 17 |
| Figure 3-1: A JGraph package containing two classes and four interfaces - - - - - | 22 |
| Figure 3-2: A JGraph method - - - - - | 25 |
| Figure 3-3: Cases of a local operation - - - - - | 35 |
| Figure 3-3: Cases of a local operation - - - - - | 35 |
| Figure 3-4: A constructor case - - - - - | 36 |

| | |
|--|----|
| Figure 4-1: MyClass and associated items | 43 |
| Figure 4-2: A JGraph method | 47 |
| Figure 4-3: The three cases of Local | 55 |
| Figure 4-4: Constructor case | 57 |

List of Abbreviations

| | |
|-----|------------------------------------|
| GUI | Graphical User Interface |
| IDE | Integrated Development Environment |
| JVM | Java Virtual Machines |
| VPE | Visual Programming Language |
| VPL | Visual Programming Language |
| TPL | Textual Programming Language |

Acknowledgments

I can hardly find proper words to express appreciation to my supervisor Dr. Phil. Cox for his guidance and all his time spent on reviewing and commenting my thesis. From figuring out problems to advising even very slight style issue in the document, he has shown to much patience to me. I have learned a lot of things from him not only for my thesis research, but for my personal development in the future also.

I would like to thank Dr. T. Smedley and Dr. R. Giles for their willingness to be members in my examining committee and their time on reviewing my thesis. I sincerely appreciate their time, interest, and suggestions.

The financial support from the Faculty of Computer Science, Dalhousie University is gratefully acknowledged.

Also, I'd like say thanks to my wife Rui Zhang. In the whole period of my study and from the very beginning, she is always there to support me. Love is a very important factor that my thesis could be finished successfully.



Introduction

In the early history of computers, the emphasis was on computation; that is, on the ability of these new machines to quickly and with minimal human intervention, solve problems of magnitudes greater than those that could be solved by people. Communication was simple and character-based. Even though the technology for communicating with computers has advanced beyond paper tape and punch cards, this character-based heritage is still with us today. An example is provided by the Unix operating system. Although UNIX is strong and powerful, it is hard for a user to communicate with because commands must be input via typed text. A major milestone in the history of computers was the introduction of personal computers (PCs). The reason the introduction of PCs was significant was that for the first time computers were cheap, did not need a special environment, and required much less specialized expertise to operate. This meant that they got into the hands of many more people. This is what drove the demand for improvements to make them more oriented to users than computer specialists.

It is now almost twenty years since low-cost, high-quality computer graphics became available, providing the means for users to interact with computers in ways other than by typing lines of text. This innovation has led to major advances in communication between computers and users by making it possible for users to accomplish tasks by directly manipulating items on the screen. For example, in a graphical operating system, the user can delete a file by dragging it to a trash can, or navigate the directory structure by opening windows on to lower level directories. Spreadsheets allow data to be laid out in a grid like a paper ledger, and formulae to be built by clicking on cells in the grid.

Although visual representations now play a significant role in software applications for users, industrial software development still relies to a very great extent on textual programming. Software developers received no benefits from graphics initially, and even now, twenty years after this phase of computing began, the use of visualization in software design and development still lags far behind its use in end-user applications. As we will discuss in more detail in Chapter 2, the design of computer hardware is inherently sequential, meaning that programs are linear sequences of instructions. As a result, programming languages are based on linear communication, that is, text. Algorithms, however, have structures which are not necessarily one-dimensional, and might therefore be more understandable if they were expressed pictorially.

When high quality graphics hardware and operating systems became available, applications began to appear with graphic user interfaces, making them much easier to use. Researchers, who realized the potential of graphics for software development, began to investigate the idea of visual programming languages (VPL), that is, languages that express algorithms in pictures rather than text. Also, software tool makers saw the potential of graphics for other aspects of software development, and began to produce various tools taking advantage of visual representations. As a result, there are visual formalisms for modelling software structures such as the Unified Modelling Language (UML) and Computer-Aided Software Engineering (CASE) tools employing visual representation such as UML to facilitate specification. In the marketplace, there are various GUI builders for directly constructing interfaces. Integrated Development Environments (IDE) usually consist of an application framework built on a textual programming language, together with a visual GUI builder, and windows and panels containing scrolling lists and other kinds of controls, providing views of various aspects of a project, such as class hierarchies. Examples include Visual C++ and Codewarrior.

Academic research has tended to concentrate on visual languages for general-purpose programming, producing some commercial products, Prograph for example. Commercial work on visual programming has produced VPLs aimed at particular application domains and users, for example LabVIEW and PhonePro. The domain-specific VPLs have generally been more successful as products, probably

because, like spreadsheets and other end-user applications, they make certain kinds of tasks much easier for certain kinds of users.

The dominance of textual programming has a lot to do with the history and evolution of computers, and the rich theoretical foundations and standards that have been built around textual languages. So, although visual programming has received a lot of study, and has been shown to provide some important benefits, it faces significant challenges in making inroads into industrial development.

It is only recently that researchers have begun to look closely at how well or how badly visual programming languages perform, and why. Studies and practical experience have shown that they have a lot to offer the professional software developer. By providing a direct visual representation of the structure of algorithms and allowing the developer to program by directly manipulating this representation, a visual language relieves the developer of the burden of managing minor syntactic details, and of parsing textual encodings to extract the algorithm structures. The programmer is therefore free to concentrate more on the important aspects of the programming problem.

As we have observed above, domain-specific VPLs such as LabVIEW have generally been more successful commercially than general-purpose VPLs like Prograph, regardless of the potential benefits of VPLs for professional developers. Unlike users of domain-specific VPLs, professional developers need tools that conform to industry standards. Hence, a VPL that conforms to a textual language standard has a much greater likelihood of acceptance.

2 Visual Languages and Software Engineering

2.1 The evolution of programming languages

In 1946, von Neumann introduced an architecture for computing machines which still provides the basis for modern computers. As Figure 2-1 shows below, a von Neumann machine consists of three parts, the Central Processing Unit (CPU), Memory and Input and Output Devices (I/O devices), which communicate with each other via communication busses. Memory contains data and programs consist of sequences of instructions. I/O devices transmit input data and results between machine and user. The CPU controls all operations, and performs computations by repeatedly executing the following steps: fetch next instruction from memory, decode it, fetch any required data, execute instruction.

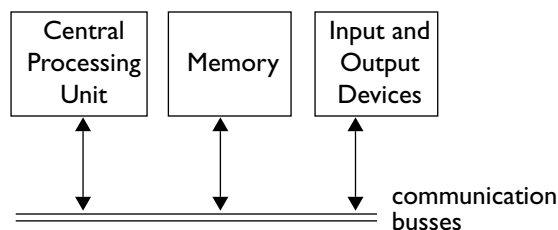


Figure 2-1: Von Neumann computer

Consequences of this architecture and operation is that execution is sequential, and programs are linear sequences of instructions. Therefore communication between the user and computer need only be sequential. Hence, input and output (I/O) devices, such as paper tape, card punches, line printers and line-oriented terminals, have mostly been designed for sequential communication dealing with strings

of characters. Programming languages have evolved from machine language, through assembly language, towards modern programming languages which provide more sophisticated structures for expressing algorithms, rather than directly reflecting the simple capabilities of the machine. Although a common thread throughout this language evolution is that programs are linear sequences of characters, the trend has been to make languages more oriented to humans than computers. Since most modern high-level programming languages such as C, C++ and Java are text-based, we will use the term “textual programming language” (TPL) to refer to them.

Early attempts to make programming languages more human-oriented were rather *ad hoc*, introducing some high-level control structures, but not based on any sound theory. The first was FORTRAN in the middle of 50s. However, the move towards higher-level languages drove research into the foundations of formal languages, grammars, automata, parsing, compiling and so forth. Consequently, TPLs have strong theoretical foundations. So there is a rich body of knowledge surrounding textual programming languages. As computing moved from a research and scientific activity to a mainstream commercial activity, programming evolved into an engineering process, requiring standardized processes and tools. As a result certain languages have become standards (e.g. C, C++ and Java) and there is a great investment in tools and methodologies based on them. So, TPLs are an indispensable part of the software engineering world.

2.2 Introduction to Visual Programming Languages(VPLs)

As the computer has evolved so has the technology of human-computer communication. A particularly important step was the development of graphic display technology and graphic user-input devices such as the mouse and light pen, and the possibilities these opened up, especially when they became widely available in the early 80s. Also, improvements in processor power made it possible for computers to support complex graphical interfaces.

As mentioned in Chapter 1, this evolution of human-computer interface technology brought many benefits to users, thereby increasing the demand for applications. Software developers, however, were

still stuck with their traditional tools. Although visual tools have been introduced into some aspects of the development process they are usually aimed at tasks other than programming, such as GUI design, modelling or specification. Visual specification of algorithms has not been generally accepted.

The term “visual programming” is used by different authors in different ways. Some definitions are as follows:

Visual programming is programming in which more than one dimension is used to convey semantics. Each potentially significant multi-dimensional object or relationship is a token (just as in traditional textual programming languages each word is a token) and the collection of one or more such tokens is a visual expression such as diagrams, free-hand sketches, icons and so on. When a programming language's (semantically-significant) syntax includes visual expressions, the programming language is a visual programming language (VPL). M. M. Burnett [5]

Visual Programming refers to any system that allows the user to specify a program in two-(or more)-dimensional fashion. Conventional textual languages are not considered two dimensional since the compilers or interpreters process them as long, one-dimensional streams. B. A. Myers [15]

A visual language manipulates visual information or supports visual interaction, or allows programming with visual expressions. The latter is taken to be the definition of a visual programming language. Visual programming languages may be further classified according to the type and extent of visual expression used, into icon-based languages, form-based languages and diagram languages. Visual programming environments provide graphical or iconic elements which can be manipulated by the user in an interactive way according to some specific spatial grammar for program construction. E. J. Golin and S. P. Reiss [7]

Visually transformed languages are inherently non-visual languages but have superimposed visual representations. Naturally visual languages have an inherent visual expression for which there is no obvious textual equivalent. M. M. Burnett [5]

Visual programming is commonly defined as the use of visual expressions (such as graphics, drawings, animation or icons) in the process of programming. These visual expressions may be used in programming environments as graphical interfaces for textual programming languages; they may be used to form the syntax of new visual programming languages leading to new paradigms such as programming by demonstration; or they may be used in graphical presentations of the behavior or structure of a program. D. W. McIntyre and E. P. Glinert [13]

A visual language is one in which pictorial, iconic, graphical syntax (as opposed to a textual syntax) is used as the primary (not just a graphical skeleton with textual flesh) means to express the logic (not just window layout) of the program being written. K.J. Schmucker [23]

From the definitions above we conclude that VPLs are characterised as follows:

- There must be inherently visual expressions associated with (semantically-significant) syntax.
- The visual expressions can be manipulated by the user interactively for the process of programming.

Unlike visual programming environments in which a program is still specified in a textual language, VPLs are used to create programs via visual expressions. VPLs are not necessarily devoid of text, however, but may use it as comments, for labels of graphical objects and so forth.

2.3 Some visual programming languages

In this section, in order to give the reader a general picture of what VPLs are and how they do their job, we give a brief description of some in this section, both general-purpose and domain-specific.

Prograph is an object-oriented visual programming language aimed for professional programmers, which adopts data flow structure [9]. Data flow is one of the more popular computing models for visual languages [12]. A method in Prograph consists of a sequence of cases, each of which is a data flow diagram. Cases are executed in order until one executes to completion. For example, Figure 2-2 shows the two cases of a method quicksort, that implements the quicksort algorithm. The first case tests to see if the incoming list is empty, and if so, outputs it as the sorted list. Otherwise control

transfers to the second case, which implements the recursive case of the algorithm. In Prograph, list is a built-in primitive type. The operation named > in the example has a specially annotated input indicating that a list is expected, and that the operation will be applied to each element of it.

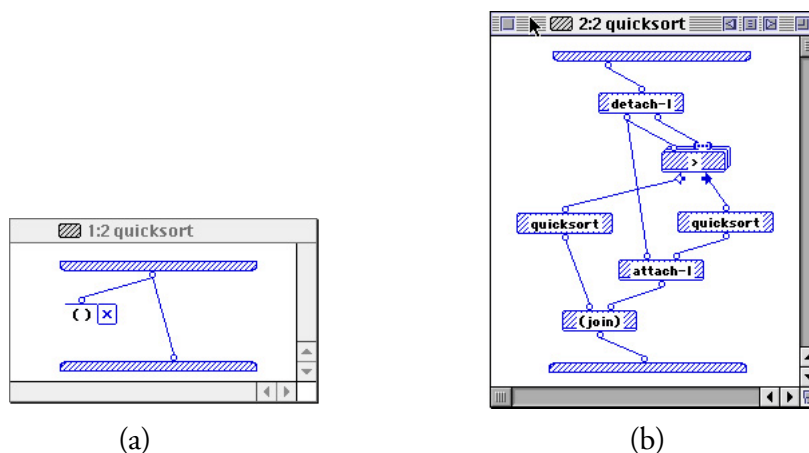


Figure 2-2: A Prograph quicksort method.

LabVIEW, another visual data flow language, is distributed by National Instruments primarily to provide a programming interface to measurement and control devices [29]. It is intended for users in engineering and science.

LabVIEW, like Prograph, uses spatial containment to indicate control structures such as loops and conditionals. Data flows from left to right in LabVIEW diagrams. Figure 2-3 below is a LabVIEW program that computes the factorial of an integer. The icons at the upper left provides the input integer, while constant 1 below it is the initial value of the factorial. The rectangle in the centre is an iteration. The icon labelled i in it denotes the iteration count, the icon labelled N indicates the number of iterations to be performed, and the matching icons with the down and up arrows on the left and right borders of the iteration denote a looped variable.

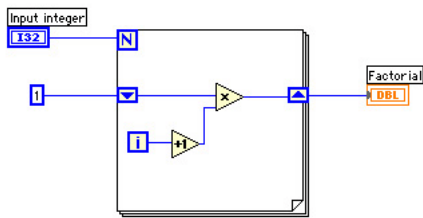


Figure 2-3: A sample program of LabVIEW

Forms/3 is a VPL based on a generalisation of the spreadsheet paradigm. A Forms/3 program consists of forms which contain cells, the contents of which are specified by a formulae. A formula is defined for each cell by a flexible combination of pointing, typing and gesturing. Figure 2-4, taken from [26], shows a sample Forms/3 program program that calculates the n^{th} element of Fibonacci sequence, which is the sum of the $(n-1)^{\text{st}}$ and $(n-2)^{\text{nd}}$ Fibonacci numbers. The program consists of three windows. Window FIB is the model for the other two windows FIB01 and FIB02, which are called instances of FIB, which inherit their model's cell and formulae unless the user explicitly provides different input. Any change to the model is propagated to its instances. When, as in this example, a cell formula in the model form references a cell in an instance of the model, the pattern of references is recognised by the system and generalised to create recursion. Forms/3 is a general purpose declarative language. The only implementation is a research prototype.

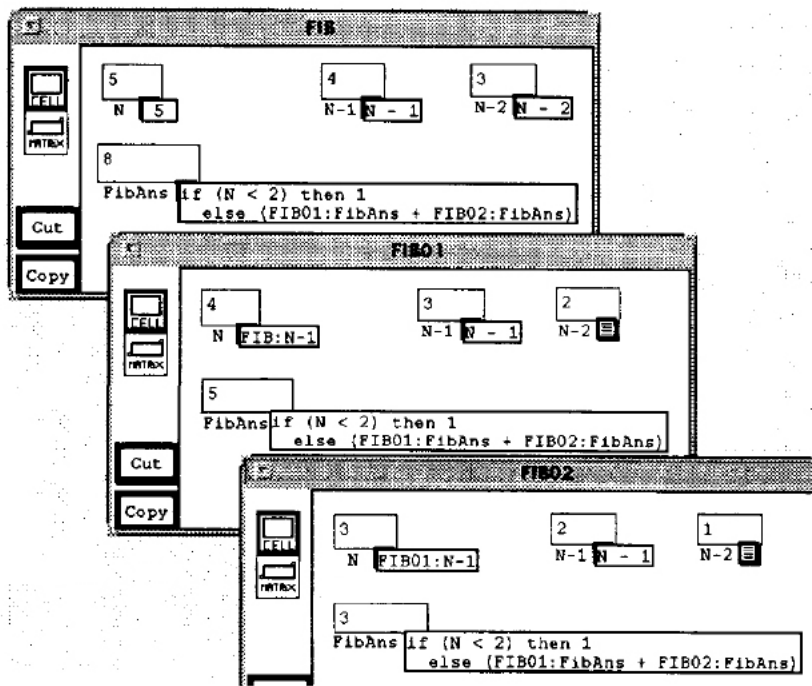


Figure 2-4: A sample Forms/3 program.

KidSim is a rule-based VPL for children in which the programmer creates graphical simulations and games by building picture transformation rules. Figure 2-5 depicts a "wall climber" program. The main window on the left is where the simulation occurs. The rules are listed in Mascot 1 window. Each rule consists of a graphical precondition on the left of the arrow and a graphical postcondition on the right. In the figure, the wall climber has just applied rule 2. The next applicable rule will be 1. KidSim is not a tool for general-purpose programming, but is aimed at making the programming of animated graphical simulations accessible to children.



Figure 2-5: A program of KidSim

From the above samples we can see that, like textual languages, VPLs are based on a variety of different programming models, have different target users, and may be general-purpose or aimed at solving programming problems in specific application domains.

2.4 Advantages of visual languages

There is evidence that visual representations can improve the human-machine interface because they enhance human cognitive abilities. As a principle, if information is made explicit and presented in a consistent and organized way, people can perform better at many tasks, including programming. In [28] Whitley concludes that “compared to textual notations, visual notations can provide better organization and can make information explicit. Moreover, properly-used visuals result in quantifiable performance benefits. Several studies show visuals outperforming text in either time or correctness, sometimes both”.

Whitley notes that visual representations are beneficial not only for objects that have concrete counter-parts in the real world, but also for “nonspatial” concepts, for which visual representations provide organization and make information explicit. Whitley discusses studies which show that the benefit of visual representations grows as the size and the complexity of a program grows, and concludes that VPLs could be useful in traditional programming where the problems are usually larger

than the problems used in controlled experiments. Some studies show that graphics can sometimes outperform text even for smaller problems. Hence, VPLs may play an important role in end-user programming where problems may be smaller than those encountered by professional programmers.

In a more recent study, Whitley and Blackwell conducted a survey of LabVIEW programmers with the aim of determining the effectiveness of the visual aspects of LabVIEW [29]. They discovered that “respondents rated the value of LabVIEW’s visual language significantly higher than the value of all other LabVIEW features rated in this survey.”

Such studies provide us with valuable data about the usefulness and usability of existing visual programming languages and environments: however, it is also important to have some means to assess a VPL in order to predict its effectiveness, or to guide VPL designers. In [11] Green and Petre introduce the cognitive dimensions framework as a “broad brush evaluation technique for interactive devices and for non-interactive notations. It sets out a small vocabulary of terms designed to capture the cognitive-relevant aspects of structure, and shows how they can be traded off against each other.”

Although the cognitive dimensions apply as much to textual programming languages and environments as to visual ones, it seems that visual languages have the potential to perform better than textual ones in some of these dimensions. In the following we will restrict our attention to those, and encourage the reader to consult [11] for the complete list.

Closeness of Mapping refers to the degree to which a language directly represents objects and actions in the problem domain. Visual languages have an inherent advantage over textual ones in this dimension since they can directly represent domain objects and relationships, while textual ones must use some special syntax to code them.

Visibility refers to whether the required material can readily be made visible, whether it can be accessed in order to be made visible, or whether it can readily be identified in order to be accessed. VPLs may provide greater opportunities for accessing and displaying information than TPLs since, like a city map, their diagrams give the observer a picture of where to look for needed information.

Secondary notation Many programming languages allow extra information to be added to programs unrelated to the formal syntax, such as indenting, commenting, choice of names and so forth. These notations make no contributions to the logic of the algorithm, but help readers of the program to understand it. Since VPLs are pictorial, they provide opportunities for useful annotations using pictorial devices which would be difficult to incorporate in textual programs.

Role-expressiveness refers to the extent to which the representation of program elements suggests their function. In this dimension, VPLs have an inherent advantage over TPLs. In TPLs, the common way to suggest functions of program elements is to use appropriate names, such as "while" to denote loop. However, in VPLs, different elements could be represented with visual representations in different shapes, different colours, or different dimensions so that their functions are obvious.

Hidden dependencies are logically significant relationships between program components which are not directly visible. VPLs have an advantage over TPLs in this dimension. In TPLs, transferring a value from its source to its destinations is accomplished by variables. Because these dependencies can be observed only by finding the various occurrences of a variable, they are not explicit. In VPLs, however, it is possible to make dependencies explicit by using by lines, directed graphs and so on. For example, in data flow a line is used to connect the source of a value and its destination.

From the above discussion, it is clear that, although textual programming languages provide the foundation for modern software development, they are not necessarily always the best choice. Visual programming languages can contribute much to the efficiency and effectiveness of programmers, so it is important to develop visual tools to enhance the software development process.

2.5 General tools for software engineers

In software engineering, the tools and methodologies used by software developers range from very high level ones for capturing the overall structure of a software system, to programming languages for coding algorithms. In this section, we will give a brief discussion with examples about those tools.

At the top level, there are various software design methodologies. Booch's object-oriented design methodology uses various kinds of diagrams to capture the structure of an object-oriented system [4]. The Unified Modelling Language (UML) [22] provides various kinds of diagrams which have only a partially defined semantics, and are used for expressing the components of a software system, the interactions between components, and the interactions between the system and its users. Entity-relationship diagrams deal with data structuring and data base specification. Tools which implement these methodologies are usually called Computer Aided Software Engineering (CASE) tools. CASE tools are mainly used as documentation tools, but some can also generate code. For example, Visual Case is a CASE tool that implements UML [3], and DeZign is a CASE tool for developing databases.

Component technology is sometimes used to provide an intermediate level of organisation between high-level specification and actual coding. Components are "black boxes" that encapsulate data and associated functionality, and communicate by sending messages to each other. A component can be shared and reused by different applications, different platforms or even different machines over a network. Some component-technology standards allow components implemented in different languages on different systems to communicate, for example the Common Object Request Broker Architecture (CORBA) [24]. Examples of component technologies are Microsoft's Component Object Model (COM)[21], JavaBeans[27] and IBM's System Object Model (SOM)[8].

At the implementation level, the software developer uses a programming language, usually implemented in an Integrated Development Environment (IDE). An IDE usually provides an application framework, a class hierarchy that supplies much of the standard functionality of a modern GUI-driven application. An IDE usually also includes a GUI builder for constructing interfaces by direct manipulation, a debugger, and tools that provide various visualisations of a software project. There are many IDEs available in the marketplace such as Microsoft Visual Studio [16], Borland Delphi [18] and C++ builder [17], and Metrowerks CodeWarrior [19].

The development of industrial software is a labour-intensive activity, so it is important that the tools used are stable and well supported, and that similar tools are available from several vendors. This

implies that development tools, which may be developed by different companies, conform to some standards. In particular, industrial software developers tend to use standard programming languages, supported by many different tools vendors, and widely used by many developers who form an informal support network.

Java is the latest evolution of the C language, which has long been a standard for industrial software development. This parentage gives Java an automatic advantage in terms of being accepted as a software development standard. In addition, unlike any other language in the past, Java has been adopted by many influential companies.

Aside from these strategic advantages, Java also has various technical characteristics that contribute to its popularity. For example, Java is object-oriented, supports multi-threading, has the capability to handle exceptions, dynamic memory management, a dynamic type casting system, extensive static type checking and simpler syntax than other object-oriented languages such as C++.

2.6 Visual software development tools for Java

There are quite a few Java-based tools that use visualizations of some aspects of the software development process. In this section, we will present some examples.

Visual Age [6] for Java is an IDE produced by IBM. The core programming language is Java, however, various kinds of visualisation are provided. First, windows and panels with scrolling lists and other controls are used to display packages, class hierarchies, code versions and so forth. Second, the JavaBeans component model is used as a basis for a restricted form of visual programming in which graphical representations of components are connected by different kinds of lines, indicating the passing of messages between components. This message-passing model in Visual Age is most suited to GUI programming, as illustrated in Figure 2-6. This diagram shows how to build a GUI and set up connections between different GUI components, directed lines indicating the flow of messages between components. By making these connections, the programmer codes behaviour into the GUI. For examples, the line from the Add button to ToDoList together with the side connection from

ToDoItem indicate that when the Add button is pressed, the value in ToDoItem will be added to the list of items in ToDoList.

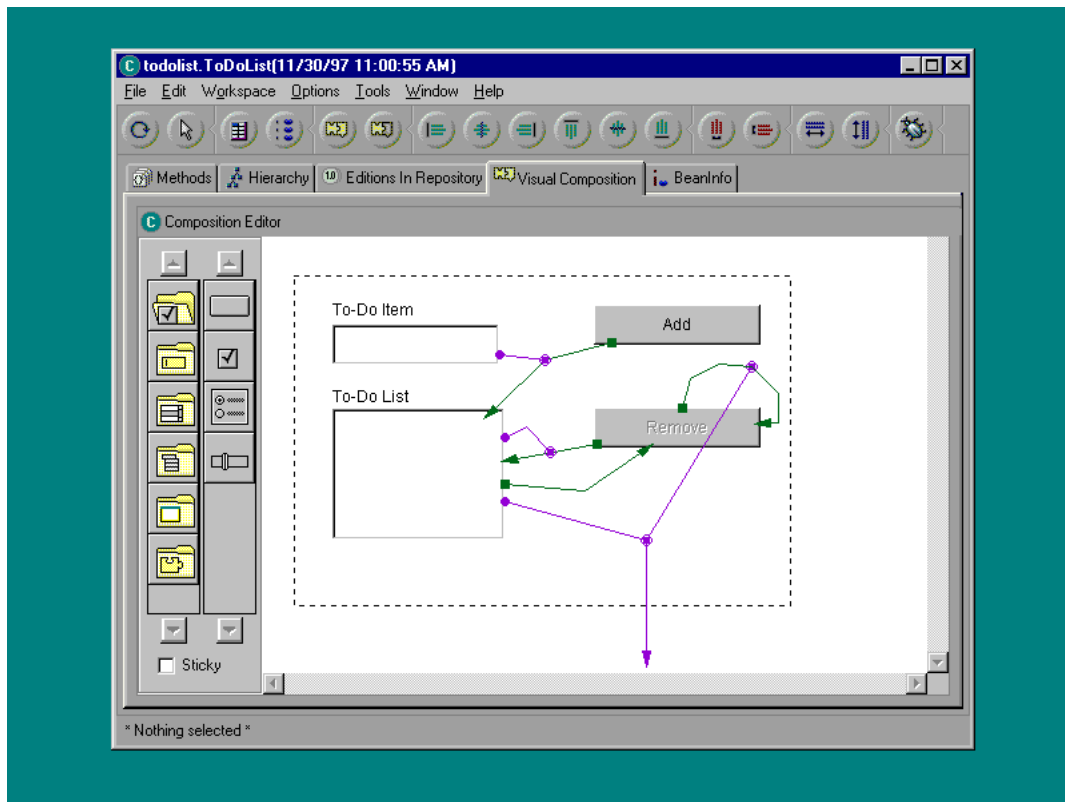


Figure 2-6: A sample Visual Age program

Java Studio is a visual programming environment developed by Sun but no longer marketed. In the design window of Java Studio, an example of which is shown in Figure 2-7 below, the programmer constructs diagrams consisting of nodes representing Java Beans, connected by wires indicating the passage of messages between nodes. A user interface corresponding to a diagram is built in an associated GUI window (not shown), which displays the graphical representations of those beans in the design window that implement GUI items. Like Visual Age, Java Studio is a component-based visual programming tool that employs message flow.

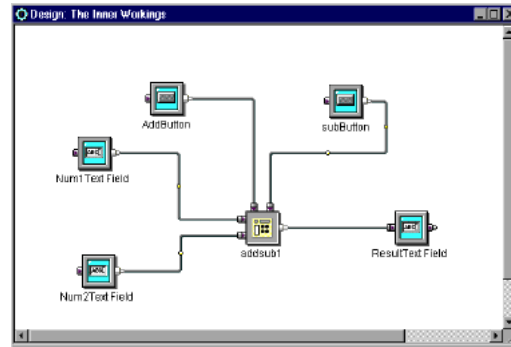


Figure 2-7: Java studio main window

The three systems described above provide limited visual programming capabilities based on components. There are other tools for Java that provide some visualisation, GUI design for example. A typical example is Visual J++ [16], a component of Microsoft Visual Studio, which in addition to visual GUI-building, provides tools for organising and managing a development project similar to those in Visual Age [6], described above. CodeWarrior is another example of an IDE for Java with similar capabilities [19].

2.7 JGraph

As discussed above, there are many Java-based software development tools that use visualisations to some extent. Some use visual representations for high-level system modelling. Others provide visual GUI editing facilities. None, however, provides complete visualisations of algorithms, and allows the developer to program algorithms by building such visualisations.

JGraph is a general-purpose visual programming language that addresses this shortcoming. A complete description can be found in [20]. As a data flow visual programming language, it borrows heavily from Prograph, using the same case structure for obtaining conditional execution, multiplexes for iteration, and controls on operations for controlling execution. From Java, JGraph inherits many features such as strong typing, identical data types and data structures, similar exception handling,

classes, interfaces and constructors, and identical levels of access (public, private etc) to classes attributes and methods.

Since JGraph is compatible with Java in the ways discussed in the above paragraph, it has the potential to bring the advantages of visual programming at the algorithm level to the world of industrial software development. However, since JGraph will certainly not replace Java, in order for it to be acceptable, tools to integrate it with Java development will be required. In particular, it will be important to allow the programmer to move freely between textual and visual representations of code.

2.8 Content of the following chapters and appendix

As background to the research reported here, we built a prototype JGraph editor which generates Java from JGraph, and imports Java code, translating it to JGraph. A user's manual is attached in Appendix A. Based on these experiments, we have defined two complete translations, and provided a critical analysis of them with respect to the goal of enabling the software developer to move freely between textual and visual representations. In Chapter 3, we give the formal definition of JGraph to provide a basis for defining a translation from JGraph to Java in Chapter 4. In Chapter 5, we define a translation in the opposite direction. Finally, in Chapter 6, we conclude our work with comparisons between Java and JGraph, an evaluation of our results, and suggestions for future work that we believe should be undertaken.

