NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

CODE MAINTENANCE AND DESIGN FOR A VISUAL PROGRAMMING LANGUAGE GRAPHICAL USER INTERFACE

by

Graham C. Pierson

September 2004

Thesis Advisor:	Mikhail Auguston
Second Reader:	Scott Coté

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** Sept 2004	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Code Maintenance and Design for a Visual Programming Language Graphical User Interface			**5. FUNDING NUMBERS**
6. AUTHOR(S) Graham Pierson			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE A**

13. ABSTRACT (maximum 200 words)

This work adds new functionality to an existing visual programming environment. It applies software maintenance techniques for use with the Java Language in a Microsoft Windows operating system environment. The previously undocumented application is intended to support programming with executable diagrams. This application has the potential to expand programming access to non-programmers, provide better software documentation and improve software maintainability. It is currently capable of supporting meta-programming tasks such as parsing and compiler building. The 11,184 legacy lines of code(LOC) were corrected, extended and documented to support future maintenance using an additional 957 LOC and changes to 45 LOC.

14. SUBJECT TERMS Code Maintenance, VPL, Visual Programming Language, Software Engineering, VisualRigal, Rigal, Graphical User Interface, Meta-programming, Code reading, notSerializableException, Software Design			**15. NUMBER OF PAGES** 179
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

CODE MAINTENANCE AND DESIGN FOR A VISUAL PROGRAMMING LANGUAGE GRAPHICAL USER INTERFACE

Graham C. Pierson
Major, United States Marine Corps
B.S., University of Puget Sound, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2004

Author: Graham Pierson

Approved by: Mikail Auguston
Thesis Advisor

Scott Coté
Second Reader

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This work adds new functionality to an existing visual programming environment. It applies software maintenance techniques for use with the Java Language in a Microsoft Windows operating system environment. The previously undocumented application is intended to support programming with executable diagrams. This application has the potential to expand programming access to non-programmers, provide better software documentation and improve software maintainability. It is currently capable of supporting meta-programming tasks such as parsing and compiler building. The 11,184 legacy lines of code(LOC) were corrected, extended and documented to support future maintenance using an additional 957 LOC and changes to 45 LOC.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

It is commonly acknowledged that even well commented code is not always easy to understand. An investment of time is required to understand even simple programming structure. As a result, supporting documentation has become part of deliverable software. Graphical techniques have been developed to better express program design and function. Universal Modeling Language can be used to provide additional information about program design at a glance. Visual programming offers a pictorial paradigm for computer programming. This technique combines graphical technique with formal language meaning to produce a type of executable diagram or model. Several past examples of visual languages include Sun Microsystem's Java Studio, open source Prograph and National Instrument's LabVIEW. It is likely the explosion of personal computer (PC) users after graphical user interface (GUI) introduction may have a parallel in visual programming language usage. As this technique matures toward mainstream acceptance, programming will become accessible to a large part of the population. Graphical techniques also have a well understood application for education and would likely support visual programming languages in classroom settings.

B. PURPOSE

This work is correction and extension of a GUI for a visual programming language, more specifically a visual meta-programming language[1], that provides visualization of control and data flow while introducing visualization of data structures. The task was to add new functionality and fix known errors in an existing application.

The greater purpose is support of basic research in visual programming languages by continuing the implementation VisualRigal (REE-gal), described below.

> [Visual Programming Languages are] stimulated by the following premises:
>
> 1. People, in general, prefer pictures over words.
> 2. Pictures are more powerful than words as a means of communication. They can convey more meaning in a more concise unit of expression.

3. Pictures do not have the language barriers that natural languages have. They are understood by people regardless of what language they speak.[2]

In practical terms this research supports the following:

- Greater access to programming for non-programmers, specifically application domain experts or customers
- Generation of visual documentation concurrent with programming, the diagrams are executable
- Improved communication between distant parts of the development team perhaps across primary languages
- More rapid understanding of program structure allowing focus on key portions of application

As an example, a naval flight officer who programs can assist directly in prototype development, improving the final products usefulness in combat and significantly reducing the development time required.[3]

C. SCOPE, METHODOLOGY AND ASSUMPTIONS

1. Scope

a. Understanding of Current GUI Design

This work examines the existing design of the VisualRigal program, which encompases 45 classes and approximately 10,000 lines of code. The goal is to gather enough information to portray the program design with UML to enable further maintenance and extension of the code.

b. Understanding of Current GUI Behavior

This work also examines high level behavior of the VisualRigal program. The goal is to document sufficient understanding of the program behavior to support future work and program use. Detailed program behavior is also documented for specific areas of correction and extension.

c. Application of the Java Swing Classes to Extend and Correct GUI Functionality

The work primary production is achieved through the application of Java classes and language to correct desired behavior and support new functionality. Because

the Java language and individual Java Virtual Machines are evolving, there is no expectation that this work represents a final solution. This belief is reinforced by the history of the VisualRigal program outlined at the beginning of Chapter III.

2. Methodology

a. Comment Current Code and Discover Design

The intention of this work is to trace program execution enough to correct and extend particular functionality. During this trace code comments will be developed that support Sun's Javadoc functionality. Additional comments have been developed to answer the questions, "Why was this done?" and "Why was this technique used?" While tracing execution, information discovered that contributes to an understanding of program structure and design has been translated into UML descriptions

b. Develop Test Plan

As an understanding of design and program requirements emerges a test plan will be developed that supports requirement fulfillment and likely fault detection.

c. Extend Functionality

Functionality has been corrected and extended on a priority bases. A successful change has been completed prior to coding additional changes. These changes are prioritized based on customer requirements and best return on resource investment. Best return is defined as greatest functionality increase for likely time invested.

d. Test with Progress

Testing has been done at the developer level while coding. Regression testing of test cases and automated suites was also utilized.

e. Develop User's Manual

A basic user's manual has been created to outline operation and document expected behavior of the program. It is included as an appendix.

f. Document Existing Design

The information collected starting with the first step of this work and throughout is consolidated in a UML representation of program design.

3. Assumptions

Understanding of UML is required for a detailed understanding of program design. Understanding of Java is required for a detailed understanding of code choices

and changes. However, a general understanding of the work is accessible to the average college graduate.

D. THESIS ORGANIZATION

1. Chapter I: Introduction

Provides a brief overview of the background, purpose and method of this work.

2. Chapter II: Background Information and Key Concept Review

Provides a brief history of visual programming and work leading to the current work.

3. Chapter III: Implementation of GUI

Provides an overview of code maintenance tools and techniques in a Java and MS Windows environment. Additionally, provides a detailed description of program design and this work's corrections and extensions of the program code as well as the implemented test plan.

4. Chapter IV: Conclusion

Examines changes required as a result of testing, an assessment of the program implementation, a review of the research questions and a discussion of likely future work with comments on priorities.

5. Chapter V: Appendixes

Copies of documentation to supplement understanding of this work, to include: original 1.0, commented original 1.1 and extended 2.0 application Java code, raw Parasoft JTest Reports and user's manual.

II. BACKGROUND INFORMATION AND KEY CONCEPT REVIEW

A. SURVEY OF VISUAL PROGRAMMING

1. History

The beginning of modern visual programming languages is credited to Goldstine and von Neumann's work with flow charts.[4] This familiar paradigm indicates the change of control as one follows the lines from node to node, and is more completely called control flow charts. This sort of visual language is classified as a control flow language.[5]

Further development has lead to a dataflow paradigm, popular in embedded systems especially for space and communication applications.[6] Here the flow of data is traced from node to node over the lines. If the node describes control decisions then a combination of control flow and data flow is attained.[5] This two dimensional view is the current state of the literature.

There is discussion of pure visual programming language, that is figures without symbols. In this classification the mixing of diagrammatic elements with symbols, like 2>4, constitutes a hybrid language.[7] This view is consistent with current discussion regarding the feasibility of model driven programming.[8] One criticism is that at some level coding is required. This does not have to be a disadvantage. If a technique provides general understanding at a glance and allows the reader to focus on those areas of interest quickly to determine perhaps conditional boundaries, it has achieved usefulness.

Spreadsheets are an example of a hybrid language.[5] There is information contained in the representation of row and columns but no one would consider using a spreadsheet without using the available symbols. Graphical representations should be used where they make sense.

GUI component libraries, filled with buttons and selection tools, appear to be moving toward standardization and are an example of a specialized hybrid language.[5]

Graphical techniques used to describe the Boolean expression above are cumbersome and cannot compete with the simplicity of basic mathematical expression.

2. Benefits

The increased accessibility of programming has already been discussed as a potential advantage for visual languages. However, the professional could benefit from the creation of accessible documentation at the same time they are building the program. If an application is built of executable diagrams, the visual code can help serve as a reference for future maintenance and current turnover on a project.

3. Risks

As with any language there is uncertainty concerning the relationship between what is coded and what is compiled. There is an implied contract that written code will execute with fidelity to written structure. Likewise, there is an expectation that the execution described in a diagram will perform consistent to the graphical representation. I call this expectation for equivalence between the graphical and logical representation the equivalence contract.

If visual programming becomes mainstream, how will vision impaired programmers work? Graphical methods are the darling of the majority. Are there interface options to support the vision minority? Will these options support the click and drag GUI paradigm? This work does not address these questions but points out the implications here.

B. VISUAL META-PROGRAMMING LANGUAGE

1. Intended Use

A graphical extension of the University of Latvia's RIGAL, a textual meta-programming language, VisualRigal is a domain specific language with graphical techniques that can be applied to a general programming visual language. It is considered a visual meta-programming language. That is, it is a programming language for manipulating and describing programming languages. Common applications for this type of language are compilers and parsers.[1]

It is this application of the language that provides a look at VisualRigal's new paradigm, visualization of data structure. VisualRigal provides control flow and data flow, as is now common, but introduces a simple graphical syntax for describing atomic data, lists and tree structures.

VisualRigal is composed of a GUI, a parser and a compiler. The compiler and parser are already fully implemented. The current work is the maintenance and extension of the GUI which produces a text file interface for the parsers consumption. Parser output is compiled and is then executable.

2. Icons

The language is fully described in the reference but a low level description of the major icons follow.

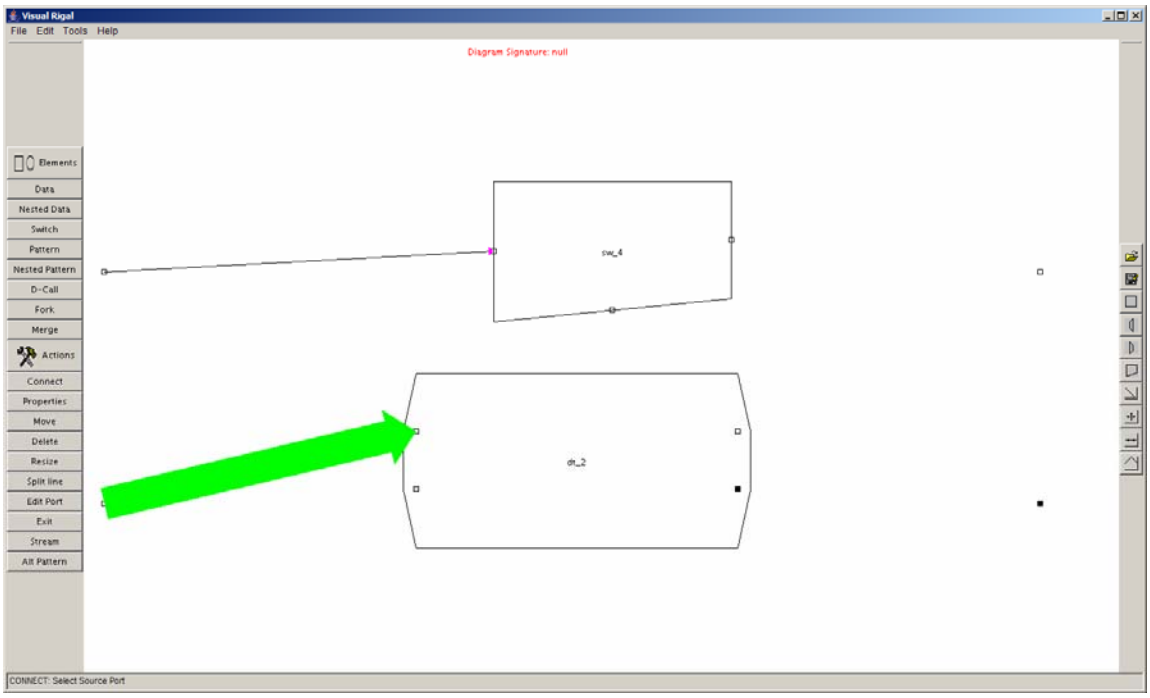


Figure 1. Sample Application Screen with Some Elements Present

a. Port

A junction box for data. Ports on the left of an object are input ports that is data enters the icon through that port. Ports on the right of an object are output ports. This convention of in or out is muddled when referencing connectors since data flows out of a diagram "input" port and into the connector. The filled in port on the lower right of an icon is a fail port whose use is optional.

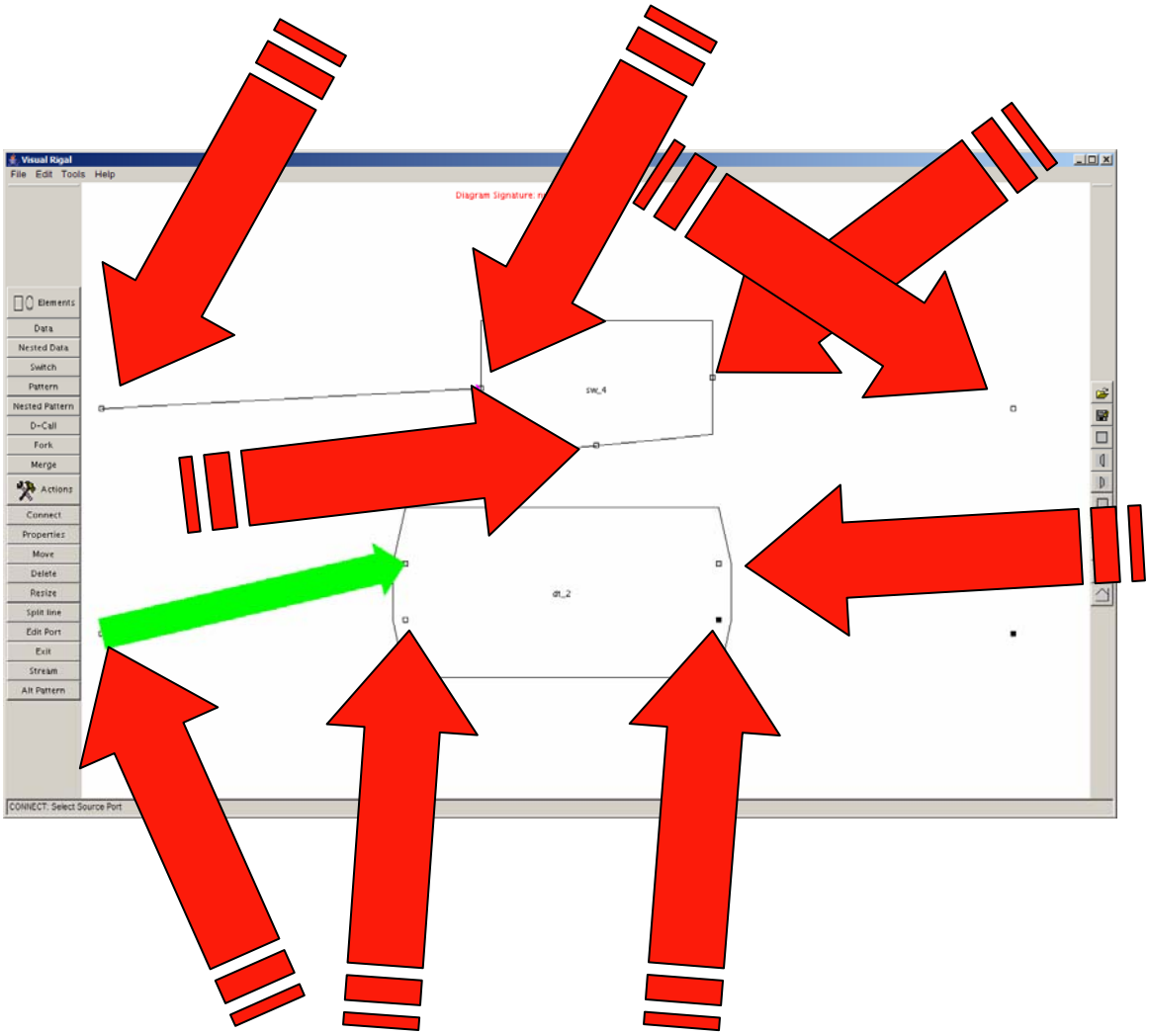


Figure 2. Large Arrows with Dashed Ends Point to Port Icons

b. Connector

A data pipe which allows only one data element at a time.

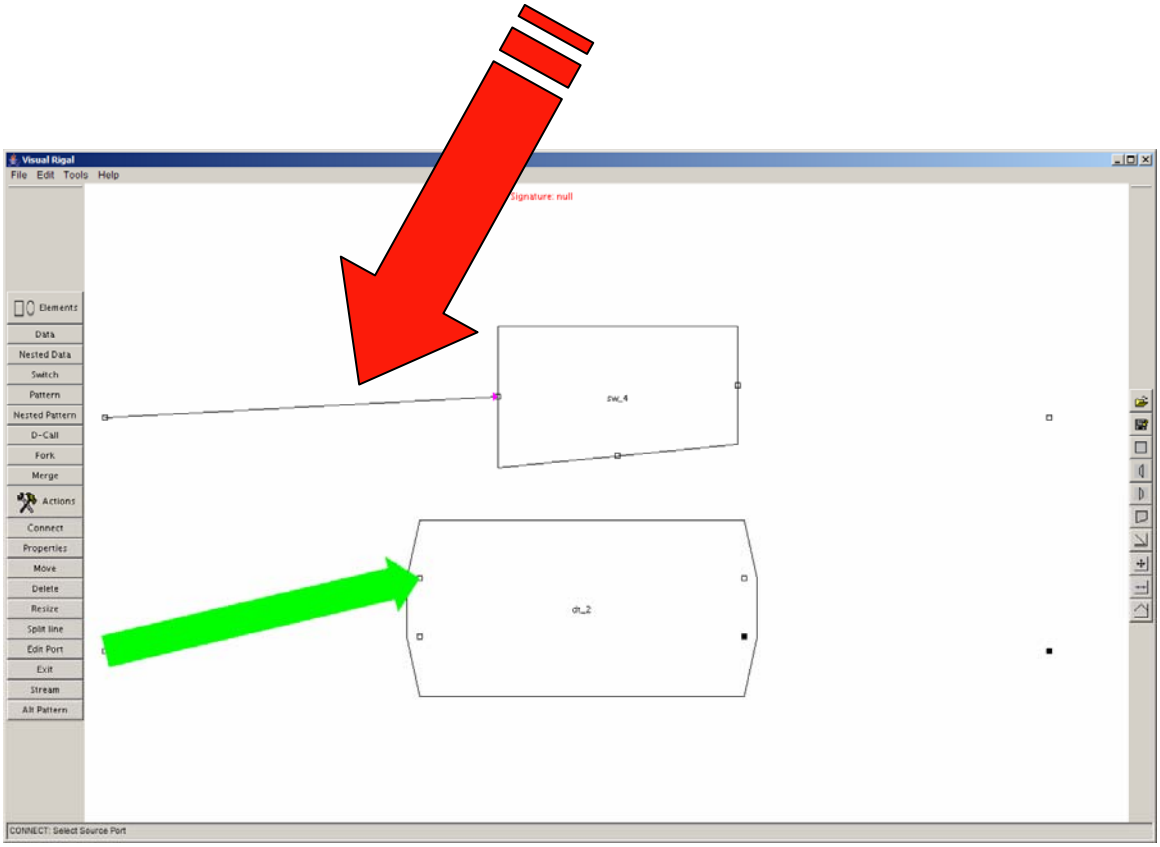


Figure 3. Connector Icon

c. Stream

A data pipe which allows a continuous flow of data, as in UNIX pipes.

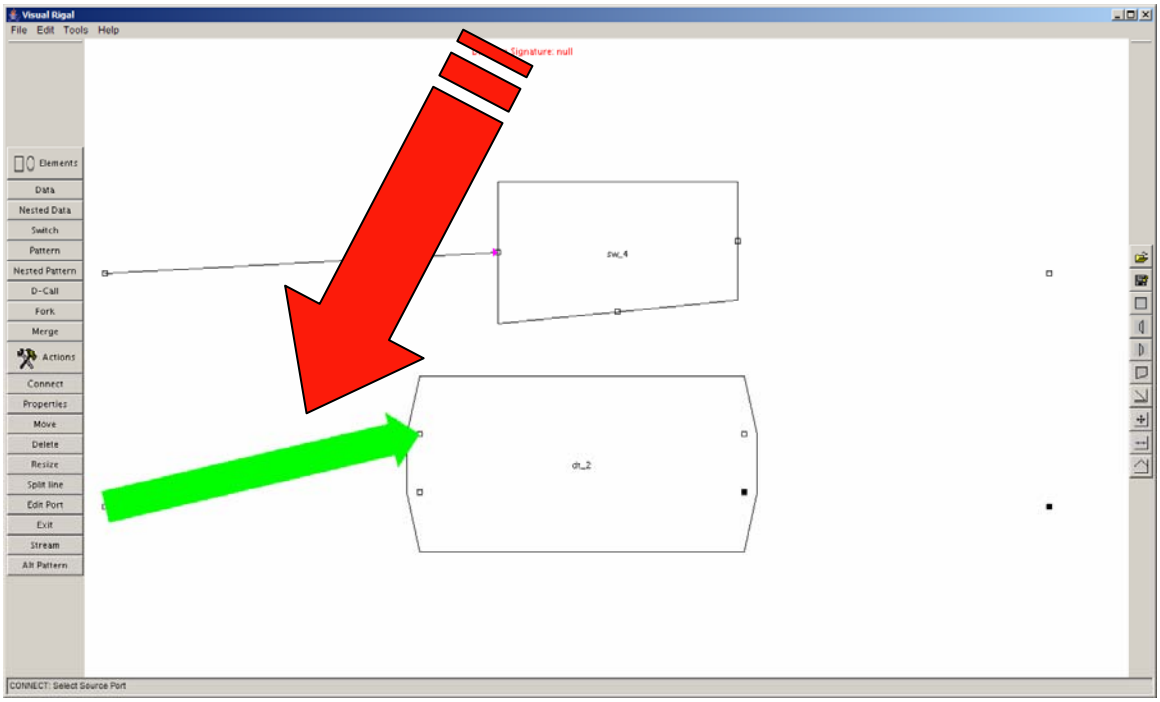


Figure 4. Stream Icon

d. Switch

A conditional or if-then statement. Uses a train track switch metaphor to depict data-flow based on truth of Boolean statement contained in the icon. As with all elements, can have 1 to many ports. All three port-sides of a switch have the same number of ports. All data entering the left side of the switch is available at the right side ports if the statement is true, or available at the bottom ports if the statement is false.

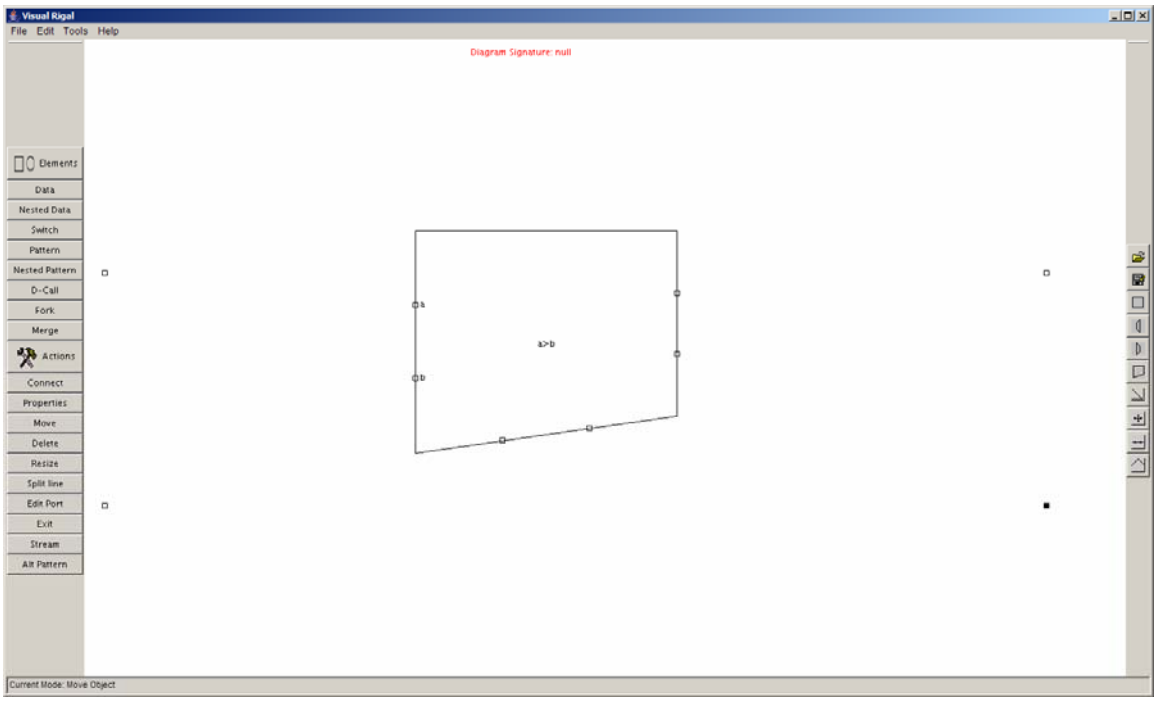


Figure 5. Switch Icon

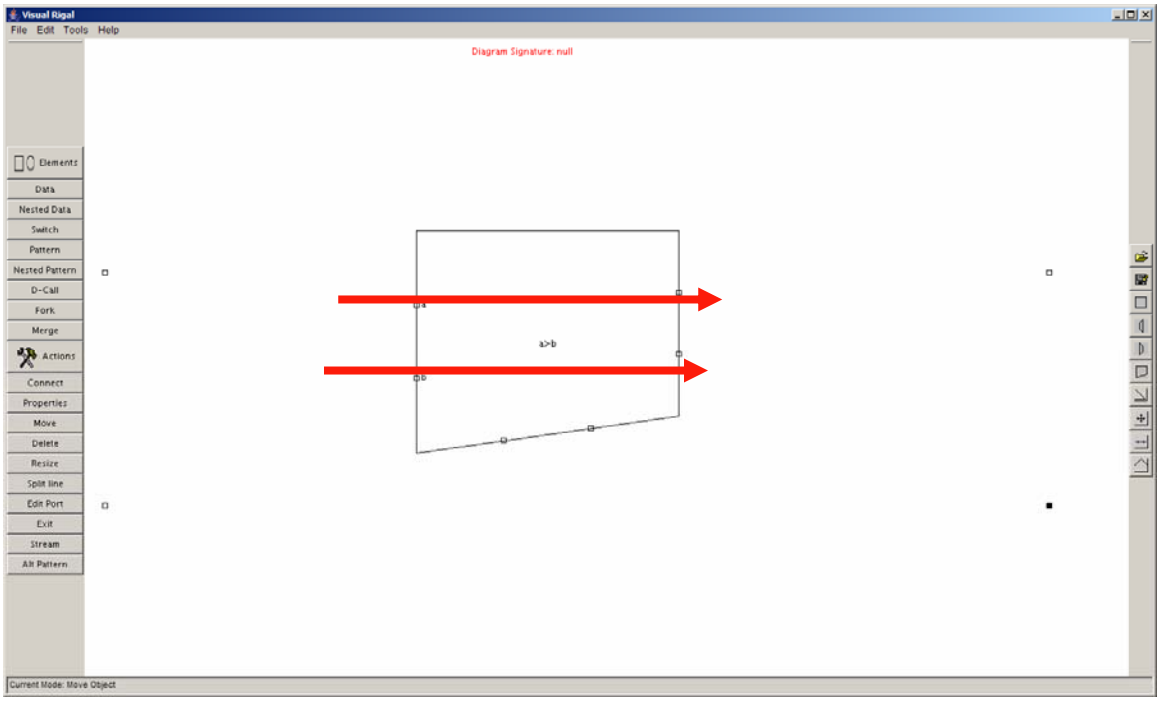


Figure 6. Data Flow If a>b is True

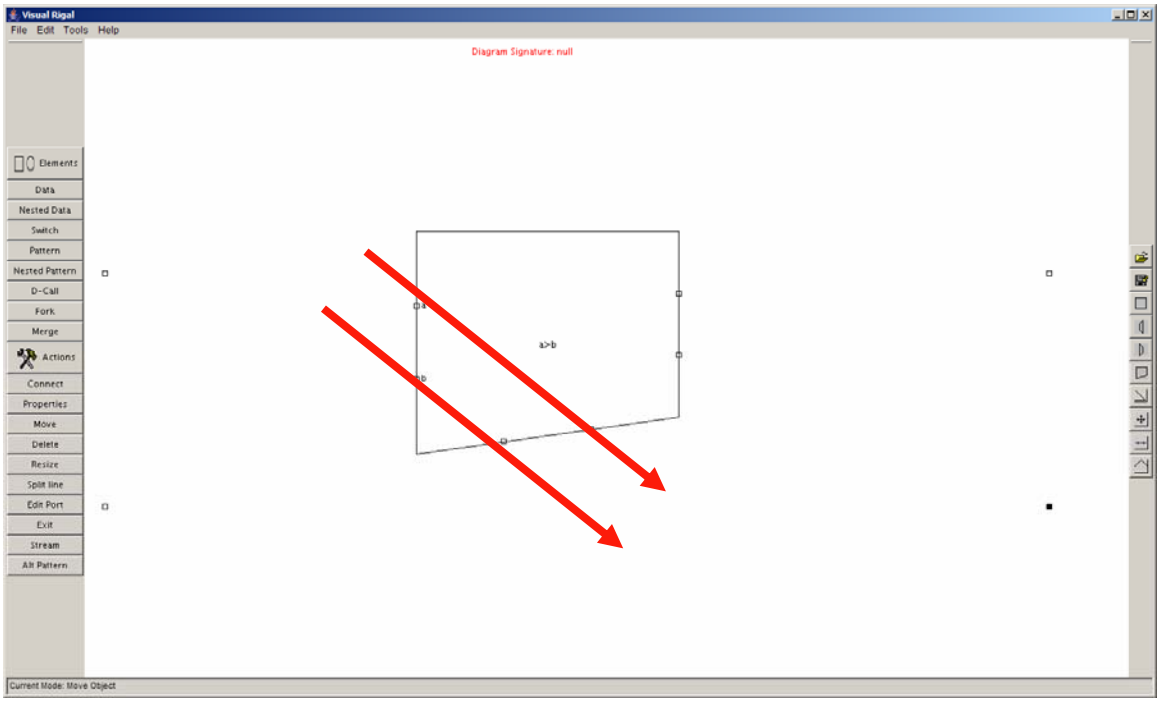


Figure 7. Data Flow If a>b is False

e. Data Box

Icon to input data into data pipe. May also hold a C type operator, +, *, ++, etc. and perform primitive mathematics using these symbols. In the example below, the smaller data box outputs the integer 5 on the first output port, while the larger outputs the sum of the two input values on the second output port.

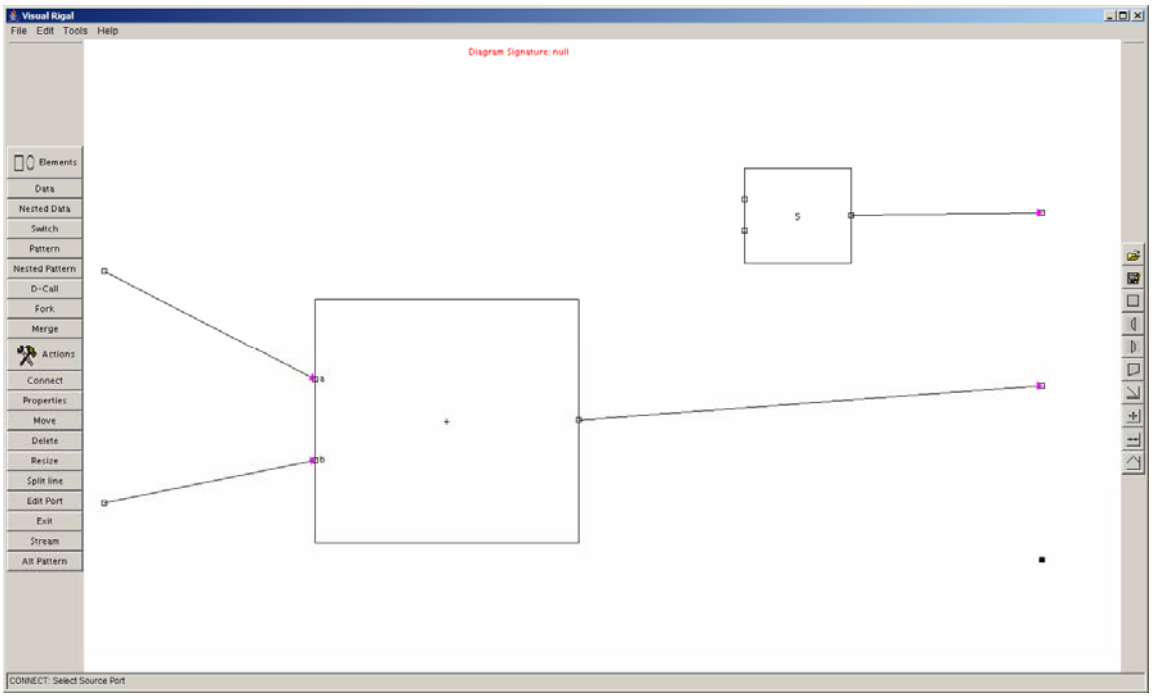


Figure 8. Data Box Icons

f. Nested Data Box

Represents the building block of a data structure. The structure can be passed through a data pipe connected to the outer Box. Can be used to assemble lists and trees. Provides visualization of data structure.