3

A formalisation of JGraph

3.1 Introduction

From this chapter on, we will address the problem of translating JGraph into Java by providing a formal, abstract definition of the JGraph language, relating this definition to the pictorial representation presented in [20], and showing how each of the abstractly defined JGraph elements corresponds to Java source. In this chapter, we will give a precise, formal specification of JGraph semantics, then in the next chapter, we will address the translation process.

In this chapter, We assume the reader is familiar with the JGraph language, the details of which can be found in [20].

The chapter is organised as follows. First, in section 3.2, we define some useful notation. Then, we will give the formal definition of JGraph. The chapter concludes with discussion and comments. The definitions in section 3.2 are fairly terse, so they are illustrated by numerous examples.

3.2 Formal definition of JGraph

Throughout this chapter we will use various notations and conventions as follows. We will use **bold** style to represent components of entities defined as tuples. For example, **Name** is a component of a project. We will frequently use the names of components of tuples as functions. For example, a normal case is defined as a 4-tuple (**Opers**, **Synchros**, **Exception**, **Outputs**), so if X is a normal case, then **Synchros**(X) denotes the second element of the tuple X.

We will denote the empty list by (). If X is a list or sequence, we will denote by $X[i]$ the i^{th} element of X .

In the following, \mathcal{N} \mathcal{T} are disjoint sets of strings called *names* and *simple types* respectively, which conform to the naming conventions of Java.

A *type* is either a simple type or an array type. An *array type* is a 1-tuple (**Elemtype**), where **Elemtype** is a type.

In the following, we assume the existence of a set \mathfrak{N} , the elements of which are called *nodes*, a 1-1 function **name** from \mathfrak{N} to \mathcal{N} and a function **type** from \mathfrak{N} to \mathcal{T} . Two lists of nodes N_1 , N_2 are said to *match* iff $|N_1| = |N_2|$ and **type**($N_1[i]$) = **type**($N_2[i]$) for each i ($1 \leq i \leq |N_1|$).

If X is a set of modifiers and Y is a Java construct, we will say that X is *legal for a Java Y* if Java permits all the modifiers in X to be simultaneously applied to the Java construct Y . For example, {"abstract", "final"} is not legal for a Java method but {"abstract", "protected"} is legal for a Java method.

3.2.1 Package

In JGraph, a package is a pair (**Name**, **Classes**) where **Name** is a name indicating the package name, **Classes** is a set, each element of which is a class or an interface.

3.2.2 Class and Interface

A JGraph *class* is an 8-tuple (**Imports**, **Modifiers**, **Name**, **Superclass**, **Interfaces**, **Attributes**, **Methods**, **Constructors**) where:

- **Imports** is a set of strings, each indicating a package imported by the class. The set is empty if no packages are imported.
- **Modifiers** is a subset of {"public", "abstract", "final"} legal for a Java class.
- **Name** is a type that is the name of the class.
- **Superclass** is a type which is the name of a class.
- **Interfaces** is a set of types each of which is the name of an interface.

- **Attributes** is a set of attributes.
- **Methods** is a set of methods.
- **Constructors** is a set of constructors.

An *attribute* is a triple (**Modifiers**, **Node**, **Value**). where:

- **Modifiers** is a subset of {"public", "protected", "private", "final", "static", "transient", "volatile"} legal for a Java class variable.
- **Node** is a node.
- **Value** is a string or *null*.

A JGraph *interface* is a 6-tuple (**Imports**, **Modifiers**, **Name**, **Superclass**, **Attributes**, **Methods**) where:

- **Imports** is a set of strings, each indicating a package imported by the class. The set is empty if no packages are imported.
- **Modifiers** is a subset of {"public", "abstract"} legal for a Java interface.
- **Name** is a type that is the name of the interface.
- **Superclass** is a type which is the name of an interface.
- **Attributes** is a set of attributes.
- **Methods** is a set of methods such that $\forall M \in \text{Methods}, \text{"abstract"} \in \text{Modifier}(M)$

3.2.2.1 Example

This example illustrates the above definitions by considering a package called MyPackage and some classes and interfaces in it, showing how the formal definition corresponds with the visual representation. Formally, this package is the pair

$$(\text{"MyPackage"}, \{C, I_1, I_2, I\})$$

where C is a class and I_1 , I_2 and I are interfaces. The structures for C and I are as follows

$$C = (\{\text{"java.*"}, \text{"javax.*"}\}, \{\text{"public"}\}, \text{"MyClass"}, \text{"SuperClass"}, \{\text{"Interface1"}, \text{"Interface2"}\}, \\ \{\{\text{"public"}\}, \text{"int"}, \text{"i"}, \text{"0"}\}, \{\{\text{"public"}\}, \text{"boolean"}, \text{"boo"}, \text{"true"}\}, \{M_1, M_2\}, \{\})$$

$I = (\{\text{"java.*"}, \text{"javax.*"}\}, \{\text{"public"}\}, \text{"MyInterface"}, \text{"SuperInterface"}, (\{\text{"public"}\}, \text{"int"}, \text{"i"}, \text{"10"}),$
 $(\{\text{"public"}\}, \text{"String"}, \text{"str"}, \text{"Hello"}), \{M_3, M_4\}, \{\})$

where M_1, M_2, M_3 and M_4 are methods, the representation for which is defined in the next section.

Figure 3-1 shows the JGraph windows that graphically represent some parts of these structures.

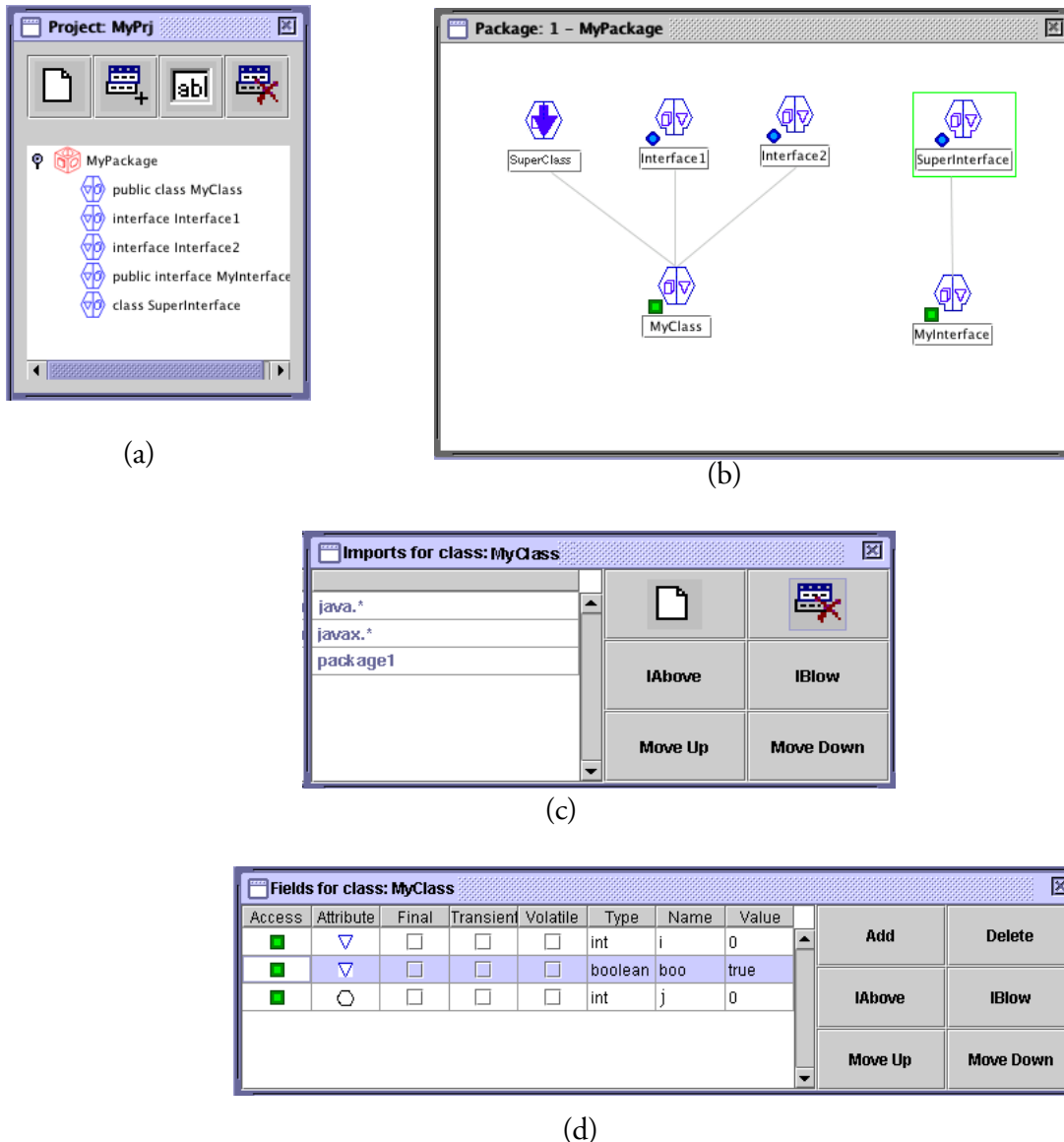


Figure 3-1: A JGraph package containing two classes and four interfaces

Figure 3-1 (a) shows the project window depicting the list of packages in the project. In particular, it includes an icon for MyPackage, and icons of the classes and interfaces in it. Figure 3-1 (b) is a class window presenting all the classes and interfaces of the particular package MyPackage and the inheritance relationships between them. Note that since an alias is not a class or interface, SuperClass does not occur in MyPackage in Figure 3-1 (a). However, the class that SuperClass refers to must be in one of the packages that MyClass imports, as shown in Figure 3-1(c). Figure 3-1(c) shows the list of imported packages of the class MyClass. Figure 3-1(d) illustrates the attributes of MyClass.

The figures in this example and those that follow were generated using our JGraph prototype. The reader should refer to the User Manual in Appendix A for the meaning of the various controls that are attached to the windows shown in these pictures.

3.2.3 Method and constructor.

If K is a class, a *method* of K is an 12-tuple (**Modifiers**, **Exception**, **Name**, **Inputs**, **Roots**, **NormalCases**, **CatchCases**, **FinallyCase**, **Flag**, **Index**, **Uplimit**, **Terminate**) where:

- **Modifiers** is a subset of {"public", "protected", "private", "abstract", "final", "native", "static", "synchronized"} legal for a Java method.
- **Exception** is a set of nodes.
- **Name** is a name.
- **Inputs** is a list of nodes distinct from each other, and from any node external to the method. See definition of external below.
- **Roots** is a list of nodes distinct from **Inputs**, and from any node external to the method, such that $|\mathbf{Roots}| \leq 1$. See definition of external below.
- **NormalCases** is a list of normal cases.
- **CatchCases** is a list of catch cases.
- **FinallyCase** is a finally case or *null*.
- If "abstract" \in **Modifiers**, then **NormalCases** = **CatchCases** = {} and **FinallyCase** = *null*.
- If C is in **NormalCases**, **CatchCases** or **FinallyCase** then C is called a *case* of the method.

- **Flag, Index, Uplimit** and **Terminate** are nodes distinct from each other, from any nodes in **Inputs** or **Roots**, and from any node external to the method.

If K is a class, a *constructor* of K is a 8-tuple (**Modifiers**, **Exception**, **Inputs**, **ConstCase**, **Flag**, **Index**, **Uplimit**, **Terminate**) where:

- **Modifiers** is a subset of {"public", "protected", "private"} legal for a Java constructor.
- **Exception** is a set of nodes.
- **Name** is a name.
- **Inputs** is a list of nodes distinct from each other, and from any node external to the constructor. See definition of external below.
- **ConstCase** is a constructor case, which may be referred to as *a case of* the constructor.
- **Flag, Index, Uplimit** and **Terminate** are nodes distinct from each other, from any nodes in **Inputs**, and from any node external to the constructor.

A node n is said to be *external to* a method or constructor of class K iff $n \in \{\mathbf{Node}(A) \mid A \in \mathbf{Attributes}(K)\}$.

A method or constructor M *contains a counted loop* iff for some case C of M , there is an operation $O \in \mathbf{Opers}(C)$ such that either O is a counted loop or is a local operation that contains a counted loop.

Refer to section 3.2.4 for the definition of counted loop.

A method or constructor M *contains a controlled loop* iff for some case C of M , there is an operation $O \in \mathbf{Opers}(C)$ such that either O is a controlled loop or is a local operation that contains a controlled loop. Refer to section 3.2.4 for the definition of controlled loop.

3.2.3.1 Example

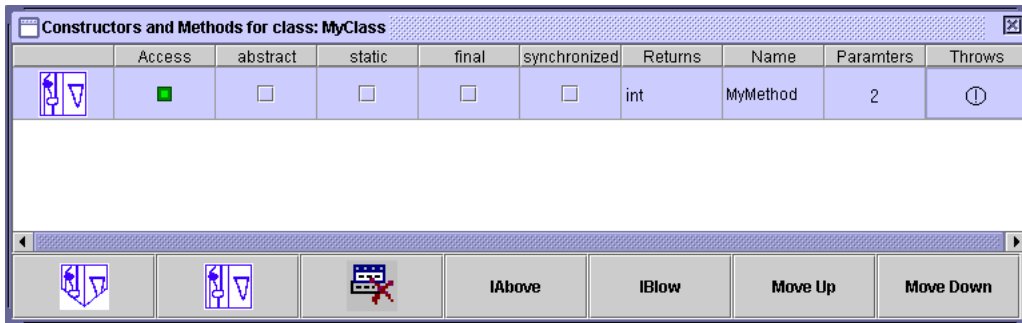
The following example illustrates a method

$$M = (\{\{\mathbf{public}\}\}, \{E\}, \text{"MyMethod"}, (I_1, I_2), (R), (N_1, N_2) \{C\}, E, G, I, U, T)$$

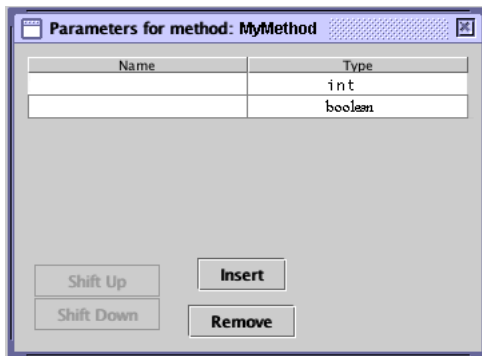
where:

- E, R, I_1, I_2, G, I, U and T are nodes.

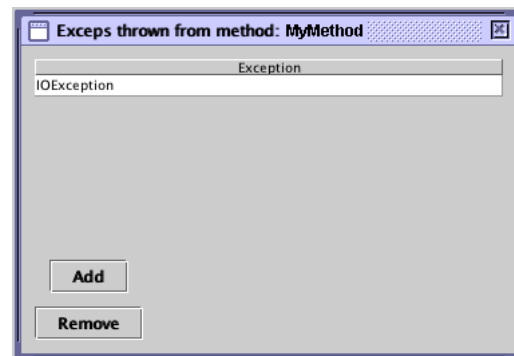
- $\text{type}(I_1) = \text{"int"}, \text{type}(I_2) = \text{"String"}, \text{type}(R) = \text{"int"}$.
- N_1, N_2 are normal cases.
- C is a catch case
- F is a finally case



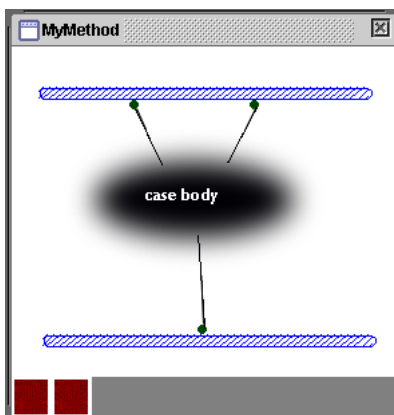
(a)



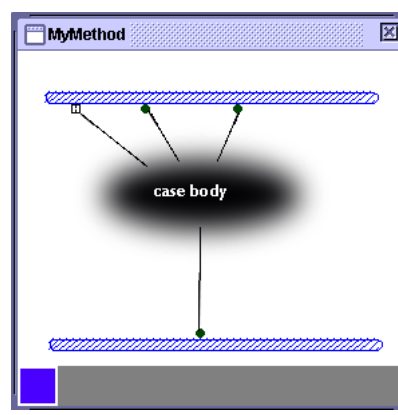
(b)



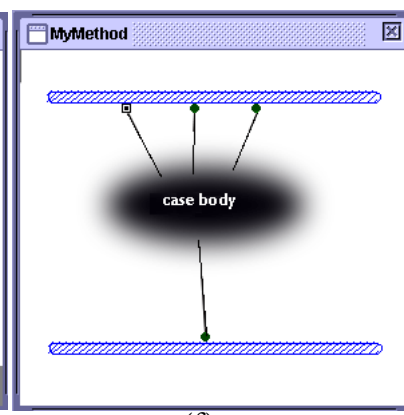
(c)



(d)



(e)



(f)

Figure 3-2: A JGraph method

Figure 3-2 (b) and (c) show, respectively, the parameter list of `MyMethod`, and the list of exceptions that `MyMethod` throws. Note that the names of parameters are empty strings, the default provided by the JGraph prototype. Since names of parameters are important only when JGraph is translated to Java, they need not be specified until required for translation, at which time they could be supplied by the programmer or generated by the translator.

Figure 3-2 (d) is the window for the normal case N_1 of `MyMethod`. The two rectangles at the bottom of this window correspond to the two normal cases N_1 and N_2 (not shown) of `MyMethod`. In the implementation, these rectangles are coloured brown to indicate that they represent normal cases. In windows displaying catch cases, as in Figure 3-2(e), or a finally case as in Figure 3-2(f), the rectangles are blue and green respectively.

Figure 3-2 (e) is the window for the catch case C of `MyMethod`. The rectangle at the bottom of this window shows that the method has only one catch case. The left root of the input bar indicated by the square icon, is the catch root of C , which will be explained more fully later.

Figure 3-2 (f) is the window for the finally case F of `MyMethod`. Note that the input bar has one more root than the input bar of the first normal case of `MyMethod`. That root, indicated by the square icon, is called a *finally input root*, and corresponds to the terminal of the output bar of the normal cases of the method, as explained later.

In Figure 3-2 (d) to (f) we represent the contents of cases by fuzzy blobs since we are not concerned with the representation of cases in this example. Cases will be discussed in section 3.2.5 below. Note that the last four components of a method, **Flag**, **Index**, **Uplimit** and **Terminate** exist only to facilitate translation to Java, so they have no representation in JGraph. We will discuss them in detail when we discuss the translation of a method to Java in section 4.2.3. We also note that not all the components of a method need to be specified. If a method does not throw exceptions, its **Exception** component will be just `{}`. Similarly, a method does not have to have catch cases if no exceptions are to be caught inside the method.

3.2.4 Operations

In order to continue defining JGraph in a strict, top-down fashion we should define cases next. However, the definitions of “case” and “operation” are interdependent, so we have chosen to define operations first.

Since each operation occurs in a case, in the following we assume the operation being defined occurs in some case C .

There are nine categories of *operation*, each of which is a tuple consisting of a selection of components as defined in the following table, where:

- Each row defines a category of operation.
- Each column corresponds to a component of a tuple.
- A grey cell indicates that the corresponding category of operation does not have the corresponding component.
- **Target** is either a node or *null* or \uparrow .
- **Name** is a name.
- If $|\mathbf{Roots}| = 1$ and $\mathbf{type}(\mathbf{head}(\mathbf{Roots})) = \text{“boolean”}$, then $\mathbf{Control} \in \{\text{☒}, \text{☑}, \text{☒}, \text{☑}, \text{☒}, \text{☑}, \text{☒}, \text{☑}, \text{☒}, \text{☑}, \text{☒}, \text{☑}\}$, otherwise $\mathbf{Control} = \text{null}$.
- **Terminals** is a finite sequence of nodes.
- **Roots** is a finite sequence of distinct nodes.
- An entry in a cell of the table indicates restrictions on the corresponding component of an operation of the corresponding category.
- **Value** is a string that can be typed into a Java program as a constant.
- **Cases** is a sequence of normal cases.
- **CatchCases** is a sequence of catch cases.
- **FinallyCase** is a finally case.
- If C is in **NormalCases**, **CatchCases** or **FinallyCase** then C is called a case of M .
- **Flag**, **Index**, **Uplimit** and **Terminate** are nodes distinct from each other, and from every node external to the operation (see definition below).

- **Inputs** is a finite sequence of nodes distinct from each other, and from **Flag**, **Index**, **Uplimit** and **Terminate** and from every node external to the operation, such that $|\mathbf{Inputs}| = |\mathbf{Terminals}|$.
- For a local operation, **Inputs** and **Terminals** match.
- **Ttypes** is a function from **Inputs** to $\{\text{Simple, Array}\} \cup \{i \mid 1 \leq i \leq |\mathbf{Roots}|\}$ such that:
 - If $\mathbf{Ttypes}(\mathbf{Inputs}[k]) = k_1$, $\mathbf{Ttypes}(\mathbf{Inputs}[j]) = j_1$ are both integers, where $k < j$, then $k_1 < j_1$.
 - If $\mathbf{Ttypes}(\mathbf{Inputs}[k]) = \text{Array}$ for some k , then $\mathbf{type}(\mathbf{Terminals}[k])$ is an array type and $\mathbf{type}(\mathbf{Inputs}[k]) = \mathbf{Elemtype}(\mathbf{type}(\mathbf{Terminals}[k]))$.
 - If $\mathbf{Ttypes}(\mathbf{Inputs}[k])$ is an integer for some k , then $\mathbf{type}(\mathbf{Inputs}[k]) = \mathbf{type}(\mathbf{Roots}(\mathbf{Ttypes}(\mathbf{Inputs}[k])))$.
- For a repeat operation,

$$\mathbf{type}(\mathbf{Inputs}[i]) = \begin{cases} \mathbf{Elemtype}(\mathbf{type}(\mathbf{Terminals}[i])) & \text{if } \mathbf{Ttypes}(\mathbf{Inputs}[i]) = \text{Array} \\ \mathbf{type}(\mathbf{Terminals}[i]) & \text{otherwise} \end{cases}$$

for each i ($1 \leq i \leq |\mathbf{Inputs}|$)

 - **Dimension** is a finite sequence, each element of which is a node or integer.
 - **Data** is a node.

If O is a local or repeat operation such that $O \in \mathbf{Opers}(C)$ for some case C , then a node n is said to be *external to O* iff n is a root of an operation in $\mathbf{Opers}(C)$, or is external to C .

A repeat operation O is called a *counted loop* iff $\mathbf{Ttypes}(R) = \text{Array}$ for some $R \in \mathbf{Inputs}(O)$.

A repeat operation O is called a *controlled loop* iff $\mathbf{Control}(O_1) \in \{\text{☒}, \text{☑}, \text{☒}, \text{☑}\}$ for some $O_1 \in \mathbf{Opers}(C_1)$ where C_1 is a case of O .

A local or repeat operation O *contains a counted loop* iff for some case C_1 of O , there is an operation $O_1 \in \mathbf{Opers}(C_1)$ such that either O_1 is a counted loop or is a local operation that contains a counted loop.

Table 3-1: JGraph operations

| | | tuple components | | | | | | | | | | | | | | | | |
|--------------------|-------------|------------------|------|---------|-----------|-------|-------|-------------|------------|-------------|-------|--------|------|-------|---------|------------|-----------|------|
| | | Target | Name | Control | Terminals | Roots | Value | NormalCases | CatchCases | FinallyCase | Types | Inputs | Flag | Index | Uplimit | Dimensions | Terminate | Data |
| operation category | constructor | not node | | | | | | | | | | | | | | | | |
| | simple | | | | ≤ 1 | | | | | | | | | | | | | |
| | get | | | | 1 | | | | | | | | | | | | | |
| | set | | | | | | | | | | | | | | | | | |
| | alloc | | | | 1 | | | | | | | | | | | | | |
| | literal | | | | 1 | | | | | | | | | | | | | |
| | local | | | | | | | | ≤ 1 | | | | | | | | | |
| | repeat | | | | | | | | ≤ 1 | | | | | | | | | |
| | array | | | | 1 | | | | | | | | | | | | | |
| | array set | node | | | | | | | | | | | | | | | | |
| | array get | node | | | | 1 | | | | | | | | | | | | |
| | match | | | | 1 | 1 | | | | | | | | | | | | |

A local or repeat operation O contains a controlled loop iff for some case C_1 of O , there is an operation $O_1 \in \mathbf{Opers}(C_1)$ such that either O_1 is a controlled loop or is a local operation that contains a controlled loop.

If O is a local operation, $O \in \mathbf{Opers}(C)$ and C is a case of some method, local or repeat M then we define:

$$\mathbf{Flag}(O) = \mathbf{Flag}(M)$$

$$\mathbf{Uplimit}(O) = \mathbf{Uplimit}(M)$$

$$\mathbf{Index}(O) = \mathbf{Index}(M)$$

Although in JGraph as defined in [20] repeat operations can have enumeration terminals, such terminals do not translate into very efficient Java, so they are omitted here.

3.2.4.1 Example

The following examples illustrate the above definition.

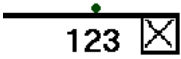
Table 3-2: Various operations and their visual representations

| JGraph operation | | Visual representation |
|------------------|---|--|
| constructor | $(null, (T_1, T_2))$ where T_1, T_2 are nodes | |
| simple | $(null, \text{"foo"}, null, (), ())$ | |
| simple | $(\uparrow, \text{"foo"}, \square, (T_1, T_2), (R))$ where T_1, T_2, R are nodes | |
| get | $(null, \text{"foo"}, null, (R))$ where R is a node | |
| get | $(\uparrow, \text{"foo"}, \square, (R))$ where R is a node | |
| set | $(null, \text{"foo"}, D)$ where D is a node | the terminal-like icon (called a terminal by Risley) represents D , the Data of the operation. |
| set | $(\uparrow, \text{"foo"}, D)$ where D is a node | |
| alloc | $(\text{"foo"}, (T_1, T_2), (R))$ where T_1, T_2 and R are nodes | |
| literal | $(null, (R), \text{"123"})$ where R is node | |
| literal | $(\square, (R), \text{"123"})$ where R is node | |

Table 3-2: Various operations and their visual representations

| | | |
|-----------|--|---|
| local | $(null, (T_1, T_2), (R_1, R_2, R_3), (C), (), (), (I_1, I_2), M)$ where $T_1, T_2, R_1, R_2, R_3, I_1, I_2$ and M are nodes, and C is a normal case Note that I_1, I_2 appear as the roots of the input bar in each case of the local operation. | |
| repeat | $(null, (T_1, T_2), (R_1, R_2), (C), (), (), P, (I_1, I_2), F, N, U, M)$ where $T_1, T_2, R, I_1, I_2, F, N, U$ and M are nodes, C is a normal case, and P is the function $P(I_1) = \text{Simple}, P(I_2) = \text{Simple}$. Note that I_1, I_2 appear as the roots of the input bar in each case of the repeat operation. | |
| repeat | $(null, (T_1, T_2, T_3), (R_1, R_2), (C), (), (), P, (I_1, I_2, I_3), F, N, U, M)$ where $T_1, T_2, T_3, R, I_1, I_2, I_3, F, N, U$ and M are nodes, C is a normal case, and P is the function $P(I_1) = 1, P(I_2) = \text{Array}, P(I_3) = \text{Simple}$. Note that this is a counted loop because of the presence of the Array input. | |
| array | $(("foo"), (R), (5,7))$ where R is a node | |
| array | $(("foo"), (R), (N))$ where R and N are nodes | |
| array get | $(G, (R), (5,7))$ where G, R are nodes | |
| array get | $(G, (R), (N))$ where G, R and N are nodes | The icon consisting of connected terminal and root represents G |

Table 3-2: Various operations and their visual representations

| | | |
|-----------|---|---|
| array get | $(\uparrow, (R), (5, N, 7))$ where R, N are nodes | |
| array set | $(G, (5, 7), D)$ where G, D are nodes | |
| array set | $(G, (M), D)$ where G, N and D are nodes | |
| array set | $(\uparrow, (5, N, 7), D)$ where N and D are nodes | |
| match | $(\boxtimes, T, (R), "123")$ where T, R are nodes |  <p>Note that R does not have visual representation. It exists in the formal structure so that the translation to Java of match is consistent with that of other operations.</p> |

3.2.5 Cases

Each case is defined with respect to the method, constructor, local operation or repeat operation in which it occurs, so in the following, let M be some arbitrary but fixed method, constructor, local operation or repeat operation.

A *normal case* of M is a 4-tuple (**Oper**s, **Synchros**, **Exception**, **Outputs**), where

- **Oper**s is a set of operations not including any constructor operations.
- $\forall O \in \mathbf{Oper}, \mathbf{Control}(O) \in \{\boxtimes, \boxplus\}$ only if $\mathbf{Roots}(M) = ()$.
- **Synchros** is a set of pairs of the form (O_1, O_2) where $O_1, O_2 \in \mathbf{Oper}$. A synchro (O_1, O_2) is said to be *from* O_1 *to* O_2 .
- **Exception** is either a node or is *null*.

- **Outputs** is a finite sequence of nodes distinct from **Flag**, **Index**, **Uplimit** and **Terminate**.

A *catch case* of M is a 5-tuple (**Opers**, **Catchroot**, **Synchros**, **Exception**, **Outputs**) defined as for a normal case except that:

- **Catchroot** is a node distinct from all roots of all operations in the case or in other cases of M .
- $\forall o \in \text{Opers}, \text{Control}(o) \notin \{\boxtimes, \boxplus\}$.

A *finally case* of M is a 5-tuple (**Previous**, **Opers**, **Synchros**, **Exception**, **Outputs**) as defined as for a normal case except that:

- $\forall O \in \text{Opers}(C), \text{Control}(O) \notin \{\boxtimes, \boxplus\}$
- **Previous** is a list of nodes that matches **Outputs**.