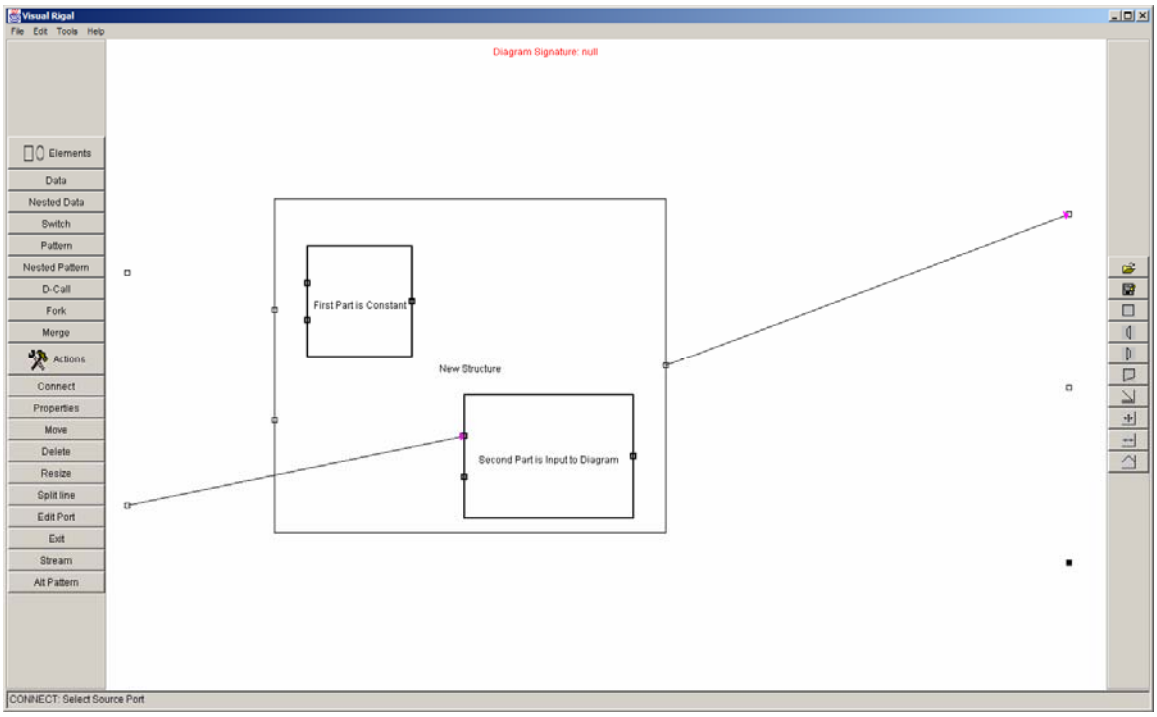


Figure 9. Creation of a Data Structure Using Nested Data Box Icon

g. Pattern Box

Performs pattern matching on data.

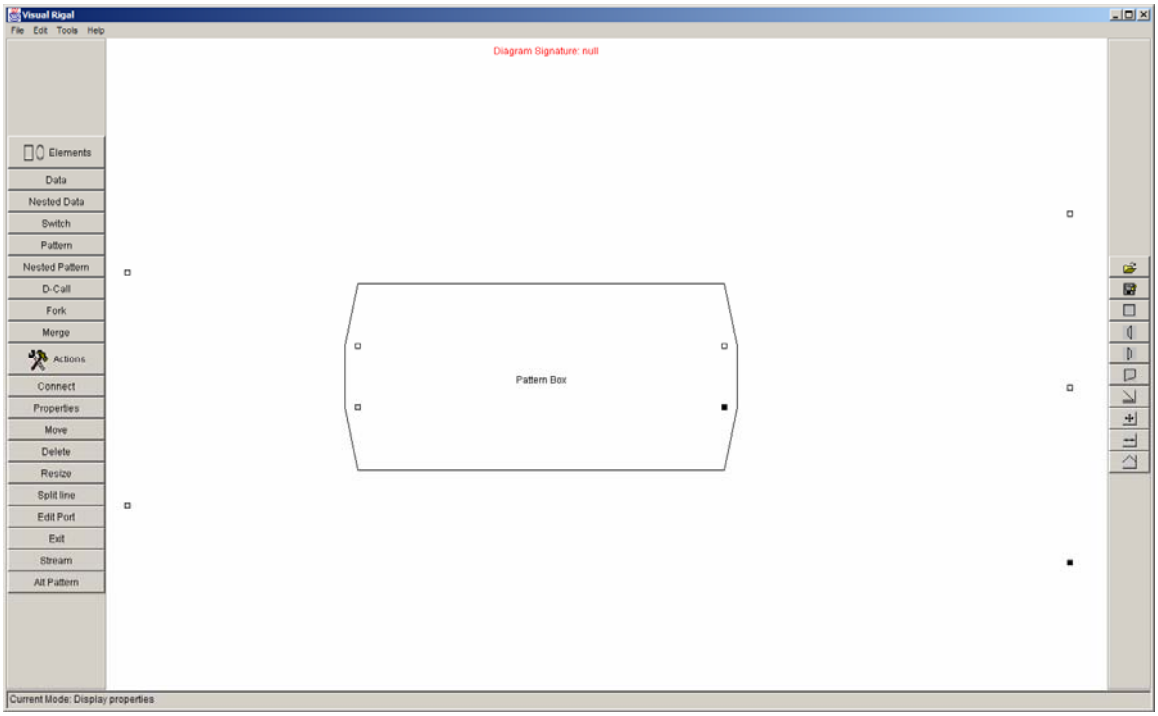


Figure 10. Pattern Box Icon

h. Nested Pattern Box

Used to deconstruct the data structures assembled in the Nested Data Boxes. Provides visualization of data structure.

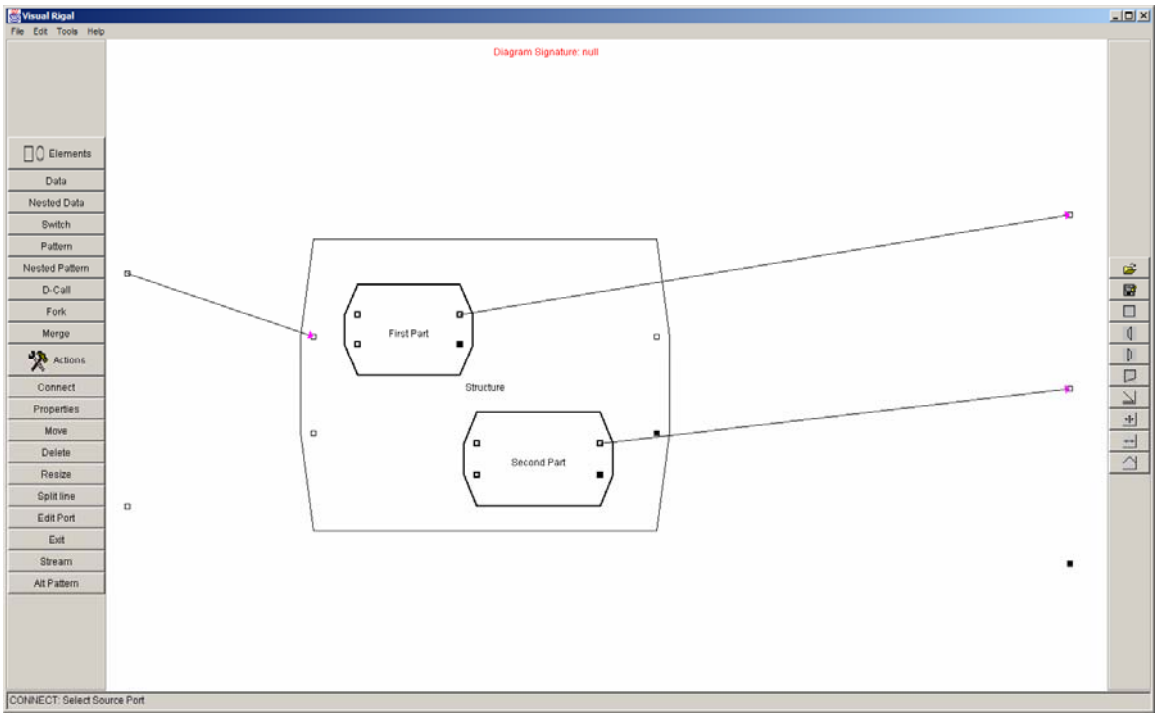


Figure 11. Nested Pattern Boxes Showing 2 Part Data Structure

i. Diagram Call

Allows reuse of diagram for subroutines or recursion.

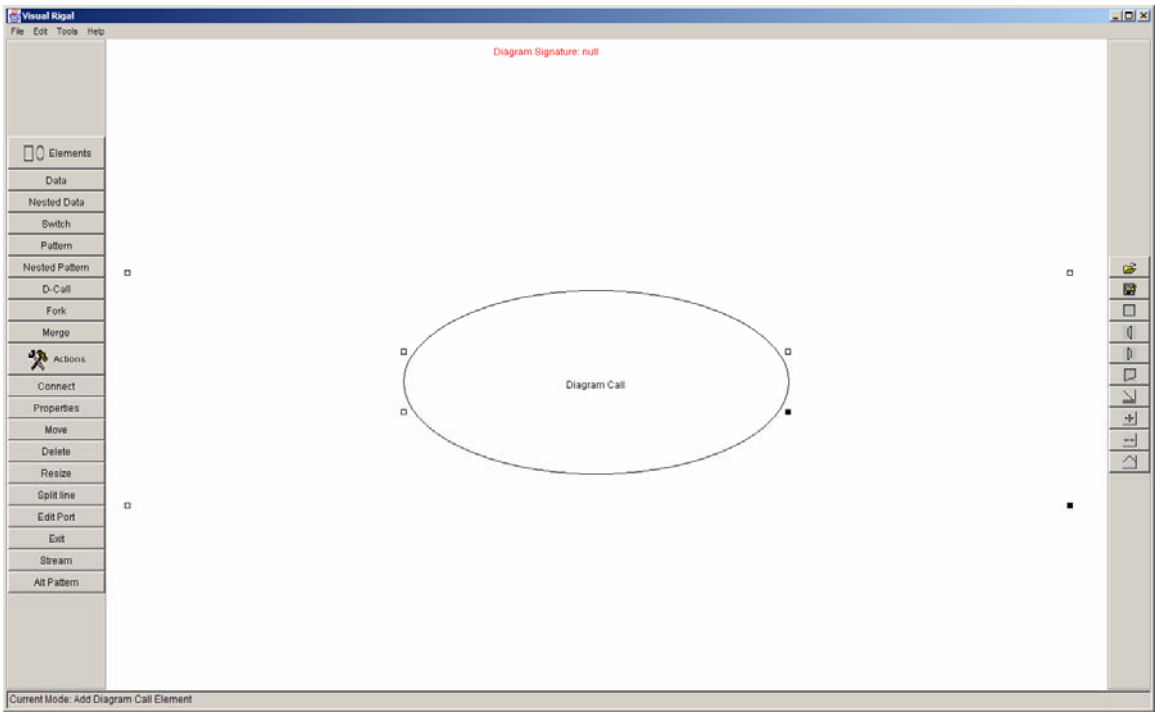


Figure 12. Diagram Call Icon

j. Fork

Duplicates data item.

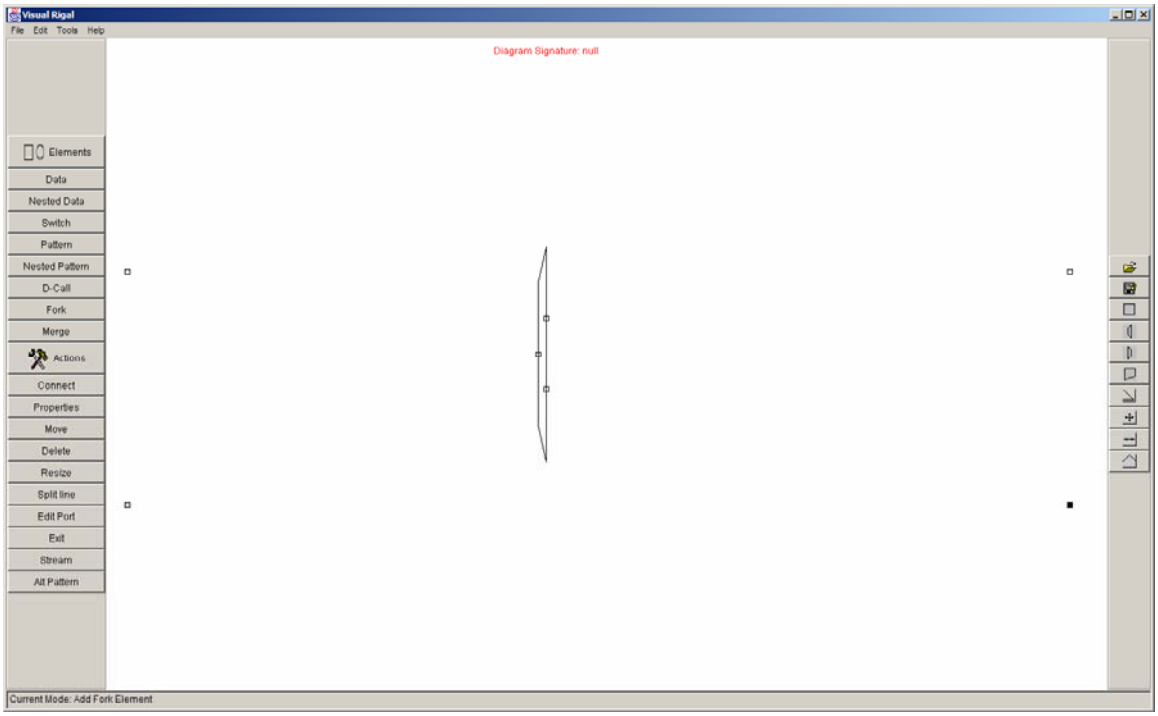


Figure 13. Fork Icon

k. Merge

Merges data streams into single stream

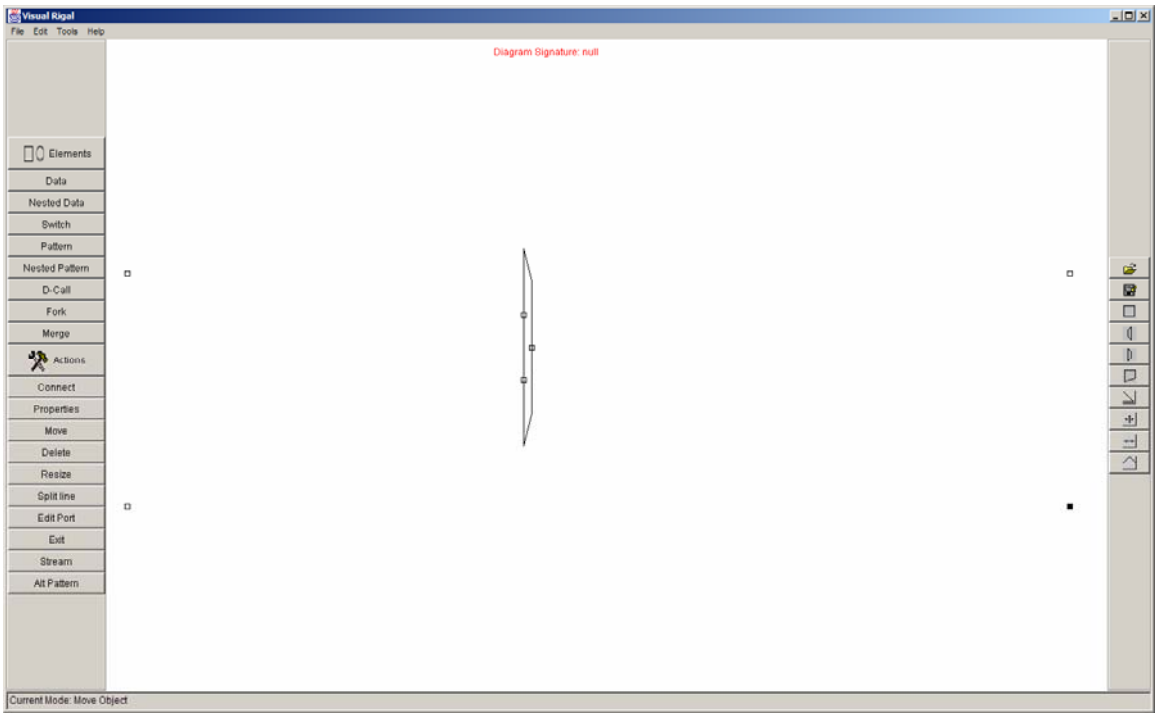


Figure 14. Merge Icon

l. Alternative Pattern

Used for pattern matching in support of parsing, similar to a switch statement. Starts looking for a pattern match at the top port. The first port with a successful match consumes the input and moves control along the route defined by the pattern matched. The example in the second figure will take an input stream and send the data following an integer to the first ouput port, following a rational number to the second ouput port and anything else to the fail port.

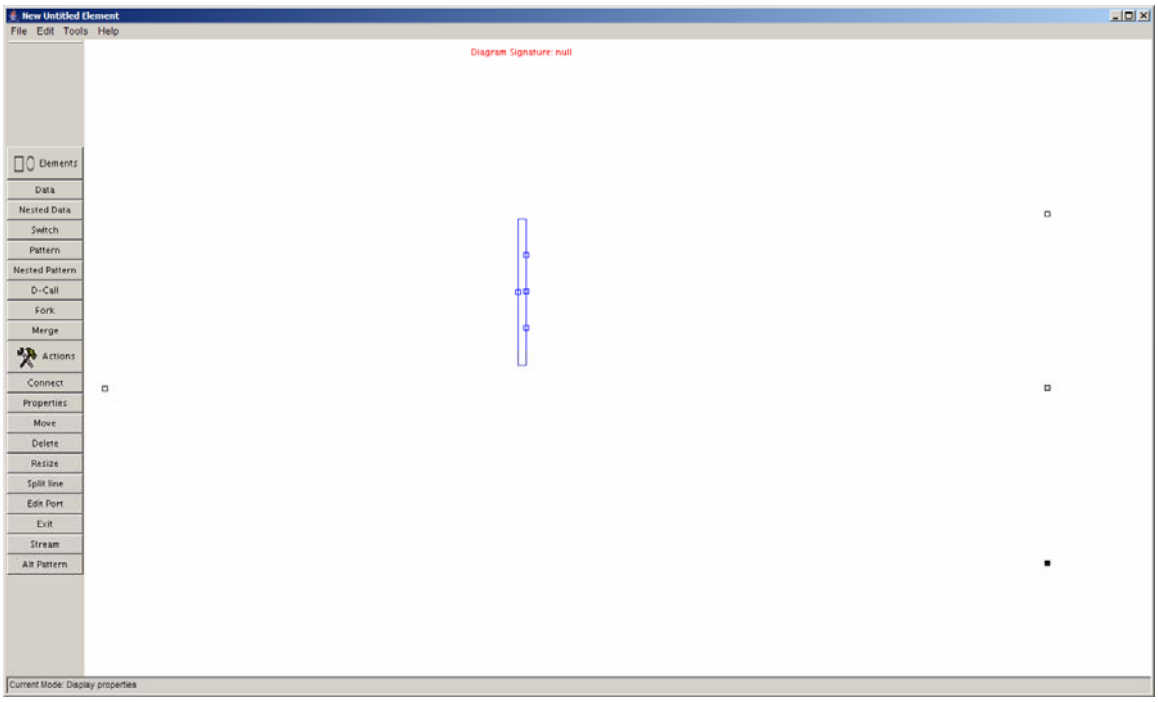


Figure 15. Alternative Pattern

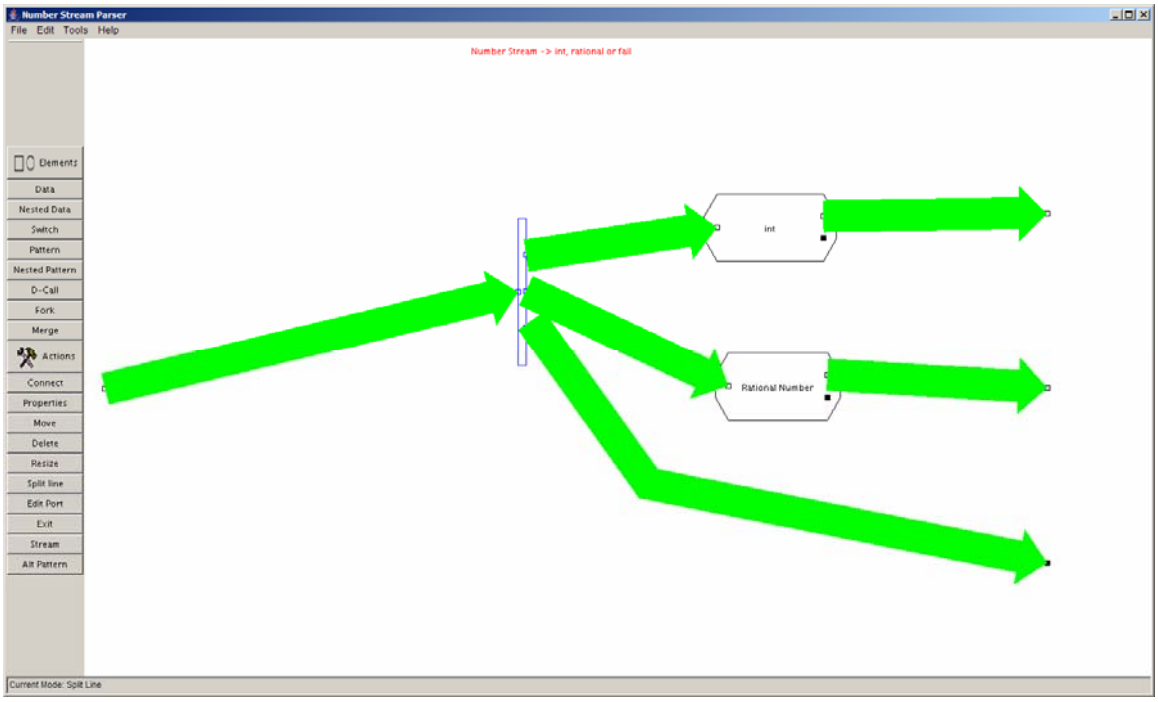


Figure 16. Parsing Usage of an Alternative Pattern

3. Application Examples

Examples are provided in the following figures. The first four define factorial functions, taking an integer input N and producing an integer N!. The first figure shows a

recursive function demonstrating the ability of VisualRigal to describe recursive functions. The second figure is an equivalent GUI diagram. Note the use of a single port switch and data boxes with a mathematical operator in each figure. The third is an equivalent iterative solution. This is followed by another GUI equivalent. The structure of the diagram suggests the existence of parallelism in the program structure. This additional information about the program comes with the visual representation.

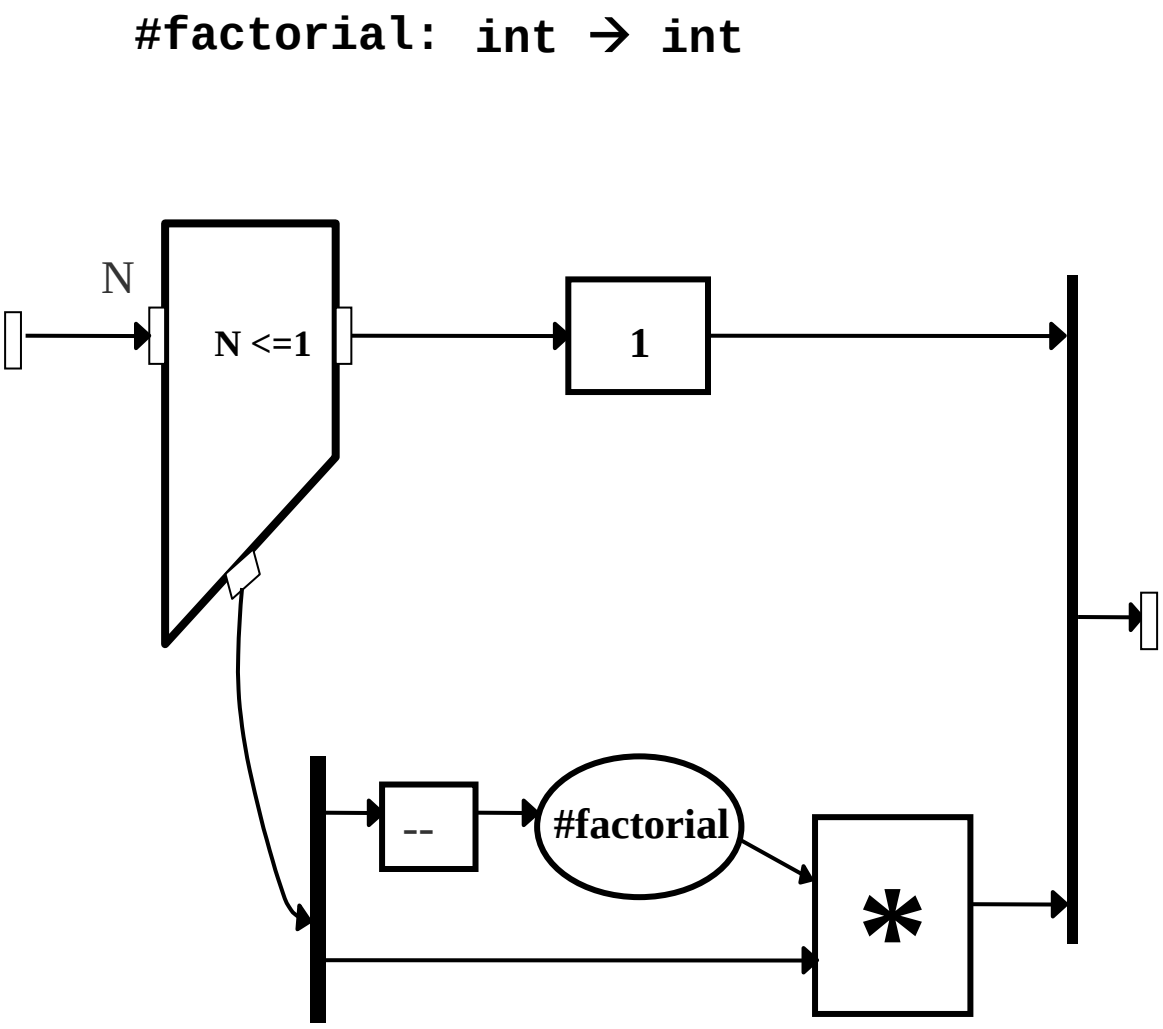


Figure 17. Recursive Factorial Example After [8]

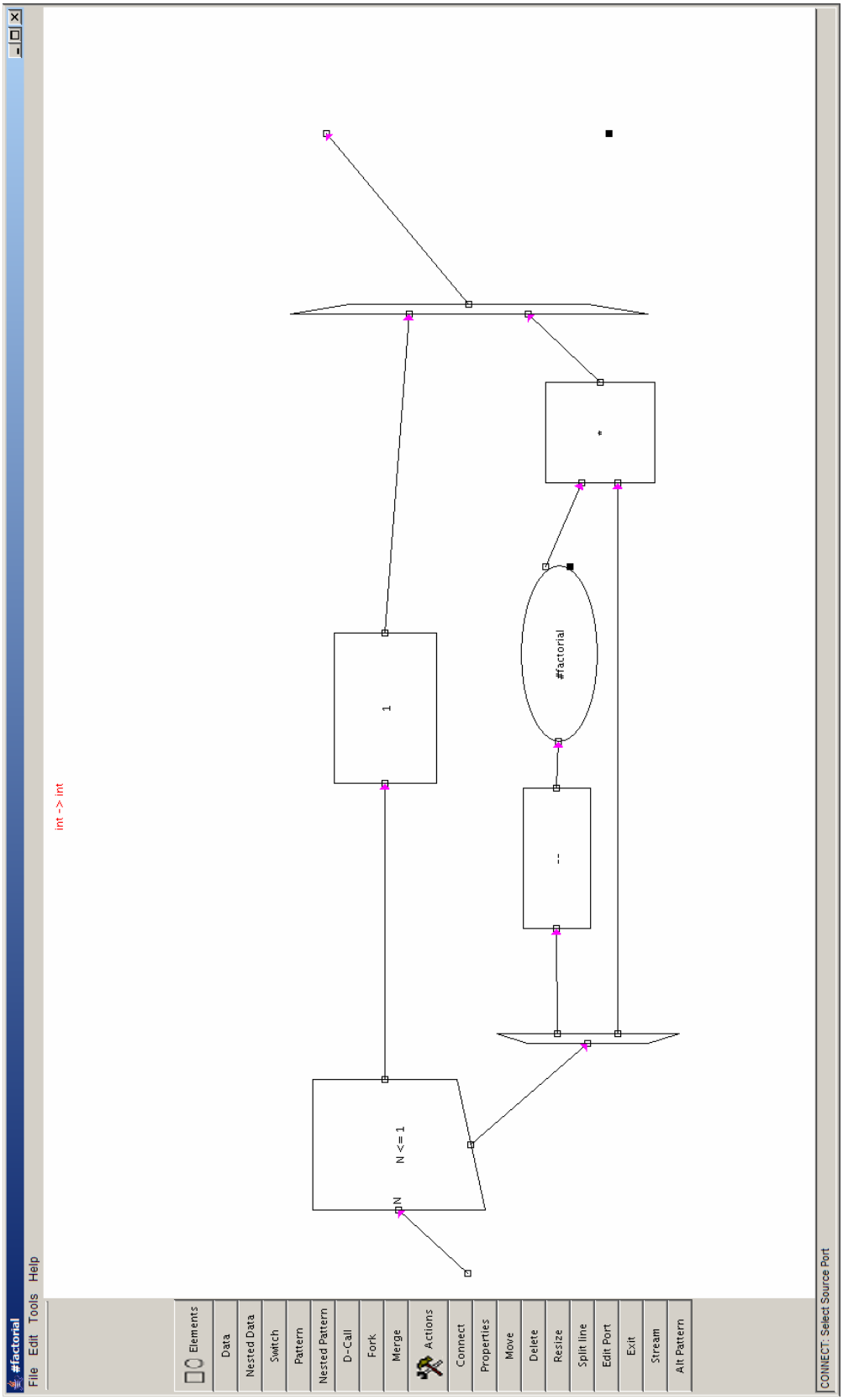


Figure 18. Recursive Factorial Example in GUI

```
#factorial: int -> int
```

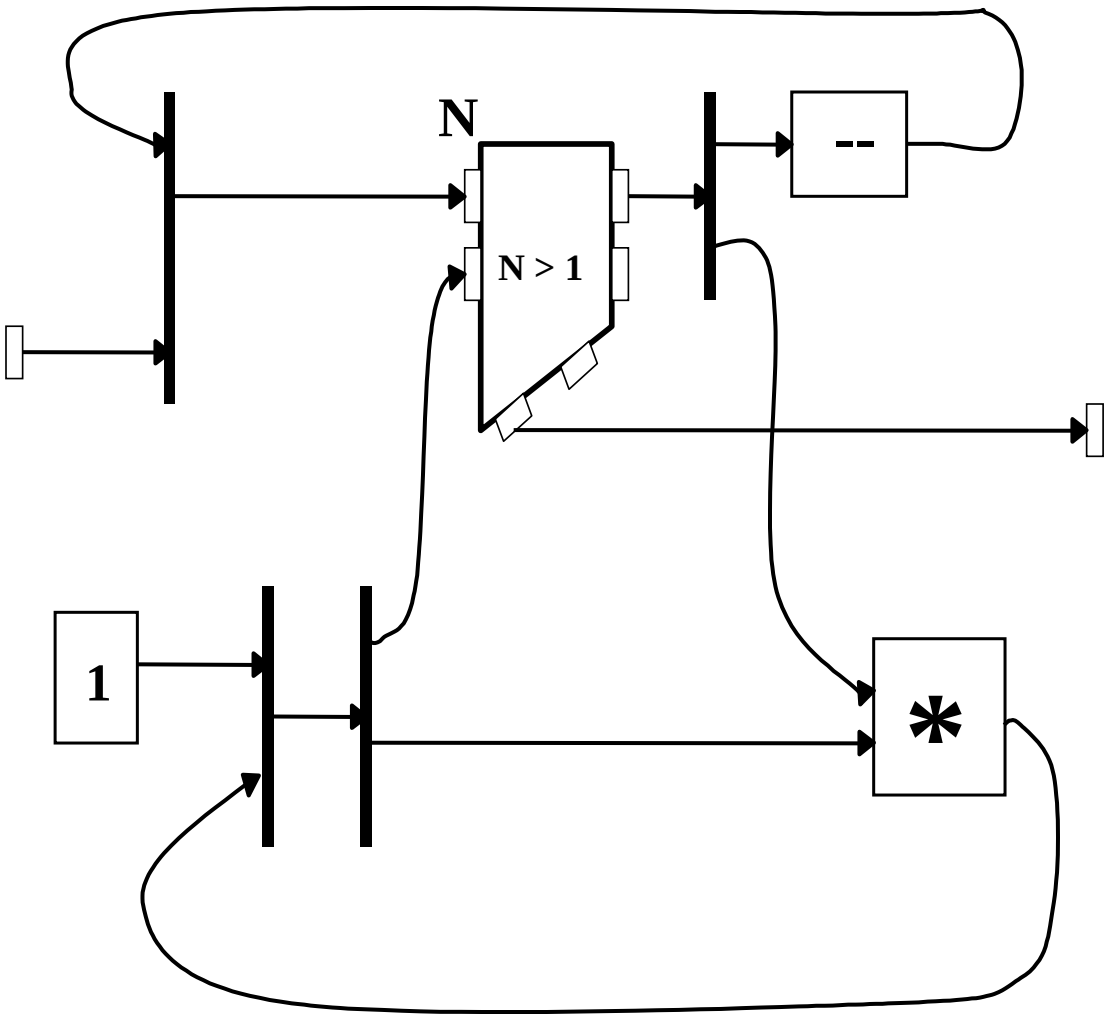


Figure 19. Iterative Factorial Example After [8]