A *constructor case* of M is defined as for a normal case except that:

- **Opers** contains exactly one constructor operation O and constructor operations can not be anywhere except in a constructor case.
- **Outputs** and **Roots**(M) must match for a normal, catch or finally case.
- The set **Opers** - $\{O\}$ can be partitioned into two subsets **Pre** and **Post** such that:
 - If $O_1 \in \text{Pre}$ then O_1 is either a simple, literal, get, alloc, array or array get operation, and
 - $|\text{Roots}(O_1)| = 1$,
 - $\text{Control}(O_1) = \text{null}$,
 - $\text{head}(\text{Roots}(O_1)) \in \text{Terminals}(O_2)$ where $O_2 = O$ or $O_2 \in \text{Pre}$,
 - if $(O_2, O_1) \in \text{Synchros}$, then $O_2 \in \text{Pre}$.

In addition, there are several further conditions that every case must satisfy as follows.

- Distinct operations in a case have no roots in common.
- **Outputs** and **Roots**(M) must match for a normal, catch or finally case.
- If N is a node and
 - either $N \in \text{Terminals}(O)$ for some $O \in \text{Opers}$
 - or $N \in \text{Dimensions}(O)$ for $O \in \text{Opers}$

or $N \in \mathbf{Target}(O)$ for some $O \in \mathbf{Opers}$

or $N \in \mathbf{Data}(O)$ for some $O \in \mathbf{Opers}$

or $N \in \mathbf{Outputs}$

or $N = \mathbf{Exception}$

then

either $N \in \mathbf{Roots}(O_2)$ for some $O_2 \in \mathbf{Opers}$

or $N \in \mathbf{Inputs}(M)$

or $N \in \mathbf{Previous}$

or $N = \mathbf{Catchroot}$

- If N is a node then N is a *local* root of a normal, catch or finally case iff $N \in \mathbf{Roots}(O)$ for some $O \in \mathbf{Opers}$, or $N \in \mathbf{Previous}$ or $N = \mathbf{Catchroot}$. N is a *local* root of a constructor case iff either $N \in \mathbf{Roots}(O)$ for some $O \in \mathbf{Post}$, or N occurs more than twice in the case. Note that “occurs in” here refers to all occurrences of N as roots, terminals, data, targets or dimensions of operations, or as the exception or an output of the case.

We will denote the set of local roots by **Local**.

- If R is a node then R is said to be *external to* the case iff
 - either $R = \mathbf{Flag}(M)$, $\mathbf{Index}(M)$, $\mathbf{Uplimit}(M)$ or $\mathbf{Terminate}(M)$
 - or $R \in \mathbf{Inputs}(M)$
 - or $R \in \mathbf{Roots}(M)$
 - or M is a local or repeat operation in $\mathbf{Opers}(C)$ for some case C , and R is a local root of C or is external to C .
- No node can be both a local root of the case and external to the case.
- Let G be the directed graph such that **Opers** is the set of vertices of G , and (O_1, O_2) is an edge of G iff $(O_1, O_2) \in \mathbf{Synchros}$, or some root of O_1 is also $\mathbf{Data}(O_2)$ or $\mathbf{Target}(O_2)$ or is an element of $\mathbf{Terminals}(O_2)$ or $\mathbf{Dimension}(O_2)$; then G is acyclic.

Note that if O is the unique constructor operation in a constructor case, $O_1 \in \mathbf{Pre}$ and $O_2 \in \mathbf{Post}$, then G can be linearly ordered in such a way that $O_1 < O < O_2$.

3.2.5.1 Example

We illustrate the definition of normal, catch and finally case by considering the following example, in which a local operation is concerned.

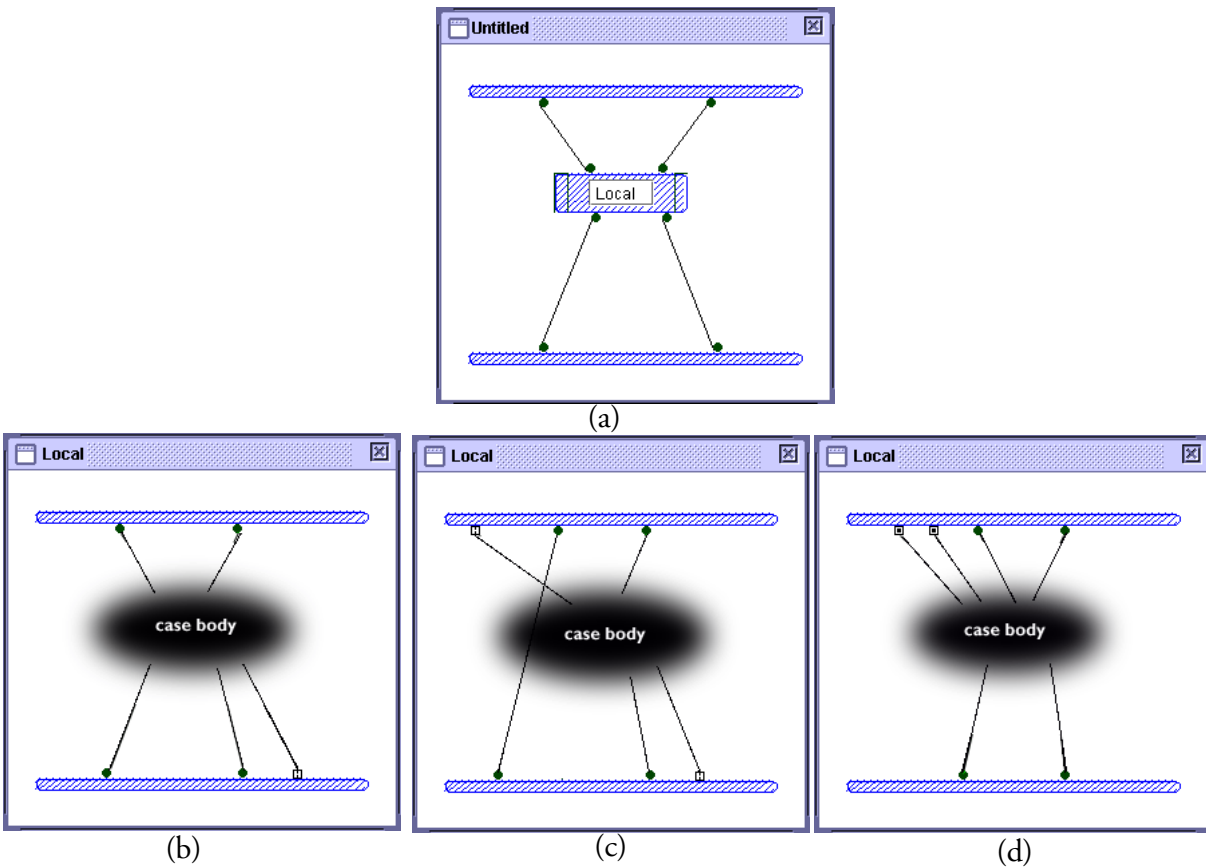


Figure 3-3: Cases of a local operation

Figure 3-3 (a) shows a case containing a local operation *Local*, which has several cases that we will consider in detail in this example. The local operation itself has the following structure:

$$L = (\text{null}, (T_1, T_2), (R_1, R_2), (C_1), (C_2), C_3, (I_1, I_2), M)$$

Figure 3-3 (b), (c) and (d) show, respectively, the normal case C_1 , catch case C_2 and finally case C_3 of the operation, which have the following structures:

$$C_1 = ((O_1, O_2), \text{null}, E, (U_1, U_2))$$

$$C_2 = ((O_3), T, \text{null}, A, (U_3, U_4))$$

$$C_3 = ((P_1, P_2), (O_4), \text{null}, \text{null}, (U_5, U_6))$$

where:

$$\text{type}(T_1) = \text{type}(I_1)$$

$$\text{type}(T_1) = \text{type}(I_1)$$

$$\text{type}(R_1) = \text{type}(P_1) = \text{type}(U_1) = \text{type}(U_1) = \text{type}(U_1)$$

$$\text{type}(R_2) = \text{type}(P_2) = \text{type}(U_2) = \text{type}(U_4) = \text{type}(U_6)$$

Note that the extra terminals on the output bars of the normal and catch cases in Figure 3-3 represent the nodes E and A that are the **Exception** components of these two cases. The extra root on the input bar of the catch case represents the node T that is the **Catchroot** component of this case. The extra two roots (finally input roots) on the input bar of the finally case represent the nodes in the **Previous** component of this case.

We illustrate the definition of a constructor case by considering the following example, which deals with the case of a constructor M of a class named MyClass.

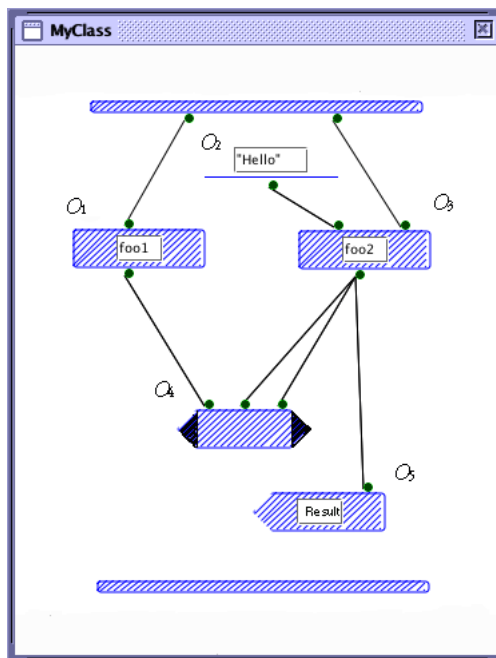


Figure 3-4: A constructor case

The constructor case in Figure 3-4 has the following structure:

$$C = ((O_1, O_2, O_3, O_4, O_5), \{\}, \text{null}, ())$$

where

$$O_1 = (\text{null}, \text{"foo1"}, \text{null}, (\mathbf{Input}(M)[0]), (N_1)),$$

$$O_2 = (\text{null}, (N_2), \text{"Hello"}),$$

$$O_3 = (\text{null}, \text{"foo2"}, \text{null}, (N_2, \mathbf{Input}(M)[1]), (N_3)),$$

$$O_4 = (\text{null}, (N_1, N_3, N_3)),$$

$$O_5 = (\text{null}, \text{"Result"}, N_3).$$

As Figure 3-4 shows, the three terminals of the constructor operation O_4 are connected to the roots of the operations before the constructor operation. The corresponding directed graph (see section 3.2.5)

G can be linearly ordered in the way described in section 3.2.5 so that $O_1, O_2, O_3 \in \mathbf{Pre}$, $O_5 \in \mathbf{Post}$.

That is $O_1, O_2, O_3 < O_4 < O_5$.

3.3 Conclusion

In this chapter we have extended the description of JGraph provided in [20] by providing a formal definition of the JGraph language, together with illustrative examples. The formalisation we have described was chosen to facilitate the process of translating JGraph programs into Java described in the next chapter. However, since the formal definitions were devised after our experiences with implementing the prototype, the data structures used in the JGraph prototype are not based on the formal definitions we have provided.

4 Translating JGraph to Java

4.1 Introduction

In this chapter, we will address the problem of translating JGraph into Java by providing a mapping from the formal definitions of JGraph elements provided in the last chapter to corresponding Java code. There are several quite obvious reasons for doing this as follows

- Illustrate the close relationship between JGraph and Java.
- Provide a “cheap” way to execute JGraph programs by generating Java code that can be compiled and executed.
- Possibly also provide a means for immediate execution in an editing/debugging environment like Prograph CPX. To provide for immediate execution, the Java code would have to be generated incrementally as the JGraph program is edited, perhaps just for JGraph elements that are determined to be syntactically correct, as described in [20], and the Java compiler called “on-the-fly”.

We assume the JGraph program we discuss is correct, and that the reader is familiar with the JGraph language, the details of which can be found in [20]. Risley, in his thesis, presented the semantics of JGraph and provided much of the information necessary to translate to Java. In this chapter, we will fill in all the details necessary for the complete process. The discussions cover the strategy for translating JGraph visual programs into Java, the style and characteristics of the Java programs generated, and how to work around the limits of Java such as the lack of a “goto” statement. Our discussion will follow the structure of JGraph in a top down fashion.

The chapter is organised as follows. First, in section 4.2, we define some useful notation. Next, we give the formal definition of the translation process. This is accomplished by defining a function τ that maps JGraph programs and program parts to Java programs and program parts. The chapter concludes with discussion and comments.

4.2 Translation to Java

In this section we show how a JGraph program can be translated into Java by defining a function τ that in general, maps a JGraph program part to a string. The one exception, however, is when τ is applied to a package, in which case it produces a set of strings. The remainder of this chapter is devoted to defining τ in a top-down fashion.

We will use underscore style to represent Java keywords, for example, public, class and static. We use the symbol ϵ to denote the empty string. Some symbols, such as semicolons and commas serve double duty as characters in strings generated by τ , and as punctuation in our notation. To avoid ambiguity, we will enclose strings in double quotes. In the following, concatenation of strings is denoted by juxtaposition. Also, every concatenation involves the insertion of a blank between the concatenated strings. For example, the concatenation operation “ab”“cd” produces the string “ab cd”.

If X is a set of strings, and α is a string, then $[X, \alpha]$ is the string defined as follows:

$$[X, \alpha] = \epsilon \text{ if } X = \emptyset$$

$$[X, \alpha] = x \text{ if } x \in X \text{ and } |X| = 1$$

$$[X, \alpha] = x \alpha [X1, \alpha] \text{ where } X = \{x\} \cup X1 \text{ and } |X| > 1$$

Also, if X and α are as above, and β and γ are strings, we define a string $[X, \alpha, \beta, \gamma]$ as follows:

$$[X, \alpha, \beta, \gamma] = \beta [X, \alpha] \gamma \text{ if } [X, \alpha] \neq \epsilon$$

$$[X, \alpha, \beta, \gamma] = \epsilon \text{ if } [X, \alpha] = \epsilon$$

Note that $[X, \alpha]$ above is not well defined. For example, suppose X is the set of names of interfaces $\{A, B, C\}$ that a class implements, then $[X, ", "]$ may be any of the strings "A, B, C", "B, C, A", "C, A, B" etc. This ambiguity is not important, however, since those elements of JGraph structures which are defined as sets correspond to elements of Java in which ordering is unimportant; for example, the order of interface names in the "implements" clause of a class definition.

If x is a list, we denote the first element of x by **head**(x), and the list obtained by removing the first element from x by **tail**(x). If y is a list, we denote by $z*y$ the list obtained by adding an item z to the beginning of y .

For a method, local or repeat operation M , let V be any set of strings such that $|V|=|\mathbf{NormalCases}(M)|$, and for each $v \in V, v \notin \mathbf{N}, v \neq \mathbf{name}(R)$ for any $R \in \mathfrak{R}$, and v conforms to Java naming conventions. Now let **Var** be an arbitrary but fixed 1-1 function from **NormalCases**(M) to V .

4.2.1 Package

At the package level, the translation process does nothing more than create a class file for each class in the package. The translation process then proceeds to the details of each of these classes. If P is a package, then

$\tau(P) = \{\text{"package" Name}(P) \text{";"} \tau(C) \mid C \in \mathbf{Classes}(P)\} \cup \{E\} \cup \{F\}$, where:

```

E = "package" Name(P) ";
    import java.awt.*;
    public class ICaseException extends Exception
    {
        public ICaseException(String Msg)
        {
            super(Msg);
        }
    }

```

and

```

F = "package" Name(P) ";
    import java.awt.*;
    public class ITermException extends Exception
    {
        public ITermException(String Msg)
        {
            super(Msg);
        }
    }"

```

The exception classes ICaseException and ITermException are added to the generated Java as part of the mechanism that deals with the case structure of JGraph methods and the terminate control.

Each of the strings in the set of strings generated by applying τ to a package corresponds to one code file containing one class definition, as required by Java.

4.2.2 Class and Interface

At this level, the correspondance with Java is exact, so the translation process is straightforward and obvious.

If C is a class then

$$\tau(C) = \tau(\text{Imports}(C)) \tau(\text{Modifiers}(C)) \text{"class"} \text{Name}(C) \tau(\text{Superclass}(C)) \tau(\text{Interfaces}(C)) \text{"\{"}$$

$$\tau(\text{Attributes}(C)) \tau(\text{Methods}(C))\text{"\}"}$$

where

$$\begin{aligned} \tau(\text{Imports}(C)) &= [\text{Imports}(C), \text{“; import”}, \text{“import”}, \text{“;”}] \\ \tau(\text{Modifiers}(C)) &= [\text{Modifiers}(C), \text{“ ”}] \\ \tau(\text{Superclass}(C)) &= \begin{cases} \text{“extends” Superclass}(C) & \text{if Superclass}(C) \neq \epsilon. \\ \epsilon & \text{otherwise.} \end{cases} \\ \tau(\text{Interfaces}(C)) &= [\text{Interfaces}(C), \text{“,”}, \text{“implements”}, \epsilon] \\ \tau(\text{Attributes}(C)) &= [\{\tau(A) \mid A \in \text{Attributes}(C)\}, \text{“;”}] \\ \tau(\text{Methods}(C)) &= [\{\tau(m) \mid m \in \text{Methods}(C)\}, \epsilon] \end{aligned}$$

Where if A is an attribute:

$$\tau(A) = \begin{cases} [\text{Modifiers}(A), \text{“ ”}] \text{type}(\text{Node}(A)) \text{ name}(\text{Node}(A)) \text{ “=” Value}(A), \\ \quad \text{if Value}(A) \neq \text{null}. \\ [\text{Modifiers}(A), \text{“ ”}] \text{type}(\text{Node}(A)) \text{ name}(\text{Node}(A)), \\ \quad \text{otherwise.} \end{cases}$$

If I is an interface then

$$\begin{aligned} \tau(I) &= \tau(\text{Imports}(C)) \tau(\text{Modifiers}(C)) \text{“interface” Name}(C) \tau(\text{Superclass}(C)) \text{“{”} \\ &\quad \tau(\text{Attributes}(C)) \tau(\text{Methods}(C)) \text{“}” \end{aligned}$$

4.2.2.1 Examples

The following example illustrates the translation of JGraph classes to Java.

Figure 4-1 (a) shows the project window containing the package list. For this example, there is only one package, named “MyPackage”, containing the class under consideration. Figure 4-1 (b) shows the class window of MyPackage depicting the inheritance relations between MyClass and other classes and interfaces. Figure 4-1 (c) and (d) show, respectively, the list of imported packages and attributes of MyClass. The Java produced by the translation defined above is as follows.

```
package MyPackage;
import java.*;
import javax.*;
```

```

import package1;

public MyClass extends SuperClass implements Interface1, Interface2
{
    private int i=0;
    protected boolean boo=true;
    <methods; section 4.2.3>
}

```

We use the notation <> to indicate code generated from a JGraph element and the section where the translation is defined.

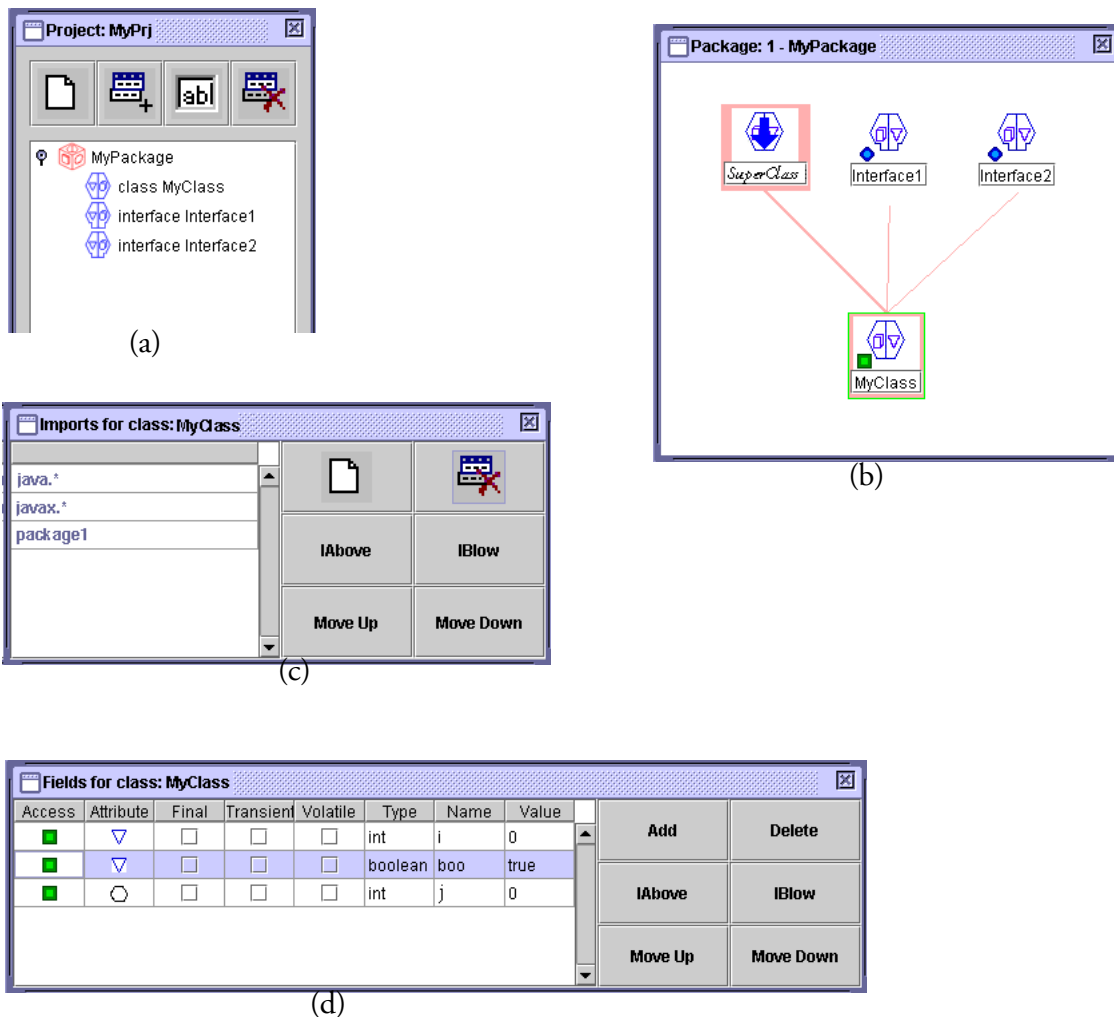


Figure 4-1: MyClass and associated items

4.2.3 Method.

Since the translation of a method is the most complex part of the process, we divide our explanation into several levels. At the top level we address the problems of translation not related to the method body. Then we will discuss the patterns of translation of a method body in terms of cases. The translation of a case is described in detail in a later section.

Let M be a method then we define $\tau(M)$ as below. Note that some of the “helper” functions we require are defined to be more general than necessary since they will be reused later to define the translations of local and repeat operations.

$$\tau(M) = \text{SIGNATURE}(M) \text{ BODY}(M)$$

where:

$$\text{SIGNATURE}(M) = [\text{Modifiers}(M), " "] \text{TYPE}(M) \text{ Name}(M) \text{ EXCEP}(M) "(" \text{PAR}(\text{Inputs}(M)) "$$

$$\text{BODY}(M) = \begin{cases} ";" \\ \text{if } \underline{\text{abstract}} \in \text{Modifiers}(M) \\ "{" \text{DECL}(M) \text{ OUTER}(M) \text{ RET}(M) " \\ \text{otherwise} \end{cases}$$

$$\text{TYPE}(M) = \begin{cases} \underline{\text{void}} \\ \text{if } \text{Roots}(M) = () \\ \text{type}(\text{head}(\text{Roots}(M))) \\ \text{otherwise} \end{cases}$$

$$\text{EXCEP}(M) = \begin{cases} \underline{\text{throws}} \{ \{ \text{type}(E) \mid E \in \text{Exception}(M) \}, ", ", " ", " " \\ \text{if } \text{Exception}(M) \neq \{ \} \\ \varepsilon \text{ otherwise} \end{cases}$$

$$\text{DECL}(M) = \text{ROOTDECL}(M) \text{ FLAGDECL}(M) \text{ INDEXDECL}(M)$$

$$\text{ROOTDECL}(M) = \begin{cases} \text{type}(\text{head}(\text{Roots}(M))) \text{ name}(\text{head}(\text{Roots}(M))) ";" \\ \text{if } \text{Roots}(M) \neq () \\ \varepsilon \\ \text{otherwise} \end{cases}$$

$$\text{FLAGDECL}(M) = \begin{cases} \text{"boolean" name}(\text{Flag}(M)) ";" \\ \text{if } M \text{ contains a controlled loop} \\ \varepsilon \text{ otherwise} \end{cases}$$

$$\text{INDEXDECL}(M) = \begin{cases} \text{"int" name}(\text{Index}(M)) ", " \text{name}(\text{Uplimit}(M)) ";" \\ \text{if } M \text{ contains a counted loop.} \\ \varepsilon \\ \text{otherwise} \end{cases}$$

$$\text{OUTER}(M) = \begin{cases} \underline{\text{try}} \{ " \text{INNER}(M) " \} \underline{\text{catch}}(\text{ITermException} \text{ Terminate}(M)) \{ " \text{SETFLAG}(M) " \} \\ \text{if } \text{Control}(O) \in \{ \boxtimes, \boxplus \} \text{ for some } O \in \text{Opers}(C) \text{ for some case } C \text{ of } M \\ \text{INNER}(M) \\ \text{otherwise} \end{cases}$$

$$\text{RET}(M) = \begin{cases} \underline{\text{return}} \text{ name}(\text{head}(\text{Roots}(M))) ";" \\ \text{if } \text{Roots}(M) \neq () \\ \varepsilon \\ \text{otherwise} \end{cases}$$

$$\text{SETFLAG}(M) = \begin{cases} \varepsilon & \text{if } M \text{ is a local operation or method} \\ \text{name(Flag}(M)) \text{ " = false;"} & \text{otherwise} \end{cases}$$

$$\text{INNER}(M) = \text{NORM}(\text{NormalCases}(M)) \text{ CATCH}(\text{CatchCases}(M)) \text{ FIN}(M)$$

$$\text{FIN}(M) = \begin{cases} \text{"finally \{ " } \tau(\text{FinallyCase}(M)) \text{ "}" } & \text{if FinallyCase}(M) \neq \text{null} \\ \varepsilon & \text{otherwise} \end{cases}$$

and if X is a list of nodes, and Y is a list of cases

$$\text{PAR}(X) = \begin{cases} \text{type(head}(X)) \text{ name(head}(X)) \text{ " ,"} \text{ PAR}(\text{tail}(X)) & \text{if } |X| > 1 \\ \text{type(head}(X)) \text{ name(head}(X)) & \text{if } |X| = 1 \\ \varepsilon & \text{otherwise} \end{cases}$$

$$\text{NORM}(Y) = \begin{cases} \text{"try \{ " } \tau(\text{head}(Y)) \text{ " } \text{catch (ICaseException " Var(head}(Y)) \text{ " } \text{ " NORM}(\text{tail}(Y)) \text{ "}" } & \text{if } |Y| > 1 \\ \tau(\text{head}(Y)) & \text{if } |Y| = 1 \end{cases}$$

and

$$\text{CATCH}(Y) = \begin{cases} \varepsilon & \text{if } Y = () \\ \tau(\text{head}(Y)) \text{ CATCH}(\text{tail}(Y)) & \text{otherwise} \end{cases}$$

4.2.3.1 Example

In this example we illustrate the translation of JGraph methods to Java by considering a method called MyMethod, partially illustrated in Figure 4-2, which has the structure:

$M = (\{\text{"public"}\}, E, \text{"MyMethod"}, (I_1, I_2), (R), (C_1, C_2, C_3), (C_4, C_5), C_6, G, I, U, E)$

where:

- $\text{name}(I_1) = \text{"i"}$.
- $\text{name}(I_2) = \text{"boo"}$.
- $\text{name}(R) = \text{"Result"}$.
- MyMethod contains both controlled and counted loops.
- $\text{name}(G) = \text{"FlagName"}$, $\text{name}(I) = \text{"IndexName"}$, and $\text{name}(U) = \text{"UpLimit"}$.
- $\text{Var}(C_2) = \text{"var2"}$, $\text{Var}(C_3) = \text{"var3"}$

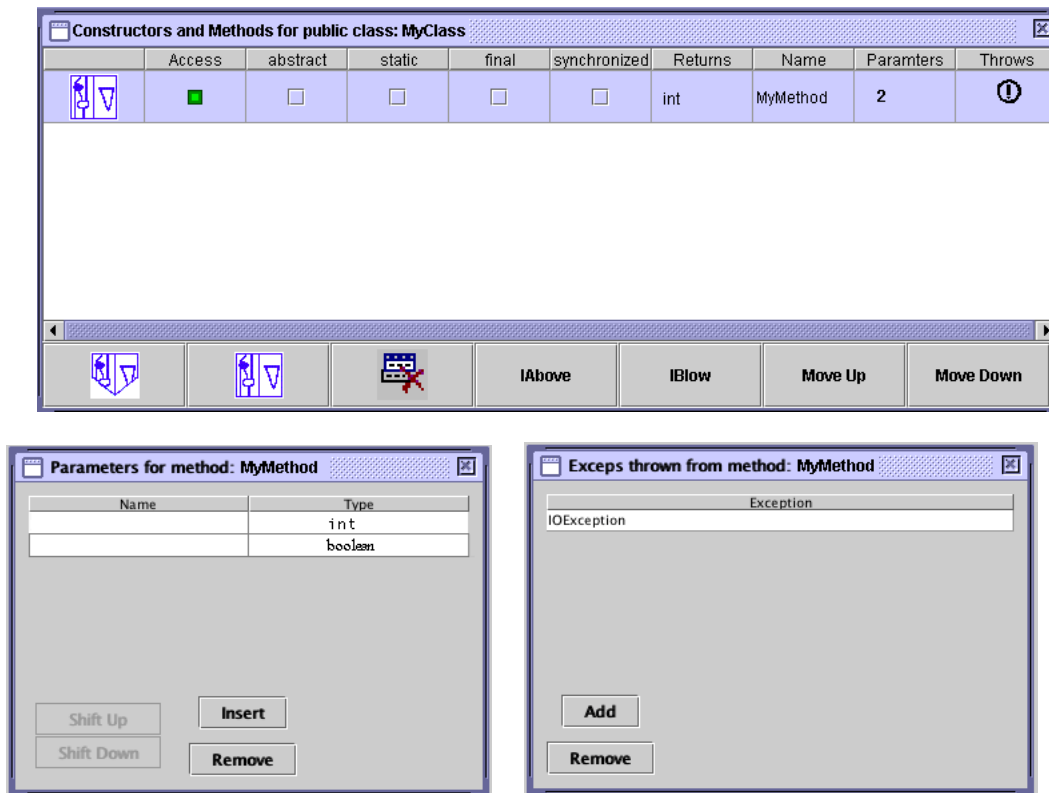


Figure 4-2: A JGraph method

Applying the above translation to this method produces the following Java code:

```

    public int MyMethod(int i, boolean boo) throws ExceptionName
    {
1 . . . . . int Result;
2 . . . . . boolean FlagName;
3 . . . . . int IndexName, UpLimit;
4 . . . . . try
5 . . . . . {
6 . . . . . <normal case, section 4.2.4>
7 . . . . . }
8 . . . . . catch ( ICaseException var2)
9 . . . . . {
10 . . . . . try
11 . . . . . {
12 . . . . . <normal case, section 4.2.4>
13 . . . . . }
14 . . . . . catch ( ICaseException var3)
15 . . . . . {
16 . . . . . <normal case, section 4.2.4>
17 . . . . . }
18 . . . . . }
19 . . . . . <catch case, section 4.2.4>
20 . . . . . <catch case, section 4.2.4>
21 . . . . . finally
22 . . . . . {
23 . . . . . <finally case, section 4.2.4>
24 . . . . . }
25 . . . . . return Result;
    }

```

This example illustrates the translation of a JGraph method into Java, where many of the more subtle and important features of the translation process occur. We now discuss these in detail.