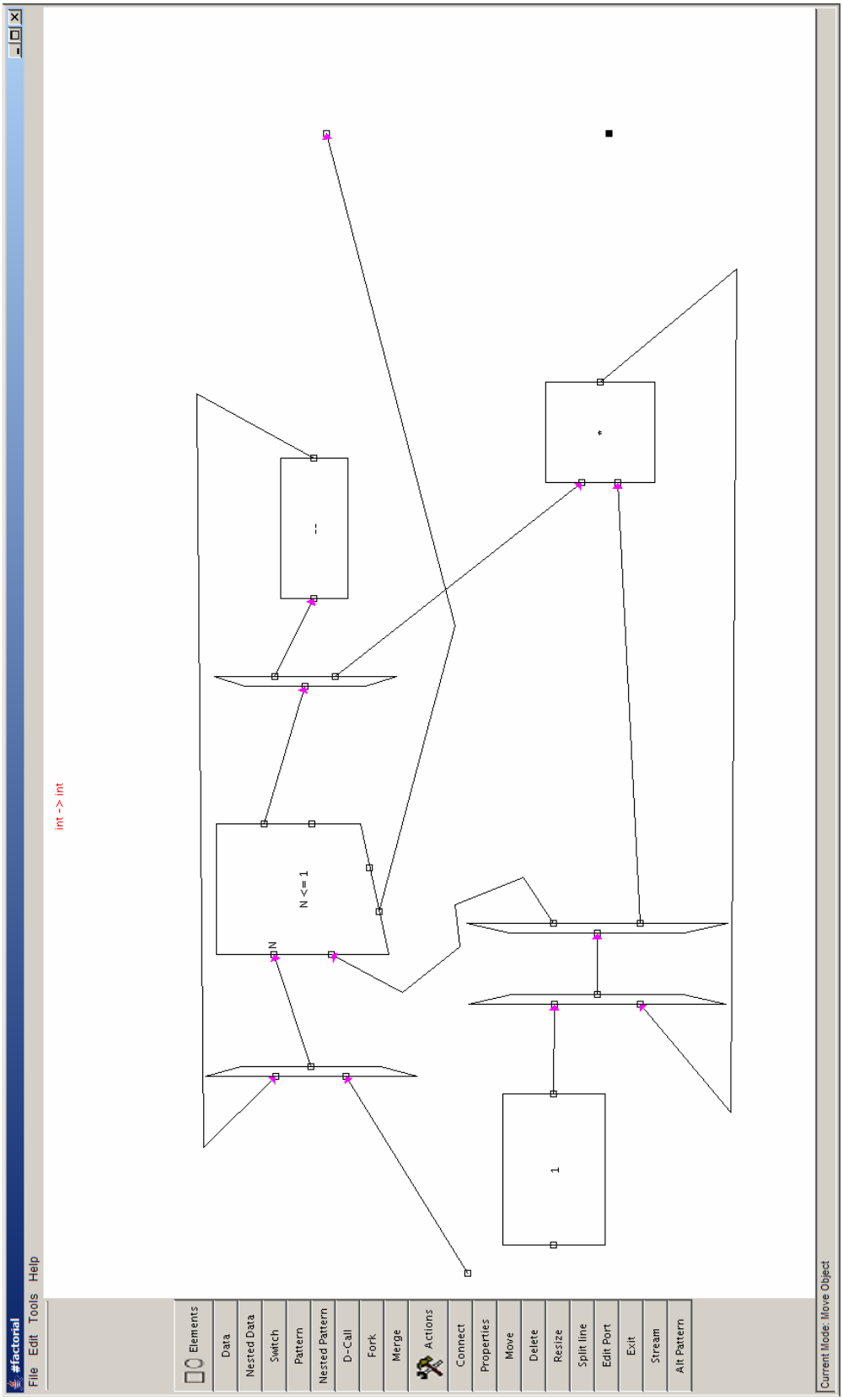


Figure 20. Iterative Factorial Example in GUI

The next diagrams demonstrate the visibility of data structure in VisualRigal. The first constructs a eational number data structure out of two input integers. The next figure takes two such data structures and adds them correctly. In this addition figure the simpiler path is when the denominators are equal. This path is represented by the top half of the diagram and is very easy to follow. The second case is the lower path, when the denominators are not equal. The diagram is harder to follow at this point giving a good indication of the complexity of the logic involved. We instinctively recognize the potential confusion at this point of the program. We are focused at a point in the routine that needs carful attention to correctness or is a likely source of a fault.

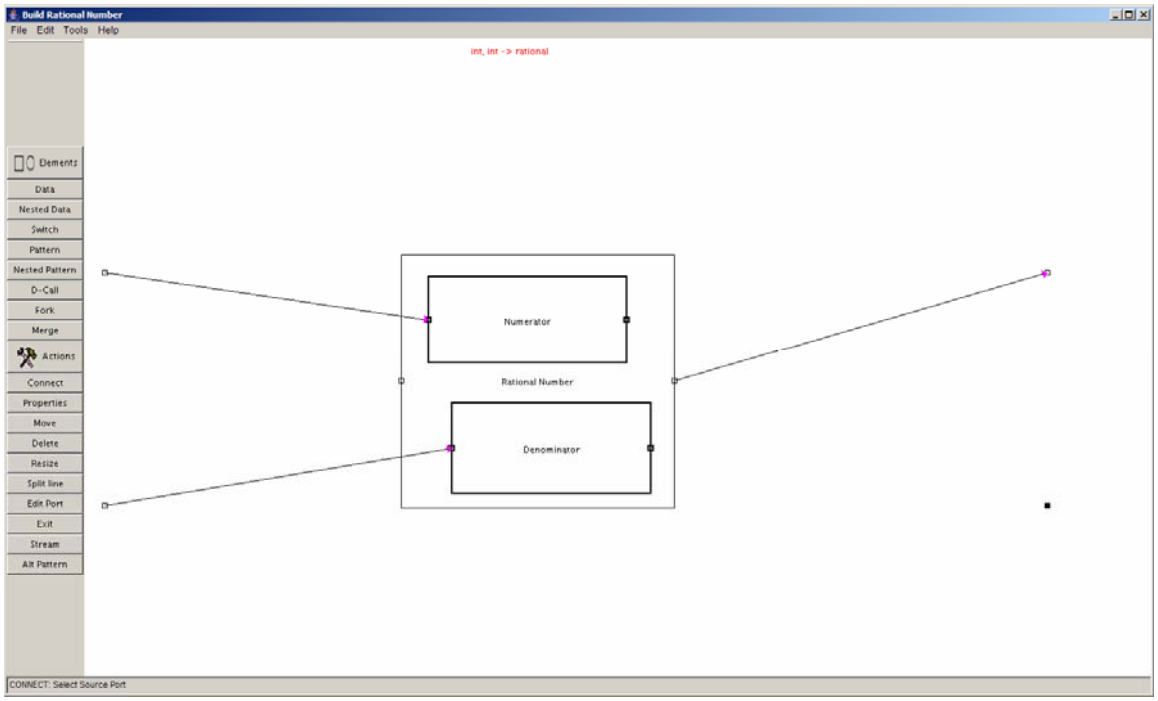


Figure 21. Function to Build Rational Number Data Structure

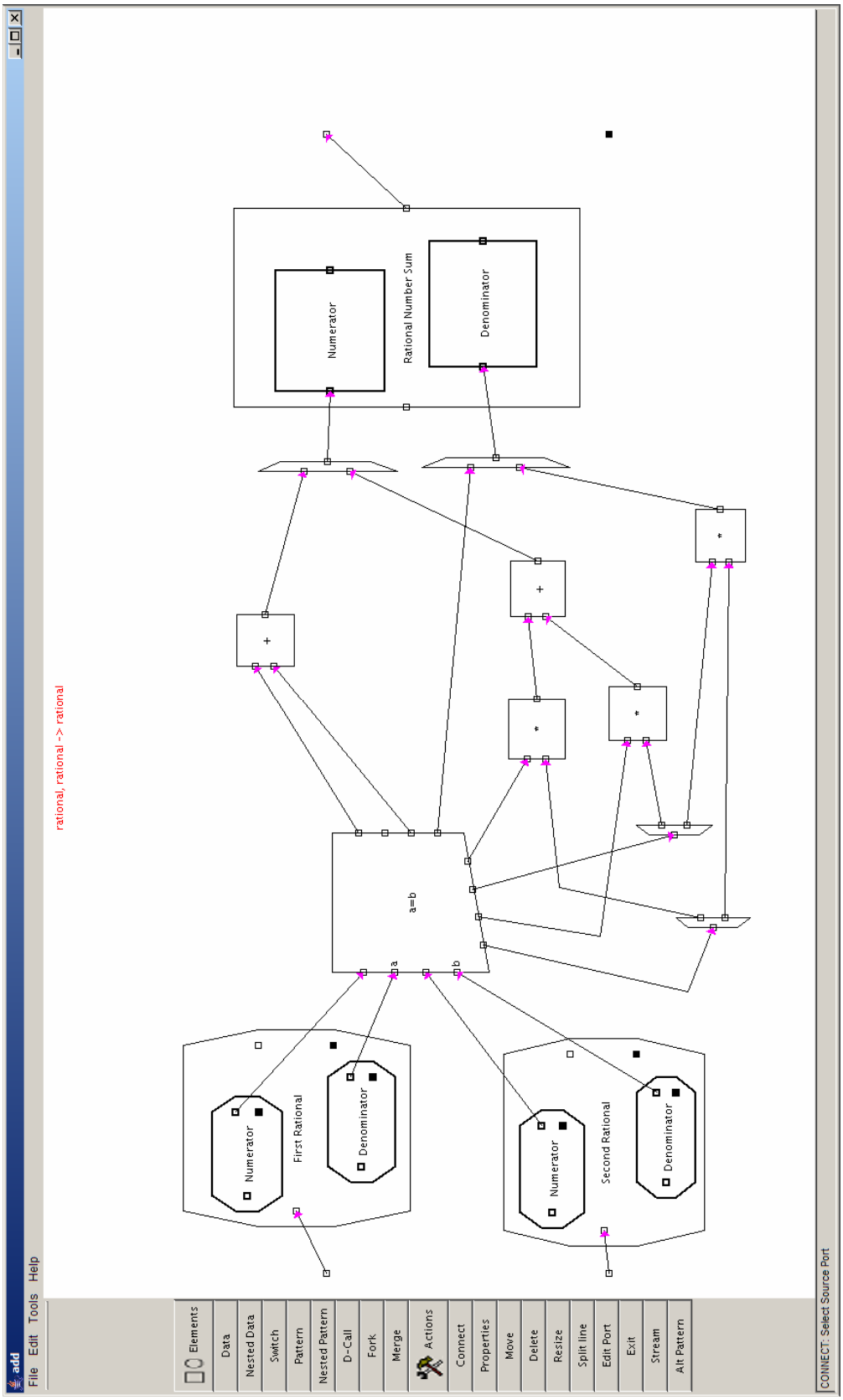


Figure 22. Function to Add Rational Number Data Structures

III. IMPLEMENTATION OF GUI

A. GUI HISTORY

The VisualRigal GUI was originally implemented by Mr. Abu Islam in 2002. Mr. Islam is a professional programmer who's work was proceeding nicely when a new java release disabled the save and restore feature of the GUI. Mr. Islam was unable to detect the new fault in his implementation before his time was over. His effort did produce a workable design implementation with over 10,000 lines of code.

The code indicates an experienced Java programmer using advanced techniques supported by the Java Swing and AWT classes. His use of self commenting code is generally excellent, providing signposts for the traveler who follows him. The code does not support standard Javadoc comments, creating difficulty when trying to maintain or extend this application.

B. CODE READING TOOLS USED

The main intent of the present work was production of a maintainable program and feature extension. Finding the best tools for a particular task was never intended. This represents the fastest way to progress at the given moment. With some exposure to particular tools or work on a UNIX based system could undoubtedly provide a faster pace of production. However, the methods listed here avoided a common project pitfall, getting caught up in tools rather than production.

1. Package Search as a Grep Substitute

The first step toward understanding any program, without other documentation, would be to start at the beginning. For a C-based language this means finding the main method. There is a painful story concerning the search through the 45 program classes that will not be related further. In any event, the preferred method for this phase is a search over all files for the main signature. This can be accomplished with an IDE path search or MS Windows search utility. This technique is simply the equivalent of an UNIX grep command.

2. Javadoc

Javadoc, when properly supported by well commented code, can be an excellent documentation resource. Javadoc is, however, dependent on the quality of comments and

transparency of method and class naming. If the class containing the main method does not identify itself with this fact, “ApplicationMain” for example, a tedious search situation ensues. The main method will be listed with the methods of one of the classes in an operational java program, but which one?

There is less text to search through in this case. The standard formatting of Javadoc even makes it obvious where to look, in the methods section alphabetically listed. But must the author once again open each classes Javadoc hoping to get lucky? Additionally, he do not know how to search HTML pages.

The Javadoc hierarchy is of some utility, showing inheritance by indentation. Assuming that main will be found in a top level class there were still 32 classes to search with this rule in mind. Not a promising start to a work in understanding approximately 10,000 lines of code. The author’s solution is presented below under New Code Implementation.

3. Simple UML Tool

BlueJ is an open source Java IDE design for use by students during there first exposure to Java. One of its features is the representation of all classes in the target file in UML form. BlueJ shows inheritance relationships and class uses. Though not a complete UML representation of the code, the class uses provide additional information the Javadoc inheritance structure does not.

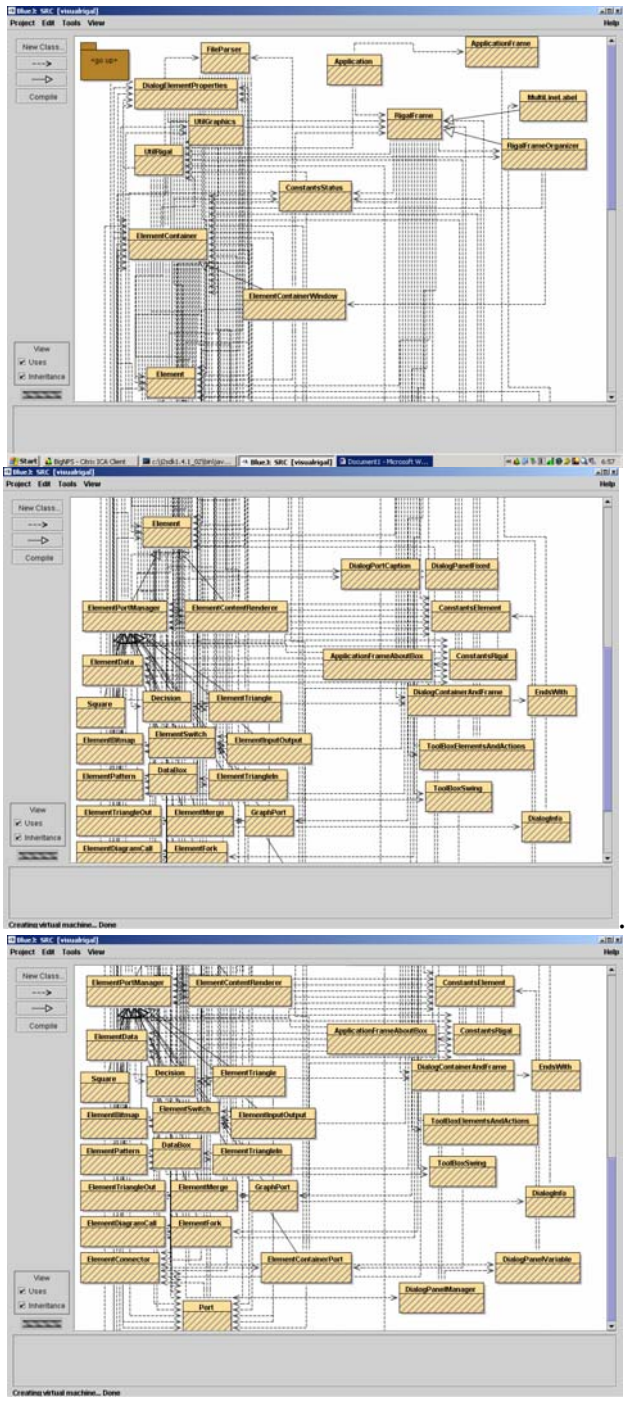


Figure 23. Three Screens of UML Depiction From BlueJ.

There are negative side-effects to BlueJ's use for this purpose. BlueJ explicitly removes package statements from Java packages. This change makes the code unusable for IDEs requiring packages.

BlueJ also adds an additional file to the folder. BlueJ opens to the same state at which it was exited. This additional file probably records this state. In any event, this file is not recognized by other IDEs making it more of a benign side effect than anything else.

The UML diagram presented by BlueJ must be formatted by hand. There is no provision for the "pretty printing" of this information nor printing. Thanks to the state saving feature, once laid out for readability the work is done provided you maintain the particular file that was formatted. Since you cannot normally use BlueJ and another IDE on the same package this reformatting can become tedious.

4. Code Tracing for Understanding by Following Instantiations and Events

Once the main method is located a traditional hand-trace of the code can begin. There are likely as many techniques for code tracing by hand as there are code tracers. The author has, in the past, used a trace utility to print out the code line reference number as it executes. This can be a useful tool for debugging spaghetti code, but it was found unnecessary for this event oriented program.

The author's tracing method is presented here for illustration of the maintenance process, working from a printout of the class containing the main method and tracing through a print out of any instantiated code as necessary. It is his habit to annotate with pencil, his notions of code purpose as he goes. In fact, these notes became the comments to document the code. In this manner these hard copies immediately become usable documentation for the code and commenting the code becomes a typing exercise, perfect for one's less cognitive periods.

Examination of classes as they instantiate develops an understanding of program state at a given moment. In the case of an event driven program, these classes are likely to contain the event driven methods allowing focus on a relatively small subset of the code for initial understanding and maintenance. As Spinellis says, don't try to

understand all of the code, only that portion effected by the proposed change.[10] The execution must be followed until a steady state is reached. Specifically the end of the main method or event driven methods may be overlooked in a program with multiple active event listeners, as is the case here.

At this point correction and extension of the code can begin. Once the event listeners are identified and their related event methods found, the program must be traced in terms of response to events. It is now time to examine particular behavior that is incorrect by tracing from the event method to completion, presumably another steady state awaiting the next event. If extension of the code is the current goal, the initiating event must be traced unless it has no legacy use. When the new program behavior is initiated by an event method not previously used the trace is unnecessary. If this initiating event is currently used, a mouse click for example, the event should be traced to completion in order to determine where the new behavior would be best inserted.

5. Assessing Method Invocation

Once it is determined that an existing method must be modified, it is necessary to asses the impact of the change. Are there unintended side-effects from this change? Will legacy use of this method provide the expected response? A first step towards answering these questions is to examine all uses of this method.

A package search for the first part of the method signature, “methodName(“ can provide a list of method invocation occurrences. Heavy use of overloaded methods for polymorphic purposes can increase the complexity of this method. In such a case a more elaborate tool may be needed, one that can associate parameter and variable types to completely define the method signature.

6. IDE Method List

Both IDEs used, NetBeans and JBuilder 5.0, have a window that alphabetically lists the methods of the class currently selected. This window can be useful for quickly locating a class method. By selecting the method name in this window, the code listing displays the portion of code beginning with the first line of the method listing. There are some disadvantages to this technique.

The methods may be listed alphabetically by access type along with variables and other foundational information about the class. If this is the case, there may be a delay in realizing you are looking in the wrong access type list for your method of interest. Private methods were hard to find listed at the end with public methods listed at the beginning and variables listed in between.

This list does not provide any information about method inheritance. If a method is defined higher in the class inheritance hierarchy you may be better off using a package search to find this method.

C. DESIGN DOCUMENTATION

A requirements analysis of this application was conducted producing an expected program design prior to code examination, shown below.

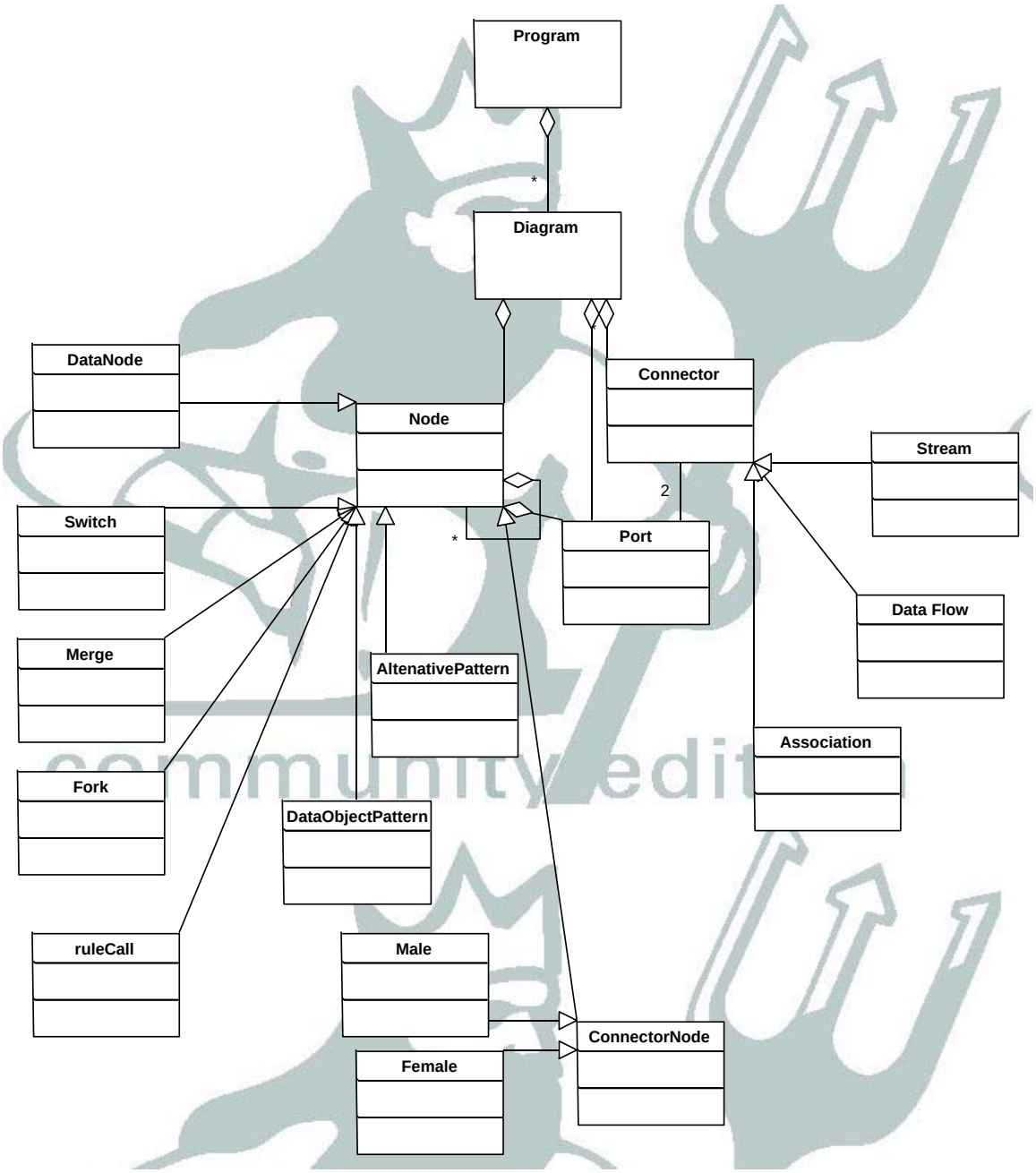


Figure 24. Expected Program Design Based on Requirements Analysis From Appendix D

Examination of the code has produced an understanding of the actual design implementation shown below. Note the grouping of classes by dashed and solid boxes: Constants, Utility Classes and Unused Code. The unused code is thought to containe classes that were original experiments for current element classes and unfinished work on classes that could become elements if the work was finished.

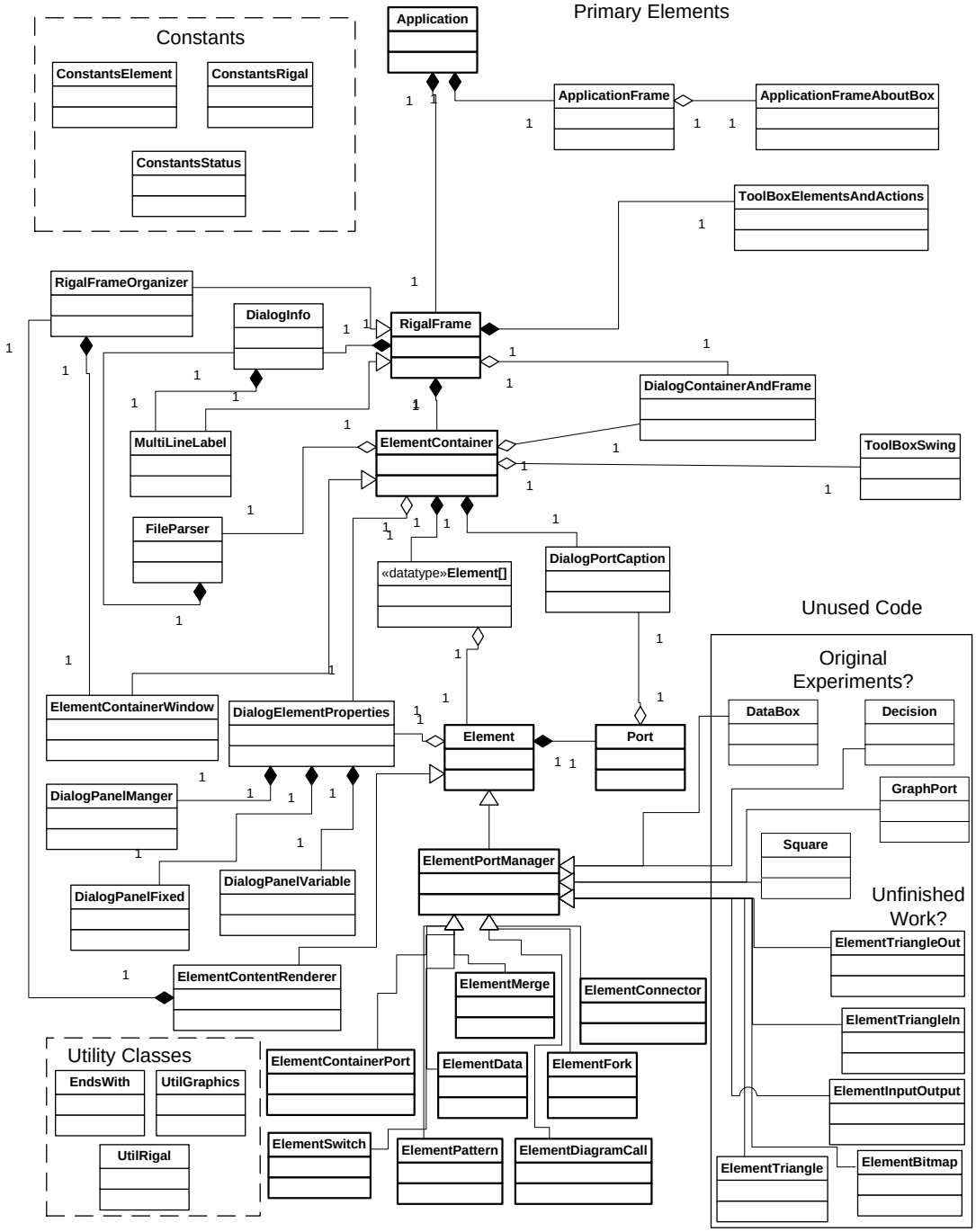


Figure 25. Actual Design of VisualRigal based on Code Examination

Extension of the code by the ElementStream and ElementAltPattern classes has produced the following modified design.

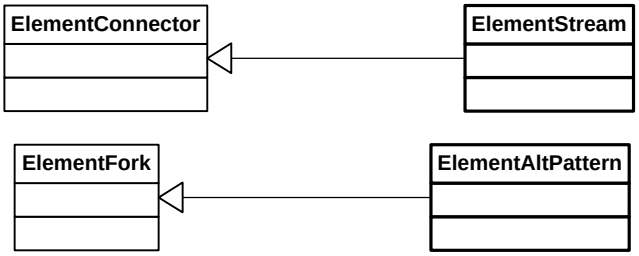


Figure 26. Thesis Class Extensions

D. NEW CODE IMPLEMENTATION

1. Finding Where to Begin

With all of the tools discussed available, except the package search or grep equivalent, getting started was the most daunting part of the task. With the intention of limiting the amount of random search, the author used the BlueJ UML diagram to begin manually sorting the classes. Classes without inheritance where set off in a separate group, per figures 1-3. There emerged 3 inheritance trees rooted in the following: RigalFrame, ElementContainer and Element. None of these roots contained main() and their children where unlikely to, so some of the classes were eliminated from the search. At this point the Application class stood out as having no indicated uses. That is no other class referenced Application as witnessed by the lack of use arrow heads. Problem solved, Application class contains the main() method.

2. Execution Path to Steady State Awaiting Event

Examination of Application class shows the constructor creating an Application Frame for unknown purposes. Since the constructor will not execute until an instance of Application is created this may be meant to support a Java applet, but ApplicationFrame is not fully developed for any purpose.

Execution path is as follows: Application.main() creates a RigalFrame → RigalFrame() creates an ElementContainer attribute → Application.main() → steady state awaiting event.

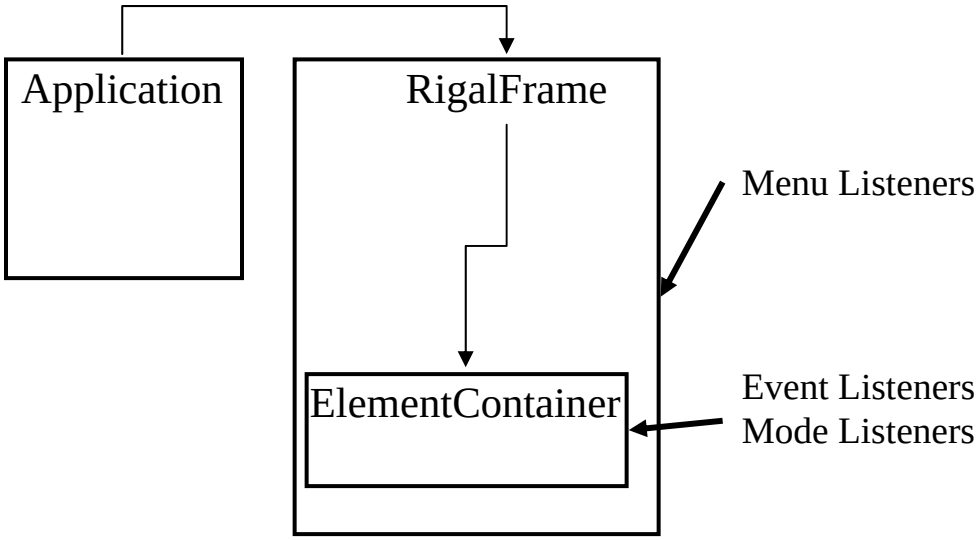


Figure 27. Instantiation Order and Source with Location of Event Driven Methods

Java listeners were found in RigalFrame and ElementContainer, these define the program interface for this event driven program. RigalFrame contains the menu item listeners and ElementContainer contains the remaining listeners, diagram area and mode button listeners.

3. Fix for 3 Compiler Errors

The 1.0 code would not compile with a Sun standard compiler due to 3 non-standard uses of the keyword “super” in class ElementPortManager.

```
public Element getElement()
{
 return super;
}

public void move(int x, int y)
{
  // move mother element
  if (super.getNested()){
     moveNestedElement(super, x, y);
     // System.out.println("called moveNestedElementOnly()");
  }
  else if (!super.getNested()){
     moveNotNestedElement(super, x, y);
     // System.out.println("called moveNotNestedElementOnly()");
  }
}
```

Figure 28. Improper Uses of Java Keyword “super”

```
public Element getElement()
{
 return (Element) this;// GCP changed super() to (Element) this
}

public void move(int x, int y)
{
  // move mother element
  if (super.getNested()){
     moveNestedElement((Element) this, x, y);// GCP changed from super() to (Element) this
     // System.out.println("called moveNestedElementOnly()");
  }
  else if (!super.getNested()){
     moveNotNestedElement((Element) this, x, y);// GCP changed from super() to (Element) this
     // System.out.println("called moveNotNestedElementOnly()");
  }
}
```

Figure 29. Correction to Improper Use of Java Keyword “super”

Evidently the Borland compiler allow the use of “super” to denote the appropriate super class for the situation, since the code did compile with JBuilder 5.0. Not only was this inconvenient for use with other compilers, IDEs and tools; but it is poor practice. Leaving the compiler to figure out the appropriate casting strikes the author as risky. It is much better to explicitly state the intended cast so there is no doubt what is produced by the compiler. This change was made to the code as shown in figure 5.

4. Add Proper Nesting Output to Text Interface

Nested data boxes must translate to nested text representations. The body of a nested data box, label followed by information enclosed by parenthesis, must appear within the parenthesis of the parent data box. Nested data boxes in the 1.0 code printed out one after another as if none of them were nested, all occupying the same level as children of the diagram and no other box.

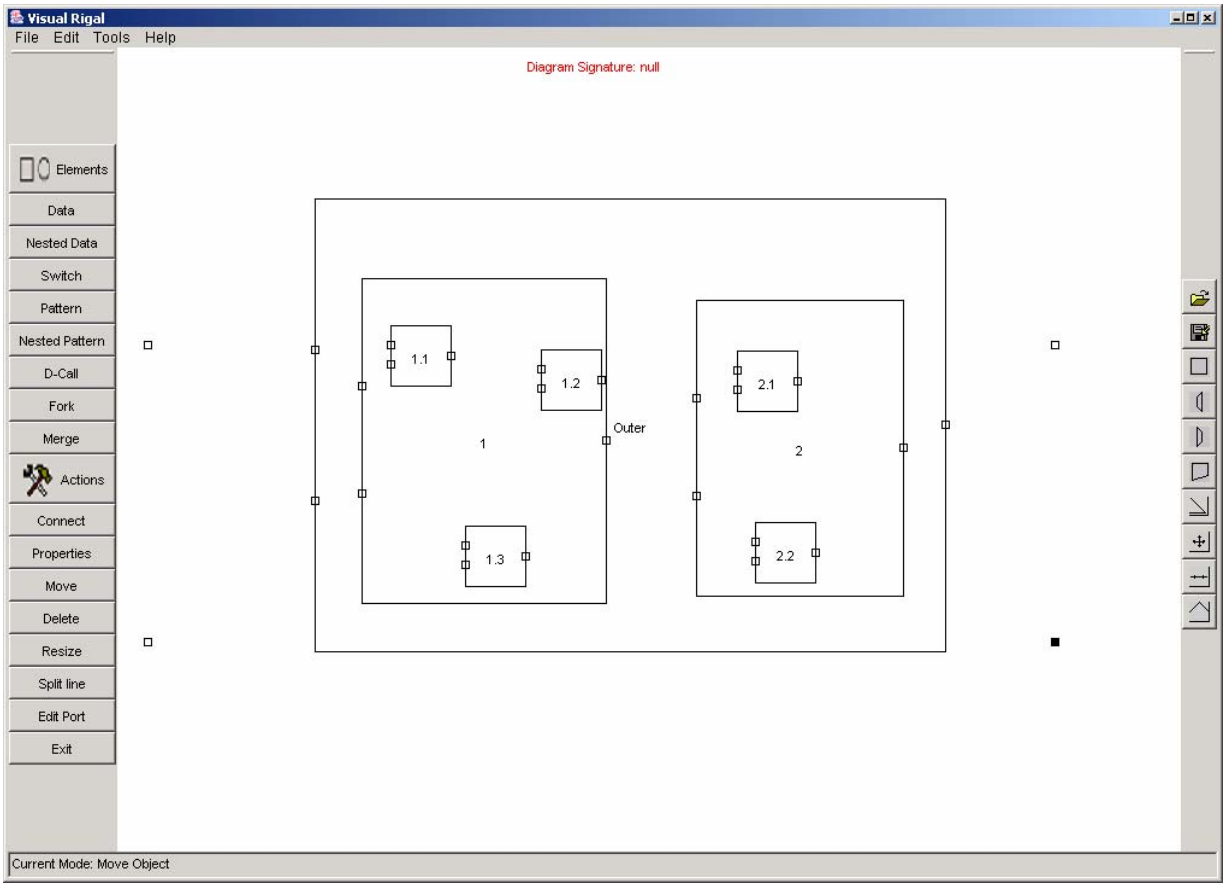


Figure 30. Sample of Graphically Nested Data Boxes

name

input_ports (p10 p11)

```
output_ports ( p12)
onfail_port    p13

data_box (
  inp_port_names ( null null)
  inp_port_ids ( p20 p21)
  out_port_ids ( p22)
  expr Outer
)


data_box (
  inp_port_names ( null null)
  inp_port_ids ( p30 p31)
  out_port_ids ( p32)
  expr 1
)


data_box (
  inp_port_names ( null null)
  inp_port_ids ( p40 p41)
  out_port_ids ( p42)
  expr 2
)


data_box (
  inp_port_names ( null null)
  inp_port_ids ( p50 p51)
  out_port_ids ( p52)
  expr 1.1
)


data_box (
  inp_port_names ( null null)
  inp_port_ids ( p60 p61)
  out_port_ids ( p62)
  expr 1.2
)


data_box (
  inp_port_names ( null null)
  inp_port_ids ( p70 p71)
  out_port_ids ( p72)
  expr 1.3
)


data_box (
  inp_port_names ( null null)
  inp_port_ids ( p80 p81)
  out_port_ids ( p82)
  expr 2.1
)


data_box (
  inp_port_names ( null null)
  inp_port_ids ( p90 p91)
  out_port_ids ( p92)
  expr 2.2
)
```

Figure 31. Text File Interface with Original Non-nested Behavior

Much of the code needed for nested boxes was already implemented. I suspect Mr. Islam was preoccupied with the save feature, discussed later, and saw no need to complete this feature without the ability to save and restore diagrams.

```
public void printText(PrintStream ps)
{
  System.out.println("printText called from ElementData");
  ps.println("");
  ps.println("data_box (");
  ps.println("   inp_port_names (" + getInputPortNames() + ")");
  ps.println("   inp_port_ids (" + getInputPortIds() + ")");
  ps.println("   out_port_ids (" + getOutputPortIds() + ")");
  ps.println("   expr " + this.getCaption());
  ps.println(")");
  ps.println("");
}
```

Figure 32. Original printText() Method with no Provision for Showing Nesting

The ElementData.printText() method was modified to only print root data boxes and call the new ElementData.printNestText() method for each of the children of the root box.

```
// print all parent data boxes, call nested print if needed
public void printText(PrintStream ps)
{
   if(!this.getNested())
   {
      Vector vMyChildren = new Vector();
      vMyChildren = getNestVector();

      System.out.println("printText called from ElementData");
      ps.println("");
      ps.println("data_box (");
      ps.println("   inp_port_names (" + getInputPortNames() + ")");
      ps.println("   inp_port_ids (" + getInputPortIds() + ")");
      ps.println("   out_port_ids (" + getOutputPortIds() + ")");
      ps.println("   expr " + this.getCaption());
      for(int i = 0; i < vMyChildren.size() ; i++){
         ((ElementData) vMyChildren.elementAt( i ) ).printNestText(ps,1);;
      }
      ps.println(")");
      ps.println("");
   }
}
```

Figure 33. New printText() Method Prints Root Boxes and Children Recursively

The ElementData.printNestText() is a copy of the current printText() method without the root-only restriction. This has the effect of recursively calling printNestText() for as many level of nodes as necessary.

```
//recursively print nested children, tabing based on current layer
  public void printNestText(PrintStream ps, int numberOfTabs) {
     Vector vMyChildren = new Vector();
     vMyChildren = getNestVector();

     ps.println("");
     printTabs(ps, numberOfTabs);
     ps.println("data_box (");
     printTabs(ps, numberOfTabs);
     ps.println("   inp_port_names (" + getInputPortNames() + ")");
     printTabs(ps, numberOfTabs);
     ps.println("   inp_port_ids (" + getInputPortIds() + ")");
     printTabs(ps, numberOfTabs);
     ps.println("   out_port_ids (" + getOutputPortIds() + ")");
     printTabs(ps, numberOfTabs);
     ps.println("   expr " + this.getCaption());
     for(int i = 0; i < vMyChildren.size() ; i++){
        ((ElementData) vMyChildren.elementAt( i ) ).printNestText(ps ,
numberOfTabs + 1 );
     }
     printTabs(ps, numberOfTabs);
     ps.println(")");
     ps.println("");
  }
```

Figure 34. New printNestText() Method Recursively Prints Child Boxes

An additional method, printTabs(), was added to increase readability by showing nesting with indentation, as is common in coding style conventions. The method simply prints the number of tabs passed as a parameter. printText() calls this with a value of 1. printNestText() has an additional parameter to pass this value, as this method recurses it increments the numberOfTabs by 1.

```
//prints given number  of 3 space tabs to stream
  public void printTabs( PrintStream ps,int numberOfTabs) {
     for(int i = 0 ; i < numberOfTabs ; i++ ){
        ps.print("   ");
     }
  }
```

Figure 35. New printTabs() Method

name

```
input_ports  ( p10 p11)
output_ports ( p12)
onfail_port    p13

data_box (
  inp_port_names ( null null)
  inp_port_ids ( p20 p21)
  out_port_ids ( p22)
  expr Outer

  data_box (
    inp_port_names ( null null)
    inp_port_ids ( p30 p31)
    out_port_ids ( p32)
    expr 1

    data_box (
      inp_port_names ( null null)
      inp_port_ids ( p50 p51)
      out_port_ids ( p52)
      expr 1.1
    )


    data_box (
      inp_port_names ( null null)
      inp_port_ids ( p60 p61)
      out_port_ids ( p62)
      expr 1.2
    )


    data_box (
      inp_port_names ( null null)
      inp_port_ids ( p70 p71)
      out_port_ids ( p72)
      expr 1.3
    )

  )


  data_box (
    inp_port_names ( null null)
    inp_port_ids ( p40 p41)
    out_port_ids ( p42)
    expr 2

    data_box (
      inp_port_names ( null null)
      inp_port_ids ( p80 p81)
      out_port_ids ( p82)
      expr 2.1
    )


    data_box (
      inp_port_names ( null null)
      inp_port_ids ( p90 p91)
      out_port_ids ( p92)
      expr 2.2
    )

  )

)
```

Figure 36. Text File Interface with New Nested Behavior

The Element class represents both a generic element and a nested generic element for Visual Rigal. It contains enough information to construct a logical tree for nested elements, though strictly only the data boxes and patter boxes may be nested and then only with like kind. Element.bNested serves as a nested flag to differentiate the condition. Element.iNestElementID provides the root-ward link to the parent by providing its element identification number. Element.vNestVector provides the leaf-ward connection by containing the element identification number of all immediate children, those one level down. Element.bHasChild provides a flag to indicate the presence of children. This last seems redundant, since we can achieve an equivalent result using !(Element.vNestVector.isEmpty()). However, this option is awkward and so less maintainable. The bHasChild increases understanding.

Control flow for nesting a data box: ElementContainer.mouseDown() → ElementContainer.nestDataBox() → ElementContainer.processNestedElement()

Equivalent ElementContainer.nestDataBox() has only the functional difference of calling ElementContainer.processNestedElement() from ElementContainer.addDataBox().

5. Note on Nested Behaviors

Most of the desired nesting behavior were implemented but were not evident due to lack of documentation. It takes significant experimentation or code tracing to determine how a correctly nested box is entered into the diagram. Because there is no visible difference between overlapping and nested boxes without trust in the program there is little willingness to make the investment required for understanding.

6. Maintain Connections During Resizing of Data Boxes

When an element having connections, as in figure 13 represented by thin arrows, is moved the connectors redraw based on the position of the element appearing to stretch in order to accommodate the motion. When a data box is resized, the connectors disappear requiring each connection to be restored by hand. The resize operation should have similar behavior to the move operation.

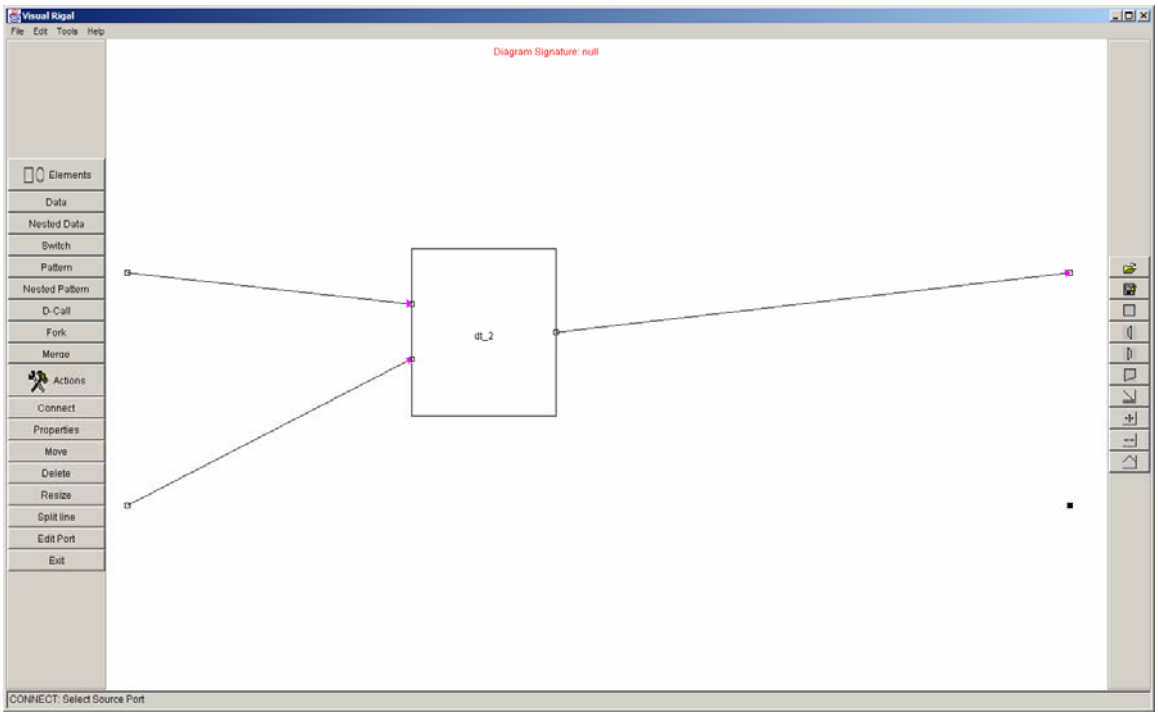


Figure 37. Element with Connections

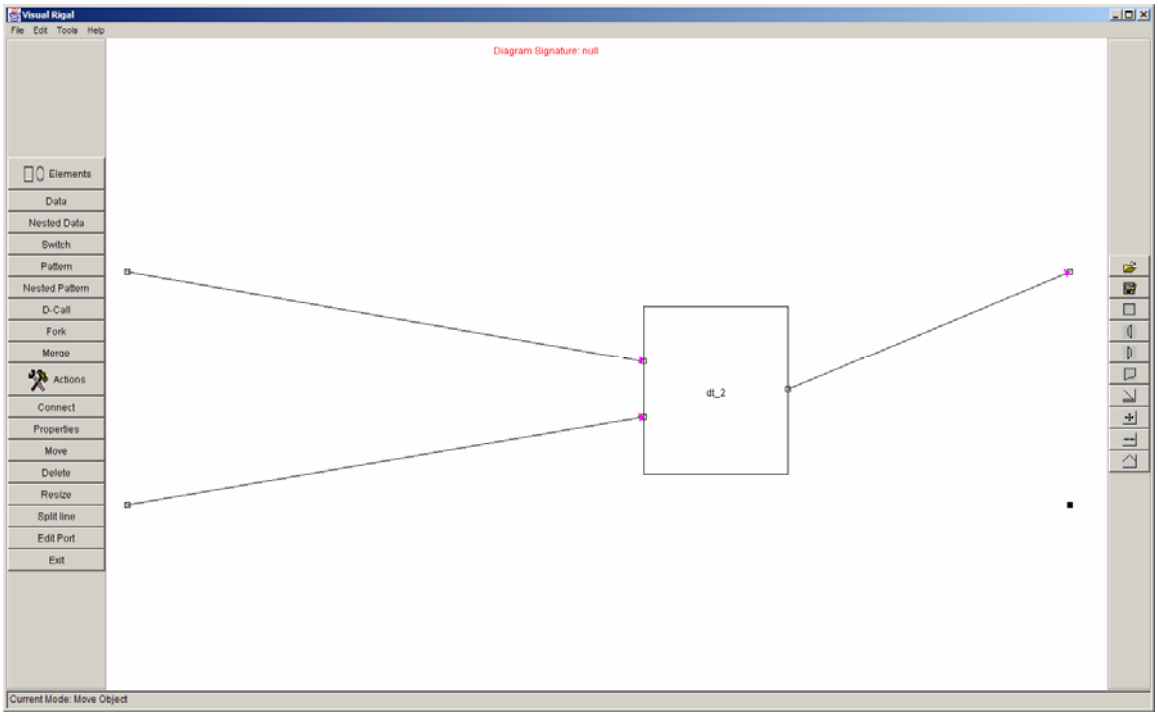


Figure 38. Same Element After a Move Operation

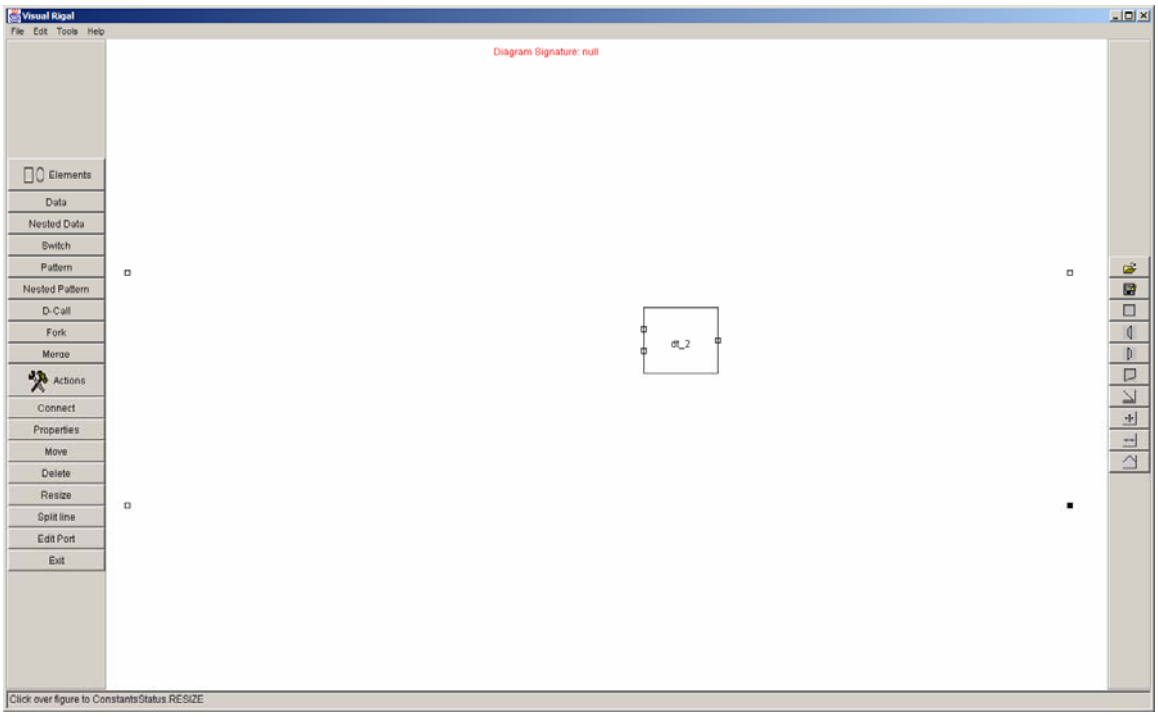


Figure 39. Same Element After Resize Operation

The move event chain and the resize event chain were examined for differences. The events diverge in the ElementContainer.mouseDrag() method based on the particular state that is current. The Element is recast as an ElementPortManger and ElementPortManager.resize() is called. This an empty method declaration that is meant to be overridden by each element's resize(). A cleaner interface could be achieved, and nothing lost, by not recasting the element and calling the resize method directly.

```
case ConstantsStatus.RESIZE:
 {
  // System.out.println(eElement.getDebugString());
  // System.out.println("Inside mouseDrag(), ConstantsStatus.RESIZE.");
     ElementPortManager epmLocal = null;
     if (eElement instanceof ElementPortManager)
  {
   // System.out.println("RESIZE condition met...");
      epmLocal = (ElementPortManager) eElement;
      epmLocal.resize(Math.abs(x-matchpoint.x), Math.abs(y-matchpoint.y));
      repaint();
   // System.out.println("RESIZE completed, repaint() executed...");
     }
 }
 break;
```

Figure 40. Unnecessary Recast and Method Call

```
public void resize (int width, int height)
{
  …
  //cleanAllPorts();// GCP strips connectors, remove
  //updateElementPorts();// GCP redraws ports, remove but use

  moveElementPorts();// GCP new method
}//end resize()
```

Figure 41. New resize() Method Showing Original Method Calls

The resize event explicitly deletes all connectors attached to this element. After which, it replaces all ports with new port instances in ElementData.updateElementPorts(). This method is present in each sub-class of ElementPortManager and is used to initialize each element. This seems like overkill, but Mr. Islam programming skill advocates for this being an intermediate step using existing code to get resize started. Once the save and restore bug arose, finishing resize would be low on the priority list. There certainly exists code to support correct behavior for the move event that I used with simple modification to change the resize behavior.

In contrast the move event chain uses a loop to translate each port on the element using the change in coordinates reported by the mouse event and Port.translate(), which is based on the Java Point.translate(). When the port moves the connection end points are updated by the next redraw since the ports define the end points.

```
public void moveNotNestedElement(Element eParent, int x, int y)
{
  // System.out.println("**** called moveNotNestedElement()");
  int difx, dify;
  difx = x - ptStart.x;
  dify = y - ptStart.y;
  Polygon polyTemp = null;

  // move mother element
  eParent.translate(difx, dify);
   for (int i=0; i<getNoOfPortsOnElement(); i++)
    {
     aPortsOnElement[i].translate(difx,dify);
    }

  polyTemp = new Polygon(getElementPolygon().xpoints,
                         getElementPolygon().ypoints,
                          getElementPolygon().npoints);

  setElementPolygon(polyTemp);

  // move children elements, if available
  if (eParent.hasChild()){
    moveElementChildren(eParent, difx, dify);
    // System.out.println("called moveElementChildren()");
  }
}
```

Figure 42. Key Port Update Technique in Move Event Chain

The quick fix therefore was to remove cleanAllPorts() and updateElementPorts() method calls from the resize event loop and replace them with a new method. The new method, ElementData.moveElementPorts(), is a modified copy of the key move event loop method, ElementPortManager.moveNotNestedElement(), figure 18. This method translates all parts of an element on the same change vector. Because of this it can exist at parent class level. For a resize event there is no common translation of each point. The relative position of each port changes and this relative position is defined in each subclass of ElementPortManger. For this reason, moveElementPorts() must join updateElementPorts() as a required method for all subclasses of ElementPortManger.

```
// new method;  refit of updateElementPorts()
//   Used loop interior from MOVE loop, traced from
// ElementContainer.mouseDrag()
// case ConstantsStatus.MOVE_ELEMENT: leading to
// ElementPortManager.moveNotNestedElement(),
// which is -->aPortsOnElement[i].translate(difx,dify); .
//   The point calculation from updateElementPorts(), determines
// the correct port position.
//   Translate uses a difference in position vice a position
// requiring trading .move() for .tranlate() .
public void moveElementPorts()
{
  int iInputPortOffset = height/(iNoOfInputPorts+1);
  int iOutputPortOffset = height/(iNoOfOutputPorts+1);
  int iInputIndex=0, iOutputIndex=0;

  // insert new ports
  for(iInputIndex=0; iInputIndex<iNoOfInputPorts; iInputIndex++)
  {
    // iInputIndex represents the portNum and idNo represents the parent Element
on which the ports belong
    // cleanPort(iInputIndex);
    //aPortsOnElement[iInputIndex]                    =                    new
Port(ptStart.x,ptStart.y+(iInputPortOffset*(iInputIndex+1)),iInputIndex,
    //                    idNo,ConstantsRigal.iOUTPUT_PORT_TYPE_ID); //
strips ports
    //aPortsOnElement[i].translate(difx,dify);// GCP need to use move

aPortsOnElement[iInputIndex].move(ptStart.x,ptStart.y+(iInputPortOffset*(iInputIndex+
1)));// GCP coordinates defined in updatePorts()
  }

  for(iOutputIndex=0; iOutputIndex<iNoOfOutputPorts; iOutputIndex++)
  {
    //aPortsOnElement[iInputIndex+iOutputIndex]                =                new
Port(ptStart.x+width,ptStart.y+(iOutputPortOffset*(iOutputIndex+1)),
    //
iInputIndex+iOutputIndex,idNo,ConstantsRigal.iINPUT_PORT_TYPE_ID);   //   strips
ports

aPortsOnElement[iInputIndex+iOutputIndex].move(ptStart.x+width,ptStart.y+(iOutputP
ortOffset*(iOutputIndex+1)));// GCP as above
    iInputIndex = 0; iOutputIndex = 0;
  }
```

Figure 43. New Method moveElementPorts()

Because of this difference the translate call is replace by a move call. Port.move() is based on the Java Point.move() which changes the point coordinates to those given as arguments. But which new coordinates should be passed to move()? These were determined by updateElementPorts(), which needed to initially define these ports. In this version these formulas were copied from updateElementPorts() and pasted into the move invocation of moveElementPorts(). This solution is expedient but a poor one for maintainability. The formulas should be removed from both methods and made into one or more methods. This would centralize the behavior without which action the formula would have to be changed in two places, a recipe for disaster if only one change were made. This correction is currently future work.

7. Changes to Allow Saving with Restore as Implemented

The original development of this interface was progressing nicely when a new release of Java disabled the save functionality. With Java 1.4.2_02 saving created a sun.io.appContext.notSerializableException and failed to save any information but did create the desired file to save into. A subsequent open command returned an EOF exception due to nothing being saved to the file initially. Both of these exceptions were only visible within an IDE since they were reported by a system.println statement. A user running the program directly would only find themselves unable to open a previously saved diagram with no explanation.

A sun.io.appContext.notSerializableException appears to mean system dependent serializable exception. No documentation could be found on the Sun public web site but forum discussions requested help for these errors, which occurred after updating the users java runtime environment, are dated September ‘03. There were no further discussion and no answers posted. After failing to find a solution from Sun and by examination of the code for obvious problems in the save event loop the author resorted to experimentation.

Achieving success with the Sun java tutorial class SerializationDemo, a modified version of this class was inserted into the program package to successfully save an instance of an ElementContainer. Using the modified SerializationDemo as a foundation steps from the program save loop were added until it broke. This happened when the program tries to make method calls to a FileDialog meant for save operations. This lead

to belief there was a side effect to the method calls. Without this comparison experiment there never would have suspicion of the FileDialog methods.

With the many deprecated classes present in the program, there was a suspicion updating to a JFileChooser would help. It did not, the same situation remained. Since appContext, according to a google search, relates to the particular platform java virtual machine, it was suspected the context was not serializable. There was no desire to save the context but what if the context was somehow attached or related to the object being saved? Serialization allows an object to be saved by creating a standard binary representation of the object and its components. The fileDialog was declared as an attribute of the RigalFrame class. It did not need to be, in fact a local declaration would serve the purpose and take up insignificantly less memory but increase class cohesion. The fileDialog fdSaveDiagram was moved to a local variable since it is not an essential attribute but a utility class. Experimentation with this change produced successful save operations.

8. Inconsistent Saves or EOF on Open

During impromptu testing, sometimes a recently saved file would throw an end of file exception when attempting to restore it. This most likely occurred because the information had never been saved in the file.

```
try
{
  // oos.flush();
  oos.close();
}
```

Figure 44. Method flush() Disabled

Upon inspection the flush method was found commented out in the RigalFrame.handleEvent() save loop. Java commonly uses information buffers as intermediate pipelines for its I/O processing. An output command sends information to the buffer, but when the information is forwarded to a file is up to the host operating system. A flush method is provided to insist the buffer be processed immediately. It is common practice to flush prior to closing an output stream, otherwise the stream may be terminated with information left in the buffer producing inconsistent behavior.

Restoration of the flush command has produced no restore problems to date. But since this situation is beyond the scope of this work to replicate a test state, there can be no certainty of success.

9. Open with Title Restore

When "Open" was selected from the menu bar, a full restoration of the diagram state at the time of saving is expected. The 1.0 restore did not recover the original title. It was also inconsistent when restoring all elements in the diagram if the diagram was saved and then opened without reinitializing the program, some were left out. Immediately after saving there were picture artifacts on the mode buttons that might erase with use depending on the platform.