Repeat operations: Iterations in JGraph, represented by repeat operations, are controlled either by indexing through arrays (counted loops), by testing boolean expressions (controlled loops) or by a combination of both. To represent a counted loop in Java, we need two variables, an index to be incremented in each iteration, and an upper limit on the index, used to stop the iteration. Clearly, if a method contains several counted loops, the same two variables can be used for each of them, so it makes sense to declare these variables at the beginning of the method body, as our example illustrates. Note that if the method were to contain no counted loops, then these variable declarations would be omitted. If the method contains a counted loop, in the sense defined above, these declarations are inserted into the Java code by the function INDEXDECL defined above using the **Uplimit** and **Index** components incorporated in the method structure for this purpose.

Similarly, if a method contains a controlled loop, as in the above example, a boolean variable is required, to be set in the body of the loop and tested in its condition. Again, only one such variable is required for all controlled loops contained in a method. The function FLAGDECL defined above inserts a declaration for it at the beginning of the method body in the Java code, if and only if the method contains a controlled loop.

Normal, catch and finally cases: In a JGraph method, exceptions thrown as a result of executing a normal case may be handled by one of the method's catch cases. Also, an exception thrown by a catch case may be handled by a catch case occurring later in the method's sequence of catch cases. This translates into Java in a very straightforward way, as a try block, containing the code corresponding to all the normal cases of the JGraph method, followed by one catch block for each catch case, possibly followed by a finally block corresponding to the JGraph method's finally case. In our example, the try and catch clauses required for this implementation cover the lines from 4 to 24.

Normal case structure: Each method has at least one normal case. Every case except the last in the sequence of normal cases may contain operations with next case controls which, when fired, cause execution to pass immediately to the next case in the sequence. Unlike conditional constructs in more conventional languages where the conditions are kept separate from other computations, within a case

in JGraph, the evaluation of conditions and other computations are mixed together. An operation with a next case control may occur at any point in the linear sequence in which the operations are executed and there may be more than one such operation.

The semantics of the next case control, although it makes sense in the context of data flow, is contrary to the principles of structured programming on which Java is based, where each code block should finish execution and return control to the enclosing code block. The only mechanisms in standard imperative programming languages for causing an abrupt transfer of control out of the middle of a code block are “goto”, “break” and exception throwing. Structured programming languages like Java don’t support “goto”, although it would certainly be the right mechanism for implementing next case. Java has a “break” statement but it is used only inside loops and cases of switch statements. Fortunately, exceptions in Java provide a mechanism which is structurally very similar to the normal case structure of a JGraph method. We can see the correspondance if we note that executing a sequence of normal cases involves either executing the head case of the sequence to completion, or executing the tail. If any next case control in the head case fires then the tail is executed. Similarly executing a “try-catch” pair in Java involves either executing the “try” to completion or executing the “catch”, and if any exception is thrown during execution of the “try”, then the “catch” is executed. Hence the head and tail of a sequence of normal cases corresponds to the “try” and “catch” clauses of a “try-catch” pair, and firing a next case control in the head case corresponds to throwing an exception in the “try” clause.

Our translation of normal cases is based on this correspondance. Wherever a next case control is fired in JGraph, the corresponding Java code, throws a special exception “ICaseException” and the next normal case will be the catch block catching the exception thrown. Lines 4 to 18 show the code correspondance. There are three normal cases in the example. That means there is at least one next case control in each of the first two normal cases. If there were no next control case in the first normal case, lines 8 to 18 would be absent.

4.2.4 Cases.

If C is a case, let P be any total ordering of $\text{Opers}(C)$ obtained by topologically sorting the graph G defined on the case C as described in Section 3.2.5. See [2] for a description of topological sort.

If C is a normal case, then:

$$\tau(C) = \{ \{ \text{type}(L) \text{ name}(L) \mid L \in \text{Local}(C) \}, \text{ ; } \} \text{ CASEBODY}(C) \{ \}$$

If C is a catch case, then:

$$\tau(C) = \{ \text{“catch” } (\text{type}(\text{Catchroot}(C)) \text{ name}(\text{Catchroot}(C))) \{ \} \{ \{ \text{type}(L) \text{ name}(L) \mid L \in \text{Local}(C) \text{ and } L \neq \text{CatchRoot}(C) \}, \text{ ; } \} \text{ CASEBODY}(C) \{ \}$$

If C is a finally case, then:

$$\tau(C) = \{ \{ \{ \text{type}(L) \text{ name}(L) \mid L \in \text{Local}(C) \text{ and } L \notin \text{Previous}(C) \}, \text{ ; } \} \{ \{ \text{type}(\text{Previous}(C)[i]) \text{ name}(\text{Previous}(C)[i]) = \text{name}(\text{Roots}(M)[i]) \mid 1 \leq i \leq |\text{Previous}(C)| \}, \text{ ; } \} \{ \text{ ; } \} \text{ CASEBODY}(C) \{ \}$$

where:

$$\text{CASEBODY}(C) = \text{OPERS}(P) \{ \{ \text{name}(\text{Roots}(M)[i]) = \text{name}(\text{Outputs}(C)[i]) \mid 1 \leq i \leq |\text{Outputs}(C)| \}, \text{ ; } \} \{ \text{ ; } \} \text{ THROW}(C) \{ \}$$

$$\text{THROW}(C) = \begin{cases} \{ \text{“throw” } \text{name}(\text{Exception}(C)) \\ \text{Exception}(C) \neq \text{null} \} \\ \{ \text{ ; } \} \\ \text{otherwise} \end{cases}$$

and if X is a sequence of operations:

$$\text{OPERS}(X) = \begin{cases} \{ \tau(\text{head}(X)) \text{ OPERS}(\text{tail}(X)) \\ \text{if } X \neq () \} \\ \{ \text{ ; } \} \\ \text{otherwise} \end{cases}$$

If C is a constructor case of a constructor of class K , let O be the unique constructor operation, P be any total ordering of $\text{Opers}(C)$ obtained by topologically sorting the graph G defined on cases as described in section 3.2.5 and let $P1$ be the linear ordering of $\text{Pre}(C)$ induced by P then:

$$\tau(C) = \{ \{ \{ \text{type}(A) \text{ name}(A) \mid A \in \text{Local}(C) \}, \text{ ; } \} \text{ CREATE}(O, \{ \}) \text{ OPERS}(P1) \text{ THROW}(C) \{ \}$$

where OPERS and EXCEP are as defined above.

In Java, the body of a constructor must perform exactly one invocation of a constructor of the parent class, which must precede any other actions performed in the constructor body. However, it is legal in Java to provide nested expressions as inputs to that constructor call. A constructor case in JGraph, is defined in such a way that the network of operations that provide inputs to the constructor operation can perform only computations that can be expressed textually as nested expressions (See section 3.2.5). Also, instead of translating these operations as a sequence of assignments, we must produce nested expressions. So the translation of a constructor case is done in two parts, the operations “before” the constructor, and those “after”.

The functional style we have been using to define translation is awkward for expressing the translation of operations before the constructor. This is because in this translation we have to search backward from the constructor operation, remembering the operations we encounter so we can set a variable to the expression on the first visit, then simply use that variable on later visits. This requires a “global variable” which a pure functional notation can not accommodate. Hence, we will express this translation as a procedural algorithm.

In the algorithm described below, we will use Java notations and conventions as much as we can as well as the various notations we have used in translations described earlier, such as juxtaposition to denote concatenation.

```

CREATE( $O$ ,  $L$ )
{
     $S = \text{“”};$ 
    if( $O \in L$ )
        return  $\text{name}(\text{head}(\text{Roots}(O)))$ ;
    else
    {
        if( $O$  is simple, get, array get or constructor operation)
        {
            if( $O$  is an alloc or array operation)
                 $S = S \text{“new”};$ 
            if ( $\text{Target}(O) = \uparrow$ )
                 $S = S \text{“super.”};$ 
            else
                if ( $\text{Target}(O)$  is a node)

```



```

    {
         $O_1$ =operation with Target( $O$ ) as root;
         $S = S$  CREATE( $O_1, L$ );
        if ( $O$  is simple, get)
             $S = S$  “.”;
    }
}
 $S = S$  IDENT( $O$ );
if( $O$  is constructor, simple or alloc)
{
     $T =$  Terminals( $O$ );
     $S = S$  “(”;
    while ( $T \neq ()$ )
    {
         $t =$  head( $T$ );
         $T =$  tail( $T$ );
        if(  $t \in$  Inputs( $K$ ))
             $S = S$  name( $t$ );
        else
        {
             $O_1 =$  operation with  $t$  as root;
             $S = S$  CREATE( $O_1, L$ );
        }
        if ( $T == ()$ )
             $S = S$  “)”;
        else
             $S = S$  “,”;
    }
}
}
if( $O$  is an alloc or array get operation)
{
     $T =$  Dimensions( $O$ );
     $S = S$  “[”;
    while  $T \neq ()$ 
    {
         $t =$  head( $T$ );
         $T =$  tail( $T$ );
        if(  $t \in$  Inputs( $K$ ))
             $S = S$  name( $t$ );
        else
        {
             $O_1 =$  operation with  $t$  as root;
             $S = S$  CREATE( $O_1, L$ );
        }
    }
}

```

```

        if ( $T == ()$ )
             $S = S "]"$ ;
        else
             $S = S "["$ ;
    }
}
if ( $O$  has a root and the root of this operation occurs more than twice in the case)
     $S = \mathbf{name}(\mathbf{head}(\mathbf{Roots}(O))) "=" S$ ;
add  $O$  to  $L$ ;
return  $S$ ;
}
}

```

where:

$$\text{IDENT}(O_1) = \begin{cases} \text{"super"} & \text{if } O_1 \text{ is a constructor operation and } \mathbf{Target}(O_1) = \uparrow \\ \mathbf{name}(K) & \text{if } O_1 \text{ is a constructor operation and } \mathbf{Target}(O_1) \neq \uparrow \\ \mathbf{Name}(O_1) & \text{if } O_1 \text{ is a simple or get operation} \\ \mathbf{Value}(O_1) & \text{if } O_1 \text{ is a literal operation} \end{cases}$$

4.2.4.1 Example.

Here we illustrate the above definitions by considering the translation of the normal, catch and finally cases of a local operation called Local, shown from left to right in Figure 4-3 and referred to below as C_1 , C_2 and C_3 respectively.

We assume that $\mathbf{Roots}(\text{Local}) = (R_1, R_2)$ where $\mathbf{name}(R_1) = \text{"r1"}$, $\mathbf{name}(R_2) = \text{"r2"}$, $\mathbf{type}(R_1) = \text{"String"}$, $\mathbf{type}(R_2) = \text{"int"}$.

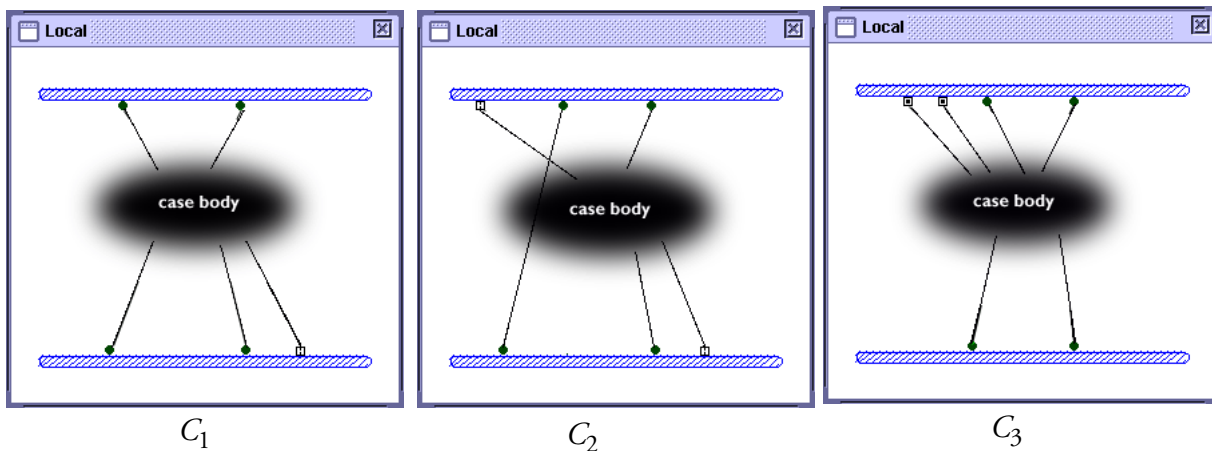


Figure 4-3: The three cases of Local

Table 4-1: Translation of cases

| Case | Corresponding Java code |
|--|---|
| <p>We assume that $\text{Exception}(C_1) = E$, $\text{Outputs}(C_1) = (U_1, U_2)$ and $\text{Local}(C_1) = \{U_1, U_2, L_1, L_2\}$ where $\text{name}(E) = \text{"except"}$ $\text{type}(L_1) = \text{"boolean"}$; $\text{name}(L_1) = \text{"l1"}$. $\text{type}(L_2) = \text{"int"}$; $\text{name}(L_2) = \text{"l2"}$. $\text{name}(U_1) = \text{"u1"}$, $\text{name}(U_2) = \text{"u2"}$.</p> | <pre> { <u>boolean</u> l1; <u>int</u> l2; <u>String</u> u1; <u>int</u> u2; <methods; section 4.2.3> r1 = u1; r2 = u2; <u>throw</u> except; } </pre> |

Table 4-1: Translation of cases

| Case | Corresponding Java code |
|---|---|
| <p>We assume that $\text{Catchroot}(C_2) = T$, $\text{Outputs}(C_2) = (U_3, U_4)$ and $\text{Local}(C_2) = \{U_4, L_1\}$ where $\text{type}(T) = \text{"MyException"}$, $\text{name}(T) = \text{"CEvar"}$, $\text{type}(L_1) = \text{"boolean"}$; $\text{name}(L_1) = \text{"l1"}$. $\text{name}(U_3) = \text{"u3"}$, $\text{name}(U_4) = \text{"u4"}$,</p> | <pre> catch (MyException CEvar) { int u4; boolean l1; <methods; section 4.2.3> r1 = u3; r2 = u4; } </pre> |
| <p>We assume that $\text{Previous}(C_3) = (P_1, P_2)$, $\text{Outputs}(C_3) = (U_5, U_6)$, $\text{Local}(C_3) = \{L_1, L_2, L_3\}$ where $\text{type}(P_1) = \text{"String"}$; $\text{name}(P_1) = \text{"p1"}$, $\text{type}(P_2) = \text{"int"}$; $\text{name}(P_2) = \text{"p2"}$, $\text{type}(L_1) = \text{"String"}$; $\text{name}(L_1) = \text{"l1"}$, $\text{type}(L_2) = \text{"boolean"}$; $\text{name}(L_2) = \text{"l2"}$, $\text{type}(L_3) = \text{"boolean"}$; $\text{name}(L_3) = \text{"l3"}$. $\text{name}(U_5) = \text{"u5"}$, $\text{name}(U_6) = \text{"u6"}$.</p> | <pre> finally { String l1; boolean l2; boolean l3; String p1 = r1; int p2 = r2; <methods; section 4.2.3> r1 = u5; r2 = u6; } </pre> |

4.2.4.2 Example

Here we illustrate the translation of constructor cases using the example in Figure 4-4. Let K be the class, and M be the constructor in which the constructor case occurs.

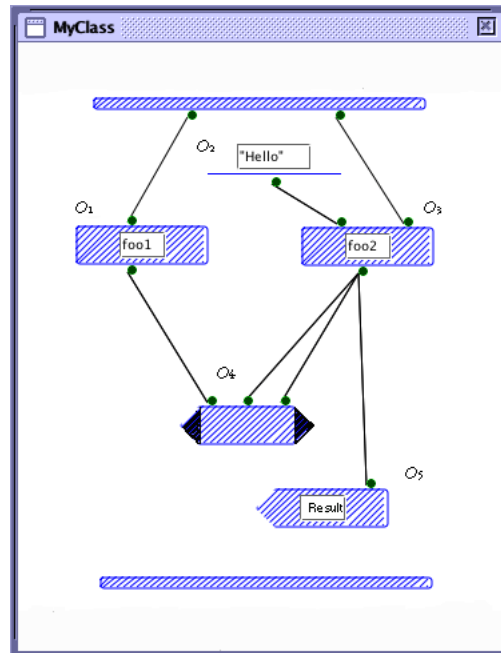


Figure 4-4: Constructor case

Table 4-2: translation of constructor case

| constructor case | Corresponding Java code |
|---|--|
| $C = ((O_1, O_2, O_3, O_4, O_5), \text{null}, \text{null}, ())$ and $\text{Local}(C) = \{L\}$ where $\text{Name}(K) = \text{"MyClass"}$, $O_1 = (\text{null}, \text{"foo1"}, \text{null}, (T_1), (R_1))$ $O_2 = (\text{null}, (R_2), \text{"Hello"})$ $O_3 = (\text{null}, \text{"foo2"}, \text{null}, (T_2, T_3), (R_3))$ $O_4 = (\text{null}, (T_4, T_5, T_6))$ $O_5 = (\text{null}, \text{"Result"}, D)$ $\text{type}(L) = \text{"boolean"}$; $\text{name}(L) = \text{"ret2"}$; We assume that $\text{Inputs}(M) = (I_1, I_2)$, and $\text{name}(I_1) = \text{"param1"}$, $\text{name}(I_2) = \text{"param2"}$ | <pre> { int ret2; MyClass(foo1(param1), ret2=foo2("Hello", param2), ret2); Result = ret2; } </pre> |

4.2.5 Operations

If O is an operation in some case of a method, local or repeat M then:

$$\tau(O) = \left\{ \begin{array}{l} \text{LHS}(O) \text{ TAR}(O) \text{ Name}(O) \text{ "(" TERM(Terminals}(O)) \text{ ")" CNTRL}(O) \\ \quad \text{if } O \text{ is a simple operation} \\ \text{name(head(Roots}(O)) \text{ TAR}(O) \text{ name}(O) \text{ CNTRL}(O) \\ \quad \text{if } O \text{ is a get operation} \\ \text{TAR}(O) \text{ Name}(O) \text{ "=" name(head(Terminals}(O)) \text{ CNTRL}(O) \\ \quad \text{if } O \text{ is a set operation} \\ \text{name(head(Roots}(O)) \text{ "=" new" Name}(O) \text{ CNTRL}(O) \\ \quad \text{if } O \text{ is an alloc operation} \\ \text{name(head(Roots}(O)) \text{ "=" Value}(O) \text{ CNTRL}(O) \\ \quad \text{if } O \text{ is a literal operation} \\ \text{"{" LOCDECL}(O) \text{ OUTER}(O) \text{ "}" } \\ \quad \text{if } O \text{ is a local operation} \\ \text{"{" FLAGDECL}(O) \text{ INDEXDECL}(O) \text{ INIT}(O) \text{ "while (" HEAD}(O) \text{ "}" NEXT}(O) \\ \text{OUTER}(O) \text{ "}" } \\ \quad \text{if } O \text{ is a repeat operation} \\ \text{name(head(Roots}(O)) \text{ "=" new" Name}(O) \text{ "[" TERM(Dimension}(O)) \text{ "]" } \\ \quad \text{if } O \text{ is an array operation} \\ \text{name(head(Roots}(O)) \text{ "=" TAR}(O) \text{ "[" TERM(Dimension}(O)) \text{ "]" } \\ \quad \text{if } O \text{ is an array get operation} \\ \text{TAR}(O) \text{ "[" TERM(Dimension}(O)) \text{ "]" = name(Data}(O)) \\ \quad \text{if } O \text{ is an array set operation} \\ \text{name(head(Roots}(O)) \text{ "=" Value}(O) \text{ "=" name(head(Terminals}(O)) \text{ ";" CNTRL}(O) \\ \quad \text{if } O \text{ is a match operation} \end{array} \right.$$

where DECL, OUTER are as defined in section 4.2.3 and:

$$\text{LOCDECL}(O) = [\{\text{type(Terminals}(O)[i]) \text{ name(Inputs}(O)[i]) \text{ "=" name(Terminals}(O)[i]) \mid 1 \leq i \leq |\text{Terminals}\}, \text{";"}, \text{"\epsilon"}, \text{";"}, \text{";"}$$

$$\text{LHS}(O) = \left\{ \begin{array}{l} \epsilon \quad \text{if } \text{Roots}(O) = () \\ \text{name(head(Roots}(O)) \text{ "="} \\ \quad \text{if } |\text{Roots}(O)| = 1 \end{array} \right.$$

$$\begin{aligned}
\text{TAR}(O) &= \begin{cases} \varepsilon & \text{if } \text{Target}(O) = \text{null} \\ \text{name}(r) \text{ "."} & \text{if } \text{Target}(O) = r \\ \text{"super."} & \text{if } \text{Target}(O) = \uparrow \end{cases} \\
\text{CNTL}(O) &= \begin{cases} \text{"; if(!" name(head(Roots(O))) "("} \{\tau(\text{Control}(O));\} & \text{if } \text{Control}(O) \in \{\boxtimes, \boxtimes\} \text{ or } \text{Control}(O) = \boxtimes \text{ and } M \text{ is a repeat operation} \\ \text{"; if(" name(head(Roots(O))) "("} \{\tau(\text{Control}(O));\} & \text{if } \text{Control}(O) \in \{\boxplus, \boxplus\} \text{ or } \text{Control}(O) = \boxplus \text{ and } M \text{ is a repeat operation} \\ \varepsilon & \text{otherwise} \end{cases} \\
\text{INIT}(O) &= \text{FLAG}(O) \text{ SIMPLE}(O) \text{ IND}(O) \text{ ARRAY}(O) \text{ LOOP}(O) \\
\text{FLAG}(O) &= \begin{cases} \text{name(Flag}(M)) \text{ "=" true ;"} & \text{if } O \text{ is a controlled loop} \\ \varepsilon & \text{otherwise} \end{cases} \\
\text{IND}(O) &= \begin{cases} \text{name(Index}(M)) \text{ "=" 0 ;"} \text{ name(Uplimit}(M)) \text{ "=" 0 ;"} & \text{if } O \text{ is a counted loop} \\ \varepsilon & \text{otherwise} \end{cases} \\
\text{SIMPLE}(O) &= [\{\text{type(Terminals}(O)[i]) \text{ name(Inputs}(O)[i]) = \text{name(Terminals}(O)[i])} \mid \\ & 1 \leq i \leq |\text{Terminals}|, \text{Ttypes}(T) = \text{Simple}, \text{";"}, \varepsilon, \text{";} \}] \\
\text{ARRAY}(O) &= [\{\text{type(Inputs}(O)[i]) \text{ name(Inputs}(O)[i]) \mid \text{Ttypes(Inputs}(O)[i]) = \text{Array}, 1 \leq i \leq |\text{Terminals}|, \text{";"}, \varepsilon, \text{";} \}; [\{\text{name(Uplimit}(O)) \text{ "=" name(Uplimit}(O)) \text{ "<" name(Terminals}(O)[i]) \text{ ".length ?"} \text{ name(Uplimit}(O)) \text{ ":"} \text{ name(Inputs}(O)[i]) \mid T \in \text{Terminals}(O) 1 \leq i \leq |\text{Inputs}| \text{ and } \text{Ttypes(Terminals}(O)[i]) = \text{Array}, \text{";"}, \text{" "}, \text{";} \}] \\
\text{LOOP}(O) &= [\{\text{type(Terminals}(O)[i]) \text{ name(Inputs}(O)[i]) \text{";"}, \\ & \text{name(Roots}(O)[\text{Ttypes(Terminals}(O)[i])] = \text{name(Terminals}(O)[i])} \mid \\ & 1 \leq i \leq |\text{Terminals}| \text{ and } \text{Ttypes(Terminals}(O)[i]) \text{ is an integer}, \text{";"}, \text{" "}, \text{";} \}] \\
\text{HEAD}(O) &= \begin{cases} \text{name(Flag}(M)) \text{ "&&" name(Index}(M)) \text{ "<=" name(Uplimit}(M))} & \text{if } O \text{ is a controlled loop and counted loop} \\ \text{name(Flag}(M)) & \text{if } O \text{ is a controlled loop but not counted loop} \\ \text{name(Index}(M)) \text{ "<=" name(Uplimit}(M))} & \text{if } O \text{ is a counted loop but not controlled loop} \\ \text{"true"} & \\ \varepsilon & \text{otherwise} \end{cases} \\
\text{NEXT}(O) &= [\{\text{name(Inputs}(O)[i]) \text{ "=" name(Roots}(O)[\text{Ttype(Terminals}(O)[i])}] \mid \\ & 1 \leq i \leq |\text{Terminals}(O)| \text{ and } \text{Ttype(Terminals}(O)[i]) \text{ is an integer}, \text{";"}, \text{" "}, \\ & \text{";"}, [\{\text{name(Inputs}(O)[j]) \text{ "=" name(Terminals}(O)[j]) \text{ "["} \text{name(Index}(O)) \text{ "]" } \mid \\ & 1 \leq j \leq |\text{Terminals}(O)| \text{ and } \text{Ttype(Terminals}(O)[j]) = \text{Array}, \text{";"}, \text{" "}, \\ & \text{";"}, \text{INCRINDEX}(O) \}]
\end{aligned}$$

$$\text{INCRINDEX}(O) = \begin{cases} \text{name}(\text{Index}(O)) \text{ “++;”} & \text{if } \text{Ttypes}(\text{Terminals}(O)[i]) = \text{Array for some } i (1 \leq i \leq |\text{Terminals}|) \\ \varepsilon & \text{otherwise} \end{cases}$$

and for any list X consisting of nodes and integers.

$$\text{TERM}(X) = \begin{cases} \varepsilon & \text{if } X = () \\ \text{name}(\text{head}(X)) & \text{if } |X| = 1 \text{ and } \text{head}(X) \text{ is a node} \\ \text{head}(X) & \text{if } |X| = 1 \text{ and } \text{head}(X) \text{ is an integer.} \\ \text{name}(\text{head}(X)) \text{ “]” } \text{TERM}(\text{tail}(X)) & \text{if } |X| > 1 \text{ and } \text{head}(X) \text{ is a node.} \\ \text{head}(X) \text{ “]” } \text{TERM}(\text{tail}(X)) & \text{if } |X| > 1 \text{ and } \text{head}(X) \text{ is an integer.} \end{cases}$$

4.2.5.1 Examples of translating operations

Here we present an extensive sequence of examples to illustrate the above definitions.