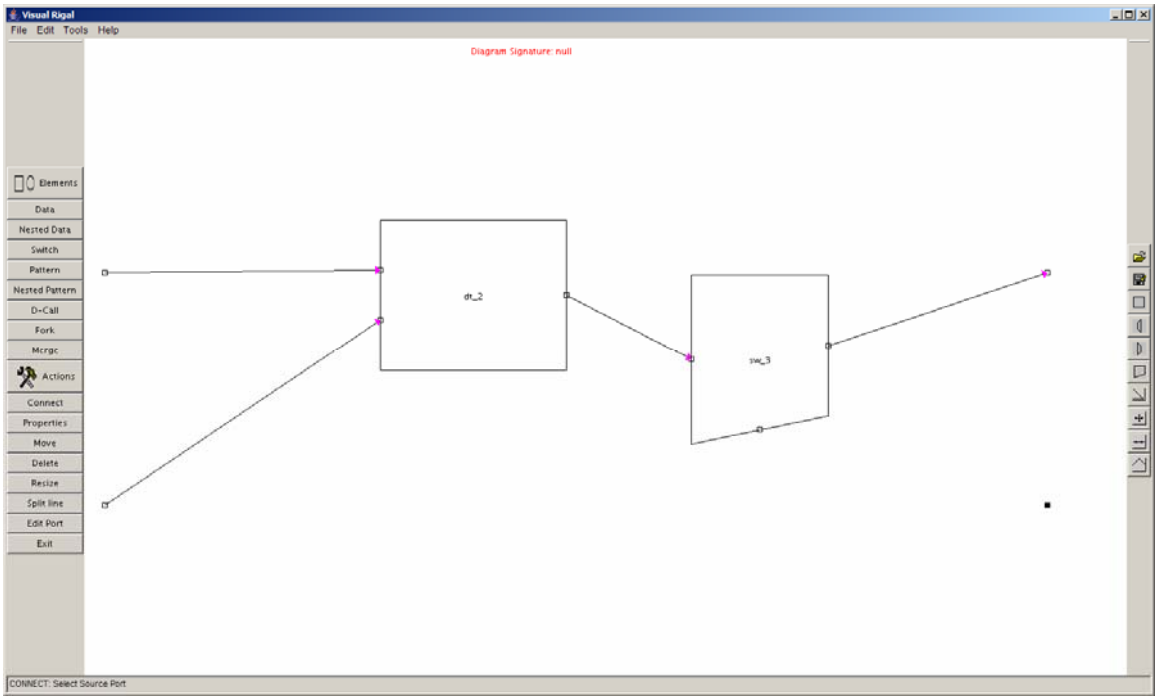


Figure 45. State of Diagram When Saved

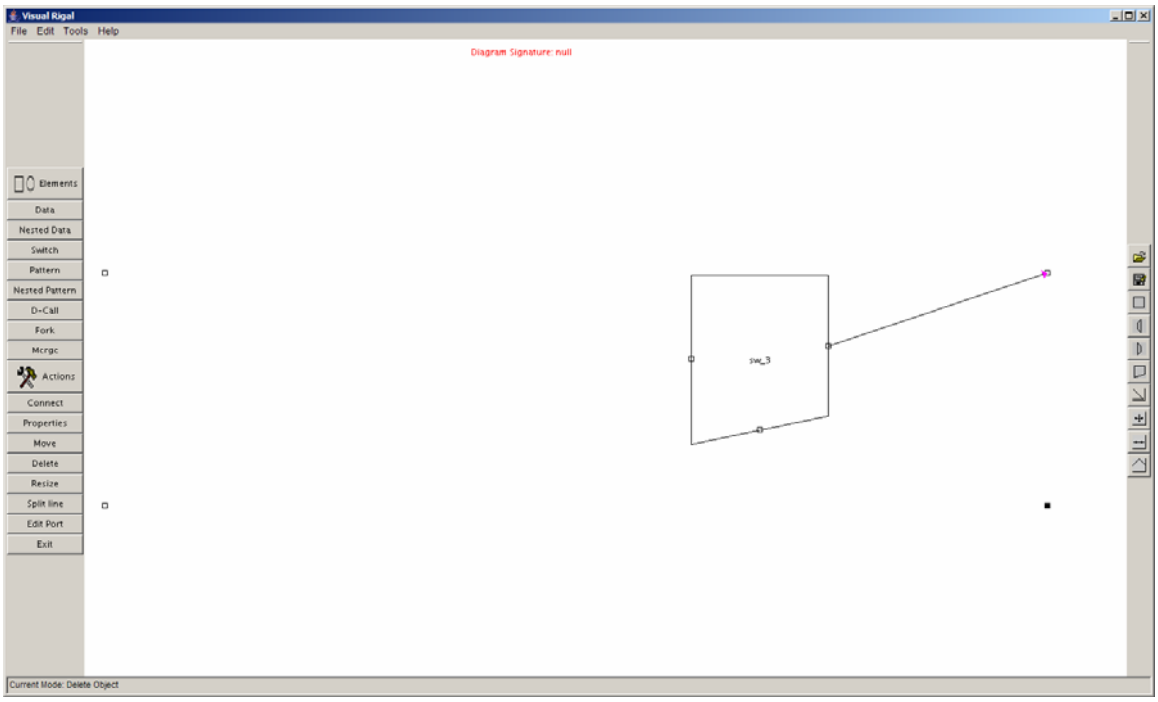


Figure 46. After Deleting Data Box

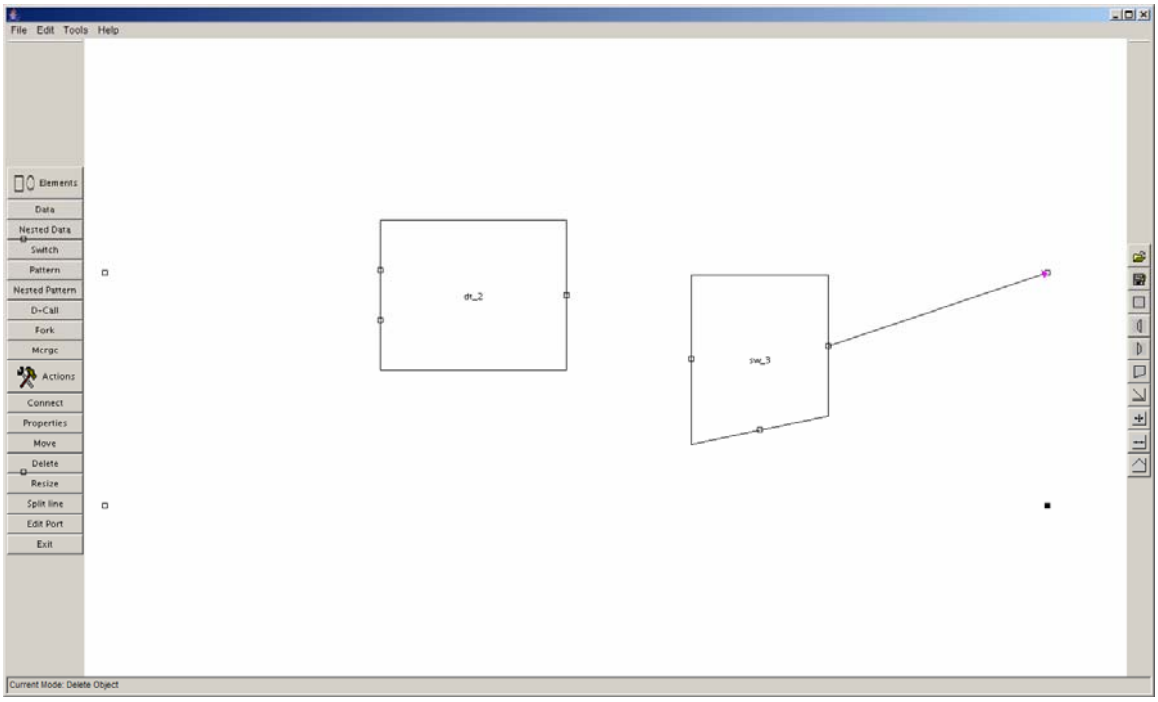


Figure 47. Immediatly After Opening the Saved Diagram

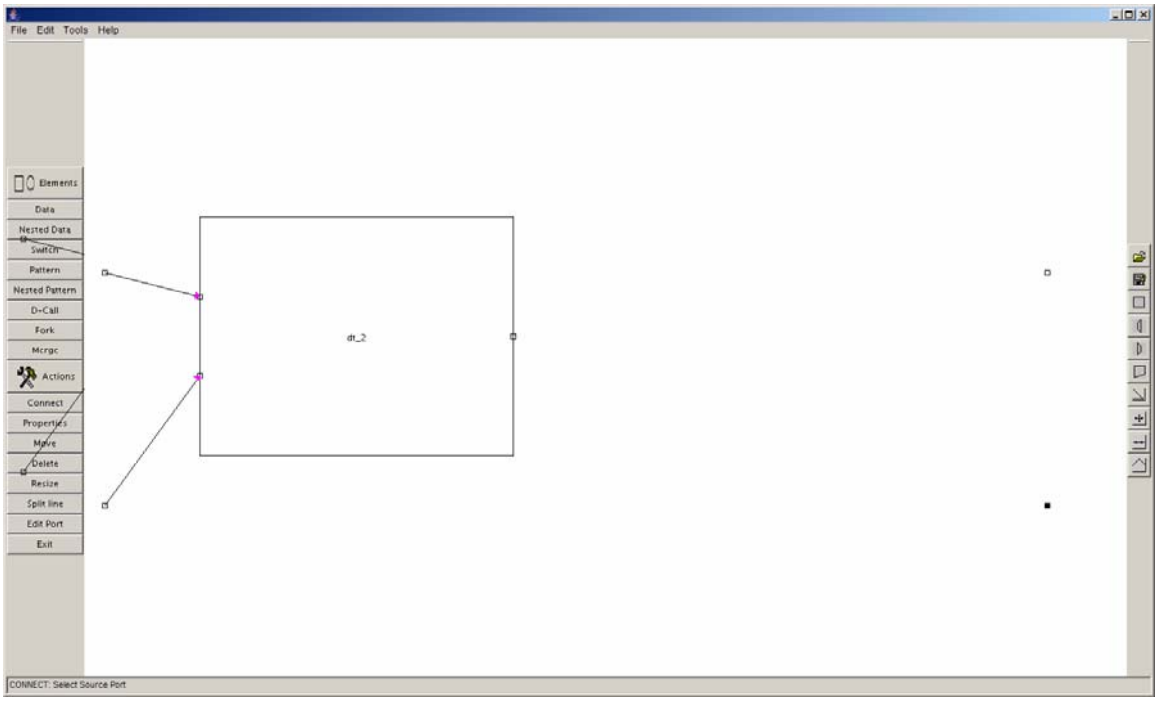


Figure 48. Diagram Restore Showing Artifacts on Mode Buttons

The problem seemed to be with imperfect management of ElementContainer details. An effort was made to clean up the single ElementContainer and prepare it for repopulation. Not everything was returned to the starting state. The saving of a diagram, pictured above, then deletion of a connected data box followed by opening of the saved diagram resulted in the pictured situation. This was due to content array manipulation and failure to fully reset the state. Extensive methods exist to clone program objects, that is make deep copies with the same information but different object handles in the Java language environment. Rather than unravel all of these techniques the program was changed to re-initialize the diagram.

With the repair of the serializable save feature there is no reason to try and manage the details of a diagram. An entire diagram can be directly saved to file as a single object. With this method corrupted diagram states have less of an impact since each is partitioned from any others. The possibility of direct diagram interaction is reduced. If a diagram is saved then an open command can discard the current diagram and reinstall the target diagram. The save function was edited to save an ElementContainer.

RigalFrame.restoreElementContainer() was modified so that now when opening a diagram everything is removed from the RigalFrame content pane and initialization behavior from the constructor is duplicated in loading the target ElementContainer from file. It is best practice to have this behavior consolidated but this is currently not the case. It is probable a large amount of the behavior can be consolidated but two different functions are being performed. One is creating new objects all together. The other is introducing a preexisting object into a new environment.

10. Clean New

When "New" was selected from the menu bar the diagram ports disappear. These ports are the visible part of an ElementPortManager which is stored in the first position of the diagram array. During cleanup operations it was removed with all of the other elements and never replaced. Following the model established above <REF>, RigalFrame.handleEvent() is modified to hide the content pane, initialize a new ElementContainer and set the pane visible again. The ElementContainer is initialized by calling a new method, freshElementContainer(). All ElementContainer initialization is moved from the RigalFrame constructor into this method and replaced with a method call. The method also checks to remove an old ElementContainer before starting fresh. Now initialization behavior resides in RigalFrame.handleEvent() and RigalFrame.freshElementContainer.

11. Restore Signature

Open always restores the diagram signature to the default, null, state. ElementContainer.paint() sets the signature to null if it needs to create a new properties dialog. Aside from the inadvisability of doing housekeeping in the paint() loop, if-else logic was added to allow restoration of a predefined signature.

12. Showing Nesting with Progressive Thickening

This is a new feature. It is unclear when looking at a diagram whether a data box is nested or simply overlapping. This change also allows the diagram to visually depict a data box's level of nesting. The convention chosen is to progressively thicken the box lines with each level of nesting. The root box would have a thickness of 1, the normal line thickness for a diagram element. The thickness of a nested box would be determined

by counting its tree distance from the root and adding this number to 1. An example shows, the thicker the lines the deeper the nesting.

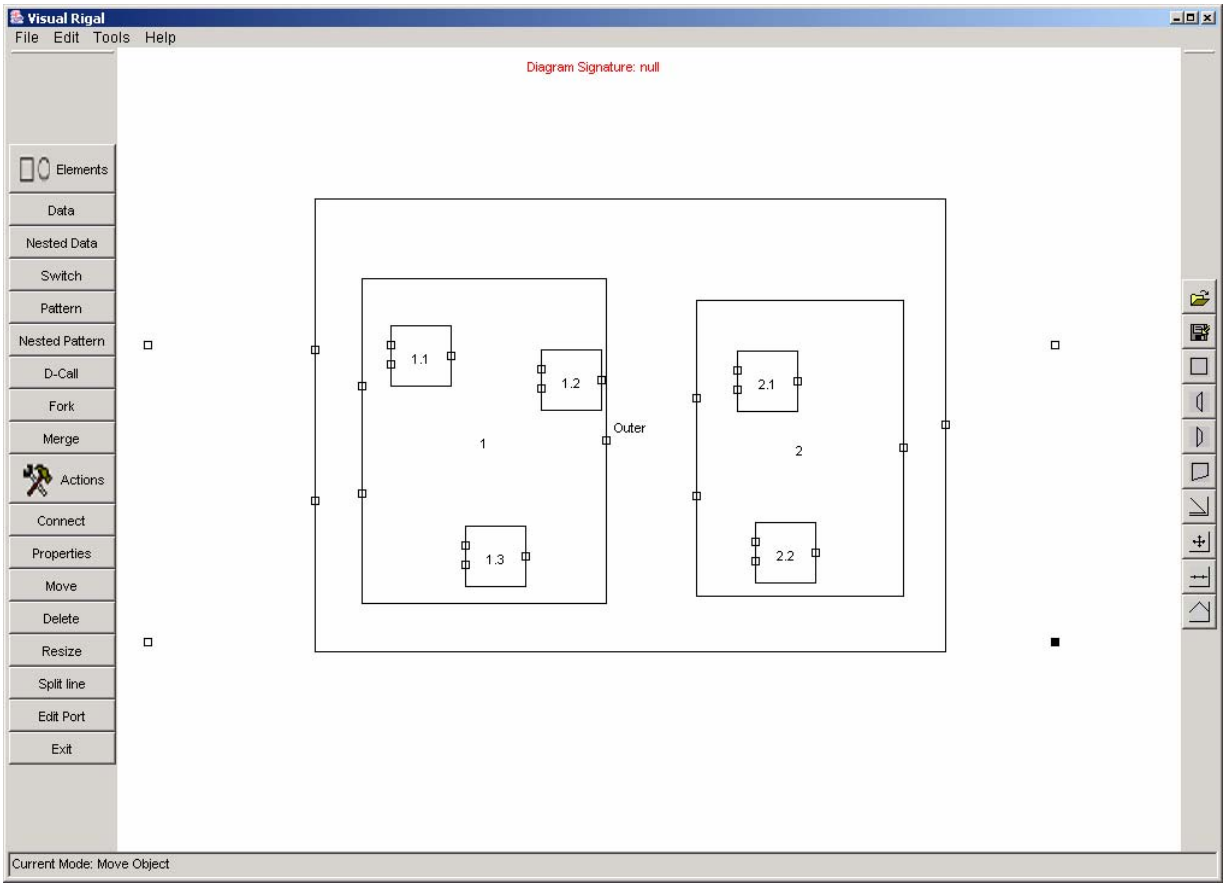


Figure 49. Nesting Without Line Thickening

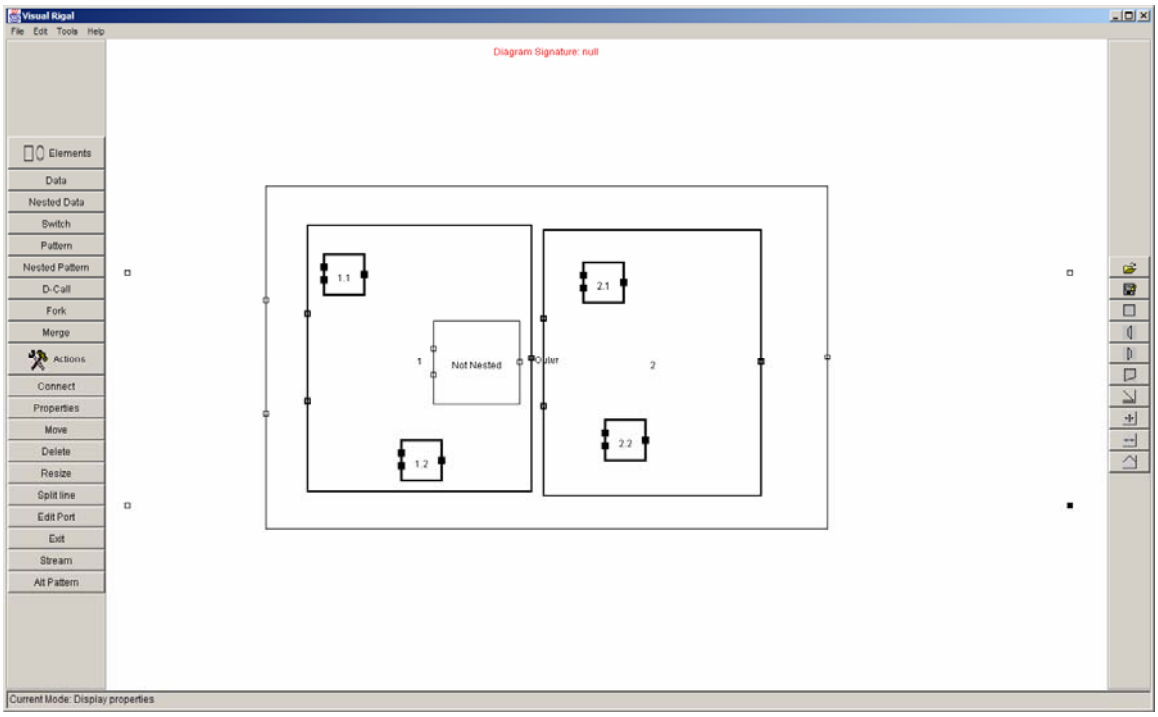


Figure 50. Nesting Without Line Thickening

With the use of Sun's ShapesDemo2D.java class for reference, it requires using a Graphics2D object vice a Graphics object. Components are drawn by the high level Element.paint(). Since Graphics2D inherits from Graphics, all legacy methods function the same. For this change, the Graphics object g is recast as a Graphics 2D and assigned to g2, then all drawing is done with g2. If the element is a data box then the default thickness of 1 is adjusted for the tree depth of the data box using the new method, elementTreeLevel().

```
public void paint(Graphics g)
{
 // System.out.println("*Element.paint(Graphics g) is called*");
 Polygon poly = new Polygon(polyPolygon.xpoints, polyPolygon.ypoints, polyPolygon.npoints);


 for (int i=0; i<poly.npoints-1; i++)
 {
  poly.xpoints[i] -= ecElementContainer.getOffsetX();
  poly.ypoints[i] -= ecElementContainer.getOffsetY();
  // g.drawLine(poly.xpoints[i], poly.ypoints[i], poly.xpoints[i+1], poly.ypoints[i+1]));
 }
 // g.drawPolyline(poly);
 // GCP line thickness control for nested nodes and stream connector
 Graphics2D g2 = (Graphics2D) g;// GCP switch graphics engine so we can control thickness

 int thickness = 1;// GCP default thickness
 if (this instanceof ElementData) {// GCP special case for nested thickness
  thickness += elementTreeLevel( 0 , ecElementContainer );// GCP add additional thickness if
needed
 }

 BasicStroke wideStroke = new BasicStroke ( (float) thickness );// GCP select line width to
show nesting
 g2.setStroke( wideStroke);// GCP set line width to show nesting

 g2.drawPolyline(polyPolygon.xpoints, polyPolygon.ypoints, polyPolygon.npoints);// GCP
change to g2
}
```

Figure 51. New paint() Method to Allow Line Thickness Change

```
/** Recursive method to find a nested element depth from its root parent or; the
 * number obtained by starting from 0 with the outer element and counting in to the
 * nested element.  Used for determining nested element line thickness.  Initially called with
 * 0 or result inaccurate.
 * @return distance from parent of this nested element
 * @param total Seed value of zero, used to collect number of levels
 * @param tempEC Element container passed to save recursive queries for same info.
 */
private int elementTreeLevel( int total , ElementContainer tempEC ) {
    // Final parent drops through and returns with no change
    if ( this.getNested()) {// others call immediate parent recursively
      int parentID = this.getNestParentId();// get parent
      Element parent = tempEC.getElementWithId( tempEC , parentID );
      // find level of parent
      // add 1 for this instance and pass it back
      total = parent.elementTreeLevel( 0 , tempEC ) + 1 ;
    }
    return total;
}//end elementTreeLevel()
```

Figure 52. New Method elementTreeLevel()

The elementTreeLevel() method is a recursive method that returns 0 for a root element or the depth for a non-root element. This value is simply added to the thickness and the polygon drawn normally. This is equivalent to using a thicker pen to draw between the same two points.

This change breaks encapsulation. A paint() method should be added to elements requiring a Graphics2D object. This method can pass the Graphics2D object to Element.paint() with a super.paint() call.

13. Twilight Nesting

This is a fix for a special situation found during testing. If a nested data box is initially placed so it is not enclosed by the parent box, it enters a twilight region of nesting. It is nested but not nested in anything. A variety of troublesome behavior can follow from further manipulation of this twilight nested box. If it is moved fully within any other box, the program senses a nested box and will not allow it to be moved out of the enclosing box. This can further increase the illusion of being nested. However, the twilight box will not move with the parent and will never print out. To this last, the box is nested so it needs a parent to print out but has none. This breaks the contract of diagram matching code and this overlapping should not be allowed. It is also a low priority on the list of fixes and features. The previously thickened lines for nested boxes will help identify this situation but the twilight situation can be easily fixed.

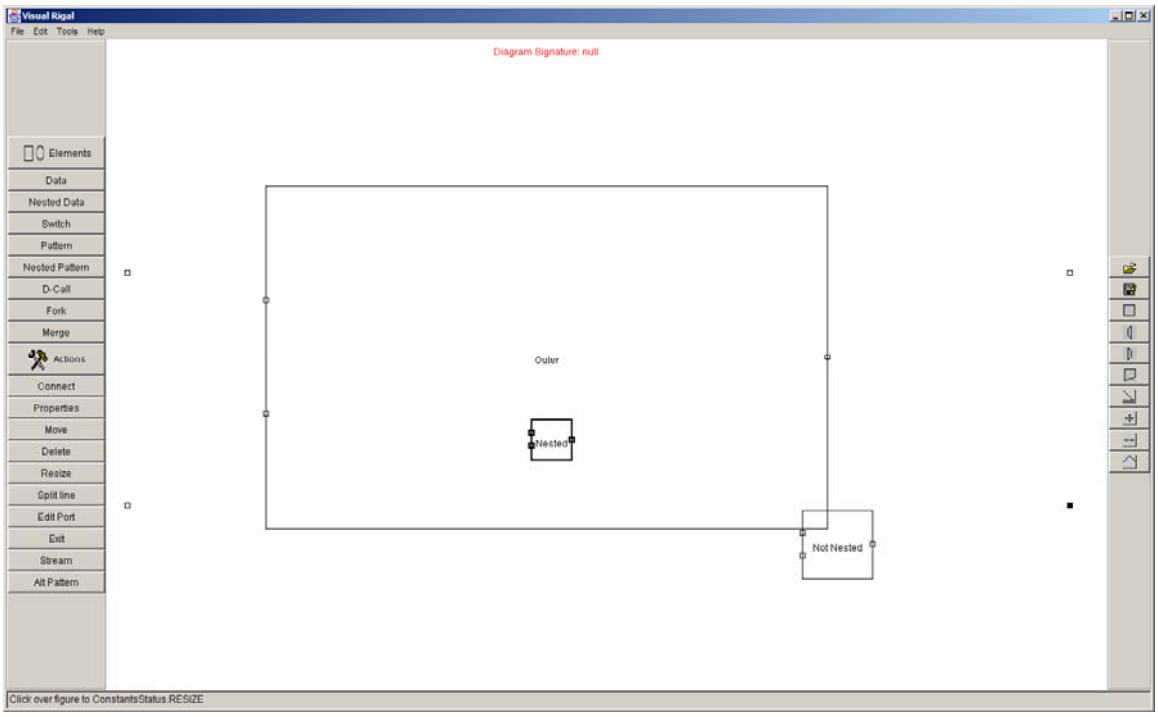


Figure 53. Twilight Nesting Example

ElementContainer.processNestedElement() sets the nested flag before checking who is the box's parent. The parent is determined by referencing the diagram to see which component encloses the target box. Since no boxes meet this condition there is no parent. This method takes advantage of the information stored as the diagram and strengthens the contract of equivalence between the diagram and the code. If there is a parent all of the nested attributes are appropriately assigned. The quick fix is to move the nested flag change into this conditional statement. Now there can be no more twilight nested boxes.

```
public void processNestedElement(Element eTemp){// Assumes nested
  Element eNest = null;
  int iNestId = getNestId(eTemp);
  //eTemp.setNested(true);// GCP original location, always set to nested
  eTemp.setNestId(iNestId);  // called once
  System.out.println("getNestId(eTemp): " + iNestId); // called twice
  eNest = getNestElement(eTemp); // called thrice
  System.out.println("eNest: " + eNest);
  if(eNest != null){
    eTemp.setNested(true);// GCP more correct place, only nested if there is a
parent
    eNest.getNestVector().add(eTemp);
    eNest.setHasChild(true);
  }
 }//end processNestedElement()
```

Figure 54. Cause of Twilight Nesting

14. Adding Streams

This is a new feature to add the stream icons to the available diagram elements and expand the language. The quickest way to achieve this goal is by extending, or inheriting from, the ElementConnector class. Both elements are to have very similar behavior, as it turns out only the graphical methods need to be overridden.

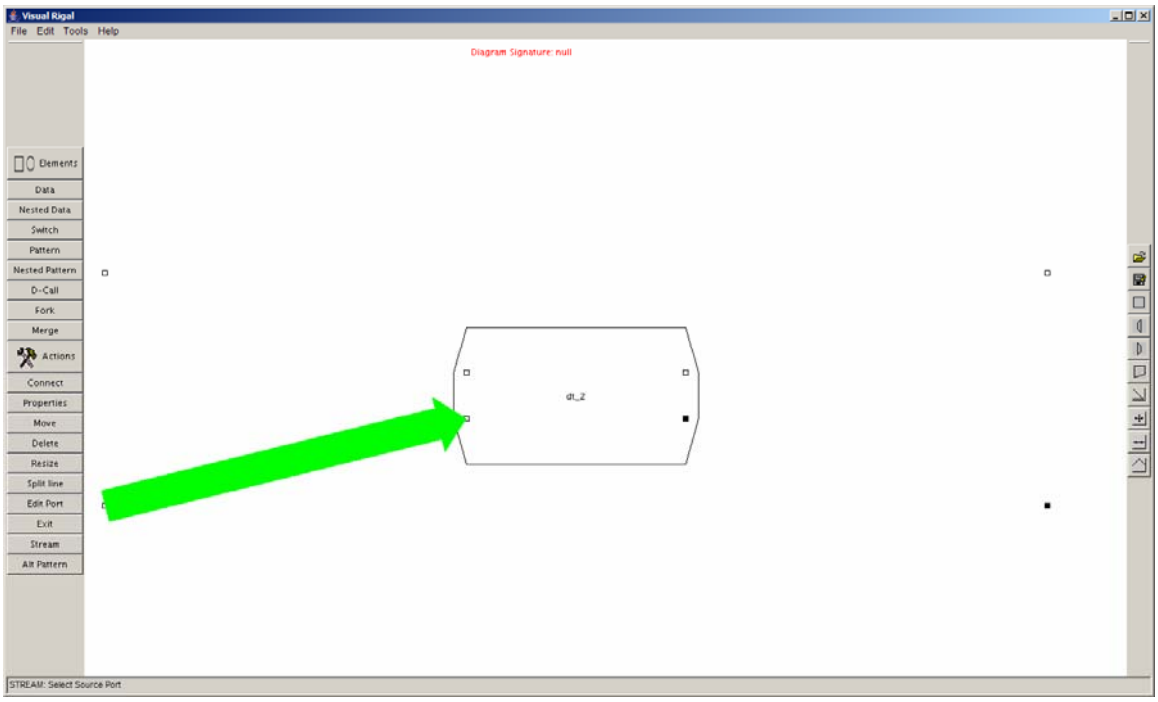


Figure 55. Example of a Stream Connector

As simple as this sounds the following table shows the changes needed to make this inheritance shortcut successful. These locations were found using a package search for “connect.”

Change in	For purpose of
ToolBoxElementandActions	Stream button
ConstantsStatus	Stream mode identification number
ElementContainer.setEditorStatusBar()	Stream status bar messages
ElementContainer.mouseDown()	Same behavior for streams and connectors with appropriate status messages for each
ElementContainer.mouseUp()	Add streams with same behavior as connectors
ElementContainer.mouseMove()	Add stream behavior
ConstantsElement	Identification number
Element.paint()	If stream then set color and thickness
ElementContainer.getText()	Prints connectors add streams
ElementContainer.addConnectorElement()	Allow streams

Table 1. Changes Required to Support Streams

This opportunity was also taken to remove redundant drawing routine in ElementPortManager.paint(). This was only significant for connectors since most elements are drawn by the Element.paint() method. Connectors must draw their own arrowheads. For these elements the drawing hierarchy is Element.paint() → ElementPortManger.paint() → ElementStream.paint(). The ElementPortManger method is only needed as a bridge along the hierarchy. The original code is left intact for illustration and is indicated by the author's bold initials, GCP.

```
public void paint(Graphics g)
{
  super.paint(g);
  //System.out.println("**ElementPortManager.paint(g) is called**");
  //System.out.println("polyPolygon: " + polyPolygon);

//   if ()
//  {

  //}
  //else
  // GCP redundant draw routine if (polyPolygon != null)
  // GCP {
  // GCP   //System.out.println("polyPolygon is NULL");
  // GCP   for (int i=1; i<polyPolygon.npoints; i++)
  // GCP   {
  //    GCP                UtilGraphics.drawLine(g,polyPolygon.xpoints[i-1]    -
ecElementContainer.getOffsetX()
  //  GCP                                                ,polyPolygon.ypoints[i-1]  -
ecElementContainer.getOffsetY(),
  //  GCP                                                  polyPolygon.xpoints[i]  -
ecElementContainer.getOffsetX(),
  //  GCP                                                  polyPolygon.ypoints[i]  -
ecElementContainer.getOffsetY(), iThick);
  // GCP   }
  // GCP }
  //if (ecElementContainer.getMode() == ConstantsStatus.CONNECT ||
        …
```

Figure 56. Duplicated Drawing Routine

15. Adding Alternative Patterns

This is a new feature to add the alternative pattern icons to the available diagram elements and expand the language. Inheritance is again leveraged for this task, as was done for stream icons. The paint() and printTest() methods needed to be overridden as before. The alternative pattern looks different from a fork but has the same port behavior graphically. The only other difference for the GUI is the text file interface representation, despite the vastly different icon meaning. This is possible because executable icon behavior is contained in the VisualRigal parser, which is already implemented, by architectural design. This is an example of the benefits a formal design might have brought to the GUI application.

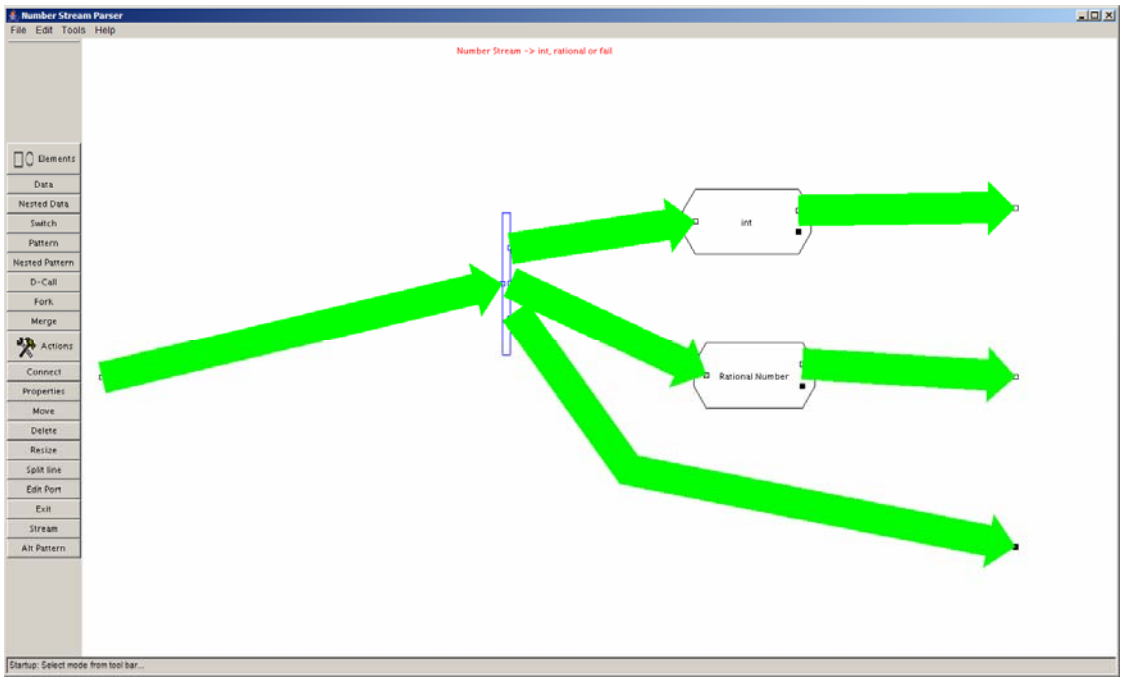


Figure 57. Example of an Alternative Pattern

Greater use of inheritance was made in this case. Many internal checks for proper behavior of an alternative pattern actually reference the super class ElementFork. A comparison of the following table with table 1 shows four less places needing change. Five of the six remaining changes simply support display of the correct status bar message. More efficency could have been attained if the status bar behavior were designed to be encapsulated within each class.

Change in	**For purpose of**
ToolBoxElementandActions	AltPattern button
ConstantsStatus	AltPattern mode identification number
ElementContainer.setEditorStatusBar()	Alt_Pattern status bar messages
ElementContainer.mouseDown()	Same behavior for alternative patterns and forks
ElementContainer.mouseUp()	Add alternative patterns with same behavior as forks
ElementContainer.mouseDrag()	Add alternative patterns with same behavior as forks

Table 2. Changes Required to Support Alternative Patterns

E. TEST PLAN

1. Assumptions

GUI testing was not be automated because of resource constraints. Automated testing of code was available with the use of Parasoft's JTest tool.

2. Methodology

The test plan is composed of testing in the two following areas:

a. Black Box Testing

Black box testing encompasses those tests done without reference to implementation details. This portion of the test plan was focused on testing support application requirements defined in the requirements analysis in appendix D. Test cases consist of a series of GUI actions designed to exercise all icons and modes at least once. No attempt was made to exercise all combinations or possible input series.

b. White Box Testing

White box testing encompasses those tests done with reference to implementation details. The primary portion of this part of the test plan will rely on automated testing using JTest. These tests were supplemented by test cases created with application design in mind.

F. TESTING RESULTS

1. Automated Test Results and Corrections

JTest checked all 47 classes and 12,141 lines of code and found violations of 29 Parasoft "Must Have Rules". These violations represent a total of 833 errors, 740 of which were corrected using JTest's quick fix feature. The JTest test report is available in appendix A for review.