Table 4-3: Translation of operations

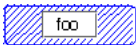


| Operation | Formal structure | Corresponding Java |
|---|--|---------------------------|
|  | $(\text{null}, \text{“foo”}, \text{null}, (), ())$ | <code>foo()</code> |
|  | $(T, \text{“foo”}, \text{null}, (), ())$ where T is a node such that: name (T) = “t”. | <code>t.foo()</code> |
|  | $(T_1, \text{“foo”}, \text{null}, (T_2), (R))$ where T_1 , T_2 and R are nodes such that: name (T_1) = “t1”, name (T_2) = “t2”, name (R) = “r”. | <code>r=t1.foo(t2)</code> |

Table 4-3: Translation of operations





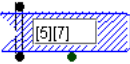
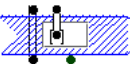

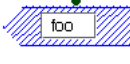

| | | |
|---|--|-----------------------------|
|  | $(T_1, \text{"foo"}, \text{null}, (T_2), (R))$ $T_1 = \uparrow$ $\text{name}(T_2) = \text{"t2"}$, $\text{name}(R) = \text{"r"}$. | $r = \text{super.foo}(t2)$ |
|  | $(\text{null}, \text{"foo"}, \text{null}, (R))$ where R is a node such that: $\text{name}(R) = \text{"r"}$ | $r = \text{foo}$ |
|  | $(T, \text{"foo"}, \text{null}, (R))$ where T, R are nodes such that: $\text{name}(T) = \text{"t"}$ $\text{name}(R) = \text{"r"}$ | $r = t.\text{foo}$ |
|  | $(T, \text{"foo"}, \text{null}, (R))$ where T, R are nodes such that: $T = \uparrow$ $\text{name}(R) = \text{"r"}$ | $r = \text{super.foo}$ |
|  | $(T, (R), (5,7))$ where T, R are nodes such that: $\text{name}(T) = \text{"t"}$ $\text{name}(R) = \text{"r"}$ | $r = t[5][7]$ |
|  | $(T, (R), (D))$ where T, R and D are nodes such that: $\text{name}(T) = \text{"t"}$ $\text{name}(R) = \text{"r"}$ $\text{name}(D) = \text{"index"}$ | $r = t[\text{index}]$ |
|  | $(T, (R), (5, N, 7))$ where T, R, N are nodes such that: $\text{name}(T) = \text{"n"}$ $\text{name}(R) = \text{"r"}$ $\text{name}(D) = \text{"index"}$ | $r = n[5][\text{index}][7]$ |
|  | $(\text{null}, \text{"foo"}, D)$ where D is a node such that: where: $\text{name}(D) = \text{"d"}$. | $\text{foo} = d$ |
|  | $(T, \text{"foo"}, D)$ where T, D are nodes such that: $\text{name}(T) = \text{"t"}$ $\text{name}(D) = \text{"d"}$ | $t.\text{foo} = d$ |

Table 4-3: Translation of operations


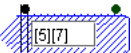
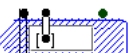



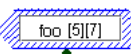

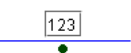
| | | |
|---|---|----------------------------|
|  | $(T, \text{"foo"}, D)$ where T, D are nodes such that: $T = \uparrow$ $\text{name}(D) = \text{"d"}$ | <u>super</u> .foo = d |
|  | $(T, (5,7), D)$ where T, D are nodes such that: $\text{name}(T) = \text{"t"}$ $\text{name}(D) = \text{"d"}$ | t[5][7] = d |
|  | $(T, (N), D)$ where T, N, D are nodes such that: $\text{name}(T) = \text{"t"}$ $\text{name}(N) = \text{"index"}$ $\text{name}(D) = \text{"d"}$ | t[index] = d |
|  | $(T, (5, N, 7), D)$ where T, N and D are nodes such that: $\text{name}(T) = \text{"n"}$ $\text{name}(N) = \text{"index"}$ $\text{name}(D) = \text{"d"}$ | n[5][index][7] = d |
|  | $(\text{"foo"}, (T_1, T_2), R)$ where T_1, T_2 and R are nodes such that: $\text{name}(T_1) = \text{"t1"}$ $\text{name}(T_2) = \text{"t2"}$ $\text{name}(R) = \text{"r"}$, | r = <u>new</u> foo(t1,t2) |
|  | $(\text{"foo"}, (), R)$ where R is a node such that: $\text{name}(R) = \text{"r"}$ | r = <u>new</u> foo() |
|  | $(\text{"foo"}, (R), (5,7))$ where R is a node such that: $\text{name}(R) = \text{"r"}$. | r = <u>new</u> foo[5][7] |
|  | $(\text{"foo"}, (R), (N))$ where R, N are nodes such that: $\text{name}(R) = \text{"r"}$, $\text{name}(N) = \text{"index"}$. | r1 = <u>new</u> foo[index] |
|  | $(\text{null}, (R), \text{"123"})$ where R is a node such that: $\text{name}(R) = \text{"r"}$ | r = 123 |

Table 4-3: Translation of operations

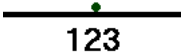


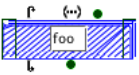


| | | |
|---|--|---|
|  | <p>(<i>null</i>, T, (R), “123”) where: name(T) = “t” name(R) = “r”</p> <p>Note that a match is simply a shorthand notation for testing for equality and has to do with specific controls. However, it is allowed to have the control <i>null</i>, in which case the match operation does not accomplish anything useful. An example of match with controls will be shown in section 4.2.6.1.</p> | <pre>r = t==123;</pre> |
|  | <p>($\{\}$, (T_1, T_2), (R_1, R_2, R_3), N, C, X, (I_1, I_2), F, D, U, E)</p> <p>where N is a sequence of normal cases, C is a sequence of catch cases, X is a finally case, T_1, T_2, I_1, I_2, F, D, U and E are nodes and type(T_1) = “<u>int</u>”, type(T_2) = “String”, name(I_1) = “i1”, name(I_2) = “i2”, name(T_1) = “t1”, name(T_2) = “t2”, name(R_1) = “r1”, name(R_2) = “r2”, name(R_3) = “r3”.</p> | <pre>{ <u>int</u> i1 = t1; String i2 = t2; <Cases; section 4.2.3.1> }</pre> |
|  | <p>($\{\}$, (T_1, T_2, T_3), $\()$, N, C, X, (I_1, I_2, I_3), F, D, U, E)</p> <p>where N is a sequence of normal cases, C is a sequence of catch cases, X is a finally case, T_1, T_2, I_1, I_2, F, D, and U are nodes and type(T_1) = “<u>int</u>”, type(T_2) = “String”, type(T_3) = “<u>boolean</u>”, name(I_1) = “i1”, name(I_2) = “i2”, name(I_3) = “i3”, name(T_1) = “t1”, name(T_2) = “t2”, name(T_3) = “t3”, name(E) = “e”</p> <p>we assume that there are terminate controls in the cases of this local operation.</p> | <pre>{ <u>int</u> i1 = t1; String i2 = t2; <u>try</u> { <Cases; section 4.2.3.1> } <u>catch</u>(ITermException e) { } }</pre> |

Table 4-3: Translation of operations

| | | |
|---|--|--|
|  | <p>$(null, (T_1, T_2, T_3), (R_1, R_2), N, C, X, E, (I_1, I_2, I_3), F, D, U)$</p> <p>where N is a sequence of normal cases, C is a sequence of catch cases, X is a finally case, $T_1, T_2, T_3, R, I_1, I_2, F, D, U$ are nodes and</p> <p>$E(I_1) = 1,$ $E(I_2) = \text{Array},$ $E(I_3) = \text{Simple},$ $\text{type}(T_1) = \text{"int"}, \text{type}(T_2) = \text{"int"},$ $\text{type}(T_3) = \text{"boolean"},$ $\text{type}(F) = \text{"boolean"}, \text{type}(D) = \text{"int"},$ $\text{type}(U) = \text{"int"},$ $\text{name}(T_1) = \text{"t1"}, \text{name}(T_2) = \text{"t2"},$ $\text{name}(T_3) = \text{"t3"},$ $\text{name}(I_1) = \text{"i1"}, \text{name}(I_2) = \text{"i2"},$ $\text{name}(I_3) = \text{"i3"},$ $\text{name}(R_1) = \text{"r1"}, \text{name}(R_2) = \text{"r2"},$ $\text{name}(F) = \text{"flag1"}, \text{name}(D) = \text{"index1"},$ $\text{name}(U) = \text{"uplimit1"},$ we assume that $\text{name}(\text{Flag}(M)) = \text{"flag"},$ $\text{name}(\text{Index}(M)) = \text{"index"},$ $\text{name}(\text{Uplimit}(M)) = \text{"uplimit"},$ and there are controlled loops and counted loops in N, C and X.</p> | <pre> boolean flag1; int index1, uplimit1; flag = true; boolean i3 = t3; index = 0 ; uplimit = 0; int i2 = t2; uplimit=uplimit<t2.length? uplimit:t2.length; int i1; r1 = t1; while(flag && index < uplimit) { i1 = r1; i2 = t2[index]; index++; <Cases; section 4.2.3.1> } </pre> |
|---|--|--|

As the example of a local operation containing operations with terminate controls shows, if a terminate control (i.e.  or ) on an operation is activated, execution of the case containing the operation stops immediately, causing execution of the method, local or repeat to which the case belongs to stop.

To achieve equivalent behaviour in Java, we introduce a special exception class `ITermException`. If execution of a JGraph operation triggers a terminate control, then in the corresponding code, an instance of `ITermException` is thrown, as we will see later in section 4.2.6.1. If a JGraph method contains a case which has an operation with a terminate control, then the body of the corresponding Java

method will consist of a try-catch structure where the try clause contains the code corresponding to the cases of the JGraph method, and the catch clause catches any ITermException thrown in the try clause. If there were no operation in any case of a JGraph method with a terminate control, then the body of the corresponding Java method would not have this try-catch structure.

4.2.6 Controls

The translation of a control is defined with respect to its context. Let $X = \mathbf{Control}(O)$ where $O \in \mathbf{Opers}(C)$ and C is a case of some method, local or repeat M , then:

$$\tau(X) = \begin{cases} \varepsilon & \text{if } X = \text{null} \\ \text{"throw new ICaseException();"} & \text{if } X \in \{\boxtimes, \checkmark\} \\ \text{"throw new ITermException();"} & \text{if } X \in \{\boxtimes, \boxplus\} \\ \mathbf{name}(\mathbf{Flag}(M)) \text{ " = false ;"} & \text{if } X \in \{\boxtimes, \checkmark\} \end{cases}$$

4.2.6.1 Examples of translating controls

Table 4-4: Translation of operations with controls




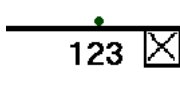
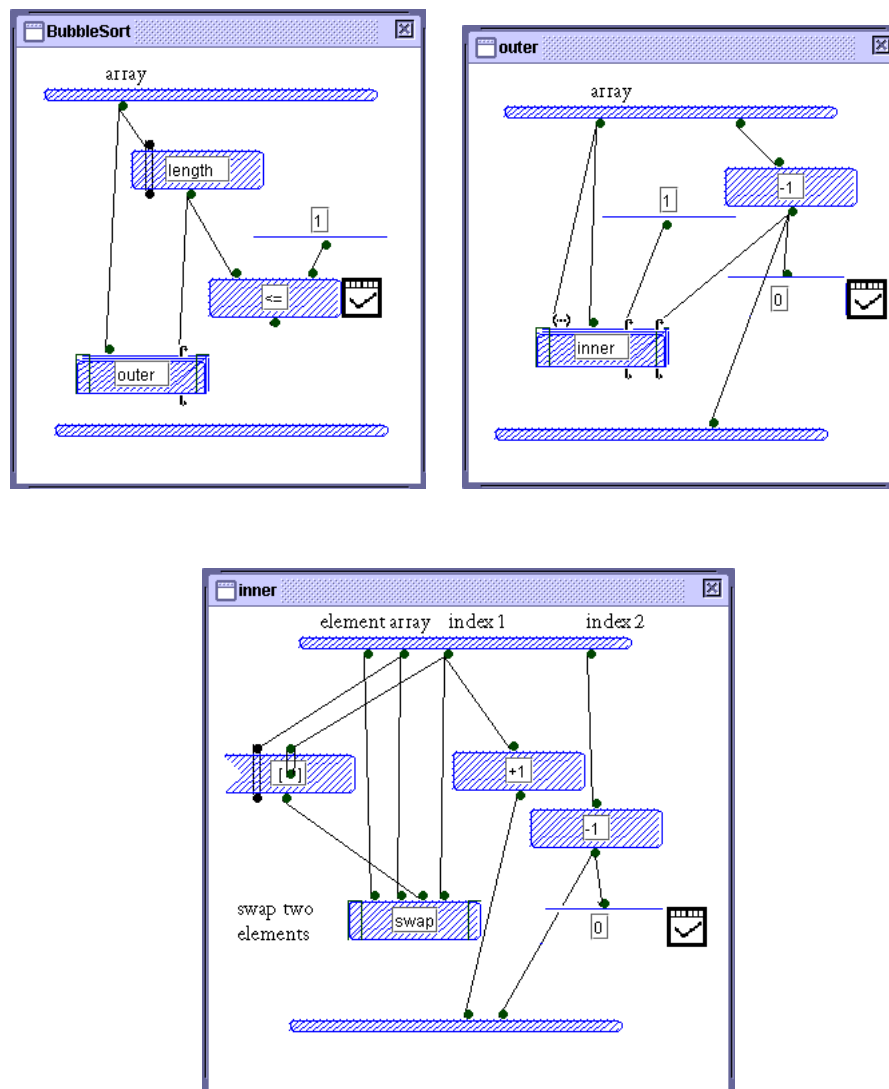
| | | |
|---|---|---|
|  | $(T, \text{"foo"}, \{\boxtimes\}, R)$ where: $\mathbf{name}(T) = \text{"t"}$ $\mathbf{name}(R) = \text{"r"}$ | $r = t.foo;$ $\mathbf{if}(!r)$ $\text{throw new ICaseException() ;}$ |
|  | $(T, \text{"foo"}, \{\boxplus\}, R)$ where: $\mathbf{name}(T) = \text{"t"}$ $\mathbf{name}(R) = \text{"r"}$ | $r = t.foo;$ $\mathbf{if}(!r)$ $\text{throw new ITermException() ;}$ |
|  | $(T, \text{"foo"}, \{\boxtimes\}, R)$ where: $\mathbf{name}(T) = \text{"t"}$ $\mathbf{name}(R) = \text{"r"}$ We assume that the operation is in a case of a repeat | $r = t.foo;$ $\mathbf{if}(!r)$ $\mathbf{flag} = \underline{\text{false}} ;$ |

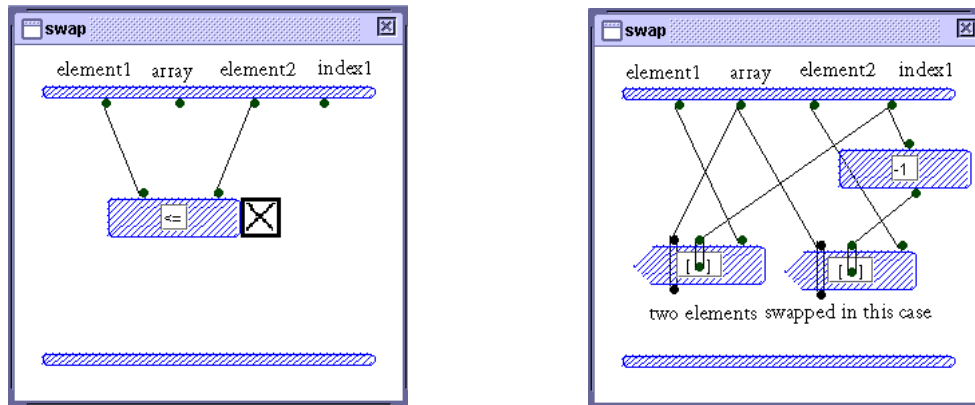
Table 4-4: Translation of operations with controls

| | | |
|---|--|--|
|  | <p>(<input checked="" type="checkbox"/>, T, (R), "123") where: $\text{name}(T) = \text{"t"}$ $\text{name}(R) = \text{"r"}$</p> | <pre>r = t==123; if(!r) throw new ICaseException() ;</pre> |
|---|--|--|

4.2.7 A general example

In this section, we illustrate the translation process presented in this chapter by considering a real example, a JGraph method BubbleSort that applies the bubblesort algorithm to sort an array in place.





The corresponding Java program generated from the JGraph program above is as follows:

```

void BubbleSort(int[] a)
{
    try{
        boolean flagname;
        int r1;
        boolean r;
        r1 = a.length;
        r = r1<=1;
        if( r ) throw new ITermException() ;
        { // code for repeat operation "outer" starts here
            boolean outer_flag;
            flagname = true;
            int[] outer_a = a;
            int outer_in1;
            int outer_r1 = r1;
            while ( flagname )
            {
                outer_in1 = outer_r1;
                int outer_rr1;
                boolean rr;
                int outer_loop1;
                try{
                    outer_rr1 = outer_in1-1;
                    rr = outer_rr1==0;
                    if( rr )throw new ITermException() ;
                    outer_loop1 = 1;
                    { // code for repeat operation "inner" starts here
                        outer_flag = true;
                        int[] inner_a = outer_a;
                        index = 0;
                        uplimit = 0;

```

```

int inner_arrelement1;
uplimit = uplimit < outer_a.length ? uplimit : outer_a.length;
int inner_r1 = outer_loop1;
int inner_r2;
int inner_out1 = outer_rr1;
while (outer_flag && index < uplimit)
{
    inner_r2 = inner_out1;
    inner_arrelement1 = outer_a[index];
    index++;
    try{
        int inner_rr2;
        boolean rr2;
        int inner_arrelement2;
        inner_rr2 = inner_r2-1;
        rr2 = inner_rr2 == 0;
        if( rr2) throw new ITermException() ;
        inner_arrelement2 = inner_a[inner_r1];
        { // code for local operation "swap" starts here
            int swap_arrelement1 = inner_arrelement1;
            int[] swap_a = inner_a;
            int swap_arrelement2 = inner_arrelement2;
            int swap_r1 = inner_r1;
            try{ // first normal case of "swap" starts here
                boolean swap_r;
                swap_r= swap_arrelement1 < swap_arrelement2;
                if( !swap_r ) throw new ICaseException();
            }catch(ICaseException e1)
            { // second normal case of "swap" starts here
                int swap_r2;
                swap_a[swap_r1]= swap_arrelement2;
                swap_r2 = swap_r1-1;
                swap_a[swap_r2] = swap_arrelement1;
            }
        }
        outer_loop1== inner_r1+1;
        inner_out1= inner_rr2;
    }catch(ITermException e2)
    {
        outer_flag = false;
    }
}
}
    outer_r1 = outer_rr1;
} catch(ITermException e3)
{

```



```
        flagname = false;  
    }  
    }  
    }  
}catch(ITermException e)  
{  
}  
}
```

4.3 Conclusion

In this chapter we have given the formal definition of the translation process to Java. It would be impossible in any reasonable amount of space to give examples that cover all combinations of features. We hope, however, that the examples we have provided will help the reader understand some of the more subtle aspects.

In the next chapter, we will discuss the problem the other way around, that is translating Java programs into JGraph.

5 Importing Java into JGraph

5.1 Introduction

In this chapter, we will address the problem of importing Java code into JGraph. The purpose is to provide JGraph with the capability to visually edit Java code. First individual elements of Java code such as keywords, operators and variables are identified by applying lexical analysis. Based on the lexeme list generated, corresponding JGraph elements are created and presented in a JGraph project for editing. We will go through this procedure in a top down fashion. We assume that the Java program imported is correct, thereby avoiding the complications associated with error processing.

The translation we will describe assumes that the source code is general Java: that is, we are not trying to identify any structures which might have originated from translating a JGraph program to Java as described in Chapter 4. At the end of the chapter we will discuss some of the issues associated with doing that. The presentation in this chapter is less formal than that of Chapter 4 relying on a series of examples.

5.2 Class files

For convenience, we assume that each class of a project to be translated is stored in a single file containing no other classes. That is each file represents an individual Java class. For example, consider the following class file. Here we use the notation <...> introduced in Chapter 4. In this chapter, it indicates that the example code includes a Java construct that will be discussed in another section.

```
package MyPackage;  
import java.awt;
```

```
import java.swing;
<class; section 5.3>
```

The JGraph generated from this file is illustrated in Figure 5-1.

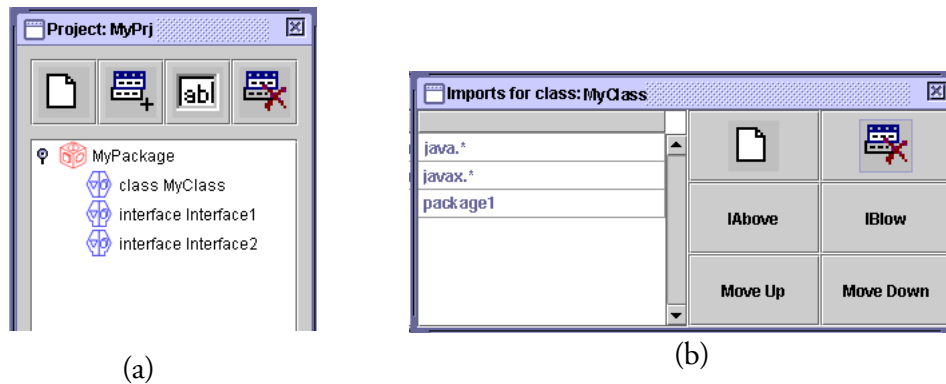


Figure 5-1: JGraph package

At the package level, Java and JGraph are slightly different. In particular, unlike JGraph, Java has no explicit representation of a package, although specific implementations of Java may. In the example, therefore, the generated JGraph has a project window, shown in Figure 5-1(a), that depicts MyPackage, as well as the list of classes and interfaces the package contains.

5.3 Class

The following examples illustrate the translation of Java classes and interfaces to JGraph.

Table 5-1: Translation of classes





| Java construct | Corresponding JGraph |
|--|---|
| <pre>class MyClass1 { <attributes; section 5.5> <constructors; section 5.7> <methods; section 5.7> }</pre> |  |
| <pre>public class MyClass2 { As above }</pre> |  |
| <pre>final class MyClass3 { As above }</pre> |  |
| <pre>public final class MyClass4 { As above }</pre> |  |
| <pre>abstract class MyClass5 { As above }</pre> |  |
| <pre>public abstract class MyClass6 { As above }</pre> |  |
| <pre>interface MyClass7 { As above }</pre> |  |

Table 5-1: Translation of classes

| Java construct | Corresponding JGraph |
|--|---|
| <pre>public interface MyClass8 { <i>As above</i> }</pre> |  |

In the examples above, different Java classes are represented by different JGraph images. Each JGraph class image consists of a class icon or interface icon in the centre, together with icons at the bottom corners of the image indicating the qualifiers of the class or interface. For example, a diamond shape at the left-bottom indicates that the class can be referenced only from inside the package. A green rectangle indicates that the class is a public class. The characters “A” and “F” indicate that the class is abstract or final.

5.4 Inheritance

In the following example, we illustrate the translation of inheritance relationships. Consider the following Java class definition, assuming that the package `MyPackage` in which it occurs does not contain a class called `SuperClass`. In the classes window of `MyPackage`, shown in Figure 5-2, `SuperClass` is represented as an alias.

```
public class MyClass extends SuperClass implements Interface1, Interface2
{
    <attributes; section 5.5>
    <methods and constructors; section 5.7>
}
```

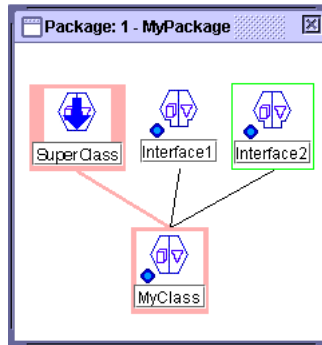


Figure 5-2: A JGraph class window showing inheritance

5.5 Attributes

The translation of Java attributes to JGraph is trivial, as shown below:

```

public int i=0;
public boolean boo=true;
static int j=0;

```

| Access | Attribute | Final | Transient | Volatile | Type | Name | Value |
|-------------------------------------|--------------------------|--------------------------|--------------------------|--------------------------|---------|------|-------|
| <input checked="" type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | int | i | 0 |
| <input checked="" type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | boolean | boo | true |
| <input checked="" type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | int | j | 0 |

Figure 5-3: Attribute list

5.6 Preprocessing

Because JGraph is a data flow language, there are some significant differences between its execution model and that of Java. In particular, JGraph transmits data by data flow, while Java transmits data via variables. Two different variables in Java may have the same name providing they do not have overlapping scopes; also, a Java variable may have a value assigned to it more than once during its lifetime.

In contrast a JGraph node can occur only once as a root, and receives a value only once. We deal with these differences by preprocessing each Java method before translating to JGraph. Much of the preprocessing is aimed at transforming the Java code such that the variables have the same characteristics as roots.

5.6.1 Simplifying control structures

For simplicity, in the following discussion, we assume that any loop statements in the method are while loops, since loops of other kinds can be translated into while loops in the obvious way.

Also, we will assume that all switch statements have been translated into conditional statements in the obvious way.

5.6.2 Multiple declarations

To remove multiple declarations, we rename variables to ensure that each variable name is declared only once, then we move the declarations to the beginning of the method body. If a variable is initialized where it is declared, we decouple the declaration part from the initialisation part and move just the declaration. The remaining initialisation becomes an assignment. By making sure that no variable name is used more than once within a method, we obtain Java code in which the pattern of variable use is similar to the pattern of occurrence of nodes in JGraph.

If a variable is declared as an array variable, we move the declaration part as discussed above. In Java, an array variable may be initialised in its declaration using an array constant, consisting of a list of values enclosed in braces. Not only does Java not allow array constants in ordinary assignment statements, but JGraph has no equivalent to array constants. So in order to obtain correct Java which is similar in structure to JGraph, the single assignment using an array constant is replaced by a sequence of assignments, one for each array element.

Table 5-2: Sample of multiple declarations

| Original code | Preprocessed code |
|--|---|
| <pre> void MyMethod(boolean b) { if(b) { int n; <body-t> }else { byte n; <body-e> } } </pre> | <pre> void MyMethod(boolean b) { int n; byte n1; if(b) { <body-t> }else { <body-e1> } } </pre> <p>where <body-e1> is the same as <body-e> except that each occurrence of n is replaced by n1;</p> |

5.6.3 Embedded sequences of statements.

In Java, a statement in a sequence of statements can itself be a sequence of statements enclosed in braces. If there are no local variables declared in such a nested sequence, then the nesting is unnecessary. After the above preprocessing step that deals with the removal of multiple declarations, all the declarations of local variables are moved to the beginning of the method. As a result, no nested sequence of statements will have local variables declared. Therefore, such sequences can be removed, as follows.

Let S be a sequence of statements $\{S_1, \dots, S_n\}$ occurring in the method such that some statement S_i in the sequence is also a sequence of statements, say $\{T_1, \dots, T_m\}$, then replace S in the method by $\{S_1, \dots, S_{(i-1)}, T_1, \dots, T_m, S_{(i+1)}, \dots, S_n\}$ and repeat until there are no embedded sequences of statements remaining.

Table 5-3: Sample of embedded sequences of statements.

| Original code | Preprocessed code |
|--|--|
| <pre> void MyMethod(boolean b) { <body-1> { <u>int</u> n; <body-2> } <body-3> } </pre> | <pre> void MyMethod(boolean b) { <u>int</u> n; <body-1> <body-2> <body-3> } </pre> |

5.6.4 Method returns

In Java, a statement of the form "return E" in a non-void method, where E is some expression, causes execution of the method to break, and the value of E to be returned to the calling procedure. If the "return" occurs in the "try" or "catch" clauses of a "try-catch-finally" structure, executing the return will cause a break to the "finally" clause. This pattern can be nested to any depth, in which case the value returned by the method will be that computed by the last "return" statement executed.

A void method may or may not contain return statements. If it does, then it behaves as described above except that it returns no value.

In contrast, since JGraph is data flow, each method must be executed to completion. Therefore, before we import a Java program into JGraph, we preprocess it so that the method body has a specific form and contains only one "return" statement.

First, a list of all return types of all methods in the program being translated is created, and for each type in the list, an exception class is added to the program, as follows:

```

package MyPackage ;

import java.awt.*;

public class IRetExcep_T extends Throwable

```

```

{
    T RetResult;
    public IRetExcep_T(T RetResult)
    {
        this.RetResult = RetResult;
    }
}

```

where we assume the return type of the method being dealt with is T and the package in which the method occurs is MyPackage. If the method returns void, then we have a simpler definition for the corresponding exception class as follows:

```

package MyPackage ;
import java.awt.*;
public class IRetExcep_void extends Throwable
{
}

```

Assuming the classes for return types have been added, we now transform each method as follows.

Suppose the method is as follows, where T is the return type.

```

<qualifiers> T name(<parameters>)
{
    <body>
}

```

Assuming T is not void, preprocessing produces the following:

```

<qualifiers> T name(<parameters>)
{

```

```

        <newbody>
    }

```

where <newbody> is obtained by replacing every statement of the form "return E;" in <body>, where E is some expression, by "throw new IRetExcep_T(E);"

If T is void, the preprocessed code is as above except that E is the empty string in each of the throw statements.

Note that if a method returns void, the code above should not contain T Result.

5.6.5 Unary increment and decrement operators

In Java, the unary operator ++ can occur as either a prefix operator or a postfix operator, and must have a variable as its operand. The value of the expression ++x is the value of x+1; the value of x++ is the value of x; and in both cases, the value of x after the expression is evaluated is the value of x+1.

The unary operator -- behaves analogously, except that it decrements its operand.

We preprocess the unary operators above by replacing all occurrences of ++x with (x = x + 1), and x++ with ((x=x+1)-1), and similarly for x-- and --x.

5.6.6 The operators +=, -=, /=, *=

We replace all the occurrences of expressions of the form x op= c, where op denotes one of the operators +, -, /, *, by x = x op c;

5.6.7 Condition of while loops and conditional statements

In Java, the condition of while loops and conditional statements could be either a boolean variable or an expression. In order to deal with embedded assignments in the condition and exceptions that might be thrown in the condition, which will be discussed in later sections, we preprocess it as follows:

1. If S = "if <condition> <body-t> else <body-e>"

S is replaced by:

$$S_1 = "E = \langle \text{condition} \rangle; \underline{\text{if}} (E) \{ \langle \text{body-t} \rangle \} \underline{\text{else}} \{ \langle \text{body-e} \rangle \}"$$

where E is a boolean variable. The declaration of E is added in the beginning of the method.

2. If S = "while <condition> <body> "

S is replaced by:

$$S_1 = "E = \langle \text{condition} \rangle; \underline{\text{while}}(E) \{ \langle \text{body} \rangle \rightarrow E = \langle \text{condition} \rangle; \}"$$

where E is a boolean variable. The declaration of E is added in the beginning of the method.

5.6.8 Embedded assignments

In this step, we discuss the removal of embedded assignments; that is, assignment expressions that occur as operands in other expressions. There are several good reasons for taking this step. Making every assignment into a statement will simplify the process of removing multiple assignments discussed later. The translation of if-else and loops will be simpler as a result, since no variables will be assigned values in the condition. The translation of Java variable-to-variable assignments into JGraph will be simplified.

As we will explain, removing embedded assignments is accomplished by adding assignment statements before the statement that contains the embedded assignment expressions. It is important, therefore, that preprocessing step 5.6.2 has been done at this point since it moves all declarations to the start of the method, guaranteeing that the new assignment statements added in this step follow declarations of the variables to which they assign values.

In this step we iteratively transform a statement containing embedded assignment expressions by replacing each embedded assignment by a new variable, introducing a preceding assignment state-

ment that sets the value of the new variable, and a following assignment statement that copies the value of the new variable into the original variable. For example the statement:

$$z = (x = y * x)$$

would be expanded to:

$$x1 = y * x;$$

$$z = x1;$$

$$x = x1;$$

If a variable on the left of an embedded assignment appears elsewhere in the expression, then we have to replace some of its occurrences in the expression with the corresponding new variable, and if there are several embedded assignments in an expression, we need to be careful about the order in which they are removed, and exactly which variable occurrences in the expression are replaced.

Assignment and method call statement

If embedded assignments occur in an assignment or method call statement, the process consists of two steps.

1. Rewriting the statement

In Java, operands of an expression are processed left to right, except for the operands of = which are processed in the opposite order. To make it easier to specify which variable occurrences get renamed in step 2, below, we transform the expression so that all operands are processed left to right. To do this, we replace every assignment expression $x = A$ in the statement by the expression $A \Rightarrow x$.

2. Expanding the statement

Suppose that after step 1, the statement has the form $S_0 A \Rightarrow x S_1$; where at least one of S_0 and S_1 is not the empty string, and S_0 contains no occurrence of \Rightarrow . Then the statement is replaced by $x_1 = A; S_0 x_1 S_1'$; $x = x_1$; where S_1' is obtained by replacing every x in S_1 by a new variable x_1 of the same type as x . The declaration of x_1 is added at the beginning of the method. This process is repeated until no occurrences of \Rightarrow remain.

We illustrate the above process by considering the following example:

$$x = (x = 5*(y = 3)) + (x = y*x)$$

After step 1 above, the original statement is re-written to:

$$(5*(3 \Rightarrow y) \Rightarrow x) + (y*x \Rightarrow x) \Rightarrow x;$$

Applying step 2 repeatedly, results in the following sequence of transformations:

$$\begin{aligned} y1 &= 3; \\ (5*y1 \Rightarrow x) + (y1*x \Rightarrow x) &\Rightarrow x; \\ y &= y1; \end{aligned}$$

Note that we have removed superfluous parentheses in generating the above. The next expansion produces:

$$\begin{aligned} y1 &= 3; \\ x1 &= 5*y1; \\ x1 + (y1*x1 \Rightarrow x1) &\Rightarrow x; \\ x &= x1; \\ y &= y1; \end{aligned}$$

Finally we obtain:

```
y1 = 3;  
x1 = 5*y1;  
x2 = y1*x1;  
x1 + x2 => x;  
x1 = x2;  
x = x1;  
y = y1;
```

Since the original statement was an assignment, we must convert the last \Rightarrow into $=$, obtaining:

```
y1 = 3;  
x1 = 5*y1;  
x2 = y1*x1;  
x = x1 + x2;  
x1 = x2;  
x = x1;  
y = y1;
```

Note that because of section 5.6.7, there would be no assignments embedded in the condition of while loops and conditional statement.

5.6.9 Missing else

As we will see below, each if-then-else statement in a Java program will be translated into a two-case local operation. In Java, such conditional statements may have no else clause: however, in JGraph, to obtain conditional behaviour, more than one case is required. To expedite their translation to JGraph, therefore, an empty else clause is added to each if-then statement.

5.6.10 Exceptions

In Java, a value assigned to a variable before an exception is thrown will be available in the code executed after the exception, provided that this code is in the scope of the variable. In JGraph, however, when control transfers out of the case where the exception is thrown the only values computed in the case which will still be available are those passed as outputs from the case and those will be available only in the finally case of the method, local or repeat to which the exception-throwing case belongs.

For that reason, in order to translate a Java program into JGraph, we need to rearrange the Java code so that in the JGraph translation, a value assigned to a node before an exception is thrown will be available in the computation conducted after the exception. The way we accomplish this rearrangement is to locate each statement that may throw an exception, then replace it with code that will generate an appropriate exception without actually throwing it, and avoid executing code that would not have been executed if an exception had been thrown in the original code. The exceptions created are assigned to a variable, which is declared at the beginning of the method to be of type `Throwable`, and used to throw an exception, if necessary, only at the end of the method. In this way we reduce all the various throws to assignments to the variable declared.

Firstly, we declare at the beginning of the method a new variable v of type `Throwable`, then we define a function `STRIPTHROW` which operates on statements and sequences of statements to replace all exception throwing by assignment to v .

Let S be a sequence of statements, then:

$$\text{STRIPTHROW}(S) = \begin{cases} S_1; \text{STRIPTHROW}(x_1); \text{if}(v == \text{null}) \{ \text{STRIPTHROW}(S_2) \} \\ \quad \text{if } S = S_1; x_1; S_2 \text{ where } S_1 \text{ is a sequence of statements that cannot throw} \\ \quad \text{exceptions, and } x_1 \text{ is a statement that may throw exceptions.} \\ S \quad \text{if } S \text{ cannot throw exceptions.} \end{cases}$$

If S is a void expression or an assignment, and S may throw exceptions, then

$$\text{STRIPTHROW}(S) = \text{try}\{S; v = \text{null};\} \text{catch}(\text{Throwable } e) \{v = e;\}$$

If S is a conditional statement that may throw exceptions, say $S = \text{if } (E) \{<\text{body-t}>\} \text{else } \{<\text{body-e}>\}$

where E is a boolean variable, then:

$$\text{STRIPTHROW}(S) = \text{if } (E)\{\text{STRIPTHROW}(\text{body-t})\} \text{else}\{\text{STRIPTHROW}(\text{body-e})\}$$

If S is a throw statement, say $S = \text{throw } T;$, then

$$\text{STRIPTHROW}(S) = \text{"v = T;"}$$

If S is a while loop statement, say $S = \text{while } (E) \{<\text{body}>\}$ where E is a boolean variable, then:

$$\text{STRIPTHROW}(S) = \text{while } (E)\{\text{STRIPTHROW}(<\text{body}>)\}$$

If S is a try-catch-finally statement, let

$$S = \text{try}\{<\text{body-t}>\} \text{catch}(E_1 \ e_1)\{<\text{body-c}_1>\} \text{catch}(E_2 \ e_2)\{<\text{body-c}_2>\} \dots \\ \text{catch}(E_m \ e_m)\{<\text{body-c}_m>\} \text{finally}\{<\text{body-f}>\}$$

then:

$$\text{STRIPTHROW}(S) = \text{STRIPTHROW}(<\text{body-t}>) \text{if } (v \neq \text{null})\{ \\ \text{if } (v \text{ instanceof } E_1) \{e_1 = v; v = \text{null}; \text{STRIPTHROW}(<\text{body-c}_1>)\} \\ \text{else if } (v \text{ instanceof } E_2) \{e_2 = v; v = \text{null}; \text{STRIPTHROW}(<\text{body-c}_2>)\} \dots \\ \text{else if } (v \text{ instanceof } E_m) \{e_m = v; v = \text{null}; \text{STRIPTHROW}(<\text{body-c}_m>)\} \text{else}\{\}\} \text{else}\{\} \\ \text{STRIPTHROW}(<\text{body-f}>)$$

Note that if a try-catch-finally does not contain a finally clause, $\text{STRIPTHROW}(<\text{body-f}>)$ would not occur in the above.

Using STRIPTHROW , we transform the body, $<\text{body}>$ of the method M. First we define a function THROWS to deal with throws clause of M, as follows:

$$\text{THROWS}(M) = \begin{cases} \text{if}(v \text{ instanceof } p_1) \text{ throw } (p_1) v; \\ \text{else if}(v \text{ instanceof } p_2) \text{ throw } (p_2) v; \dots \\ \text{else if}(v \text{ instanceof } p_n) \text{ throw } (p_n) v; \\ \text{else}\{\} \\ \quad \text{if the declaration of } M \text{ includes the clause } \underline{\text{throws}} \ p_1, p_2 \dots p_n \text{ for some } n > 0 \\ \varepsilon \quad \text{if } M \text{ has no throws clause.} \end{cases}$$

If M is non-void and returns value of type T , we replace $\langle \text{body} \rangle$ by
 $\text{STRIPTHROW}(\langle \text{body} \rangle); \text{THROWS}(M) \text{ return } ((\text{IRetExcep_}T)v).\text{RetResult};$

If M is void and has a throws clause, we replace $\langle \text{body} \rangle$ by
 $\text{STRIPTHROW}(\langle \text{body} \rangle); \text{if } (v \neq \text{null}) \text{ THROWS}(M) \text{ else}\{\}$

Otherwise, we replace $\langle \text{body} \rangle$ by $\text{STRIPTHROW}(\langle \text{body} \rangle)$.

This process is illustrated by the example as follows.

| Code after step 5.5.8 | Preprocessed code |
|--|---|
| <pre>String MyMethod(<u>boolean</u> b) <u>throws</u> E₃ { <u>try</u> { <u>if</u>(b) <u>throw new</u> E₁(); <u>else</u> <u>throw new</u> E₂(); }<u>catch</u>(E₁ e₁) { System.out.println("E₁ is thrown"); <u>throw new</u> E₃(); <u>return</u> "Hello World"; }<u>catch</u>(E₂ e₂) { System.out.println("E₂ is thrown"); }<u>finally</u> { System.out.println("finally is reached"); } } where E₁, E₂, E₃ are three different exception types.</pre> | <pre><u>void</u> MyMethod(<u>boolean</u> b) <u>throws</u> E₃ { Throwable v; <u>if</u>(b) v = <u>new</u> E₁(); <u>else</u> v = <u>new</u> E₂(); <u>if</u>(v != <u>null</u>) <u>if</u>(v <u>instanceof</u> E₁) { e₁=v; v = <u>null</u>; System.out.println("E₁ is thrown"); v = <u>new</u> E₃(); }<u>else</u> <u>if</u>(v <u>instanceof</u> E₂) { e₂ = v; v = <u>null</u>; System.out.println("E₂ is thrown"); } System.out.println("finally is reached"); } <u>else</u>{ <u>if</u>(v <u>instanceof</u> E₃) <u>throw</u> (E₃)v; <u>else</u> { } <u>return</u> ((IRetExcep_T) v).RetResult; }</pre> |

5.6.1.1 Isolating variables.

In JGraph, conditionals are dealt with by multi-case locals, loops by repeat operations, and exception handling by multi-case locals. Each of these structures involves cases with their own roots, different

from the roots in the enclosing case. In Java, on the other hand, the variables that occur in loops, if-then-else statements and try-catch-finally statements can also occur outside. Consequently, in this preprocessing step, we transform the Java code by introducing new local variables into these structures, different from variables occurring outside them.

If R is a variable then R is *external* to a statement if it is not an attribute of the class of the method and has an occurrence in the method which is not in the statement.

Let X be the set of all variables that are external to but occur in a statement S where S is a loop, if-then-else or try-catch-finally statement., and for each x in X , let x' be a new variable; then the declaration " $T\ x'$;" is added to the beginning of the method, and S is replaced by S_N as follows.

Note that the preprocessing step as described in section 5.6.10 guarantees that there is no finally clause in the method body, every catch case is in the particular form catch($E\ e$){ $v = e$;}, and there is no more than one catch case in any try-catch statements.

1. If $S = \underline{\text{if}}\ \langle\text{condition}\rangle\ \langle\text{body-t}\rangle\ \underline{\text{else}}\ \langle\text{body-e}\rangle\ "$

S is replaced by:

$$S_1 = \text{"IN } \underline{\text{if}}\ \langle\text{condition1}\rangle\ \{\langle\text{body-t}_1\rangle\ \text{OUT}\} \underline{\text{else}}\ \{\langle\text{body-e}_1\rangle\ \text{OUT}\}"$$

where

- if x is assigned a value in $\langle\text{body-t}\rangle$ or $\langle\text{body-e}\rangle$ then OUT is the statement " $x = \text{COPY}(x')$;" and otherwise OUT is the empty string.
- $\langle\text{condition1}\rangle$, $\langle\text{body-t}_1\rangle$ and $\langle\text{body-e}_1\rangle$ are obtained by replacing all occurrences of x by x' in $\langle\text{condition}\rangle$, $\langle\text{body-t}\rangle$ and $\langle\text{body-e}\rangle$ respectively.
- if x occurs in $\langle\text{condition}\rangle$, or x is used before it is assigned in $\langle\text{body-t}\rangle$ or in $\langle\text{body-e}\rangle$, or x is assigned in $\langle\text{body-t}\rangle$ but not in $\langle\text{body-e}\rangle$ or vice versa, then IN is the statement " $x' = \text{COPY}(x)$;" and otherwise IN is the empty string.

This process is repeated for all the variables in X , obtaining S_N .

Each assignment statement that includes the function COPY is a value passing statement that essentially does a variable-to-variable assignment, but corresponds to the transmission of a value between the outside and the inside of a JGraph entity, for example, from a terminal on the outside of a local to the corresponding input bar root on the inside. Although value passing statements are Java assignments, in the final translation to JGraph, they will be treated differently from variable-to-variable assignments. Note that the Java code that results from the introduction of these value passing statements can not be correct since the function COPY is undefined, and in fact could not be defined since it applies to variables of any type. However, if every value passing statement is replaced by an assignment, the code will be correct.

This process is illustrated by the example below.

| Code after step 5.5.9 | Preprocessed code |
|---|--|
| <pre> void MyMethod(boolean b) { int i=0; int k=-1; if(b) { int n = i; <body-t> }else { int m = k; <body-e> k = m; } } </pre> <p>Assuming that i is not assigned a value anywhere in <body-t></p> | <pre> void MyMethod(boolean b) { int i'; int i=0; i' = COPY(i); k' = COPY(k); if(b) { int n = i'; <body-t'> k = COPY(k'); }else { int m = k'; <body-e'> k' = m; k = COPY(k'); } } </pre> |

2. If $S = \text{"while <condition> <body>"}$

S is replaced by:

$$S_1 = \text{" IN while <condition_1>\{ <body_1> OUT\}}$$

where

- if x occurs in $\langle \text{condition} \rangle$ or is used before it is assigned in $\langle \text{body} \rangle$ then IN is the statement " $x' = \text{COPY}(x)$;" and otherwise IN is the empty string.
- if x is assigned a value before it is used in $\langle \text{body} \rangle$ then OUT is the statement " $x = \text{COPY}(x')$;". Otherwise, if x is both used and assigned and is used before it is assigned a value in $\langle \text{body} \rangle$ then OUT is " $\text{LP}(x') ; x = \text{COPY}(x')$;". Otherwise OUT is the empty string.
- $\langle \text{body}_1 \rangle$ is obtained by replacing all occurrences of x by x' in $\langle \text{body} \rangle$.
- $\langle \text{condition}_1 \rangle$ is obtained by replacing all occurrences of x by x' in $\langle \text{condition} \rangle$

This process is repeated for all the variables in X , obtaining S_N .

The function LP identifies variables in Java that will correspond to the loop terminals and loop roots of a repeat operation in JGraph. Note that the Java code that results from the introduction of the loop function LP cannot be correct since LP is undefined, and in fact could not be defined since it applies to variables of any type.

This process is illustrated by the example below.

| Code after step 5.5.9 | Preprocessed code |
|--|---|
| <pre> void MyMethod(boolean b) { int i = 0; while(b) { i += 1; System.out.println(i); if(i == 100) { b = false; } else {} } } </pre> | <pre> void MyMethod(boolean b) { int i'; boolean b'; boolean b"; int i = 0; b' = COPY(b); while(b') { i' += 1; System.out.println(i'); i" = COYP(i'); b" = COPY(b'); if(i" == 100) { b" = false; b' = COPY(b"); } else { b' = COPY(b"); } i = COPY (i'); LP(b'); b = COPY(b'); } } </pre> |

3. If $S = \text{"try } \langle \text{body} \rangle \text{ catch}(E \ e) \{ \langle \text{body-e} \rangle \}$ ", S is replaced by:

$$S_1 = \text{"IN } \text{try } \{ \langle \text{body-1} \rangle \text{ OUT} \} \text{ catch } (E \ e) \{ \langle \text{body-e} \rangle \text{ OUT} \} \text{"}$$

where

- if x is used before it is assigned in $\langle \text{body} \rangle$ then IN is the statement " $x' = \text{COPY}(x)$;" and otherwise IN is the empty string.

- if x is assigned a value in $\langle \text{body} \rangle$, or x is v then OUT is the statement " $x = \text{COPY}(x)$ ";. Otherwise, OUT is the empty string.
- $\langle \text{body-1} \rangle$ is obtained by replacing all occurrences of x by x' in $\langle \text{body} \rangle$.

This process is repeated for all the variables in X , obtaining S_N .

Note that $\langle \text{body} \rangle$ is not possibly in the form "throw T;" since it has been eliminated in section 5.6.10.

This process is illustrated by the example below.

| Code after step 5.5.9 | Preprocessed code |
|--|--|
| <pre> int i = 0; int n = 100; try { S; v = null; }catch(E e) { v = e; } </pre> <p>where E is an Exception type, S is a statement that may throw an exception of type E. Assuming that variables i and n are used in S.</p> | <pre> int i = 0; int n = 100; n' = COPY(n); i' = COPY(i); try { S'; v = null; }catch(E e) { v = e; } </pre> <p>where S' is the same as S except that each occurrence of n and i are replaced by n' and i'.</p> |

5.6.12 Removing multiple assignments

Since JGraph is a data flow language each of its "variables", that is, roots, can be assigned a value only once. This property of data flow languages is called *single assignment*. In Java, however, a variable can, in general, be assigned a value more than once. So we need to modify a given Java program to remove multiple assignments.

Step 5.6.2 guarantees that all variables in a method are uniquely named and therefore makes it easy to identify multiple assignments since two assignments to variables of the same name must, in fact, be assignments to the same variable. Also, in looking for multiple assignments, we can restrict our attention to assignment statements since embedded assignment expressions were removed in step 5.6.8. We also note that because of step 5.6.9, every conditional statement has an else clause.

We say that two assignment statements in a method are *compatible* iff the variables being assigned are different, or the assignments are in different clauses of a try-catch statement or an if-then-else statement; otherwise the two assignments are incompatible.

Let the body of the method be S_1 “ $x = A;$ ” S_2 “ $x = B;$ ” S_3 , where the two assignments are incompatible.

The method body is replaced by S_1 “ $x = A;$ ” S_2 “ $x_1 = B;$ ” S_3' , where S_3' is obtained by replacing each occurrence of x in S_3 by x_1 except that wherever $LP(x)$ occurs in S_3 , it is replaced by $LOOP(x, x_1)$, where x_1 is a new variable of the same type as x . The declaration of x_1 is added to the beginning of the method. The process is repeated for all the multiple assignments until it no longer applies.

This process is illustrated by the example below.

Like LP , $LOOP$ identifies variables in Java that will correspond to the loop terminals and loop roots of a repeat operation in JGraph, and like LP , $LOOP$ could not be defined since it applies to variables of any type. However, if $LOOP(x, x')$ is replaced by the assignment $x = x'$, the code will be correct.

| Code after step 5.5.10 | Preprocessed code |
|--|--|
| <pre> void MyMethod(boolean b) { int i'; boolean b'; boolean b"; int i = 0; b' = COPY(b); while(b') { i' += 1; System.out.println(i'); i" = COYP(i'); b" = COPY(b'); if(i" == 100) { b" = false; b' = COPY(b"); }else { b' = COPY(b"); } i = COPY (i'); LP(b'); b = COPY(b'); } } </pre> | <pre> void MyMethod(boolean b) { int i'; int i"; boolean b'; boolean b"; boolean b₁; boolean b₂; int i = 0; b' = COPY(b); while(b') { i' += 1; System.out.println(i'); i" = COYP(i'); b" = COPY(b'); if(i" == 100) { b₁ = false; b₂ = COPY(b₁); }else { b₂ = COPY(b"); } i = COPY (i'); LOOP(b', b₂); b = COPY(b₂); } } </pre> |

5.6.13 Variable to variable assignments

We remove all assignments of the form $x = y$, where x and y are variables, and replace every occurrence of x by y .

5.7 Methods and Constructors

Here we illustrate how Java methods and constructors are translated into JGraph by considering a class with several methods and one constructor. Here we are not dealing with the translation of the method bodies, so we have omitted them from the Java code, and show only those JGraph windows that correspond to the explicit parts of the Java code.

```

MyClass(int i, boolean boo)
{
    <expressions; section 5.8.1>
    <controls; section 5.9.1, 5.9.2, 5.9.3>
    <assignments; section 5.8.2>
}
int MyMethod1(int i, boolean boo) throws IOException
{
    As above
}
public String MyMethod2(boolean boo)
{
    As above
}
protected void MyMethod3(void) throws Exception
{
    As above
}
private int MyMethod4(boolean boo)
{
    As above
}

```

Some of the corresponding JGraph windows are shown in Figure 5-4. To avoid repetition, the details of parameters and exceptions are shown for only one method.

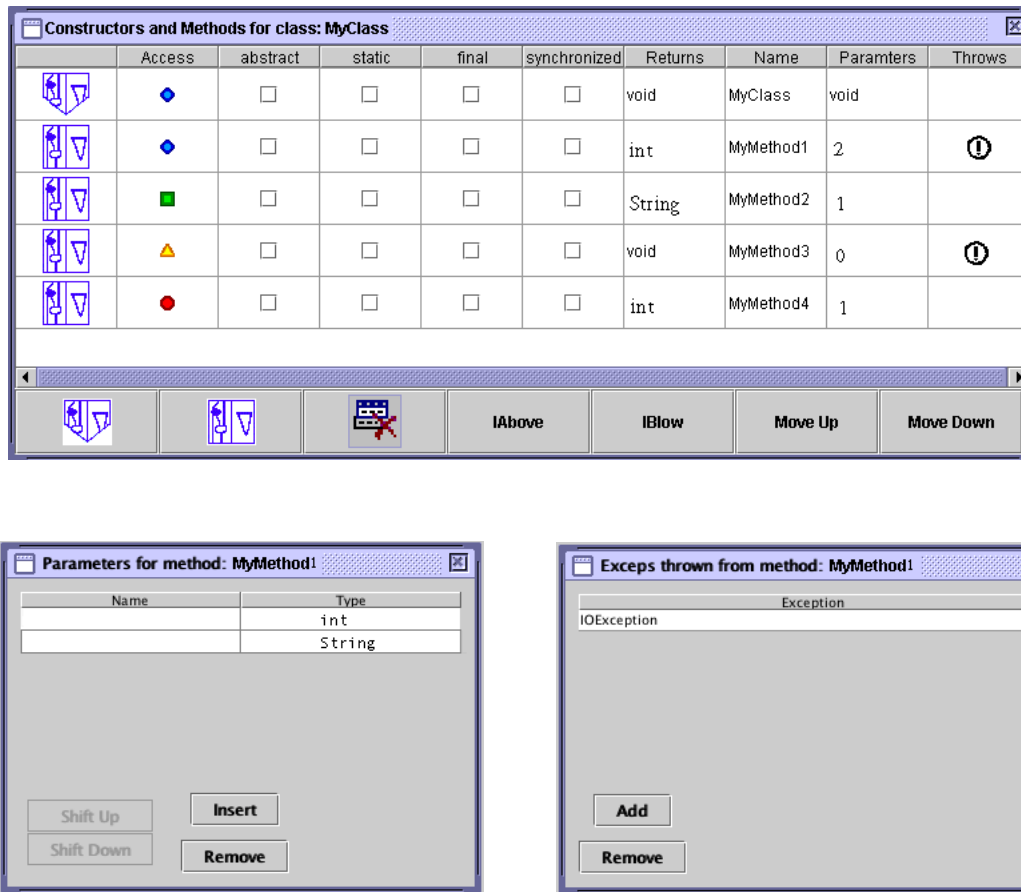


Figure 5-4: JGraph methods corresponding to Java


5.8 Expressions

In this section we will address the translation of expressions, where an expression is built using the standard Java operators including assignment. We ignore declarations since they have no representation in the corresponding JGraph except as types of roots.

Since JGraph is a dataflow language, it does not in general have assignments like Java since there are no variables. There is an exception to this general rule though, which is when the left-hand side of an assignment is an expression that represents an attribute of an instance. In this case JGraph actually performs an assignment in much the same way as Java. So first we will deal with Java expressions that do not involve assignments, then consider assignment expressions. All Java expressions either return

one value or no value. We will refer to these two categories as “non-void” and “void” respectively. In all examples of non-void expressions, we will identify the root in the corresponding JGraph structure that provides the value of the expression.

In Java, a variable is an expression. Since JGraph has no variables, there is no direct representation in JGraph of a Java expression that is simply a variable. In Java, however, a variable cannot occur as an expression unless it has been assigned a value. So the root which corresponds to an expression which is simply a variable is the root of the expression that provides the value for the variable.

In the following discussion, we will use fuzzy blobs to represent JGraph structures equivalent to Java expressions and statements. So if X is a Java expression, then  represents the corresponding JGraph structure. If a fuzzy blob represents a non-void expression, then there will be a unique root on some operation in it that corresponds to the expression in the sense that the value for the expression will be produced at that root. Since variables can be assigned within a Java expression, so within a corresponding fuzzy blob there will be other roots corresponding to these assigned variables.

5.8.1 Expressions which are not assignments

Table 5-4: Examples of translating functional expressions

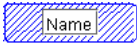
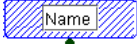
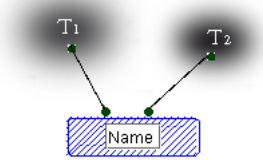
| Java Expression | Corresponding JGraph |
|--|--|
| Name() where Name refers to a void method |  |
| Name() where Name refers to a non-void method |  where the root corresponds to the expression. |
| Name(T_1, T_2) where Name refers to a void method and T_1 and T_2 are expressions |  |

Table 5-4: Examples of translating functional expressions

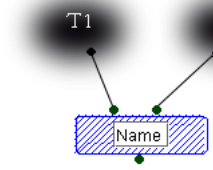
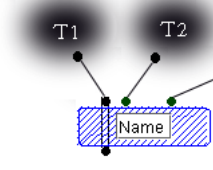
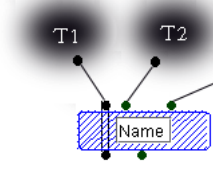
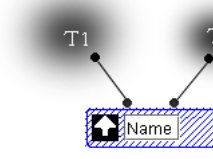
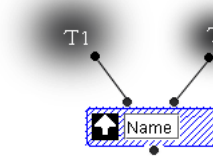
| Java Expression | Corresponding JGraph |
|--|--|
| <p data-bbox="389 346 560 388">$\text{Name}(T_1, T_2)$</p> <p data-bbox="308 399 617 514">Where Name refers to a non-void method and T_1 and T_2 are expressions</p> |  <p data-bbox="657 504 1307 577">where the root of the operation Name corresponds to the expression.</p> |
| <p data-bbox="365 630 576 672">$T_1.\text{Name}(T_2, T_3)$</p> <p data-bbox="308 682 609 798">where Name refers to a void method and T_1, T_2 and T_3 are expressions</p> |  |
| <p data-bbox="365 913 576 955">$T_1.\text{Name}(T_2, T_3)$</p> <p data-bbox="308 966 625 1081">Where Name refers to a non-void method and T_1, T_2 and T_3 are expressions</p> |  <p data-bbox="657 1071 1307 1144">where the root of the operation Name corresponds to the expression.</p> |
| <p data-bbox="349 1176 592 1218"><u>super</u>.Name(T_1, T_2)</p> <p data-bbox="308 1228 609 1344">where Name refers to a void method and T_1 and T_2 are expressions</p> |  |
| <p data-bbox="349 1438 592 1480"><u>super</u>.Name(T_1, T_2)</p> <p data-bbox="308 1491 617 1606">Where Name refers to a non-void method and T_1 and T_2 are expressions</p> |  <p data-bbox="657 1596 1307 1669">where the root of the operation Name corresponds to the expression.</p> |

Table 5-5: Examples of translating operator expressions

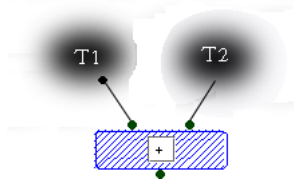
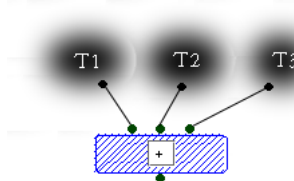
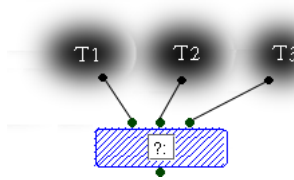
| Java Expression | Corresponding JGraph |
|---|--|
| $T_1 + T_2$ where T_1 and T_2 are expressions |  <p>where the root of the operation + corresponds to the expression.</p> |
| $T_1 + T_2 + T_3$ where T_1 , T_2 and T_3 are expressions |  <p>where the root of the operation + corresponds to the expression. Note that associative binary operations like + can have any number of inputs.</p> |
| $T_1 ? T_2 : T_3$ where T_1 , T_2 and T_3 are expressions |  <p>where the root of the operation ?: corresponds to the expression.</p> |

Table 5-6: Object Get

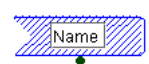
| Java Expression | Corresponding JGraph |
|---|---|
| Name where Name refers to an attribute |  <p>where the root corresponds to the expression.</p> |

Table 5-6: Object Get

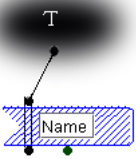

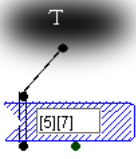
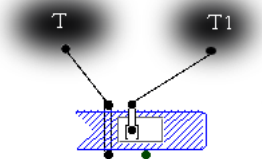
| Java Expression | Corresponding JGraph |
|---|---|
| <p>$T.Name$ where Name refers to an attribute and T is an expression</p> |  <p>where the root of the operation Name corresponds to the expression.</p> |
| <p>$super.Name$ where Name refers to an attribute</p> |  <p>where the root corresponds to the expression.</p> |
| <p>$T[5][7]$ where T is an expression</p> |  <p>where the root of the operation corresponds to the expression.</p> |
| <p>$T[T_1]$ where T and T_1 are expressions</p> |  <p>where the root of the operation corresponds to the expression.</p> |

Table 5-7: Allocations

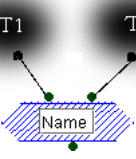
| Java Expression | Corresponding JGraph |
|---|--|
| <p>$new Name(T_1, T_2)$ where Name refers to a class</p> |  <p>where the root of the operation Name corresponds to the expression</p> |

Table 5-7: Allocations

| Java Expression | Corresponding JGraph |
|--|---|
| <code>new Name()</code> where Name refers to a class | A blue hatched hexagonal node containing the text 'Name'. A small black dot is positioned below the node, representing the root of the expression. where the root corresponds to the expression |
| <code>new foo[5][7]</code> where foo refers to a class or type | A blue hatched hexagonal node containing the text 'foo [5][7]'. A small black dot is positioned below the node, representing the root of the expression. where the root corresponds to the expression |
| <code>new foo[T]</code> where foo refers to a class or type and T is an expression | A blue hatched hexagonal node containing the text 'foo [T]'. A small black dot is positioned below the node. A line connects this dot to a larger, shaded node labeled 'T' above it, representing the root of the operation corresponding to the expression. where the root of the operation corresponds to the expression |

Table 5-8: Constants

| Java Expression | Corresponding JGraph |
|-----------------|---|
| "Hello" | A white rectangular node containing the text '"Hello"'. A horizontal blue line is drawn below the node, and a small black dot is positioned below the line, representing the root of the expression. where the root corresponds to the expression. |

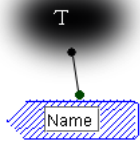
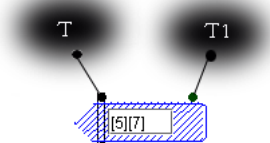
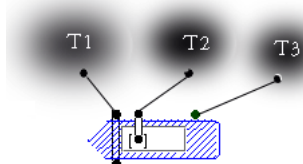
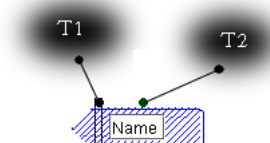
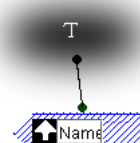
5.8.2 Assignment expressions

In Java an assignment expression has the form $X=Y$, where Y is an expression and X is either a variable which is not an attribute of a class, or an expression that evaluates to an attribute of an instance or an element of an array. Since JGraph is data flow, it has no variables, and therefore, no construct equivalent to Java's assignment to a variable. The use of a variable after it is assigned in Java will be represented in JGraph by a data link connecting the root corresponding to the expression that provides the

variable its value, to a terminal in the JGraph structure corresponding to the expression in which the variable is used.