2. Manual Test Results

a. Deleted Nested Box Still Reported in Text File Interface

Testing found nested boxes that were deleted appeared in the text file interface as if they had never been deleted. This violates the implied contract to maintain equivalence between the visual and logical depiction of code.

b. Diagram Signature is Not Reported in Text File Interface

The diagram signature never appears in the text file.

c. Resize of Elements Other Than Data Boxes Removes Connections

All elements, other than data boxes, loose their connections when resized. This is the same situation described previously for data boxes, prior to the correction described above.

d. Unable to Change Number of Element Ports

The number of output ports for the fork and alternative pattern elements and the input ports for the merge element do not change despite initiating a change in the properties dialog.

e. Diagram Element is Deleted While Adding Another

After executing a test script for several actions, the addition of a connector removes the last element previously added to the diagram.

G. CORRECTIONS BASED ON TESTING BEHAVIOR

1. Corrections to Deleting Nested Box Behavior

The application deletion routine, ElementContainer.deleteObjectWithId(), removed the target element's reference from the element array which holds all elements in the diagram, ElementContainer.aElementsInElementContainer. However, this routine leaves the same reference in the parent child vector, Element.vNestVector. The diagram element array is used to display elements on the screen. The parent child vector is used to print a logical representation of nested elements to the interface text file.

The deletion routine was modified to remove the child reference, and then further modified to be a tree node deletion routine. That is, if the target nested element is a parent itself, the children are introduced to the grandparent and become the grandparent's children.

```
public synchronized void deleteObjectWithId(int id)
{
 int i;
 for (i=0; i<iIndexOfElementsInElementContainer; i++)
 {
  if (aElementsInElementContainer[i].getId() == id)
  {
   if( aElementsInElementContainer[i].getNested() ){//GCP if nested
    Element eParentOfDeleting = aElementsInElementContainer[i].getNestElement();
    Vector vParentVector = eParentOfDeleting.getNestVector();
    Element eDeletingElement = getElementWithId( this,id );
    vParentVector.remove( eDeletingElement );//GCP remove nest reference in parent

    Vector vVectorOfChildren = eDeletingElement.getNestVector();
    Iterator iterChildVector = vVectorOfChildren.iterator();
    Element eChild = null;

    while ( iterChildVector.hasNext() ) {//GCP add children to  grand-parent
     eChild = (Element) iterChildVector.next();
     vParentVector.add( eChild );
     eChild.setNestId( eParentOfDeleting.getId() );
    }
     //GCP reorder vParentVector
   }else{//GCP top level element
     if ( aElementsInElementContainer[i].hasChild() ) {//GCP with children

      Element eDeletingElement = getElementWithId( this,id );
      Iterator vDeletingChildren = eDeletingElement.getNestVector().iterator();
      Element eElement = null;//GCP holding site
      //make them top level elements
      while ( vDeletingChildren.hasNext() ) {

            eElement =  (Element) vDeletingChildren.next();
       eElement.setNested( false );
       eElement.setNestId( -1 );
      }//end while

     }//end if has child

   }//end if-else

      aElementsInElementContainer[i] = null;
      break;
  }// end if id
 }// end for i

 if (aElementsInElementContainer[i] == null)
```

Figure 58. Changes to Element Deletion Routine

2. Correction to Have Diagram Signature Reported in Text File Interface

The signature was added in the reporting method right after the title is reported.

```
public void getText(PrintStream ps)
{
  ps.println("name      " + this.getCaption());
  ps.println("signature (" + this.getSignature()+ ")");//GCP add signature
  ps.println("");
  ps.println("input_ports  ("+ncp.getInputPortIds()+")");
```

Figure 59. Change to Report Signature

3. Correction to Change number of Element Ports on Fork, Merge and Alternative Pattern

This behavior was commented out for fork and merge elements. This was probably for previous troubleshooting. Each line of code was restored to correct this fault.

4. Correction to Prevent Diagram Element is Deletion While Adding a Different Element

This was an insidious logic fault since the logical failure is not initially apparent. When an element is targeted for deletion there are two possible paths, one for non-connector elements and one for connectors.

```
if (eElement instanceof ElementPortManager)       {
  if (((ElementPortManager)eElement).isConnected())         {
    ((ElementPortManager)eElement).disconnect();
  }
  deleteObjectWithId(eElement.getId());
  iHighlitedLine=-1;
  iMatchLine = -1;
  bMoved=true;
  repaint();
 }
else if (eElement instanceof ElementConnector)       {
    ((ElementConnector)eElement).disconnect();
    iHighlitedLine=-1;
    iMatchLine = -1;
    bMoved=true;
    repaint();  // this calls the update(Graphics) method and this method has
          // been overridden in ElementContainer.java
  }
```

Figure 60. Two Possible Branches for Element Deletion

The first path is entered based on the conditional statement (eElement instanceof ElementPortManager) where eElement is the target element. If eElement inherites from ElementPortManager this is evaluates to true. The second is based on (eElement instanceof ElementConnector) which evaluates to true if eElement is an ElementConnector. The problem arises because connectors inherit from EleemntPortManger as shown in the application UML diagram, figure 20. For this

reason the second branch can never execute. The only difference in the two branches is the conditional disconnect action. Connectors are by definition connected so this attribute is never set for them.

When a connector is deleted, the reference is removed from the ElementContainer.aElementsInElementContainer array preventing the connector from being drawn. Because the connector is never disconnected from its ports, the ports maintain the connected state with reference to the connector. A new element will occupy the last position in the ElementContainer.aElementsInElementContainer array. If a subsequent connection is made to one of the ports maintaining reference to the deleted connector, the first step is to disconnect and delete this connector. Since it is already absent from the element array above, the element at the same index position is deleted. Because the initial fault occurs several steps before a problem is apparent there is additional difficulty localizing the it.

The first branch conditional is modified to exclude connector elements restoring the intended functionality, as shown.

if (eElement instanceof ElementPortManager && !(eElement instanceof ElementConnector)){// **GCP correct connector delete removes last in diagram replace by connector**

Figure 61. Correction for Inadvertent Deletion When Connecting

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CONCLUSIONS

A. ASSESSMENT OF IMPLEMENTATION

1. Strengths

a. Defacto Design is Similar to Design Produced by Requirements Analysis

While formal design methodology was not utilized in the initial work, the defacto design, shown in figure 20, is very similar to a design from the formal process, figure 19. The intuitive solution produced by an experienced programmer is remarkable similar to a deliberate design. This is not surprising, since they are both attempt to solve the same problem. The common parts of the design allowed relatively easy extension to the application.

b. Support of Equivalence Contract with Java Component Abstract Interface

The use of abstract interface java.awt.Component for visual language elements promotes closer equivalence between the graphical and logic representations. Java's graphical routines are based on displaying Components. The same class attributes and methods used to draw the Component are used in processing events targeted on their manipulation. When a text interface is created the class attributes and methods are again involved.

2. Weaknesses

a. Class Interfaces Not Fully Developed for Extension

Trouble for maintenance and extension lurks where the defacto design and the formally derived design differ. Indicative of this situation is the unexpected deletion of an element when adding a connector. The defacto design classifies a connector as a type of element, on the same level as a data box. The original author clearly thought of connectors as separate from elements as witnessed by his expectation for ElementPortManagers to be exclusive of connectors. By distinguishing between elements, or nodes as in figure 19, the original deletion solution would have worked as coded.

3. Lacks Documentation

The difficulty caused by lack of documentation have been well documented previously, and this fact is mentioned here only for completeness. This problem has been corrected to a great extent by this work.

B. LESSONS RELEARNED

During the course of correcting, extending and documenting 11,184 legacy lines of code(LOC) to support future maintenance adding an additional 957 LOC and changes to 45 LOC, the following basic software engineering priciples were illustrated:

1. Formal Design Improves Maintainability

Formal design fully examines the application at a high level of abstraction. This has the result of considering functionality that may not be initially implemented. Later, when new functionality is added there is a higher likelihood the design will support it. A formal design is intended to be a general solution to the problem. To the extent the problem has been fully described, the design should accommodate future extension.

2. Documented Code Improves Maintainability

The difficulty posed by trying to understand code lightly documented has been elaborated on by examples above. A formal design process will produce some useful documentation, figure 19, but using code comments to illuminate the reason for using particular technique can greatly improve comprehension time which, in turn, improves maintainability.

3. Understanding of Design and Requirements Improves Extension Solutions

The authors solution to the addition of stream elements required a variety of changes throughout the code. As understanding of the design and implementation increased it became clear the simpler solution used in adding alternative patterns is possible. Any time that can be saved understanding a large piece of software can be translated into efficient maintenance and extension solutions.

4. Object Oriented Design Improves Diagram to Interface Link

Java components serve as a model for diagram elements. This model is observed by its graphical representation of a screen and its logical representation in the interface text file. If both representations use the same information to create them and changes to

that information are controlled by the model, there is a good likelihood of equivalence. Encapsulation of the information and its manipulation methods is one of the principals of object oriented design.

5. Some Testing is Better Than None

Though it is likely not possible to fully test an application, some reasoned priority of testing can produce excellent results. By asking, "How can I test under these constraints?" instead of, "Should I test under these constraints?" the testing faults discussed above were found.

6. Testing is Necessary to Expose Unexpected Behavior

The relatively long sequence of events required to expose the inadvertent deletion fault made it a challenging fault to detect. This behavior could never have been intended. The solution creating it would have worked given an alternate design, discussed under weaknesses above. Only through formal testing can a fault of this nature be found before a user or customer find it for you.

C. FUTURE WORK

1. Update Code

The ElementData resize solution needs to be transferred to other elements. The solution to maintain connections is generalized previously, however, each element requires a unique solution. This is a time consuming cut and paste exercise but was not completed by this work.

There are many deprecated methods used throughout the code and as it ages this problem will increase. Some of the methods needing updating will provide challenging design considerations. For example all of the mouse____() methods have been subsumed by the handleEvent() method. A clever programmer might change the names to handleMouse___() and use handleEvent() to farm out the work, avoiding a giant method difficult to maintain.

2. Add New VisualRigal Elements

There are several additional VisualRigal elements that are not implemented by previous work. The stream and alternative pattern elements were implemented by this work.

3. Adaptation of VisualRigal to the General Programming Domain

While some elements of VisualRigal would be useful to illustrate general programming constructs, the conditional switch for example, the language is design for a specialized domain. The emphasis on fundamental data structures does offer potential for more general application, but this must be demonstrated.

LIST OF REFERENCES

[1] M. Auguston, V. Berzins, B. Bryant, ”Visual Meta-Programming Language“, Proceedings of OOPSLA 2001 Workshop on Domain-Specific Visual Languages, 2001, pp. 69-82

[2] N. Shu, “Visual Programming Languages: A Perspective and a Dimensional Analysis”, Visual Programming Environments: Paradigms and Systems, IEEE Computer Society Press, 1990, p. 41

[3] Conversations with NPS student Major Robert Taylor, USMC, 2004

[4] L. Belady, C. Evangelisti, L. Power, “GREENPRINT: A Graphic Representation of Structures Programs”, Visual Programming Environments: Paradigms and Systems, IEEE Computer Society Press, 1990, p. 79

[5] Conversations with M. Auguston, NPS, 2004

[6] P. Cox, T. Pietrzykowski, “Using a Pictorial Representation to Combine Dataflow and Object-Orientation in a Language independent Programming Mechanism”, Visual Programming Environments: Paradigms and Systems, IEEE Computer Society Press, 1990, p. 313

[7] M. Boshernitsan, M. Downes, “Visual Programming Languages: A Survey”, University of California, Berkeley, 1997, p. 3

[8] IEEE Software Magazine, Volume 20 Issue 5, 2003

[9] M. Auguston, Unpublished Presentations Slides for “Control Constructs in Visual Meta-Programming Language”, Proceedings of the Tenth International Conference on Distributed Multimedia Systems, 2004

[10] D. Spinellis, “Code Reading”, Addison-Wesley, 2003,

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIXES

A. TEST REPORT FOR TESTING PARASOFT MUST HAVE RULES

04/09/12 19:25:33	CODING STANDARDS

CODING STANDARDS

Project name	Errors qfix / total		Files checked / total		Lines checked / total	
Thesis	740	833	47	47	12141	12141
Total [0:00:32]	**740**	**833**	**47**	**47**	**12141**	**12141**

Errors Summary by:CategorySeverity

[**230**] **Global Static Analysis** (GLOBAL)
- [**27**] Avoid globally unused package-private fields. (UPAF-1) qfix
- [**1**] Declare package-private classes as inaccessible as possible. (DPAC-1) qfix
- [**2**] Declare package-private methods as inaccessible as possible. (DPAM-1) qfix
- [**200**] Declare package-private fields as inaccessible as possible. (DPAF-1) qfix

[**3**] **Security** (SECURITY)
- [**3**] Make all inner classes "private". (INNER-1)

[**2**] **Miscellaneous** (MISC)
- [**1**] Avoid non-public classes with "public" constructors. (PCTOR-2) qfix
- [**1**] Declare "private" constant fields "final". (FF-1) qfix

[**112**] **Optimization** (OPT)
- [**2**] Avoid unnecessary "instanceof" evaluations. (UISO-1)
- [**64**] Use abbreviated assignment operators. (AAS-3) qfix
- [**4**] Close input and output resources in "finally" blocks. (CIO-1)
- [**42**] Avoid unnecessary casting. (UNC-1) qfix

[**3**] **Javadoc Comments** (JAVADOC)
- [**3**] Provide Javadoc comments for "public" classes and interfaces. (PJDCC-1) qfix

[9] **Class Metrics** (METRICS)
[9] Cyclomatic Complexity. (TCC-2)
[109] **Possible Bugs** (PB)
[63] Avoid casting primitive data types to lower precision. (CLP-2)
[6] Avoid comparing floating point types. (DCF-2)
[11] Avoid a "switch" statement with a bad "case". (SBC-3) qfix
[22] Avoid using "+" on Strings to concatenate instead of add numbers. (DCP-3) qfix
[7] Use 'equals ()' when comparing Objects. (UEI-3) qfix
[156] **Coding Conventions** (CODSTA)
[156] Avoid using literal constants. (USN-2) qfix
[3] **Garbage Collection** (GC)
[3] Avoid potential memory leaks in ObjectOutputStreams by calling 'reset ()'. (OSTM-2)
[48] **Object Oriented Programming** (OOP)
[17] Avoid "public" instance fields. (APF-2) qfix
[30] Avoid hiding inherited instance fields. (AHF-1) qfix
[1] Avoid overriding an instance "private" method. (OPM-2) qfix
[123] **Unused Code** (UC)
[75] Avoid unused "import" statements. (UIMPORT-2) qfix
[3] Avoid assignments to variables that are never read. (AVNR-2)
[1] Avoid unused "private" fields. (PF-2) qfix
[43] Avoid unused local variables. (AUV-2) qfix
[1] Avoid unused "private" methods. (PM-2) qfix
[35] **Naming Conventions** (NAMING)
[35] Avoid lowercase letters in "final" "static" field names. (USF-2) qfix
[315] **Severity 1**
[230] **Global Static Analysis** (GLOBAL)
[27] Avoid globally unused package-private fields. (UPAF-1) qfix
[1] Declare package-private classes as inaccessible as possible. (DPAC-1) qfix
[2] Declare package-private methods as inaccessible as possible. (DPAM-1) qfix
[200] Declare package-private fields as inaccessible as possible. (DPAF-1) qfix
[3] **Security** (SECURITY)
[3] Make all inner classes "private". (INNER-1)
[1] **Miscellaneous** (MISC)
[1] Declare "private" constant fields "final". (FF-1) qfix

[48] **Optimization** (OPT)
[2] Avoid unnecessary "instanceof" evaluations. (UISO-1)
[4] Close input and output resources in "finally" blocks. (CIO-1)
[42] Avoid unnecessary casting. (UNC-1) qfix
[3] **Javadoc Comments** (JAVADOC)
[3] Provide Javadoc comments for "public" classes and interfaces. (PJDCC-1) qfix
[30] **Object Oriented Programming** (OOP)
[30] Avoid hiding inherited instance fields. (AHF-1) qfix
[414] **Severity 2**
[1] **Miscellaneous** (MISC)
[1] Avoid non-public classes with "public" constructors. (PCTOR-2) qfix
[9] **Class Metrics** (METRICS)
[9] Cyclomatic Complexity. (TCC-2)
[69] **Possible Bugs** (PB)
[63] Avoid casting primitive data types to lower precision. (CLP-2)
[6] Avoid comparing floating point types. (DCF-2)
[156] **Coding Conventions** (CODSTA)
[156] Avoid using literal constants. (USN-2) qfix
[3] **Garbage Collection** (GC)
[3] Avoid potential memory leaks in ObjectOutputStreams by calling 'reset ()'. (OSTM-2)
[18] **Object Oriented Programming** (OOP)
[17] Avoid "public" instance fields. (APF-2) qfix
[1] Avoid overriding an instance "private" method. (OPM-2) qfix
[123] **Unused Code** (UC)
[75] Avoid unused "import" statements. (UIMPORT-2) qfix
[3] Avoid assignments to variables that are never read. (AVNR-2)
[1] Avoid unused "private" fields. (PF-2) qfix
[43] Avoid unused local variables. (AUV-2) qfix
[1] Avoid unused "private" methods. (PM-2) qfix
[35] **Naming Conventions** (NAMING)
[35] Avoid lowercase letters in "final" "static" field names. (USF-2) qfix
[104] **Severity 3**
[64] **Optimization** (OPT)

[64] Use abbreviated assignment operators. (AAS-3) qfix

[40] **Possible Bugs** (PB)

[11] Avoid a "switch" statement with a bad "case". (SBC-3) qfix

[22] Avoid using "+" on Strings to concatenate instead of add numbers. (DCP-3) qfix

[7] Use 'equals ()' when comparing Objects. (UEI-3) qfix

B. CHANGES TO ORIGINAL CODE

This appendix represents all changes to the original code. It is provided as a reference. It consists of a list of solution titles followed by code segments. The class and method of the code are indicated and a cut and paste operation is expected. Note the use of elipses, **…**, for large methods to indicate several lines of code not shown. In such cases enough code remains to correctly identify the position of the code segment. Bold face contents containing the authors initials, GCP, indicate the key location for the change.

Contents:

Fix for 3 compiler errors in ElementPortManager:

```
public Element getElement()
{
  return (Element) this;// GCP changed super() to (Element) this
}

 public void move(int x, int y)
 {
   // move mother element
   if (super.getNested()){
      moveNestedElement((Element) this, x, y);// GCP changed from super() to
(Element) this
      // System.out.println("called moveNestedElementOnly()");
   }
   else if (!super.getNested()){
      moveNotNestedElement((Element) this, x, y);// GCP changed from
super() to (Element) this
      // System.out.println("called moveNotNestedElementOnly()");
   }
 }
```

Successful attempt to add proper nesting data boxes printText, in ElementData:

```
// print all parent data boxes, call nested print if needed
public void printText(PrintStream ps)
{
   if(!this.getNested())
   {
      Vector vMyChildren = new Vector();
      vMyChildren = getNestVector();

      System.out.println("printText called from ElementData");
      ps.println("");
      ps.println("data_box (");
      ps.println("   inp_port_names (" + getInputPortNames() + ")");
      ps.println("   inp_port_ids (" + getInputPortIds() + ")");
      ps.println("   out_port_ids (" + getOutputPortIds() + ")");
      ps.println("   expr " + this.getCaption());
      for(int i = 0; i < vMyChildren.size() ; i++){
         ((ElementData) vMyChildren.elementAt( i ) ).printNestText(ps,1);;
      }
      ps.println(")");
      ps.println("");
   }
}
```

```
        //recursively print nested children, tabing based on current layer
          public void printNestText(PrintStream ps, int numberOfTabs) {
             Vector vMyChildren = new Vector();
             vMyChildren = getNestVector();

             ps.println("");
             printTabs(ps, numberOfTabs);
             ps.println("data_box (");
             printTabs(ps, numberOfTabs);
             ps.println("   inp_port_names (" + getInputPortNames() + ")");
             printTabs(ps, numberOfTabs);
             ps.println("   inp_port_ids (" + getInputPortIds() + ")");
             printTabs(ps, numberOfTabs);
             ps.println("   out_port_ids (" + getOutputPortIds() + ")");
             printTabs(ps, numberOfTabs);
             ps.println("   expr " + this.getCaption());
             for(int i = 0; i < vMyChildren.size() ; i++){
                ((ElementData) vMyChildren.elementAt( i ) ).printNestText(ps ,
numberOfTabs + 1 );
             }
             printTabs(ps, numberOfTabs);
             ps.println(")");
             ps.println("");
          }

        //prints given number  of 3 space tabs to stream
          public void printTabs( PrintStream ps,int numberOfTabs) {
             for(int i = 0 ; i < numberOfTabs ; i++ ){
                ps.print("   ");
             }
          }
```

Changes to maintain connections during resizing, in ElementData:

```
          public void resize (int width, int height)
          {
            …
            //cleanAllPorts();// GCP strips connectors, remove
            //updateElementPorts();// GCP redraws ports, remove but use

            moveElementPorts();// GCP new method
          }//end resize()

          // new method;  refit of updateElementPorts()
          //   Used loop interior from MOVE loop, traced from
```

```
        // ElementContainer.mouseDrag()
        // case ConstantsStatus.MOVE_ELEMENT: leading to
        // ElementPortManager.moveNotNestedElement(),
        // which is -->aPortsOnElement[i].translate(difx,dify); .
        //   The point calculation from updateElementPorts(), determines
        // the correct port position.
        //   Translate uses a difference in position vice a position
        // requiring trading .move() for .tranlate() .
        public void moveElementPorts()
        {
          int iInputPortOffset = height/(iNoOfInputPorts+1);
          int iOutputPortOffset = height/(iNoOfOutputPorts+1);
          int iInputIndex=0, iOutputIndex=0;

          //System.out.println("iNoOfInputPorts: " + iNoOfInputPorts);
          //System.out.println("iNoOfOutputPorts: " + iNoOfOutputPorts);

          // insert new ports
          for(iInputIndex=0; iInputIndex<iNoOfInputPorts; iInputIndex++)
          {
            // System.out.println("iNoOfInputPorts: " + iNoOfInputPorts);
            // System.out.println("value of iInputIndex before 1st loop line: " +
iInputIndex);
            // iInputIndex represents the portNum and idNo represents the parent Element
on which the ports belong
            // cleanPort(iInputIndex);
            //aPortsOnElement[iInputIndex] = new
Port(ptStart.x,ptStart.y+(iInputPortOffset*(iInputIndex+1)),iInputIndex,
            //                          idNo,ConstantsRigal.iOUTPUT_PORT_TYPE_ID);
            //aPortsOnElement[i].translate(difx,dify);// GCP need move

aPortsOnElement[iInputIndex].move(ptStart.x,ptStart.y+(iInputPortOffset*(iInputIndex+
1)));// GCP formula from updateElementPorts()
            // System.out.println("value of iInputIndex after 1st loop line: " + iInputIndex);
            }
            // System.out.println("value of iInputIndex after 1st for loop: " + iInputIndex);
            // System.out.println("iInputIndex: " + iInputIndex);
            // System.out.println("iOutputIndex: " + iOutputIndex);

            for(iOutputIndex=0; iOutputIndex<iNoOfOutputPorts; iOutputIndex++)
            {
              //aPortsOnElement[iInputIndex+iOutputIndex] = new
Port(ptStart.x+width,ptStart.y+(iOutputPortOffset*(iOutputIndex+1)),
              //
iInputIndex+iOutputIndex,idNo,ConstantsRigal.iINPUT_PORT_TYPE_ID);
```

```
aPortsOnElement[iInputIndex+iOutputIndex].move(ptStart.x+width,ptStart.y+(iOutputP
ortOffset*(iOutputIndex+1)));// GCP formula from updateElementPorts()
              //System.out.println("value of iInputIndex+iOutputIndex after 2nd loop line: "
+ (iInputIndex+iOutputIndex));
            }
              //System.out.println("value of iInputIndex+iOutputIndex after 2nd for loop: "
+ (iInputIndex+iOutputIndex));
              iInputIndex = 0; iOutputIndex = 0;
          }
```

Changes to allow saving (and restore as implemented), in RigelFrame:

```
public class RigalFrame extends JFrame
  implements java.io.Serializable
  // implements ComponentListener
{
  ElementContainer ecElementContainer = null;
  // ElementContainer ecNew = null;
  FileDialog fdSaveText, fdOpen;// Remove fdSaveDiagram GCP, 3 additional
change locations
  TextField txtStatus;
…
}//end class RigalFrame

// constructor used in application
  public RigalFrame(String title)
  {
…
    fdSaveText = new FileDialog(this,"Save Diagram As Text",
FileDialog.SAVE);
    fdSaveText.setFilenameFilter(new EndsWith(".TXT"));
    // fdSaveText.setFilenameFilter("*.TXT");
    // moved and modified to handleEvent() GCP
    //fdSaveDiagram.setFilenameFilter(new EndsWith(".BIN"));//SUPPRESS,
could move also GCP
    fdOpen = new FileDialog(this,"Open Diagram", FileDialog.LOAD);
      …
  }// end RigalFrame(String)

public boolean handleEvent (Event e)
  {
    …
    ObjectOutputStream oos = null;
    // System.out.println("File -> Save Text called");
```

```
            FileDialog fdSaveDiagram = new FileDialog(this,"Save Diagram",
FileDialog.SAVE);//GCP Moved and modified
            fdSaveDiagram.pack();
               …
          }//end handleEvent()
```

Change for inconsistent saves / failed saves, in RigalFrame:

```
          public boolean handleEvent (Event e)
          {
            …
               try
               {
                 oos.flush();// GCP restored standard use, probably commented out for
not serializable error troubleshooting
                  oos.close();
               }
            …
          }//end handleEvent()
```

Changes for clean “New” and “Open” with title restore, in RigalFrame:

```
          public boolean handleEvent (Event e)
          {
            if (((String)e.arg).equals("Save Diagram")) // File -> Save menu item
begins
              {
                …

               try
               {
                System.out.println("inside the try block of save Diagram()...");
                // OutputStream fos = new FileOutputStream("MyFile.bin");
                // ObjectOutputStream oos = new ObjectOutputStream(fos);

                oos = new ObjectOutputStream(
                     new BufferedOutputStream(
                     new FileOutputStream(sFilePath + sFileName)));

               // Element[] aElements = ecElementContainer.getArrayOfElements();
               // if (aElements != null){
                  //System.out.println("aElements is NOT null");
                 // System.out.println("****debugging for serialization");
```

```
                //
System.out.println(UtilRigal.getDebugElementArrayString(aElements));
                 // oos.writeObject(ecElementContainer.getArrayOfElements());
                 oos.writeObject(ecElementContainer);// GCP save entire
container to save work, allowed with serialization fix
               // }//end if
                System.out.println("passed -->
oos.writeObject(ecElementContainer)");
              }//end try


          public boolean handleEvent (Event e)
          {
                …
            if (((String)e.arg).equals("New"))
              {
                if (psOutFile != null) psOutFile.close();

                //ecElementContainer.restart(); GCP replaced with:
              this.getContentPane().setVisible( false );// GCP hide change
              //removeElementContainer( this, ecElementContainer );// GCP swap
out, done in freshElementContainer()
              freshElementContainer();// GCP creates brand new initialized
object
              this.getContentPane().setVisible( true );

                psOutFile = null;
                sFileName = null;
                this.setTitle("New Untitled Element");
              }


        public void restoreElementContainer(String sFile)
         {
           System.out.println("restoring Element Container...");
           ObjectInputStream ois = null;
           ElementContainer ecTemp = null;
           // sFile = sFilePath + sFileName;

           try
             {
              //Element aeTemp[] = null;// GCP removed
              //Element[] aeRetrieved = null; // GCP removed
```

```
            ois = new ObjectInputStream(
                new BufferedInputStream(
                new FileInputStream(sFile)));
              // new FileInputStream("MyFile.bin")));

            Object obj = ois.readObject();
            //aeRetrieved = (Element[]) obj; // GCP removed
            if ( obj != null ) {// GCP troubleshooting, but not bad practice
              ecTemp = (ElementContainer) obj;// GCP restored
              // restoreSerializedElements(ecElementContainer, ecTemp);
              //restoreSerializedElements(ecElementContainer, aeRetrieved); //
GCP removed


              this.getContentPane().removeAll();// GCP clean start


              txtStatus = new TextField();// GCP moved from constructor
              txtStatus.setEditable(false); // GCP moved from constructor

              // **** restore diagram
              ecElementContainer = ecTemp;// GCP stored container restored
logically
              ecTemp = null;// GCP good housekeeping
              ecElementContainer.setRigalFrame( this );// GCP environment
introductions normally done in EC constructor
              ecElementContainer.setEditorStatusBar( ConstantsStatus.STARTUP
);// GCP attribute initialization normally done in EC constructor
               this.setTitle( ecElementContainer.getTitle() );// GCP restore title
              // ecNew = new ElementContainer(this); // GCP moved from
constructor

              ToolBoxSwing tbSwing = new ToolBoxSwing();// GCP moved
from constructor
              // tbElements = new tbElementsAndActions(ecElementContainer); //
GCP moved from constructor

              tbElementsAndActions = new
ToolBoxElementsAndActions(ecElementContainer); // GCP moved from
constructor
              addElementContainer(this, ecElementContainer); // GCP moved
from constructor
```

```
                this.getContentPane().add("South", txtStatus); // GCP moved from
constructor

                ecElementContainer.setMoved();
                ecElementContainer.paint(this.getGraphics());
                ecElementContainer.repaint();
                System.out.println(" ecElementContainer.paint() done");

//System.out.println(UtilRigal.getDebugElementArrayString(aeRetrieved));
              }
              else {
                System.out.println(" Null object in file");
              }


            }
            catch (ClassNotFoundException ex)


          public boolean handleEvent (Event e)
          {
            // File -> Open menu item begins
              if (((String) e.arg).equals("Open"))
              {
                  if (sFileName == null)
               {
                 …

                 this.setVisible( false );// GCP for restore
                 restoreElementContainer(sFilePath + sFileName);
                 this.setVisible( true );// GCP for restore

                 …
                  }
               else // sFileName != null
               {
                 System.out.println("File->Open called...sFileName != null");
                 this.setVisible( false );// GCP for restore
                 restoreElementContainer(sFilePath + sFileName);
                 this.setVisible( true );// GCP for restore

                  }
```

```
        }

      /**
       *
       */
      private void freshElementContainer() {
        if ( ecElementContainer != null ) {
          //removeElementContainer( this, ecElementCOntainer ); GCP more
involved
          getContentPane().removeAll();

        }//end if

        txtStatus = new TextField();
        txtStatus.setEditable(false);

        // **** create a new diagram
        ecElementContainer = new ElementContainer(this);
        // ecNew = new ElementContainer(this);
        ToolBoxSwing tbSwing = new ToolBoxSwing();
        // tbElements = new tbElementsAndActions(ecElementContainer);
        tbElementsAndActions = new
ToolBoxElementsAndActions(ecElementContainer);
        addElementContainer(this, ecElementContainer);
        this.getContentPane().add("South", txtStatus);


      }//end freshElementContainer()


      // constructor used in application
      public RigalFrame(String title)
      {
            …
        fdOpen = new FileDialog(this,"Open Diagram", FileDialog.LOAD);
        fdOpen.setFilenameFilter(new EndsWith(".BIN"));
        // fdOpen.setFilenameFilter("*.TXT");
        }

        freshElementContainer();// GCP allow element containers to be
reinitialized for "NEW and "OPEN"

        // this.getContentPane().add("West", tbActions);
```

Changes for signature restore, in ElementContainer

```
public void paint(Graphics g)
{
  …
  if (dcaf == null)
    {
      System.out.println("new dcaf is created ");
      dcaf = new DialogContainerAndFrame(getRigalFrame(), this);
      dcaf.getInputPortCombo().setSelectedIndex(1);
      dcaf.getOutputPortCombo().setSelectedIndex(0);
      dcaf.setInputPortIndex(1);
      dcaf.setOutputPortIndex(0);
      if ( sSignature != null ) {// GCP to maintain signature during restore
        dcaf.setDiagramSignature( sSignature );
      }// end if else follows
      else {// GCP to initialize signature for first time
        dcaf.setDiagramSignature("Diagram Signature: null");
      }//end else

      paintDiagramSignature(g, dcaf.getDiagramSignature());
    }
    else
    {
      paintDiagramSignature(g, dcaf.getDiagramSignature());
    }
}//end paint()
```

Changes for progressive thickness of nested elements, in Element

```
public void paint(Graphics g)
{
  // System.out.println("*Element.paint(Graphics g) is called*");
  Polygon poly = new Polygon(polyPolygon.xpoints, polyPolygon.ypoints,
polyPolygon.npoints);


  for (int i=0; i<poly.npoints-1; i++)
  {
    poly.xpoints[i] -= ecElementContainer.getOffsetX();
    poly.ypoints[i] -= ecElementContainer.getOffsetY();
    // g.drawLine(poly.xpoints[i], poly.ypoints[i], poly.xpoints[i+1],
poly.ypoints[i+1]));
  }
```

```
            // g.drawPolyline(poly);
            // GCP line thickness control for nested nodes and stream connector
            Graphics2D g2 = (Graphics2D) g;// GCP switch graphics engine so we can
control thickness

            int thickness = 1;// GCP default thickness
            if (this instanceof ElementData) {// GCP special case for nested thickness
              thickness += elementTreeLevel( 0 , ecElementContainer );// GCP add
additional thickness if needed
            }

            BasicStroke wideStroke = new BasicStroke ( (float) thickness );// GCP select
line width to show nesting
            g2.setStroke( wideStroke);// GCP set line width to show nesting

            g2.drawPolyline(polyPolygon.xpoints, polyPolygon.ypoints,
polyPolygon.npoints);// GCP change to g2
          }

          /** Recursive method to find a nested element depth from its root parent or; the
           * number obtained by starting from 0 with the outer element and counting in to
the
           * nested element.  Used for determining nested element line thickness.  Initially
called with
           * 0 or result inaccurate.
           * @return distance from parent of this nested element
           * @param total Seed value of zero, used to collect number of levels
           * @param tempEC Element container passed to save recursive queries for same
info.
           */
          private int elementTreeLevel( int total , ElementContainer tempEC ) {
              // Final parent drops through and returns with no change
              if ( this.getNested()) {// others call immediate parent recursively
                int parentID = this.getNestParentId();// get parent
                Element parent = tempEC.getElementWithId( tempEC , parentID );
                // find level of parent
                // add 1 for this instance and pass it back
                total = parent.elementTreeLevel( 0 , tempEC ) + 1 ;
              }
              return total;
          }//end elementTreeLevel()

                                                  in ElementContainer
          public void processNestedElement(Element eTemp){// Assumes nested
            Element eNest = null;
            int iNestId = getNestId(eTemp);
```

```
            eTemp.setNestId(iNestId);  // called once
            System.out.println("getNestId(eTemp): " + iNestId); // called twice
            eNest = getNestElement(eTemp); // called thrice
            System.out.println("eNest: " + eNest);
            if(eNest != null){
              eTemp.setNested(true);// GCP more correct place, only nested if there is a
parent
              eNest.getNestVector().add(eTemp);
              eNest.setHasChild(true);
            }
          }//end processNestedElement()



        public void addConnectorElement(ElementConnector ecNew, Port portStart,
Port portEnd){
                 // System.out.println("add new databox");
              // System.out.println("\n****DBGS****" + ecNew.getDebugString());
              upsizeArrayOfElementsInElementContainer();
                 aElementsInElementContainer[iIndexOfElementsInElementContainer] =
ecNew;
                 eElement =
aElementsInElementContainer[iIndexOfElementsInElementContainer];
              if (ecNew instanceof ElementConnector) {// GCP add streams
                eElement.setType("Connector");
                eElement.setElementTypeId(ConstantsElement.iCONNECTOR_NODE);

              }//end if, else follows
              else {
                eElement.setType("Stream");
                eElement.setElementTypeId(ConstantsElement.iSTREAM_NODE);
              }//end if-else GCP add streams

              iIndexOfElementsInElementContainer++;
```

Changes for Stream Connector, in ConstantsElement

```
          // node ids
          public final static int iDEFAULT_NODE  = 0;
          public final static int iDATA_NODE  = 1;
          public final static int iFORK_NODE  = 2;
          public final static int iMERGE_NODE  = 3;
          public final static int iSWITCH_NODE  = 4;
          public final static int iCONTAINER_NODE = 5;
```

```
public final static int iCONNECTOR_NODE = 6;
public final static int iPATTERN_NODE = 7;
public final static int iDCALL_NODE = 8;
public final static int iSTREAM_NODE = 9;// GCP for new stream connector
```

in Element.paint()

```
if (this instanceof ElementData) {// GCP special case for nested thickness
  thickness += elementTreeLevel( 0 , ecElementContainer );// GCP add
additional thickness if needed
}else if (this instanceof ElementStream){// GCP add stream stem attempt
  thickness = 10;
  g2.setColor(Color.green.brighter());
}//end else if

BasicStroke wideStroke = new BasicStroke ( (float) thickness );// GCP select
line width to show nesting
g2.setStroke( wideStroke);// GCP set line width to show nesting

g2.drawPolyline(polyPolygon.xpoints, polyPolygon.ypoints,
polyPolygon.npoints);// GCP change to g2

wideStroke = new BasicStroke ( (float) 1 );// GCP select line width for
normal
g2.setStroke( wideStroke);// GCP set line width back
g2.setColor(Color.black);
}
```

in ToolBoxElementsAndActions

```
public ToolBoxElementsAndActions(ElementContainer ecElementContainer)
{
        …
 JButton jbConnect = new JButton("Connect");
 JButton jbStream = new JButton("Stream");// GCP adds stream connectors
        …
    gbc.gridwidth = 1;
    gbc.gridx = 0;
    gbc.gridy = 19;
    gbl.setConstraints(jbExit, gbc);
    tbElementsAndActions.add(jbExit, gbc);

    gbc.gridwidth = 1;// GCP adding Stream button
    gbc.gridx = 0;
    gbc.gridy = 20;
    gbl.setConstraints(jbStream, gbc);
```

```
            tbElementsAndActions.add(jbStream, gbc);
          …
    jbEditPort.addActionListener(bhButtonHandler);
    jbExit.addActionListener(bhButtonHandler);
    jbStream.addActionListener(bhButtonHandler);// GCP add Streams

 }

public class ButtonHandler implements ActionListener
 {
  RigalFrame rf = null;

   public ButtonHandler(RigalFrame rf)
   {
     this.rf = rf;
   }

   public void actionPerformed(ActionEvent e)
   {
      …
     else if ((e.getActionCommand()).equals("Exit"))
     {
       System.exit(0);
     }
     else if ((e.getActionCommand()).equals("Stream"))// GCP add Streams
     {
          ecElementContainer.setEditorStatusBar(ConstantsStatus.STREAM);
     }
   }
 }
}
```

in ConstantsStatus

```
  public final static int RESIZE = 117;
  public final static int EDITPORT  = 118;
  public final static int STREAM  = 119;// GCP add Streams
```

in ElementContainer
.getText()

```
      // print all ElementConnectors
    ElementConnector ncTemp = null;
    ElementStream nsTemp = null;// GCP add streams
    Port portStartTemp = null;
    Port portEndTemp = null;
```

```
            ps.println("");
            for (int i=0; i<iIndexOfElementsInElementContainer; i++)
            {
              nTemp = aElementsInElementContainer[i];
              if (nTemp != null)
              {
                // now print Connector nodes GCP
                int iElType = nTemp.getElementTypeId();// GCP holds the info for
comparisons next
                // GCP connectors or streams
                if ( ( iElType == ConstantsElement.iCONNECTOR_NODE ) | ( iElType ==
ConstantsElement.iSTREAM_NODE ) )
                {
                 if ( iElType == ConstantsElement.iCONNECTOR_NODE ) {
                    //nTemp.printText(ps);
                   ncTemp = (ElementConnector) nTemp;
                   portStartTemp = (Port) ncTemp.getStartPort();
                   portEndTemp = (Port) ncTemp.getEndPort();

                   if (portStartTemp != null && portEndTemp != null)
                   {
                     ps.println("arrow ( p" + portStartTemp.getParentId() +
portStartTemp.getId() +
                                 " p" + portEndTemp.getParentId() + portEndTemp.getId() +
" )");
                   }
                 }//end if, else follows
                 else {
                    //nTemp.printText(ps);
                   nsTemp = (ElementStream) nTemp;
                   portStartTemp = (Port) nsTemp.getStartPort();
                   portEndTemp = (Port) nsTemp.getEndPort();

                   if (portStartTemp != null && portEndTemp != null)
                   {
                     ps.println("stream ( p" + portStartTemp.getParentId() +
portStartTemp.getId() +
                                 " p" + portEndTemp.getParentId() + portEndTemp.getId() +
" )");
                   }
                 }//end if-else


                }
              }
            }
```

```
}

public Element getObjectWithId(int id)



public void setEditorStatusBar(int param)
{
 //LangFrame rfRigalFrame = (LangFrame)getParent();
 //if (rfRigalFrame == null) rfRigalFrame = this.rfRigalFrame;
 if (iCurrentMode == ConstantsStatus.STREAM) {// GCP adds streams
   System.out.println("Inside setEditorStatusBar(STREAM)");
   iCurrentMode = param;
   bMoved = true;
   repaint();
 }//end if, else follows
 else if (iCurrentMode == ConstantsStatus.CONNECT)
 {
  System.out.println("Inside setEditorStatusBar(CONNECT)");


public void setEditorStatusBar(int param)
{
        …
 case ConstantsStatus.CONNECT:
   rfRigalFrame.setStatus("CONNECT: Select Source Port");
   bOutputPortSelected = false;
   ptAnchorStart = null;
   startPort = null;
   endPort = null;
   repaint();
   break;

 case ConstantsStatus.STREAM: // GCP add streams
   rfRigalFrame.setStatus("STREAM: Select Source Port");
   bOutputPortSelected = false;
   ptAnchorStart = null;
   startPort = null;
   endPort = null;
   repaint();
   break;

public boolean mouseMove(Event evt, int xin, int yin)
{
        …
```

```
            //System.out.println("Mouse MOVE sensed: offset_x =" + offset_x +",
offset_y= " + offset_x);
           // GCP add streams
           if (iCurrentMode == ConstantsStatus.STREAM && bOutputPortSelected) {
             ptCurrent = new Point(x,y);
             bMoved = true;
             repaint();
           }//end if, else follows
           else if (iCurrentMode == ConstantsStatus.CONNECT &&
bOutputPortSelected)

         public boolean mouseDown(Event evt, int xin, int yin)
         {
                 …
                   rfRigalFrame.setStatus("CONNECT: Select ENDING port");
                   ptAnchorStart = new Point(startPort.x,startPort.y);
               }
               else
               {
                 rfRigalFrame.setStatus("You must select a SOURCE port first and then a
DESTINATION port.");
               }
                }
             }
             break;

             case ConstantsStatus.STREAM:// GCP add streams
             Port pPortTemp = getSensedPort(x,y);
             if ((pPortTemp != null &&  pPortTemp.getTypeId() ==
ConstantsRigal.iOUTPUT_PORT_TYPE_ID) ||
             (pPortTemp != null &&  pPortTemp.getTypeId() ==
ConstantsRigal.iON_FAIL_PORT_TYPE_ID))
             {
               System.out.println("Start port clicked, pPortTemp: " + pPortTemp);
               bOutputPortSelected = true;
               rfRigalFrame.setStatus("STREAM: Select ENDING port");
               ptAnchorStart = new Point(pPortTemp.x,pPortTemp.y);
               startPort = pPortTemp;
             }
             else if (pPortTemp != null && pPortTemp.getTypeId() ==
ConstantsRigal.iINPUT_PORT_TYPE_ID)
             {
               endPort = pPortTemp;
               System.out.println("End port clicked, pPortTemp: " + pPortTemp);
             }
```

```
            iMatchLine = sense(x,y);
            // if (iMatchLine != -1) System.out.println("PORT match");
            // if (iMatchLine != -1  && eElement instanceof ElementPortManager)
            if (eElement instanceof ElementPortManager)
            {
                System.out.println("ConstantsStatus.STREAM MODE, inside
ElementContainer.mouseDown()");
                ElementPortManager npmElementPortManager = (ElementPortManager)
eElement;
             // if source port is not already selected
             if (bOutputPortSelected)
             {
                 endPort = npmElementPortManager.getSensedPort();
              if (startPort != endPort)
              {
                  if (endPort.getTypeId() ==
ConstantsRigal.iOUTPUT_PORT_TYPE_ID ||
                  endPort.getTypeId() == ConstantsRigal.iON_FAIL_PORT_TYPE_ID)
                  {
                     upsizeArrayOfElementsInElementContainer();
                     System.out.println("endPort.isConnected(): " +
endPort.isConnected());
                     System.out.println("startPort.isConnected(): " +
startPort.isConnected());
                     System.out.println("endPort.getTypeId(): " + endPort.getTypeId());

                     if (endPort.isConnected())
                     {
                       endPort.disconnect();
                     }
                     if (startPort.isConnected())
                     {
                       startPort.disconnect();
                     }
                    // draw new arrow if source and dest ports are different
                    ElementStream cConn = new ElementStream(this, new
Point(startPort.x,startPort.y),
                                        new Point(endPort.x,endPort.y));

                     addConnectorElement(cConn, startPort, endPort);
                     }
                   setEditorStatusBar(ConstantsStatus.STREAM);
              }
              else
              {
```

```
                rfRigalFrame.setStatus("Please select a destination port. You must select a
SOURCE port " +
                " first and then a DESTINATION port.");
               }
                }
                else
              { // source port is not selected yet, create source port
                   startPort = npmElementPortManager.getSensedPort();
                // System.out.println("startPort: " + startPort);
                System.out.println("startPort.getTypeId(): " + startPort.getTypeId());
                System.out.println("startPort.getTypeId(): " + startPort.getTypeId());

                if (startPort.getTypeId() == ConstantsRigal.iINPUT_PORT_TYPE_ID ||
                   startPort.getTypeId() == ConstantsRigal.iON_FAIL_PORT_TYPE_ID)
                {
                 bOutputPortSelected = true;
                    rfRigalFrame.setStatus("STREAM: Select ENDING port");
                    ptAnchorStart = new Point(startPort.x,startPort.y);
                }
                else
                {
                 rfRigalFrame.setStatus("You must select a SOURCE port first and then a
DESTINATION port.");
                }
                 }
              }
              break;

              default:


            // inside mouseUp()
            case ConstantsStatus.CONNECT:
              rfRigalFrame.setCursor(Frame.DEFAULT_CURSOR);
              ptCurrent = null;
              ptAnchorEnd = null;
              //ptAnchorStart  = null;
              iMatchLine = -1;
              matchdrag = false;
              return true;

               // inside mouseUp()
            case ConstantsStatus.STREAM:// GCP add streams
              rfRigalFrame.setCursor(Frame.DEFAULT_CURSOR);
              ptCurrent = null;
              ptAnchorEnd = null;
```

```
            //ptAnchorStart  = null;
            iMatchLine = -1;
            matchdrag = false;
            return true;
```

.paint()

```
            // set the arrowhead from the current point if it is adjacent or
            // keep the arrowhead unchanged

          }
          else if (iCurrentMode == ConstantsStatus.DRAWLINE || iCurrentMode ==
ConstantsStatus.CONNECT || iCurrentMode == ConstantsStatus.STREAM)
          {// GCP add stream
```

in ElementPortManger

```
        public synchronized Element sense(int x, int y)
        {
           if (ecElementContainer.getMode() == ConstantsStatus.CONNECT ||
              ecElementContainer.getMode() == ConstantsStatus.EDITPORT ||
              ecElementContainer.getMode() == ConstantsStatus.STREAM )// GCP add
streams


        public void paint(Graphics g)
        {
          super.paint(g);
          //System.out.println("**ElementPortManager.paint(g) is called**");
          //System.out.println("polyPolygon: " + polyPolygon);

       //    if ()
         //  {

          //}
          //else
          // GCP redundant draw routine if (polyPolygon != null)
          // GCP {
          // GCP   //System.out.println("polyPolygon is NULL");
          // GCP   for (int i=1; i<polyPolygon.npoints; i++)
          // GCP   {
          // GCP     UtilGraphics.drawLine(g,polyPolygon.xpoints[i-1] -
ecElementContainer.getOffsetX()
          // GCP                          ,polyPolygon.ypoints[i-1] -
ecElementContainer.getOffsetY(),
```

```
            // GCP                  polyPolygon.xpoints[i] -
ecElementContainer.getOffsetX(),
            // GCP                  polyPolygon.ypoints[i] -
ecElementContainer.getOffsetY(), iThick);
            // GCP   }
            // GCP }
            //if (ecElementContainer.getMode() == ConstantsStatus.CONNECT ||
```

ElementStream Class ##

```
package visualrigal;

import java.awt.*;
import java.util.*;
import java.io.*;

/**
 * Title:
 * Description: with better stream arrow graphics- ports are visible, no labels
 * Copyright:    Copyright (c) 2002
 * Company:
 * @author
 * @version 1.0
 */

public class ElementStream extends ElementConnector
implements java.io.Serializable {

   double AngleHead = 0;
   double AngleTail= 0;
   double Delta_X_Head = 0;
   double Delta_X_Tail = 0;
   double size = 50.0;
   private transient DialogElementProperties dnp = null;
   double headLineShift = 3.0;
   double tailLineShift = 2.4;

  public ElementStream(ElementContainer ecElementContainer, Point ptStart,
Point ptEnd) {
    super( ecElementContainer,  ptStart,  ptEnd);
    iElementTypeId = ConstantsElement.iSTREAM_NODE;
   }


  // this is the method that is called after a stream is created
```

```
            // when you connect two ports, in the STREAM mode
             public void paint(Graphics g)
             {
              int pointTotal = 0;//to remember how many points we have
              Point head = null, tail = null,//to hold the original values
                   midPoint = null,//hold the arrowhead base midpoint
                   EndPoint1 = null, EndPoint2 = null;//holds arrowhead base endpoints

              //System.out.println("****** Begin ElementConnector.paint()");

              //g.fillPolygon(polyPolygon);

              if (polyArrowhead == null)
              {
               polyArrowhead = getElementConnectorArrowhead();
               getTailAngle();
              }

              //Thick line requires a shift from the endpoints to look correct,
              //not stick out of arrowhead obscuring port

              //save head point
              pointTotal = polyPolygon.npoints;
              head = new Point( polyPolygon.xpoints[pointTotal-1] ,
polyPolygon.ypoints[pointTotal-1] );

              //replace with arrowhead base midpoint
              EndPoint1 = new Point( polyArrowhead.xpoints[2] , polyArrowhead.ypoints[2]
);//the last point in the arrowhead
              EndPoint2 = new Point( polyArrowhead.xpoints[1] , polyArrowhead.ypoints[1]
);//the middle point in the arrowhead

              midPoint = midpoint( EndPoint1 , EndPoint2 );

              //shift down the line from the head toward the tail because of thickness overlap
              if (Delta_X_Head >= 0) {//because of different behavior +x vs. -x

               //head side shift
               midPoint.x += ( ( size / headLineShift ) * ( Math.cos ( AngleHead ) ) );
               midPoint.y += ( ( size / headLineShift ) * ( Math.sin ( AngleHead ) ) );

              }//end if, else follows
              else {

               //head side shift
               midPoint.x -= ( ( size / headLineShift ) * ( Math.cos ( AngleHead ) ) );
```

```
            midPoint.y -= ( ( size / headLineShift ) * ( Math.sin ( AngleHead ) ) );

          }//end if-else

          //Temporarily assign the shifted point
          polyPolygon.xpoints[pointTotal-1] = midPoint.x;
          polyPolygon.ypoints[pointTotal-1] = midPoint.y;

          //Now do the tail end
          //save tail point
          tail = new Point( polyPolygon.xpoints[0] , polyPolygon.ypoints[0] );

          //at the same time shift up the line from the tail toward the head for the same
reason
          if (Delta_X_Tail >= 0) {//because of different behavior +x vs. -x

            //tail side shift
            polyPolygon.xpoints[0] -= (int)( ( size / tailLineShift ) * ( Math.cos (
AngleTail ) ) );
            polyPolygon.ypoints[0] -= (int)( ( size / tailLineShift ) * ( Math.sin (
AngleTail ) ) );

          }//end if, else follows
          else {

            //tail side shift
            polyPolygon.xpoints[0] += (int)( ( size / tailLineShift ) * ( Math.cos (
AngleTail ) ) );
            polyPolygon.ypoints[0] += (int)( ( size / tailLineShift ) * ( Math.sin (
AngleTail ) ) );

          }//end if-else

          super.paint(g);

         //put the original head point back in the polygon
          polyPolygon.xpoints[pointTotal-1] = head.x;
          polyPolygon.ypoints[pointTotal-1] = head.y;

          //put the original tail point back in the polygon
          polyPolygon.xpoints[0] = tail.x;
          polyPolygon.ypoints[0] = tail.y;

          if (polyArrowhead != null)
          {
            g.setColor(Color.green.brighter());
```

```
            g.fillPolygon(polyArrowhead);
            g.setColor(Color.black);
            Polygon p = new Polygon(polyArrowhead.xpoints, polyArrowhead.ypoints,
polyArrowhead.npoints);

            for (int i=0; i<p.npoints; i++)
            {
                p.xpoints[i] -= ecElementContainer.getOffsetX();
                p.ypoints[i] -= ecElementContainer.getOffsetY();
            }
            //g.fillPolygon(p);
          }
          //g.setColor(Orig);
        /*
          System.out.println("arrow
="+polyArrowhead+"NP="+polyArrowhead.npoints);
           for (int I=0; I<polyArrowhead.npoints; I++)
            System.out.println("x="+polyArrowhead.xpoints[I]+"
Y="+polyArrowhead.ypoints[I]);
        */
          // System.out.println("****End ElementConnector.paint()");
         }


          public Polygon getElementConnectorArrowhead()
          {
           Polygon poly = new Polygon();
           // add the first_point
           poly.addPoint(ptEnd.x, ptEnd.y);
           Delta_X_Head = (double)(polyPolygon.xpoints[polyPolygon.npoints-2] -
polyPolygon.xpoints[polyPolygon.npoints-1]);
           double Delta_Y = (double)(polyPolygon.ypoints[polyPolygon.npoints-2] -
polyPolygon.ypoints[polyPolygon.npoints-1]);


           if (Delta_X_Head == 0)
            AngleHead = Math.PI/2.0;
           else
            AngleHead = Math.atan(Delta_Y/Delta_X_Head); // + Math.PI/2;
           //System.out.println("Segment 01: DX="+Delta_X_Head+" DY="+Delta_Y+"
angle "+AngleHead);

           if (Delta_X_Head >= 0)
           {
            int X = (int)((double)ptEnd.x + (size * Math.cos(AngleHead - Math.PI/4.0)));
            int Y =(int)((double) ptEnd.y + (size * Math.sin(AngleHead - Math.PI/4.0)));
```

```
    poly.addPoint(X,Y);
  //System.out.println("Segment 02: X="+X+" Y="+Y);
    X = (int)((double)ptEnd.x + (size * Math.cos(AngleHead + Math.PI/4.0)));
    Y = (int)((double)ptEnd.y + (size * Math.sin(AngleHead + Math.PI/4.0)));
    poly.addPoint(X,Y);
    // close the arrowhead triangle, repeat first_point as end_point
    poly.addPoint(ptEnd.x, ptEnd.y);
 // System.out.println("Segment 03: X="+X+" Y="+Y);
  }
  else
  {
    int X = (int)((double)ptEnd.x - (size * Math.cos(AngleHead - Math.PI/4.0)));
    int Y =(int)((double) ptEnd.y - (size * Math.sin(AngleHead - Math.PI/4.0)));
    poly.addPoint(X,Y);
  //System.out.println("Segment 04: X=" + X + " Y=" + Y);
    X = (int)((double)ptEnd.x - (size * Math.cos(AngleHead + Math.PI/4.0)));
    Y = (int)((double)ptEnd.y - (size * Math.sin(AngleHead + Math.PI/4.0)));
    poly.addPoint(X,Y);
    // close the arrowhead triangle, repeat first_point as end_point
    poly.addPoint(ptEnd.x, ptEnd.y);
 // System.out.println("Segment 05: X="+X+" Y="+Y);
  }
 //  System.out.println("Segment 06: X="+ptEnd.x+" Y="+ptEnd.y);

  return poly;
}
/**
 * Return midpoint of line between two given points
 */
private Point midpoint( Point point1, Point point2  ) {
  Point answer = new Point();
  int Diff_X = ( point1.x - point2.x );
  int Diff_Y = ( point1.y - point2.y );

  answer.x = point2.x + ( Diff_X / 2 );
  answer.y = point2.y + ( Diff_Y / 2 );

  return answer;
}//end midpoint()

 public DialogElementProperties getDialog(){

   return dnp;
 }
```

```
  /**
   * Find tail segment information to make port visible by line correction
   */
  private void getTailAngle() {

    Delta_X_Tail = (double)(polyPolygon.xpoints[0] -polyPolygon.xpoints[1]);
    double Delta_Y = (double)(polyPolygon.ypoints[0] -polyPolygon.ypoints[1]);


    if (Delta_X_Tail == 0)
      AngleTail = Math.PI/2.0;
    else
      AngleTail = Math.atan(Delta_Y/Delta_X_Tail); // + Math.PI/2;

  }//end getTailAngle()

}//end class ElementStream
```

Changes for nesting pattern boxes, add to ElementPattern

```
  public Rectangle getBounds(){//prevented nesting
    Rectangle rectRetVal = null;
    rectRetVal = new Rectangle(this.getX(), this.getY(), this.getWidth(),
this.getHeight());

    return rectRetVal;
  }

  public int getX()//prevented proper nested movement
  {
    return (int) ptStart.getX();
  }

  public int getY()
  {
    return (int) ptStart.getY();
  }
```

add proper nesting pattern boxes printText, ElementPattern

```
  // print all parent data boxes, call nested print if needed
  public void printText(PrintStream ps)
  {
     if(!this.getNested())
```

```
            {
               Vector vMyChildren = new Vector();
               vMyChildren = getNestVector();

               System.out.println("printText called from ElementPattern");
               ps.println("");
               ps.println("pattern_box (");
               ps.println("   inp_port_names (" + getInputPortNames() + ")");
               ps.println("   inp_port_ids (" + getInputPortIds() + ")");
               ps.println("   out_port_ids (" + getOutputPortIds() + ")");
               ps.println("   expr " + this.getCaption());
               for(int i = 0; i < vMyChildren.size() ; i++){
                  ((ElementPattern) vMyChildren.elementAt( i ) ).printNestText(ps,1);;
               }
               ps.println(")");
               ps.println("");
            }
         }

        //recursively print nested children, tabing based on current layer
         public void printNestText(PrintStream ps, int numberOfTabs) {
            Vector vMyChildren = new Vector();
            vMyChildren = getNestVector();

            ps.println("");
            printTabs(ps, numberOfTabs);
            ps.println("pattern_box (");
            printTabs(ps, numberOfTabs);
            ps.println("   inp_port_names (" + getInputPortNames() + ")");
            printTabs(ps, numberOfTabs);
            ps.println("   inp_port_ids (" + getInputPortIds() + ")");
            printTabs(ps, numberOfTabs);
            ps.println("   out_port_ids (" + getOutputPortIds() + ")");
            printTabs(ps, numberOfTabs);
            ps.println("   expr " + this.getCaption());
            for(int i = 0; i < vMyChildren.size() ; i++){
               ((ElementPattern) vMyChildren.elementAt( i ) ).printNestText(ps ,
numberOfTabs + 1 );
            }
            printTabs(ps, numberOfTabs);
            ps.println(")");
            ps.println("");
         }

        //prints given number  of 3 space tabs to stream
         public void printTabs( PrintStream ps,int numberOfTabs) {
```

```
            for(int i = 0 ; i < numberOfTabs ; i++ ){
               ps.print("   ");
            }
         }
```

add dialog boxes for nested pattern boxes, in ElementContainer

```
         public void showDialogByElement(Element eElement)
         {
          …
            else if (eElement.getElementTypeId() ==
ConstantsElement.iPATTERN_NODE)
            {
             ElementPattern nPolymorphic = (ElementPattern) eElement;
             DialogElementProperties dnpTemp = nPolymorphic.getDialog();
             if (dnpTemp != null)
             {
              System.out.println("dnpTemp is NOT null");
              dnpTemp.showDialog();
             }
             else
             {
              dnpTemp = new DialogElementProperties(rfRigalFrame,nPolymorphic);
              dnpTemp.showDialog();
             }
            }
            else if (eElement.getElementTypeId() ==
ConstantsElement.iDCALL_NODE)
```

delete nested boxes from parent child vector so they do not print out from there, in ElementContainer

```
         public synchronized void deleteObjectWithId(int id)
         {
          int i;
          for (i=0; i<iIndexOfElementsInElementContainer; i++)
          {
            if (aElementsInElementContainer[i].getId() == id)
            {
              if( aElementsInElementContainer[i].getNested() ){//GCP if nested
                Element eParentOfDeleting =
aElementsInElementContainer[i].getNestElement();
                Vector vParentVector = eParentOfDeleting.getNestVector();
                Element eDeletingElement = getElementWithId( this,id );
                vParentVector.remove( eDeletingElement );//GCP remove nest reference
in parent
```

```
                Vector vVectorOfChildren = eDeletingElement.getNestVector();
                Iterator iterChildVector = vVectorOfChildren.iterator();
                Element eChild = null;

                while ( iterChildVector.hasNext() ) {//GCP add children to  grand-parent
                  eChild = (Element) iterChildVector.next();
                  vParentVector.add( eChild );
                  eChild.setNestId( eParentOfDeleting.getId() );
                }
                  //GCP reorder vParentVector
              }else{//GCP top level element
                  if ( aElementsInElementContainer[i].hasChild() ) {//GCP with children

                    Element eDeletingElement = getElementWithId( this,id );
                    Iterator vDeletingChildren =
eDeletingElement.getNestVector().iterator();
                    Element eElement = null;//GCP holding site
                    //make them top level elements
                    while ( vDeletingChildren.hasNext() ) {

                        eElement =  (Element) vDeletingChildren.next();
                      eElement.setNested( false );
                      eElement.setNestId( -1 );
                    }//end while

                  }//end if has child

                }//end if-else

                  aElementsInElementContainer[i] = null;
                  break;
                }// end if id
              }// end for i

              if (aElementsInElementContainer[i] == null)
```

Encapsulate stream and nesting thickness data and pattern box graphics, Element

```
            public void paint(Graphics g)
            {
              // System.out.println("*Element.paint(Graphics g) is called*");
              Polygon poly = new Polygon(polyPolygon.xpoints, polyPolygon.ypoints,
polyPolygon.npoints);
```

```
            for (int i=0; i<poly.npoints-1; i++)
            {
              poly.xpoints[i] -= ecElementContainer.getOffsetX();
              poly.ypoints[i] -= ecElementContainer.getOffsetY();
              // g.drawLine(poly.xpoints[i], poly.ypoints[i], poly.xpoints[i+1],
poly.ypoints[i+1]));
            }
            // g.drawPolyline(poly);

            g.drawPolyline(polyPolygon.xpoints, polyPolygon.ypoints,
polyPolygon.npoints);// GCP change to g2

          }
```

In ElementData and ElementPattern

```
          public void paint(Graphics g)
          {
            // System.out.println("*ElementData.paint(Graphics g) is called*");

            // GCP line thickness control for nested nodes
            Graphics2D g2 = (Graphics2D) g;// GCP switch graphics engine so we can
control thickness

            int thickness = 1;// GCP default thickness
            thickness += elementTreeLevel( 0 , ecElementContainer );// GCP add
additional thickness if needed

            BasicStroke wideStroke = new BasicStroke ( (float) thickness );// GCP select
line width to show nesting
            g2.setStroke( wideStroke);// GCP set line width to show nesting

            super.paint(g2);

            thickness = 1;// GCP return to default thickness
            wideStroke = new BasicStroke ( (float) thickness );// GCP select default
thickness
            g2.setStroke( wideStroke);// GCP set line width to default thickness
            g2.setColor(Color.black);// GCP set default color
          }
```

In ElementStream

```
public void paint(Graphics g)
{
  …

  else {

    //tail side shift
    polyPolygon.xpoints[0] += (int)( ( size / tailLineShift ) * ( Math.cos (
AngleTail ) ) );
    polyPolygon.ypoints[0] += (int)( ( size / tailLineShift ) * ( Math.sin (
AngleTail ) ) );

  }//end if-else

  // GCP line thickness control for stream connector
  Graphics2D g2 = (Graphics2D) g;// GCP switch graphics engine so we can
control thickness

  g2.setColor(Color.green.brighter());

  int thickness = 40;//GCP width of stream line
  BasicStroke wideStroke = new BasicStroke ( (float) thickness );// GCP select
line width
  g2.setStroke( wideStroke);// GCP set line width

  super.paint(g2);

  thickness = 1;// GCP back to default thickness

  wideStroke = new BasicStroke ( (float) thickness );// GCP select line width to
normal
  g2.setStroke( wideStroke);// GCP set line width to normal
  g2.setColor(Color.black);

 //put the original head point back in the polygon
```

Add signature to text file, in ElementContainer

```
public void getText(PrintStream ps)
{
  ps.println("name      " + this.getCaption());
  ps.println("signature (" + this.getSignature()+ ")");//GCP add signature
  ps.println("");
  ps.println("input_ports  ("+ncp.getInputPortIds()+")");
```

Ordered nested boxes, Element

```
/**
 * Called to reassess the ordering of nested boxes in the parent vNestVector,
 * should contain this
 * This moved or was added so update order
 */
public void reorderNestVector() {
  if ( this.getNested() ) {//if a parent then sort
     Element temp = null;//holding space for swap

    Element parent = (Element) getNestElement();//find parent
    Vector vNestVector = parent.getNestVector();// the vector that needs ordering

    int numberInVector = vNestVector.size();

    //bubble sort
    for(int i = 0 ; i < numberInVector - 1 ; i++){
      for (int j = 0 ; j < numberInVector - 1 - i ; j++ ) {

        //comparator call
        if ( ( (Element) vNestVector.get( j )).follows( ( (Element) vNestVector.get(
j + 1 ) ) ) ) {
          //Swap
          temp = ( (Element) vNestVector.get( j ));
          vNestVector.set( j , vNestVector.get( j + 1 ) ) ;
          vNestVector.set( j + 1 , temp ) ;
        }//end if

      }//end for j

    }//end for i END bubble sort


  }//end if
  //else no parent- do nothing

}//end reorderNestVector()


/**
 * comparator for determining vNestVector ordering of nested boxes
 * returns true if this should follow next in ordering,
 * returns false otherwise
```

```
 */
private boolean follows( Element next ) {

  boolean answer = false;// default response

    Point pointA = null, pointB = null;
    //load pointB with next position points for comparison
    pointB = next.getStartPoint();
    //load pointA with this
    pointA = this.getStartPoint();

    //comparing top left corners, the one further right on the x-axis follows unless
    //they are equal with respect to the x-axis then the one further down on the y-
axis
    //follows
    if ( ( pointA.x > pointB.x ) || ( ( pointA.x == pointB.x ) && ( pointA.y >
pointB.y) ) ) {
      answer = true;

    }//end if

    return answer;

}//end follows()
```