The second kind of JGraph assignment, in which the left hand side is an expression that evaluates to an attribute of an instance or element of an array, corresponds to a set operation in JGraph. The various possibilities are illustrated below:

Table 5-9: Object Set and Array Set

| Java Expression | Corresponding JGraph |
|--|--|
| <p style="text-align: center;">Name=T</p> <p>where Name refers to an attribute and T is an expression</p> |  |
| <p style="text-align: center;">T[5][7]=T₁</p> <p>where T and T₁ are expressions</p> |  |
| <p style="text-align: center;">T₁[T₂]=T₃</p> <p>where T₁, T₂ and T₃ are expressions</p> |  |
| <p style="text-align: center;">T₁.Name=T₂</p> <p>where Name refers to an attribute and T₁ and T₂ are expressions</p> |  |
| <p style="text-align: center;"><u>super</u>.Name=T</p> <p>where Name refers to an attribute and T is an expression</p> |  |

5.8.2.1 General example:

The following example provides a more comprehensive example of the translation of expressions:

```
x.b.a = Func(Z);
```

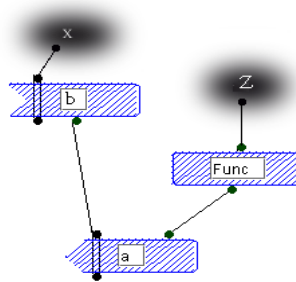


Figure 5-5: A general example

5.8.3 Bodies of methods and control structures

After the preprocessing steps, the body of each Java method, and the body of each control structure consists of a sequence of statements each of which is one of the following:

- Void expression
- Assignment expression
- Declaration
- Control structure

If a method has a throws clause, it will have a conditional statement at the end of its body, in which exceptions of the types specified in the clause are thrown. After preprocessing, these will be the only throw statements in the method. The translation of Java expressions to JGraph has been discussed above in general, and therefore, void expressions have been dealt with. The translation of assignment expressions has been covered in section 5.8.2. Declarations produce no corresponding JGraph structures. The translation of control structures will be covered in section 5.9. Hence, it remains in this section to describe how a sequence of statements is translated.

We will describe the translation of a sequence of statements in a recursive fashion. First, the empty sequence of statements produces no JGraph structures. Now consider a non-empty sequence $S ; Y$ where S is the first statement and Y is the remainder of the sequence. Figure 5-6 shows the corresponding JGraph structure. Each connection indicates that a value produced in the structure represented by blob S is used in the structure represented by blob Y . Note that because of preprocessing step 5.6.12 above, there are no multiple assignments in the method, so the root at the tail of the connection is uniquely defined.

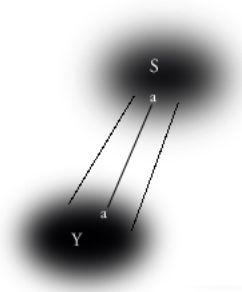


Figure 5-6: A general example of translating statements

5.8.4 Method

The following example illustrates the translation to JGraph of Java methods that are not void. After the preprocessing described in section 5.6.4 for method returns and the preprocessing steps described in section 5.6.10 for try-catch-finally statement, the method has the following form.

```

public int MyMethod(int param1, String param2) throws E
{
    Throwable v;
    Throwable v1;
    .....
    <body>
    v1 = COPY(v);

```

```

if (v1 instanceof E)
{
    throw (E) v1;
} else
{
}

return ((IRetExcep_int) v).RetResult;
}

```

The corresponding JGraph is as follows::

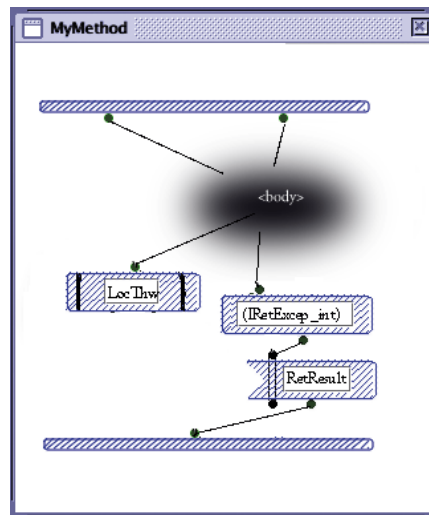


Figure 5-7: JGraph method obtained from a non-void Java method that throws exceptions

The roots on the input bar correspond to the parameters of the method. The terminal on the output bar corresponds to the return value. The terminal on the local operation `LocThw` corresponds to `v`. Figure 5-8 depicts the cases of the local operation `LocThw` resulting from translating the conditional statement in the above code as described in section 5.9.1 below. The root on the input bars of the cases corresponds to `v1`.

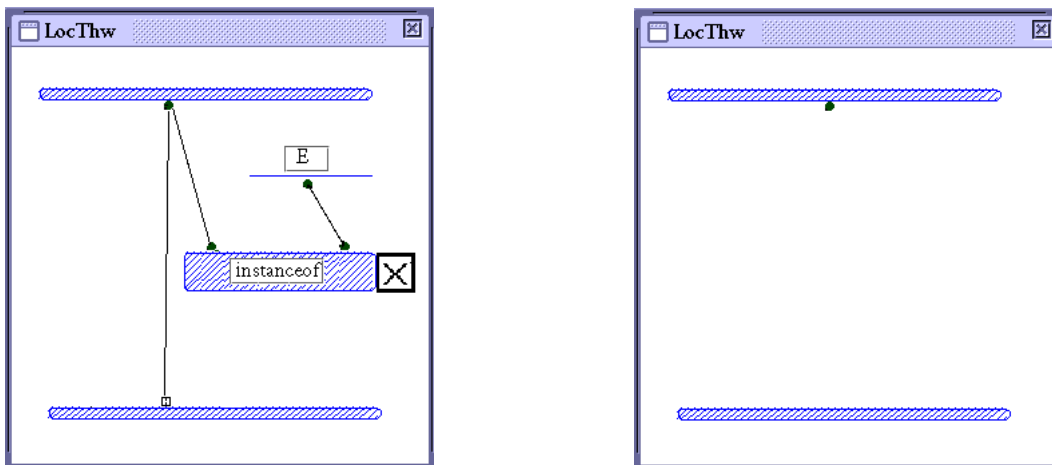


Figure 5-8: The cases of local methods of MyMethod

The following example illustrates the translation to JGraph of void Java methods. After the preprocessing described in section 5.6.4 for method returns and the preprocessing steps described in section 5.6.10 for try-catch-finally statements, the method has the following form.

```

public void MyMethod(int param1, String param2) throws E
{
    Throwable v;
    Throwable v1;
    .....
    <body>
    v1 = COPY(v);
    if(v1 != null)
    {
        if (v1 instanceof E) throw (E) v1;
        else{}
    }
}

```

```

    }else{}
}

```

The corresponding JGraph is as follows:

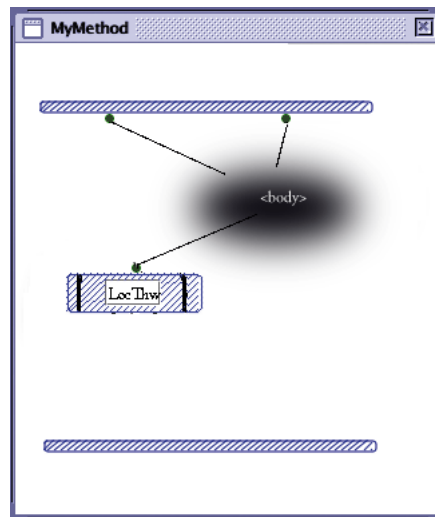


Figure 5-9: JGraph method obtained from a void Java method that throws exceptions

Note that in the examples above, if the method does not throw exceptions to its caller, that is, it is not declared with a throws clause, the local operation LocThw would be omitted.

Figure 5-10 depicts the cases of the local operation LocThw resulting from translating the conditional statement in the above code as described in section 5.9.1 below. The root on the input bars of both cases corresponds to v_1 .

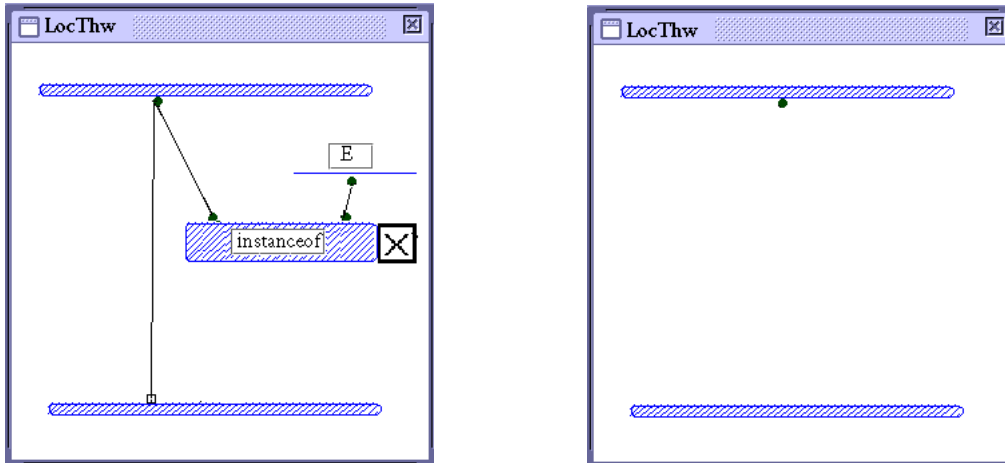


Figure 5-10: The cases of local methods of MyMethod

5.9 Control structures

5.9.1 If statements

Consider the following conditional statement and associated value passing statements preceding it, resulting from preprocessing step 5.6.11.

```

x1 = COPY(x);
y1 = COPY(y);
z1 = COPY(z);
if(z1)
{
    <body1>
    u = COPY(u1);
    v = COPY(v1);
}
else
{
    <body2>

```



```

u = COPY(u2);
v = COPY(v2);
}

```

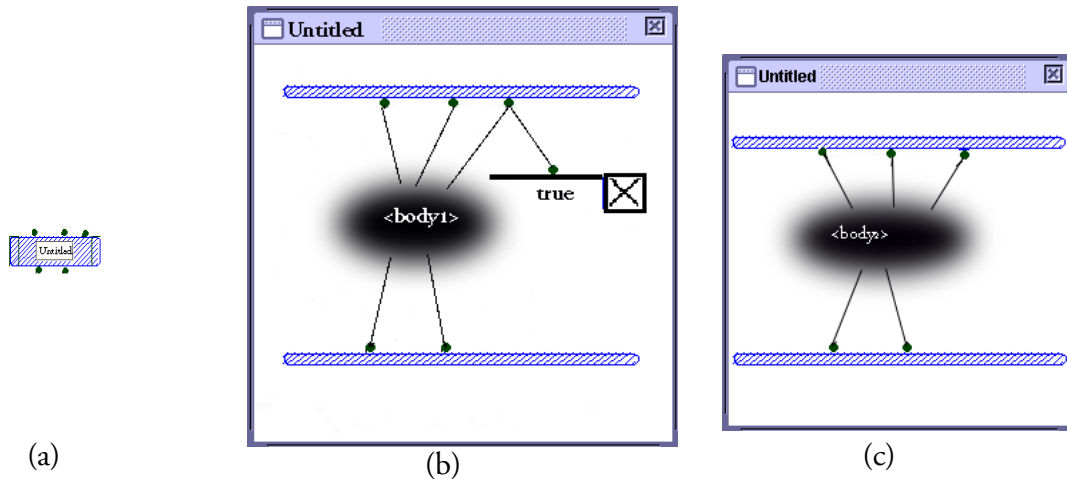


Figure 5-11: Translation of if-else structure

Figure 5-11 above shows that the if statement is translated into a local operation that consists of two normal cases. The terminals on the local operation correspond to the variables x , y , z respectively; the roots on the local operation correspond to the variables u and v , respectively; the roots on the input bars correspond to the variables x_1 , y_1 and z_1 respectively; the terminals on the output bar of the first case correspond to the variables u_1 and v_1 , respectively; and the terminals on the output bar of the second case correspond to the variables u_2 and v_2 , respectively. The first case contains JGraph structures obtained by translating $\langle \text{body}_1 \rangle$, together with a match operation that tests the condition of the if-then-else, reduced in step 5.6.7 to a single variable. The second case contains the JGraph structures obtained by translating $\langle \text{body}_2 \rangle$. Note that the output bar terminals in each case are guaranteed to be connected into the blobs since the variables corresponding to the output bar terminals are those which are assigned values that are used outside the conditional structure, and are identified by the value passing statements at the ends of the if and else bodies.

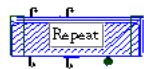
5.9.2 While statements

Consider the following while loop statement and associated value passing statements preceding it, resulting from preprocessing step 5.6.11.

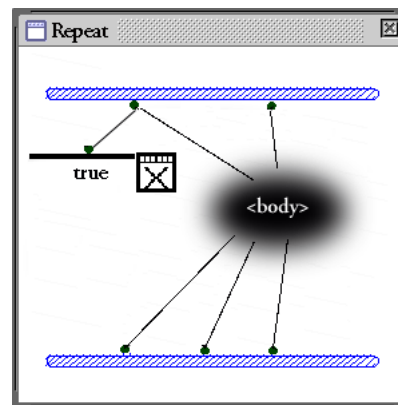
```

x1 = COPY(x); y1 = COPY(y);
while(x1)
{
  <body>
  LOOP(x1, x2); x3 = COPY(x2);
  LOOP(y1, y2); y3 = COPY(y2);
  z3 = COPY(z2);
}

```



(a)



(b)

Figure 5-12: Translation of while loop

The while loop is translated into a JGraph repeat operation, shown in Figure 5-12 (a), which has one case, depicted in Figure 5-12 (b). The terminals of the operation correspond to x , y ; the roots of the operation correspond to x_3 , y_3 and z_3 respectively; the roots on the input bar correspond to x_1 and y_1 respectively, the terminals on the output bar correspond to x_2 , y_2 and z_2 respectively.

Note that after preprocessing the only loops remaining in the Java code are while loops.

5.9.3 Try-catch structure

Consider the following try-catch statement and associated value passing statements preceding it, resulting from preprocessing step 5.6.11.

```

x1 = COPY(x);
u1 = COPY(u);
z1 = COPY(z);

try
{
    u2 = A;
    v1 = null;
    v = COPY(v1);
    u3 = COPY(u2);
}catch(Throwable e)
{
    v = COPY(e);
    u3 = COPY(u1);
}

```

We assume that in the code above, the variables x_1 and z_1 are the only variables occurring in A .

Note that the only try-catch structures that remain after preprocessing are those resulting from preprocessing step 5.6.10, so the original body of the try clause can only contain either an assignment or a void expression. If the try body contains a void expression, then the $u_1 = \text{COPY}(u)$, $u_3 = \text{COPY}(u_2)$, and $u_3 = \text{COPY}(u_1)$ statements above would not be present.

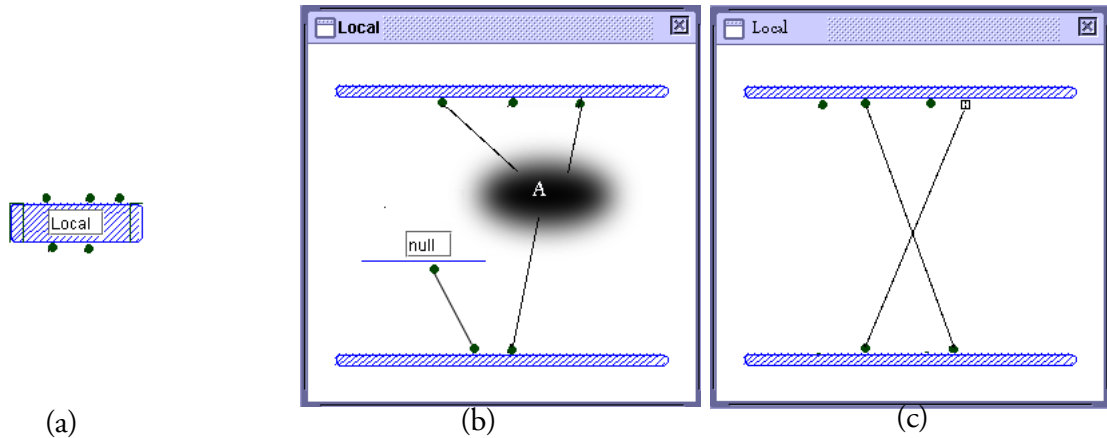


Figure 5-13: Translation of try-catch structure

Figure 5-13 (a) depicts the JGraph local resulting from the Java try-catch statement, consisting of the try case in (b) and catch case in (c). The terminals of the local operation correspond to the variables x , y and z respectively; the roots of the local operation correspond to the variables v and u respectively. The roots of the input bar correspond to the variables x_1 , y_1 and z_1 respectively; the terminals of the output bar on Figure 5-13(b) correspond to v_1 and u_1 respectively; and the terminals of the output bar on Figure 5-13(c) correspond to e and y_1 respectively.

<Dr. cox, since the next section is pretty much stand alone, I will make sure the JGraph code is correct regarding the preprocessing steps in this weekend>

5.10 A comprehensive example

In this section, we illustrate the translation process on an example, a method that bubblesorts an integer array, which demonstrates many of the features of the transformations we have described.

```
void BubbleSort(int [] a)
{
    int i, j, n, tmp;
    n = a.length;
```

```
for(i = 0; i < n-1; i++)  
{  
    for(j = 0; j < n-1-i; j++)  
        if(a[ j + 1] < a[j])  
            {  
                tmp = a[j];  
                a[j] = a[j+1];  
                a[j+1] = tmp;  
            }  
    }  
}
```

The JGraph program corresponding to the program above is shown in Figure 5-14. For simplicity, we will not present the preprocessed code.

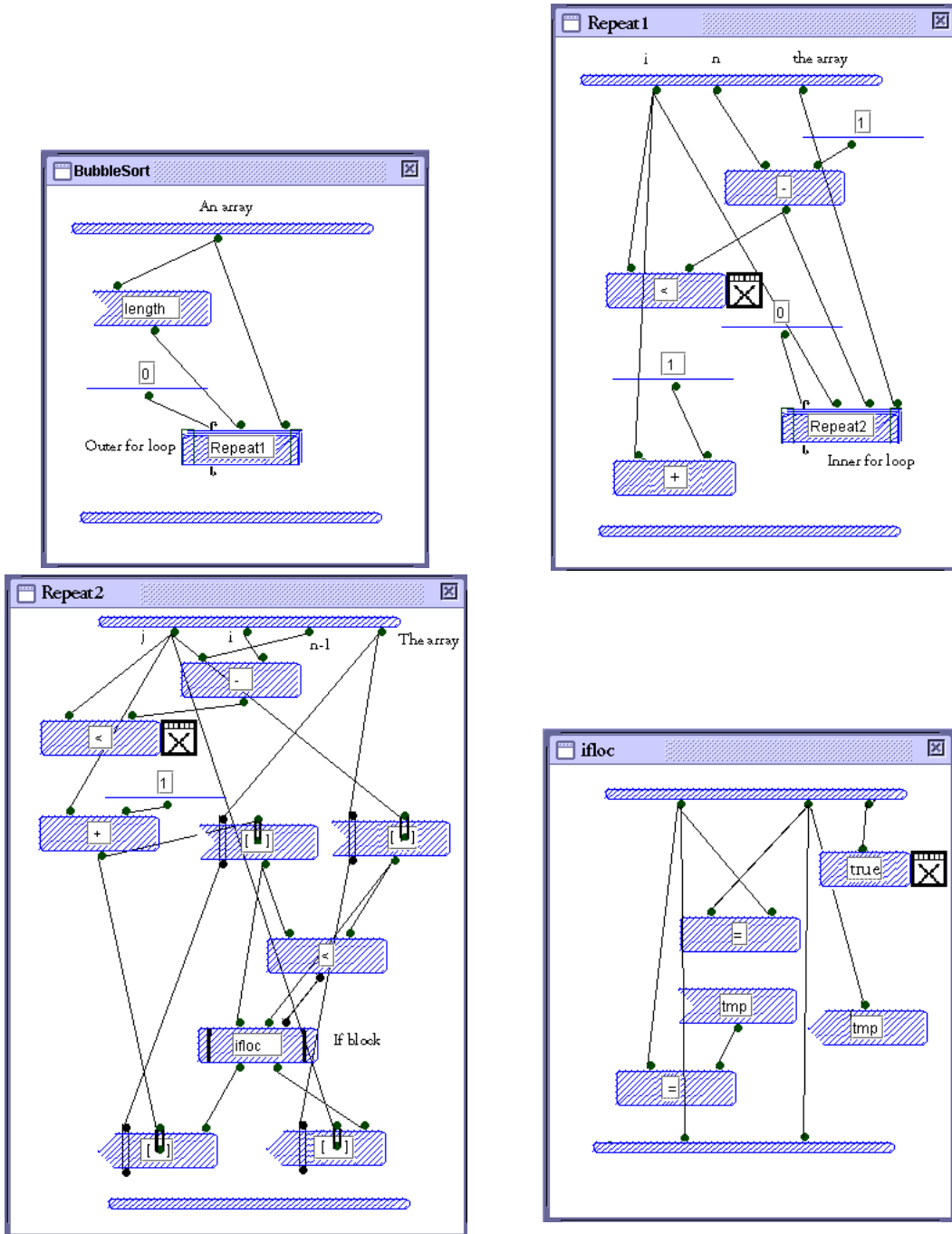


Figure 5-14: Bubble

5.11 Conclusion

In this chapter we have discussed the process of translating from Java to JGraph. It would be difficult in any reasonable amount of space to give examples that cover all combinations of features. We hope, however, that the examples we have provided will help the reader understand some of the more subtle aspects.

In the next chapter, we will conclude our discussions by informally comparing the features of Java and JGraph programs generated by the two translations.

6

Conclusions and Future Work

6.1 Introduction

Although visual programming has been extensively studied for twenty years or more and has been shown to have substantial benefits, it has not been adopted to any great extent by the industrial software development community. The work reported here is part of a larger project, the goal of which is to build a general purpose visual programming environment that adheres to the Java standard. In earlier work, a visual language JGraph was designed, and as background to the work reported here, a prototype has been implemented.

In order for a Java-compatible visual language to be part of mainstream software engineering, various tools will be required. For example, a programmer needs to be able to move easily between textual and visual representations of a program. We have conducted a preliminary investigation of the problem of translating between JGraph and Java.

In order to provide a firm foundation for describing a translation from JGraph to Java, in Chapter 3 we presented a formal definition of JGraph syntax, and gave examples to illustrate its connection to the visual representation. In Chapter 4, we defined a function that maps correct JGraph programs, expressed in the notation of Chapter 3 into equivalent Java, again illustrated by examples.

In Chapter 5 we address the problem of translating Java to JGraph. Because of JGraph's data flow features, in order to turn Java programs, which are typical control flow programs, into data flow pro-

grams, we introduced several preprocessing steps before the final translation into JGraph. These steps are illustrated with a series of examples.

In this chapter, we compare Java and JGraph for both similarities and differences, and how these have affected the two translations we have described. We also provide an assessment of our results in terms of how well they achieve the goal of allowing a programmer to easily move between visual and textual programming. Finally, we will suggest some possible future directions for this work.

6.2 Comparison of Java and JGraph

In this section we will ignore the obvious major difference between JGraph and Java, namely, that one is visual and the other textual. Although some of the more substantial structural differences we will discuss below are due to this difference in the mode of representation, it is the structural differences that concern us, rather than the reason that they arise.

First we note the ways in which the two languages are similar. In Java and JGraph, the concepts of package and class are identical. Classes and interfaces in both languages have the same qualifiers in the same combinations; attributes have the same qualifiers. Items can be imported from other packages.

The concept of method is also identical in both Java and JGraph. A method must have a name and a parameter list which may be empty. If the method is not the constructor of a class, it must have a return type which may be void. For Java and JGraph, a method can be declared with the same qualifiers in the same combinations.

Both Java and JGraph provide exceptions, which can be thrown anywhere, and can be handled by a try-catch-finally structure. In both languages, an unhandled exception must be explicitly thrown by the method in which the throw occurs. To this extent, the two languages provide the same exception throwing and handling capabilities via structures that are essentially similar. However, because of the interaction of exception handling and data flow in JGraph, there are some significant differences as well, which we will discuss later in this section.

As we know, JGraph is a data flow programming language and Java is control flow. In data flow, execution of a program is primarily driven by the flow of data through a network of processing elements, together with superimposed control structures for conditional execution, looping and so forth. In contrast, in a control flow language, execution is driven by stepping through a sequence of statements. The superimposed control in JGraph is achieved by enclosing diagrams in cases, sequences of which make up the bodies of methods and control structures. These properties of JGraph have two important consequences. First, each node, which corresponds to a variable in a textual language, can be assigned a value only once; and second, the scope of a node is limited to the case in which it occurs, so the only values that come out of a case are those that are passed through the output bar, when or if execution of the case completes. On the other hand, a variable in Java can be assigned a value multiple times and its scope is the block in which it is declared, including inside nested statements. Although these differences are apparently minor, they lead to major structural differences between equivalent programs in the two languages.

In Java, a method body does not have to execute to completion since a return statement can appear anywhere. In JGraph, however, one case of a method body must be executed to completion.

In JGraph, conditional execution is accomplished by sequence of cases, executed in order until one finishes. The computing of conditions is mixed with other computations in each case, and control can jump from anywhere in a case to the next case. In contrast, in Java conditions and computations are separate, so once the commitment has been made to execute either the then or the else part of an if-then-else, it will be executed to the end, barring exception throwing or returning.

In JGraph, iteration of execution is handled by the repeat operation. The computing of conditions is mixed with other computations in cases of a repeat, so if evaluating a condition causes a terminate control to fire, iteration stops immediately without finishing execution of the case. A repeat can have array inputs, indicating that successive iterations apply to successive array elements. Array inputs also provide loop control since iteration stops if an array input is exhausted. In Java, there are several forms of loop statement such as while, do-while and for. Conditions and computations are separate, so once

the commitment has been made to execute loop body, it will be executed to the end, unless a return statement is encountered or an exception thrown.

In JGraph there are two ways an exception can be thrown in a case, by a method call or by an exception terminal on the output bar of the case. If exceptions are thrown in a method call, execution of the case is terminated so all results computed in the case will be lost. If an exception is thrown at the output bar, however, values of output bar terminals are available to the associated finally case. In Java, however, exceptions can be thrown anywhere in a try clause, and any value computed for a variable is available to the associated catch and finally clauses, provided they are within the scope of the variable.

Because JGraph is data flow, and is expressed visually, there is no strict linear order of execution imposed by the way a program is drawn, unless the programmer forces a particular execution order using synchronos. In Java there is a strict linear order imposed by the order of statements, even though it may be possible to reorder statements without changing the meaning of the program. This difference between the two languages does not result in any significant structural differences between equivalent programs.

6.3 Characteristics of Java generated from JGraph and vice versa.

In JGraph, changes in the normal progression of execution through a data flow diagram are accomplished by controls, which cause abrupt termination of execution and transfer of control elsewhere. To mimic this in Java, we have had to resort to exception throwing. Hence the generated Java code relies heavily on the exception mechanism for ordinary control flow, and contains the special extra classes needed to accomplish this.

A Java program generated from JGraph will contain a large number of variables resulting from single assignment and the fact that in JGraph variables are local to cases.

During translation of a Java program to JGraph, try-catch-finally statements are largely eliminated and replaced by conditionals. Some try-catch structures are reintroduced to deal with exceptions in a

very localised fashion. The end result is that in the JGraph code, there will be no finally cases, and each catch case simply outputs the caught exception. Any normal case that may throw an exception contains at least one operation that calls a method that may throw an exception, but has no exception root on its output bar. Finally, the only place an exception terminal may occur is on the output bar of the case of a method. Note that every method will have exactly one case.

The generated JGraph will include exception classes declared for holding the return values of methods, the exception classes will be imported to JGraph too as additional classes. The number of exception classes could be big if a project has many methods with many return types.

There are various other trivial characteristics of a JGraph program generated from Java. For example, the conditions of conditional statements and loops are just variables, and the repeat operations do not have any array inputs.

6.4 Assessment of the translations

Although we have solved the problems of transforming JGraph programs into Java and vice versa in the previous chapters, the translations are not just inverses of each other. We have not shown an example of applying the two translations end-to-end, however, it should be obvious to the reader that doing so will produce code that is quite different from the original, regardless of whether we start with a Java program or a JGraph program. An example of this non-reversibility is as follows.

When a JGraph program is translated to Java, a sequence of cases in the original program corresponds to nested try-catch-finally statements in the resulting Java program. During the reverse translation, these nested try-catch-finally statements are replaced by nested if-then statements, each of which becomes a two-case local operation. If the original sequence consisted of more than two cases, then it will not be the same as the final JGraph obtained after two translations, since it has case sequences of length at most two. Even if the sequence of cases in the original JGraph had only two cases, translating to Java and back does not produce the same code, as the following example shows. Note that for

simplicity we have not included the value-passing statements generated in step 5.6.11 in the example code.

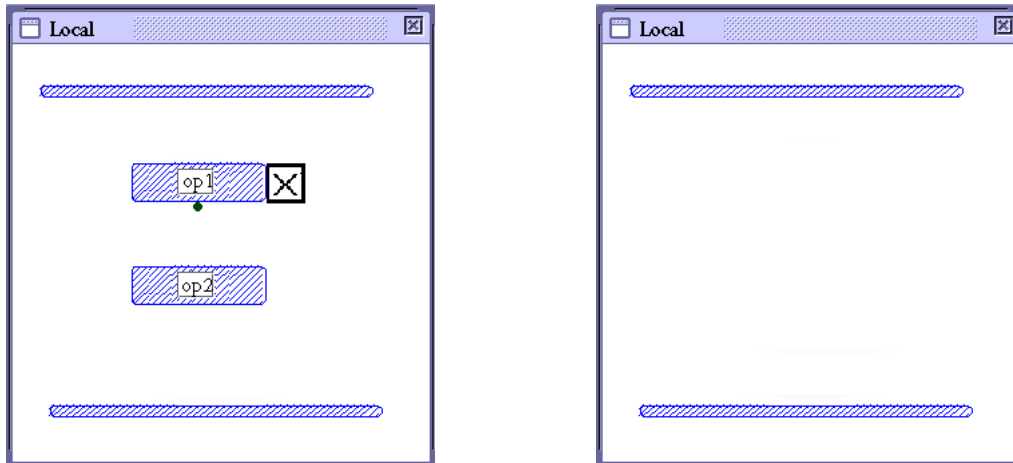


Figure 6-1: The cases of a local operation

Figure 6-1 shows the case windows of a two-case local operation Local. The Java code resulting from translating Local is as follows:

```

{
    boolean bl;
    <other declarations>
    try
    {
        bl = op1();
        if (!bl) throw new ICaseException();
        op2();
    }catch(ICaseException e)
    {

```

```

    }
}

```

which will be transformed, by applying preprocessing steps as described in Chapter 5, into the code below before it is translated back into JGraph again:

```

<declarations>
bl = op1();
if (!bl)
{
    v = new ICaseException();
} else {}
if(v == null)
{
    op2();
} else {}
if(v != null)
{
    if(v instanceof ICaseException)
    {} else {}
} else {}

```

We can see that the original two-case local has turned into three conditional statements, each of which will become a two-case local after translation into JGraph.

Because of the major differences between JGraph and Java discussed in section 6.2 above, both translations introduce special mechanisms and structures, resulting in code that would not be written by a programmer proficient in the target programming language. For example, a Java programmer is unlikely to write code which embodies the single assignment rule of data flow and achieves condi-

tional execution by throwing exceptions. Similarly, a proficient JGraph programmer is unlikely to create a program in which exception throwing and handling is replaced by deeply nested conditional statements.

One question which arises naturally from the above discussion is the following. If we translate back and forth several times, does the process iterate towards some fixed point? The answer to the question is clearly negative, as illustrated by the above example, where each round of translations multiplies the number of two-case locals by three. The size of the code therefore increases at each step so no fixed point exists.

6.5 Future work

The goal of the JGraph project, of which the research reported here is a part, is to bring the benefits of visual programming to industrial software development. The success of this project depends on adhering to a standard. Although JGraph is compatible with Java, for it to be acceptable to professional developers, some mechanism is necessary to allow easy transition back and forth between visual and textual representations of a program.

As background to the research reported here I have implemented a prototype JGraph editor, and experimented with translating between between Java and JGraph. Based on that experience, I have defined two translations and assessed their usefulness in terms of the overall goals of the project.

As discussed in section 6.3, the two translations are not inverses of each other, and although the code produced by each is correct, it is not of the quality a proficient programmer would produce. An important question to be addressed, therefore, is whether reversible transformations, producing better quality code, can be devised. This may require changes to the design of JGraph. On the implementation side, the prototype should be extended to include features of JGraph not currently supported, and to serve as a test bed for improved translations.

References

- [1] Ambler. A. L & Burnett M. M., *Influence of Visual Technology on the Evolution of Language Environments*, ieeec, Vol 22, Issue 10, (Oct1989), pp. 9-22.
- [2] Brassard G. & Bratley P., *Fundamentals of Algorithmics*, Prentice Hall, (1996).
- [3] Booch G., Jacobson I., & Rumbaugh J., *The Unified Modeling Language User Guide*, Addison Wesley Professional, (1999).
- [4] Booch G., *Object Oriented Design: with applications*, The Benjamin/Cummings Publishing Company, Inc., (1991).
- [5] Burnett M. M. , Visual Programming. *Encyclopedia of Electrical and Electronics Engineering* (John G. Webster, ed.), John Wiley & Sons Inc., New York, (1999).
- [6] Carrel-Billiard M & Akerley J., *Programming with VisualAge for Java*, IBM Redbook, (1998).
- [7] Golin E. J. & Reiss S. P., *The Specification of Visual Language Syntax*, IEEE Workshop on Visual Languages, (1990), pp.105-110.
- [8] Coskun N. & Sessions R., Class Objects in SOM, *IBM Personal Systems Developer* (Summer 1992): 67-77.
- [9] Cox P. T., Giles F.R. & Pietrzykowski T. (1989). Prograph: A Step toward Liberating the Programmer from Textual Conditioning, Proceedings of the 1989 IEEE Workshop in Visual Languages, pp.150-156.
- [10] Cox P. T. & Song B., *A formal Model for Component-based Software*, IEEE Symposium on Visual/Multimedia Approaches to Programming and Software Engineering, Stresa, Italy, (2001).

- [11] Green T. R. G & Petre M., *Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework*, Journal of Visual Languages and Computing, (1996).
- [12] Hills D., *Visual Languages and Computing Survey: Data Flow Visual Programming Languages*, Journal of Visual Languages and Computing v3 no.1, Academic Press, (1992).
- [13] McIntyre D. W. and Glinert E. P., *Visual Tools for Generating Iconic Programming Environments*, (1992).
- [14] Muller P. A., *Instant UML*, Wrox Press, (1997).
- [15] Myers B. A., *Taxonomies of Visual Programming and Program Visualization*, jvlc, (1990), pp.97-123.
- [16] Microsoft Company, *Visual Studio .Net*, <http://msdn.microsoft.com/vstudio>, accessed 2002.
- [17] Borland Software Corporation, *C++ Builder: Revolutionary C++ for Internet*, <http://www.borland.com/cbuilder>, accessed July 2002.
- [18] Borland Software Corporation, *Delphi: Next-generation e-business development*, <http://www.borland.com/delphi>, accessed July 2002.
- [19] Metrowerks, a Motorola company, *CodeWarrior Development Tools*, <http://www.metrowerks.com/mw/default.htm>, accessed July 2002.
- [20] Risley, *JGraph: A Java Compatible Visual Language*. MCS Thesis, Faculty of Computer Science, Dalhousie University, (2000).
- [21] Rogerson D., *Inside COM*, Microsoft Press, (1996).
- [22] Rumbaugh J., Jacobson I., & Booch G., *The Unified Modeling Language Reference Manual*, Addison Wesley Professional, (1999).

- [23] Schmucker K.J., *Rapid Prototyping using visual programming tools*, Tutorial Notes, CHI'96, Vancouver, (1996).
- [24] Schmidt D. *Overview of CORBA*, <http://www.cs.wustl.edu/~schmidt/corba-overview.html>.
- [25] Sun Microsystems, *JavaBeans Specification*, <http://www.javasoft.com/beans/spec.html>, (1997).
- [26] Yang S. & Burnett M.M., *From Concrete forms to Generalized Abstractions through Perspective-Oriented Analysis Of Logical Relationships*. Proceedings 1994 IEEE Symposium on Visual Languages.
- [27] Wayner P., *Java Beans for Real Programmers*(For Real Programmers Series), Morgan Kaufmann Publishers, (June 1998).
- [28] Whitley K. N., *Visual programming languages and the empirical evidence for and against*, Journal of Visual Languages and Computing, Vol. 8, No. 1, (Feb 1997), pp. 109-142.
- [29] Whitley K. N and Blackwell A. F., *Visual Programming in the Wild: A Survey of LabVIEW Programmers*, Journal of Visual Languages and Computing, Vol. 12, No. 4, (Aug 2001), pp. 435-472.

APPENDIX A

JGraph Users Manual

Instruction :

JGraph is a visual programming language. This document describes how to use the JGraph prototype. It assumes that the reader is already familiar with the JGraph language. By using the prototype of JGraph, users can do following things:

1. Build and edit JGraph programs.
2. Generate Java code from JGraph programs.
3. Import Java program into JGraph.

We will describe how to work with JGraph prototype in the following chapters. In this document, the material will be presented in the order in which the user encounters the various GUI items.

1. Summary of JGraph

Figure 1-1 Mainframe of JGraph

When JGraph starts up, the Main Frame appears. This frame contains all other JGraph windows, and provides top-level control of the JGraph application via the seven menus shown in Figure 1-1.

There are seven menus in the menu bar, providing the following functions.

The File menu contains items that create a new project, open an existing project, import java program into JGraph and save the current JGraph program.



Figure 1-2 File menu

The menu Class provides items for changing the characteristics of the selected class. At one time, there could be only one class selected. The selected class in a class window can be set to “public”, “abstract”, “final” or “alias” by selecting items in this menu.



Figure 1-3 Class menu

The menu Method is used for setting the characteristics of the selected method. That is, selecting different items sets a method to “public”, “private”, “protected” or “abstract”. At one time, there could be only one method selected.



Figure 1-4 Method menu

The menu **Opers** is used for setting the type of the selected operation in a case window. At one time, there is only one operation selected. The available operation types are “Simple operation”, “Get operation”, “Set operation”, “Allocation operation”, “Local operation”, “Literal operation” and “Repetition operation”.

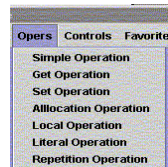


Figure 1-5 Opers menu

The **Controls** menu is used for adding controls to the selected operations in a case window and changing the type of selected roots or terminators. The available controls types are “Next case (false)”, “Next case (true)”, “Term False case”, “Term True case”, “Finish False case”, “Finish True case”. The available types of roots and terminators are: “Origination”, “Reference”, “Array”, “Loop”, “Enum”, “Finally” and “Thrown”.

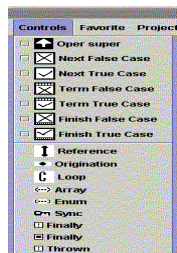


Figure 1-6 Controls menu

The menu Favorite provides users with some options for tailoring the environment to suit their preference. In the present implement, this menu simply allows the user to hide the project panel to reduce desktop clutter.

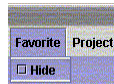


Figure 1-7 Favorite menu

The menu Project is responsible to generate Java source code and compile it.

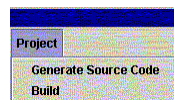


Figure 1-8 Project menu

2. Start working with a project.

There are several ways to start working with a project. The user can create a new project, open an existing project or create a project by importing a Java program into JGraph. These actions are performed by selecting the New, Open, and Import Java code Items respectively from the File menu.

To import java source code into JGraph, the user must create a list of all the Java files to be included in the imported project, and then select the list file in the Dialog box to open a project.

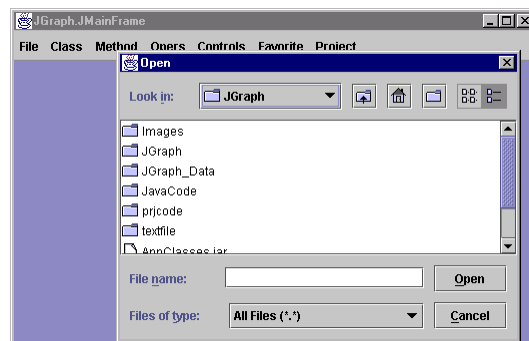
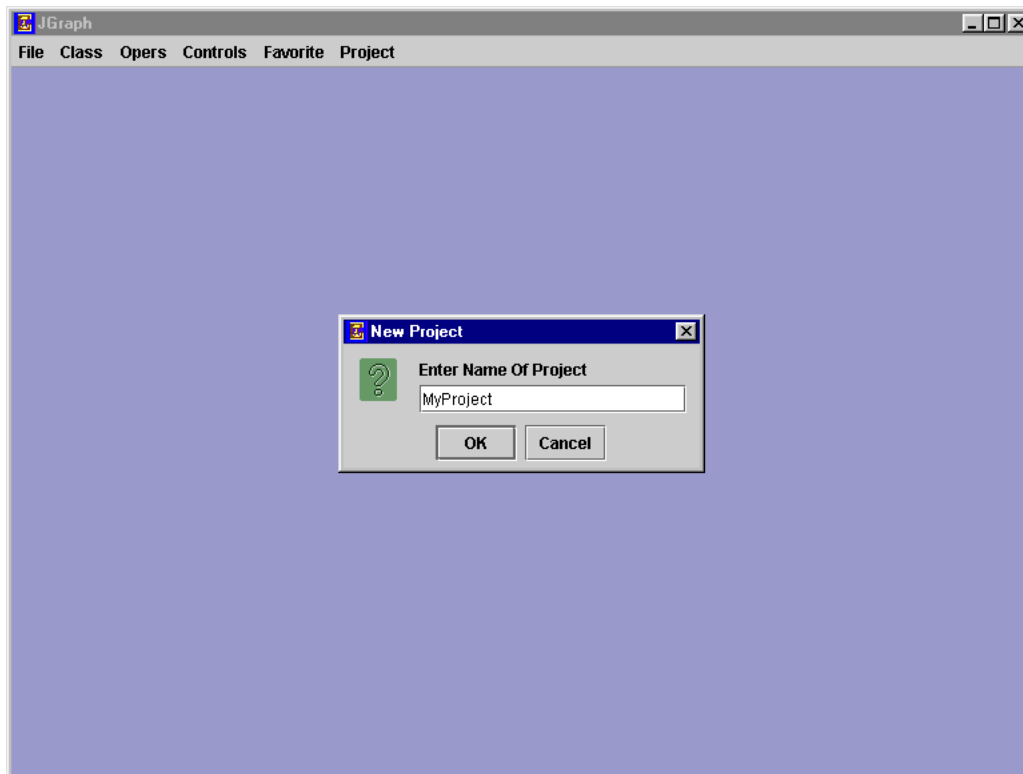


Figure 2-1 The dialog box for the user to select a project.

After a project is selected, a project panel will appear so that users can work on it. The project panel contains all the classes and the related methods in the project in a tree.

If users feel inconvenient because the panel take some place, they can hide that panel by going to Menu "Favorite" and select "Hide".

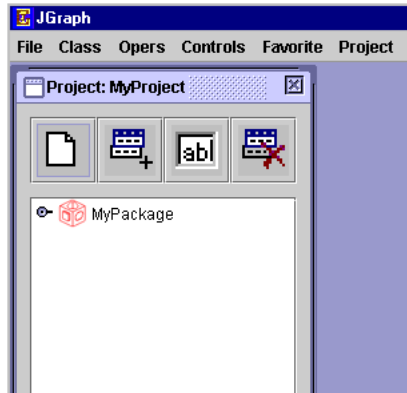


Figure 2-2 The project panel

Table 1:





| icon | description |
|---|--|
|  | Create a new package |
|  | |
|  | Change the name of the current package |
|  | Delete the current package. |

Table 2-1 The project buttons

The project panel appears after selected a project as shown in Figure 2-5. In this case, the project name is prjcode.

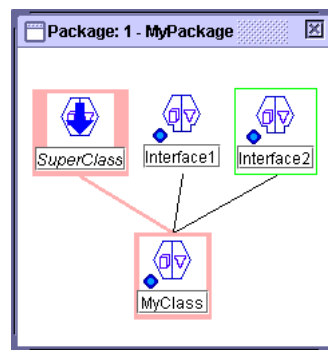
3 Edit classes.

Double click the mouse on the node “prjcode” in the project panel; an internal frame for editing classes will appear. Clicking the mouse anywhere in the frame will add a new class in the project and a class icon will be shown in the frame representing the class. Selecting the class icon and then pressing the “Delete” key will delete the class from the project. The class icon consists of two parts; double clicking the mouse on the right part of the icon will activate an internal frame via which the user can

edit the attributes and the packaged lists for the class. The user can also browse the list of methods associated to the class and modify some items such as method name and parameters in the frame.

Double clicking the mouse on the left part of the class icon will activate an internal frame in which the user can edit methods in the class.

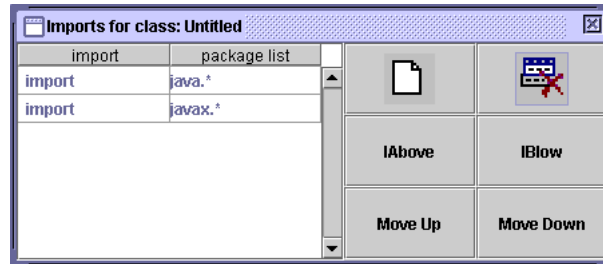
The relationship between the class icon and the internal frames mentioned above is shown as Figure 3-1:



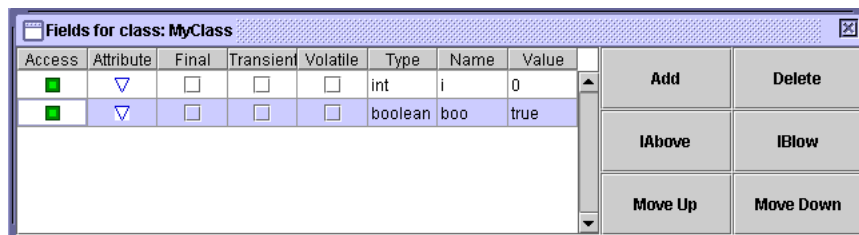
(a)

| | Access | abstract | static | final | synchronized | Returns | Name | Parameters | Throws |
|--|--------|--------------------------|--------------------------|--------------------------|--------------------------|---------|-----------|------------|--------|
| | ■ | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | void | MyMethod1 | void | |
| | ◆ | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | void | MyMethod2 | void | |

(b)



(c)



(d)

Figure 3-1

In order to set up inheritance between two classes, just select the class icon to denote the subclass, press and hold the “Alt” key, move the mouse to the super class icon and click on it, then release the “Alt” key. The inheritance relationship between the two classes will be done as shown in the graph below. JClass1 is the super class of Jclass2:

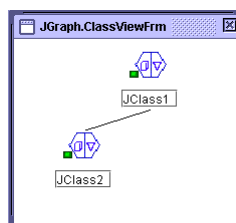


Figure 3-2

4. Edit methods.

Same way as editing classes above, the user can add a new method by clicking the mouse anywhere in the corresponding internal frame activated by double clicking the left part of a class icon, and delete it by selecting it and pressing the "Delete" key.

The Method icon consists of two parts, double click the mouse on the right part of the icon will activate an internal frame via which the user can edit the attributes and browse the list of parameters for the method as well. Double click the mouse on the left part of the method icon will activate an internal frame in which the user can edit cases in the method.

The relationship between the method icon and the internal frames mentioned above is shown in Figure 4-1:

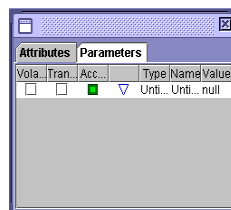
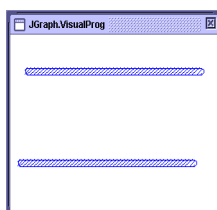
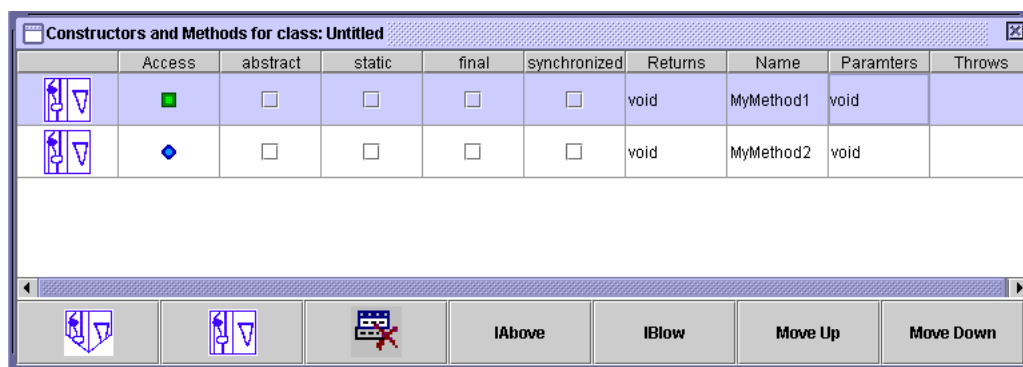


Figure 4-1

5. Edit cases.

There are two long bars in a case by default. One of them locates on the topside of the case window denoting the beginning of a case. Another bar locates at the bottom of the case window denoting the termination of a case. Of course, the user can drag and drop them anywhere off their default location in the window. Clicking on the lower side of the start bar will create the parameters for that case. The parameters are represented by small circles. The user can specify details of them by double clicking on the specific small circles. The return terminals are represented by small circles also.

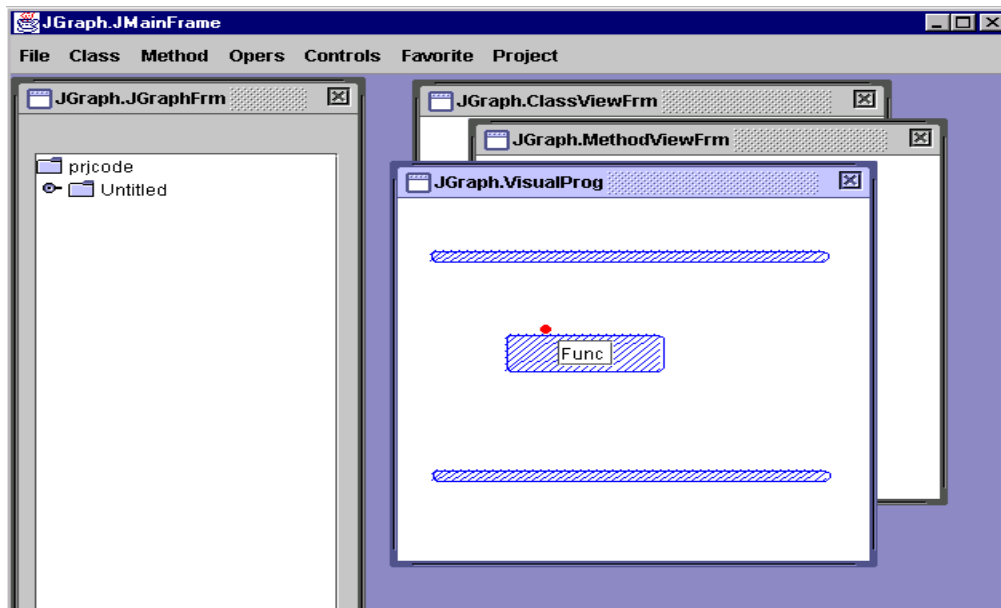


Figure 5-1

5.1 Specify the parameters.

Click the mouse on the lower side of the top bar will create a new circle denoting a input parameter. Double click the mouse on it will pop up a small dialog to specify the type and the name of the parameter.

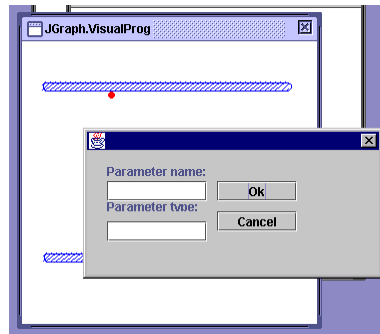


Figure 5-1-1

5.2 Edit operations

Clicking the mouse in the blank area of the case window cause to generate a new operation. Selecting the operation, going to the “Oper” menu and selecting an appropriate type to set the type of the operation. Clicking the mouse on the topside of the operation will create a terminal. Clicking the mouse on the lower side of the operation will create a root. Double clicking the mouse on the terminal or root will pop up a small dialog box as Figure 5-1-1 to set the terminal or the root respectively.

If the operation is a local method operation, double clicking the mouse on it will create a new case window within which the user can edit the local method.

If the operation is a local class operation, double clicking the mouse on the right side or the left side of the operation will activate internal frames as Figure 3.1 to edit a local class.

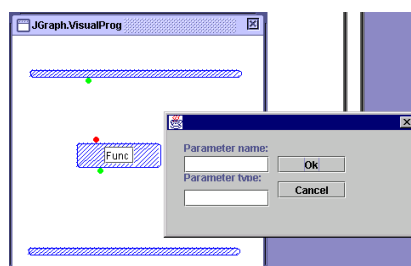


Figure 5-2-1

5.3 Edit controls

After selected an operation, going to “Control” menu and selecting a menu item will apply a corresponding control to the operation.

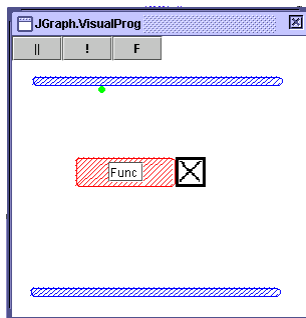


Figure 5-3-1

5.4 Connection between roots and terminals.

After selected a root or terminal, press and hold the “Alt” key and move the mouse to the terminal or root, it will set up the connection between the root and the terminal.

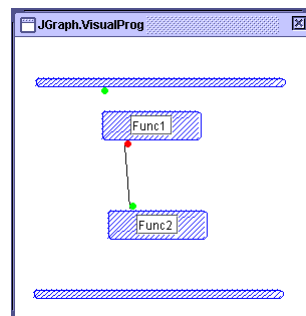


Figure 5-4-1

5.5 Synchronization.

After selected an operation, press and hold the “Shift” key and move the mouse to another operation, it will set up the synchronization between two operations.

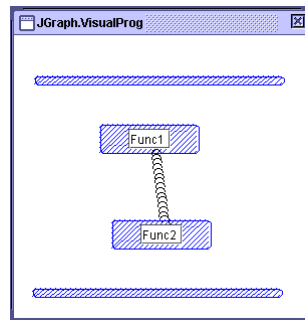
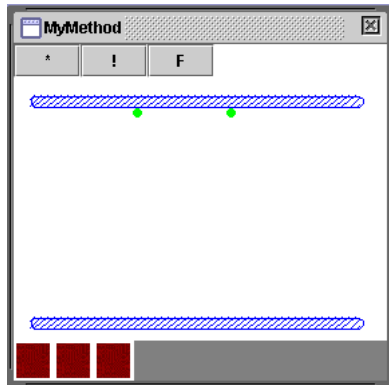


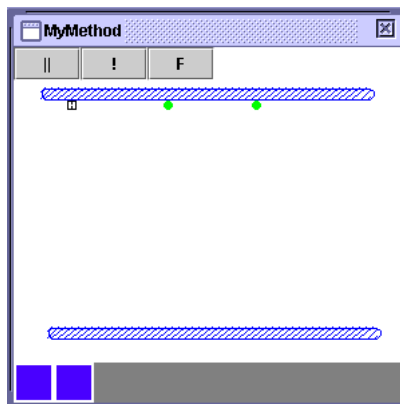
Figure 5-5-1

5.6 Navigate the cases.

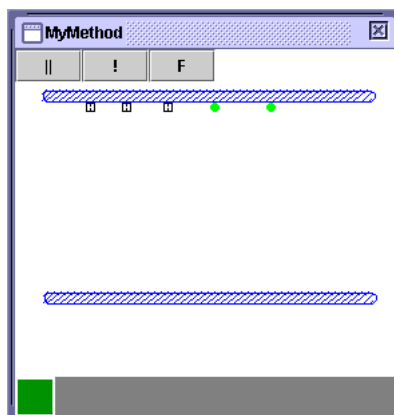
Move the mouse to the topside of the case window. A list of buttons will appear. Clicking the mouse on the left button will make a bar on the bottom of the case button visible. On the bar there list all the normal cases denoted by square boxes. Double click the mouse on the appropriate square box will activate the corresponding case window to edit, as Figure 5-6-1 (a) illustrates. Otherwise, click the middle button on the topside of the case window will make a bar on the bottom of the case button for catching exceptions visible. On the bottom bar there lists all the cases (denoted by square boxes as well) available for catching exceptions, as Figure 5-6-1 (b) illustrates. Double clicking the mouse on the appropriate box will activate the corresponding case window to edit. If clicking the right button on the topside of the case window, a bar on the bottom of the case will be made visible. There lists the only potential square box for the finally statement in Java, as Figure 5-6-1 (c) illustrates. Double clicking on it will activate the corresponding case window to edit. The user can also delete a case from the bottom. The user just need to select the corresponding square box and then press the “Delete” key.



(a)



(b)



(c)

Figure 5-6-1

6. Generate the Java source code.

It is quite simple to generate Java code from the JGraph program. It just need go to project menu and select “Generate source code” menu item. The correspondring Java source code will be generated in the directory where the project is opened from.

7. Build the generated Java program.

Go to the project menu and select “Build” menu item. The JGraph application will call Java Compile to compile the generated Java program.