In ElementContainer

```
public void processNestedElement(Element eTemp){// Assumes nested
  Element eNest = null;
  int iNestId = getNestId(eTemp);

  eTemp.setNestId(iNestId);  // called once
  System.out.println("getNestId(eTemp): " + iNestId); // called twice
  eNest = getNestElement(eTemp); // called thrice
  System.out.println("eNest: " + eNest);
  if(eNest != null){
    eTemp.setNested(true);// GCP more correct place, only nested if there is a
parent
    eNest.getNestVector().add(eTemp);
    eNest.setHasChild(true);

    eTemp.reorderNestVector(); //GCP check nested ordering each time a new
element is nested
  }
}//end processNestedElement()
```

```
public boolean mouseUp(Event evt, int xin, int yin )
{
 System.out.println("Mouse UP sensed");
 System.out.println("Mouse UP sensed");
 int x = xin + offset_x;
 int y = yin + offset_y;

 System.out.println("mouseUp, iCurrentMode: " + iCurrentMode);
  switch(iCurrentMode)
  {
     case ConstantsStatus.MOVE_ELEMENT:
        if ( eElement.getNested() ) {
          eElement.reorderNestVector();//GCP check nested order after a move
        }//end if

        break;

  case ConstantsStatus.DATA_ELEMENT: // 103
```

Add AltPattern, ToolBoxElementsAndActions

```
  JButton jbFork = new JButton("Fork");
  JButton jbAltPattern = new JButton("Alt Pattern");// GCP adds alt patterns
  JButton jbPattern = new JButton("Pattern");


            gbc.gridwidth = 1;// GCP adding Stream button
            gbc.gridx = 0;
            gbc.gridy = 20;
            gbl.setConstraints(jbStream, gbc);
            tbElementsAndActions.add(jbStream, gbc);

            gbc.gridwidth = 1;// GCP adding AltPattern button
            gbc.gridx = 0;
            gbc.gridy = 21;
            gbl.setConstraints(jbAltPattern, gbc);
            tbElementsAndActions.add(jbAltPattern, gbc);
```

```
jbFork.addActionListener(bhButtonHandler);
jbAltPattern.addActionListener(bhButtonHandler);// GCP add altPatterns
jbMerge.addActionListener(bhButtonHandler);


  else if ((e.getActionCommand()).equals("Fork"))
  {
      ecElementContainer.setEditorStatusBar(ConstantsStatus.FORK);
  }
  else if ((e.getActionCommand()).equals("Alt Pattern"))// GCP add altpattern
  {
      ecElementContainer.setEditorStatusBar(ConstantsStatus.ALT_PATTERN
);
  }
  else if ((e.getActionCommand()).equals("Merge"))
```

In ConstantStatus

```
public final static int STREAM  = 119;// GCP add Streams
public final static int ALT_PATTERN  = 120;// GCP add Streams
```

In ElementContainer

```
public boolean mouseDown(Event evt, int xin, int yin)
{
 // System.out.println("Mouse DOWN sensed");
 ElementConnector ncElementConnector = null;
 int x = xin + offset_x;
 int y = yin + offset_y;

 switch (iCurrentMode)
 {
    …

     case ConstantsStatus.FORK:
    {
      // System.out.println("execute addForkElement()");
      addForkElement(new ElementFork(this, "Fork", new Point(x,y), 10, 70));
    }
    break;
    case ConstantsStatus.ALT_PATTERN:// GCP add alt pattern
    {
      // System.out.println("execute addAltPatternElement()");
      addForkElement(new ElementAltPattern(this, "Alt_pattern", new
Point(x,y), 10, 70));
```

```
            }
            break;
            case ConstantsStatus.SWITCH:


        public boolean mouseUp(Event evt, int xin, int yin )
        {
         System.out.println("Mouse UP sensed");
         System.out.println("Mouse UP sensed");
         int x = xin + offset_x;
         int y = yin + offset_y;

         System.out.println("mouseUp, iCurrentMode: " + iCurrentMode);
          switch(iCurrentMode)
          {
            ...
          case ConstantsStatus.FORK:
           {
             if (bNewDialog)
              {
                ElementFork ndTemp = (ElementFork)
aElementsInElementContainer[iIndexOfElementsInElementContainer-1];
                DialogElementProperties dnpTemp = new
DialogElementProperties(getRigalFrame(), ndTemp);
                ndTemp.setDialog(dnpTemp);
                dnpTemp.showDialog();
                bNewDialog = false;
              }
           }
           break;
          case ConstantsStatus.ALT_PATTERN:// GCP add alt pattern
           {
             if (bNewDialog)
              {
                ElementAltPattern ndTemp = (ElementAltPattern)
aElementsInElementContainer[iIndexOfElementsInElementContainer-1];
                DialogElementProperties dnpTemp = new
DialogElementProperties(getRigalFrame(), ndTemp);
                ndTemp.setDialog(dnpTemp);
                dnpTemp.showDialog();
                bNewDialog = false;
              }
           }
           break;
          case ConstantsStatus.DRAWLINE:
```

```
public boolean mouseDrag(Event evt, int xin, int yin)
{
 // System.out.println("Mouse DRAG sensed");
 int difx, dify;
 int x = xin + offset_x;
 int y = yin + offset_y;

 switch (iCurrentMode)
 {
 …

 case ConstantsStatus.FORK:
   ElementFork LocalElementFork = null;
   if (eElement instanceof ElementFork) {
        LocalElementFork = (ElementFork)eElement;
   }
   LocalElementFork.resize(Math.abs(x-ptAnchorStart.x),Math.abs(y-
ptAnchorStart.y));
   repaint();
   break;

 case ConstantsStatus.ALT_PATTERN:// GCP add alt pattern
   ElementAltPattern LocalElementAltPattern = null;
   if (eElement instanceof ElementAltPattern) {
        LocalElementAltPattern = (ElementAltPattern)eElement;
   }
   LocalElementAltPattern.resize(Math.abs(x-ptAnchorStart.x),Math.abs(y-
ptAnchorStart.y));
   repaint();
   break;

 case ConstantsStatus.ADDDATABOX:


public void setEditorStatusBar(int param)
{
 …

 switch(iCurrentMode)
 {
  …
 case ConstantsStatus.FORK:
   rfRigalFrame.setStatus("Current Mode: Add Fork Element");
   break;
```

```
            case ConstantsStatus.ALT_PATTERN:
              rfRigalFrame.setStatus("Current Mode: Add Alt Pattern Element");
              break;
            case ConstantsStatus.RESIZE:
```

ElementAltPattern Class##

```
package visualrigal;

import java.awt.*;
import java.io.*;

/**
 * Title:
 * Description:
 * Copyright:    Copyright (c) 2002
 * Company:
 * @author
 * @version 1.0
 */

public class ElementAltPattern extends ElementFork
  implements java.io.Serializable {

  public ElementAltPattern(ElementContainer parent, String name, Point ptStart,
int width, int height) {

    super(parent, name, ptStart, 3, 3);
    iElementTypeId = ConstantsElement.iFORK_NODE;
  }

  public ElementAltPattern(ElementContainer parent, String name, Point ptStart,
int width, int height, int id)
  {
    super(parent, name, ptStart, 3, 3, id);
    iElementTypeId = ConstantsElement.iFORK_NODE;
  }

  public void printText(PrintStream ps)
  {
    System.out.println("printText called from ElementAltPattern");
    ps.println("");
    ps.println("alt_pattern ( input_port_ids (" + getInputPortIds() + " )");
    ps.println("out_port_ids ( " + getOutputPortIds() + " ) )");
```

```
}


public void resize (int width, int height)
{
  width = 10;
  //height = 70;
  this.width = width;
  this.height = height;


  Polygon polyPolygon = new Polygon();
  polyPolygon.addPoint(ptStart.x, ptStart.y );
  polyPolygon.addPoint(ptStart.x + width, ptStart.y);
  polyPolygon.addPoint(ptStart.x + width, ptStart.y + height);
  polyPolygon.addPoint(ptStart.x, ptStart.y + height);
  polyPolygon.addPoint(ptStart.x, ptStart.y );
  this.polyPolygon = polyPolygon;

  cleanAllPorts();
  updateElementPorts();
}

  public void paint(Graphics g){
  // System.out.println("*ElementALtPattern.paint(Graphics g) is called*");

  g.setColor(Color.blue);// GCP set color

  super.paint(g);

  g.setColor(Color.black);// GCP set default color
}//end ElementAltPattern Class
```

Fix multi-ports for fork, alt pattern and merge, DialogElementProperties

```
public void addButtonListeners()
{
    …

        else if (eElement.getElementTypeId() ==
ConstantsElement.iFORK_NODE)
```

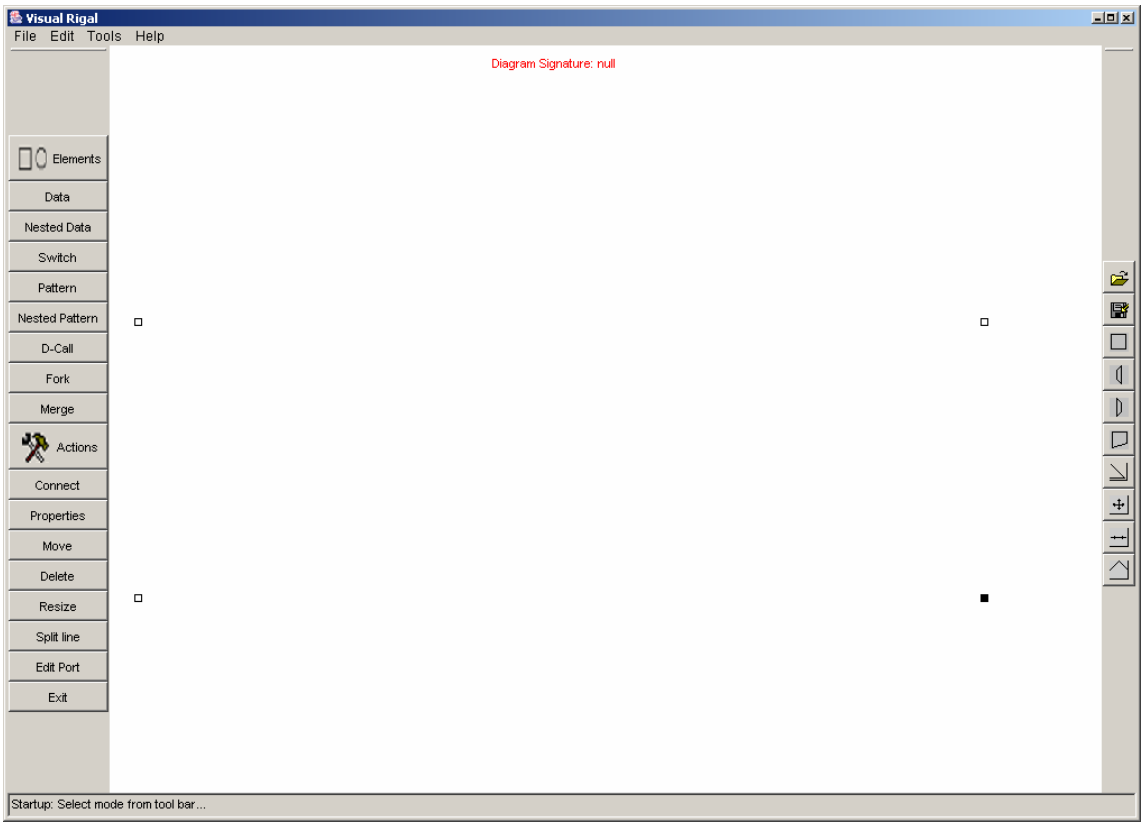
```
                {
                   ElementFork ndTemp = (ElementFork) eElement;
                   System.out.println("ndTemp: " + ndTemp);
                   ndTemp.cleanAllPorts();
                   ndTemp.setNoOfInputPorts(iInputPortIndex+1);
                   ndTemp.setNoOfOutputPorts(iOutputPortIndex+1);// GCP repair fork
and alt pattern multiple output ports
                   System.out.println("ndTemp.getTotalNoOfPorts(): " +
ndTemp.getTotalNoOfPorts());
                   System.out.println("ndTemp.getTotalNoOfPorts(): " +
ndTemp.getTotalNoOfPorts());
                   ElementPortManager npTemp = (ElementPortManager) eElement;
                   npTemp.setNoOfPorts(ndTemp.getTotalNoOfPorts());
                   ndTemp.updateElementPorts();
                }
                else if (eElement.getElementTypeId() ==
ConstantsElement.iMERGE_NODE)
                {
                   ElementMerge ndTemp = (ElementMerge) eElement;
                   System.out.println("ndTemp: " + ndTemp);
                   ndTemp.setNoOfInputPorts(iInputPortIndex+1);// GCP repair merge
multiple input ports
                   ndTemp.cleanAllPorts();
                   ndTemp.setNoOfOutputPorts(iOutputPortIndex+1);
                   System.out.println("ndTemp.getTotalNoOfPorts(): " +
ndTemp.getTotalNoOfPorts());
                   System.out.println("ndTemp.getTotalNoOfPorts(): " +
ndTemp.getTotalNoOfPorts());
                   ElementPortManager npTemp = (ElementPortManager) eElement;
                   npTemp.setNoOfPorts(ndTemp.getTotalNoOfPorts());
                   ndTemp.updateElementPorts();
                }
```

Fix delete connector then add connector deletes previous node from diagram, ElementContainer

```
public boolean mouseUp(Event evt, int xin, int yin )
{
 …
  */
  case ConstantsStatus.DELETEOBJECT:
    if (iMatchLine != -1)
    {
```

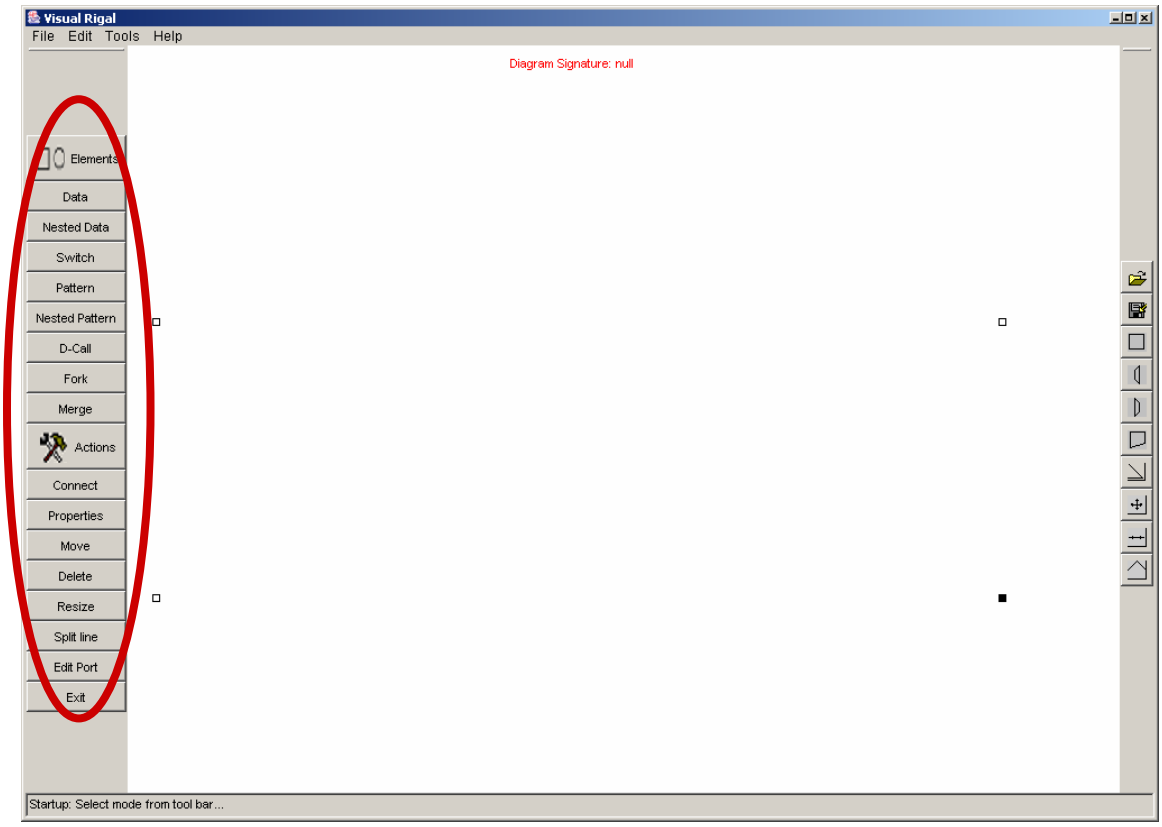
if (eElement instanceof ElementPortManager && !(eElement instanceof ElementConnector)){// **GCP correct connector delete removes last in diagram replace by connector**

C. VISUALRIGAL USER'S MANUAL



Appearance at initial start-up

The VisualRigal graphical user interface(GUI) appears as shown upon initial start up. The GUI is controlled by menu bars and a mouse. The primary menu is the Mode Tool Bar shown below.



Location of Mode Tool Bar

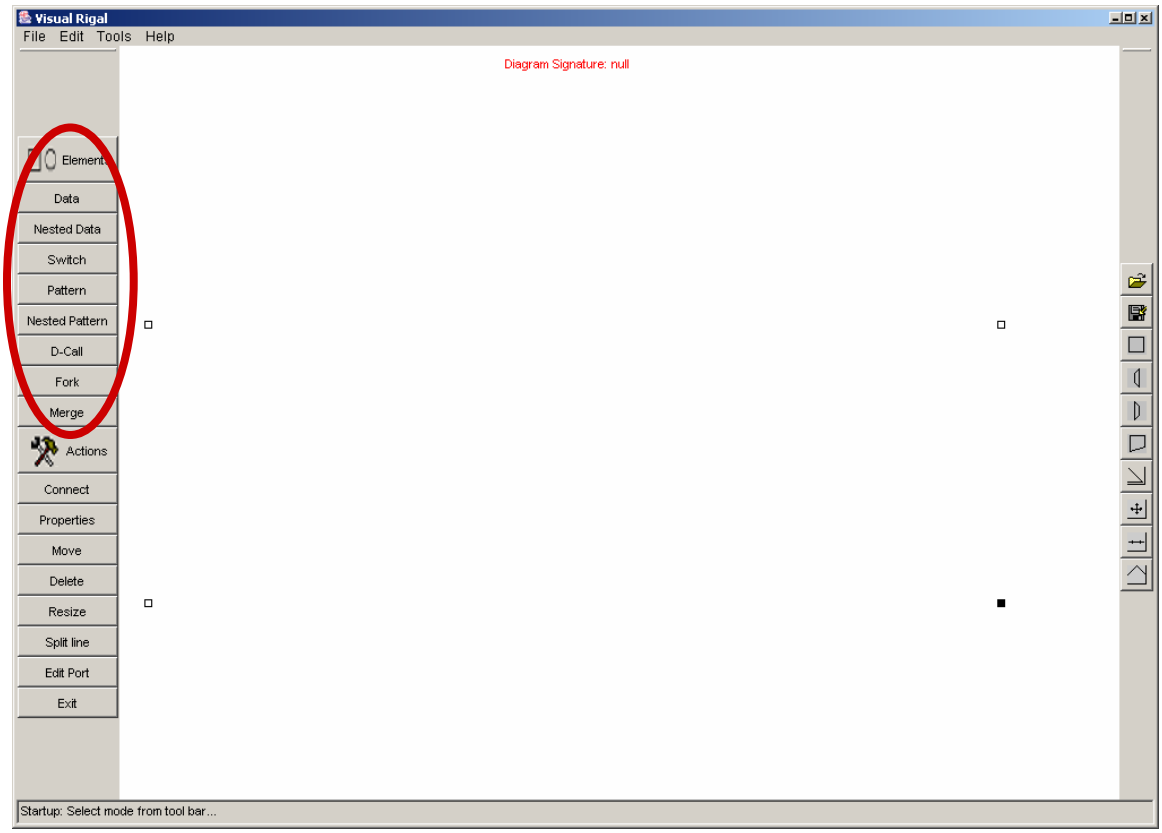
It is further divided into the Add Element Buttons and the Actions on Elements Buttons. An element can be added to the diagram area by clicking the appropriate button and then clicking on the diagram. While holding the button down non-nested elements may be resized. For non-nested elements a properties dialog box will appear with button release.

Nested elements are added by selecting a Nested Data or Nested Pattern button and clicking at the location of the intended top left corner for this new element. If this new element is fully enclosed by a like element, it will be nested. A visual indication of success is the progressive thickening of element lines with each level of nesting. If the new element is not fully enclosed by any other element it will not be nested but reside at the top level of the diagram. In all respects this element will be like any other non-nested element.

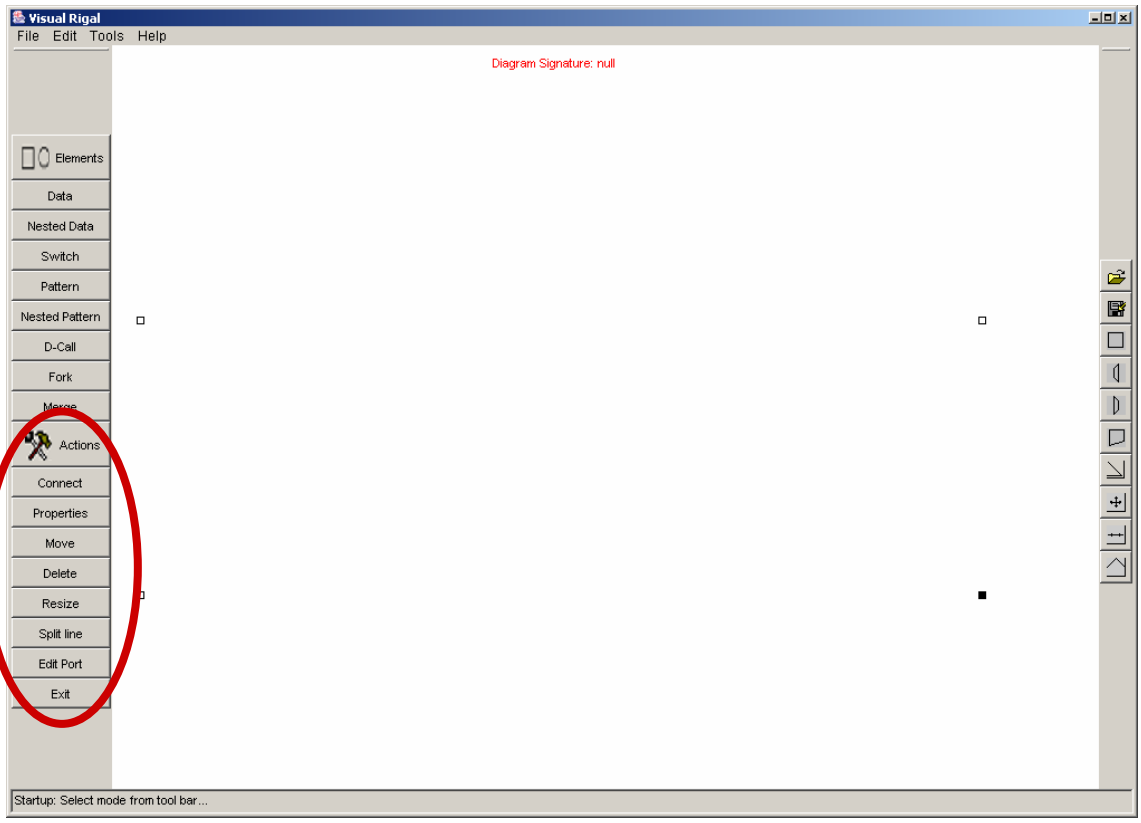
Action instructions follow:

Connect- click on a diagram input port or an element output port, pointer turns into a hand when over the port. Drag the line to a diagram output port or an element input port and click. Connection is made.
Properties- click an element or the diagram background to edit properties.
Move- click and drag an element by the top left corner into desired position and release.
Delete- click in an element to delete it from the diagram.
Resize- click and drag from the bottom right corner to change size of picture.

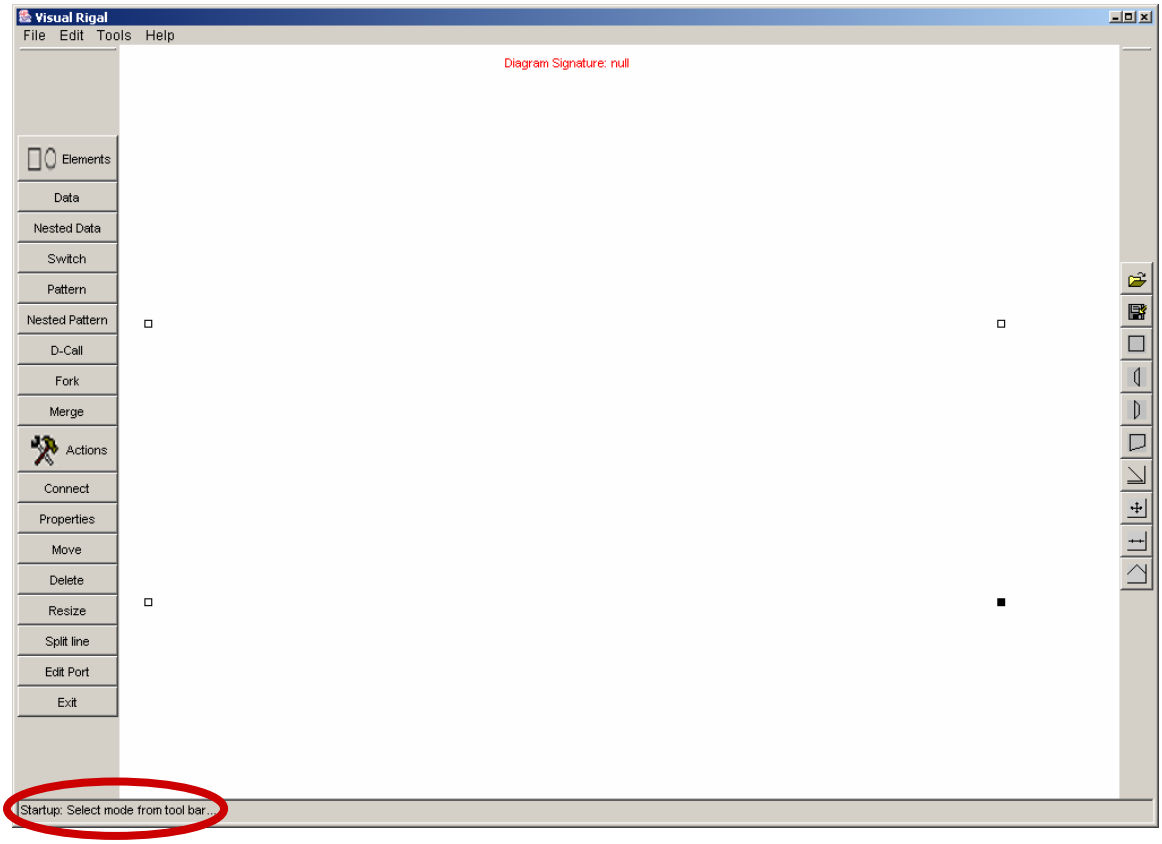
Split Line- click and drag on a connector or stream line segment to create an additional point on the route described by the line. This is useful for cleaning up diagrams.
Edit Port- click on a port to edit its caption. The pointer turns into a hand when over the port.
Exit- quit application
Stream- same as connect
Alt Pattern- incorrectly placed element



Add Element Buttons



Actions on an Element Buttons



Message Bar

The message bar shown above displays instructions each step of the way.

Top Menu

File Menu

New- start with a clean slate

Open- restore a previously saved diagram

Save Diagram- saves the diagram, may be restored with Open

Save Text- saves a text file interface that represents the logical equivalent of the diagram. Cannot be restored from this file.

Exit- quit the application

Other Menus- currently unused

D. REQUIREMENTS ANALYSIS

Requirements Analysis
SW 4583 Project

LT Les Glosby
Mr. Robert Luna
Maj. Graham Pierson
Capt. Ray Pursel

Version 4.2
18 Jun 04

THIS PAGE INTENTIONALLY LEFT BLANK

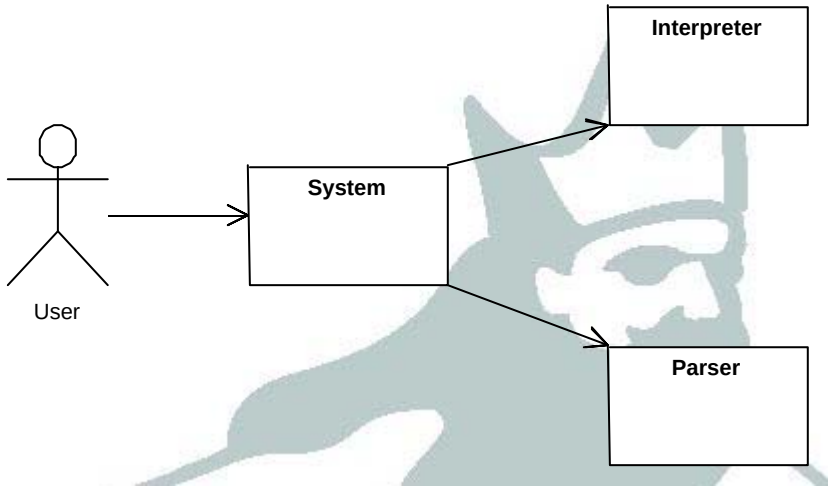
INTRODUCTION

PURPOSE OF THE SYSTEM

”The task is to design a front end for a prototype implementation of a visual programming language. This is a research prototype; the aim is to demonstrate the new capabilities of two-dimensional data flow diagram notation for describing algorithms and data structures. Although data flow diagrams are well known and broadly used in software design practice, nobody has tried this particular combination of data flow notation with a two dimensional representation of data in order to write executable programs. The OOPSLA 2001 paper provides more detailed discussion and examples of this approach.” [1]

SCOPE OF THE SYSTEM

System boundaries are described as follows:



DEFINITIONS, ACRONYMS AND ABBREVIATIONS

Can be found in the Glossary.

REFERENCES

1. Problem Statement for Term Project, SW4583, Spring 2004

2. M. Auguston, V. Berzins and B. Bryant, “Visual Meta-Programming Language”, 2001.

OVERVIEW

CURRENT SYSTEM

PROPOSED SYSTEM

OVERVIEW

The proposed system, a graphical user interface (GUI) for a visual programming language, is well described in the references. The methodology for development and its products are described here as an overview of the process. This document portrays an order for presentation but not actual development path. As shown below, this work was not a straight path to its present form.

The references were studied for **requirements**. Developers created **scenarios** using the prototype and references. The scenario catalog has been extended as need was discovered during the iterative process. From study of the scenario commonalities the team developed **use-cases**. Again use-cases were added as required in the iterative process. The current use-case taxonomy was created by analysis of the existing cases. Some use-cases were not developed and others received extensive elaboration as required for progress by the development team. **Objects** were identified through analysis of use-cases and partitioned as described. **Dynamic models** were developed for each use case as required, using the generated objects. In this case, developers constructed these models after problems in object design emerged. Finally, a **user interface** mock-up is referenced but is represented by a java jar file included with the design documents.

FUNCTIONAL REQUIREMENTS

Vision Statement Excerpt

"The required system should provide the following basic **functionality**.

• Visual editor that supports drawing and storing of two-dimensional diagrams. Typical operations include add, delete, resize, and move a node, connect/disconnect nodes, add/delete/edit textual information associated with the whole diagram and with separate nodes.

• User can store a diagram and later load it and continue editing.

• When editing of a diagram is completed, user will store it as a text file and pass to the parser (which is independent component, not part of this project). If parser detects presence of syntax or semantic errors it notifies the editor and the editor displays the error messages.

• After a successful parsing, the diagram can be executed by an interpreter (which also is an independent component, not part of this project). If the execution is performed in a debugging mode, the interpreter sends a message to the editor indicating the current step in the execution process, and the editor highlights the corresponding part of the diagram so that the user can trace the execution on the screen." [1]

Vision Statement Analysis

A textual analysis was performed on the above statement and the following functions found:

Select and Paste

Select, Cut and Paste

Resize

Connect

Move

Save

Compile

Load

Receive and Display Error

Receive and Display Current Object of execution

Copy and Paste

Lab Demonstration Analysis

An analysis was performed on the first lab session and the following functions found:

Change Properties

Select and … (make iterative)

Select several create Pattern Group

Edit Pattern Group (likely extension)

Selecting nested objects as a group (likely extension)

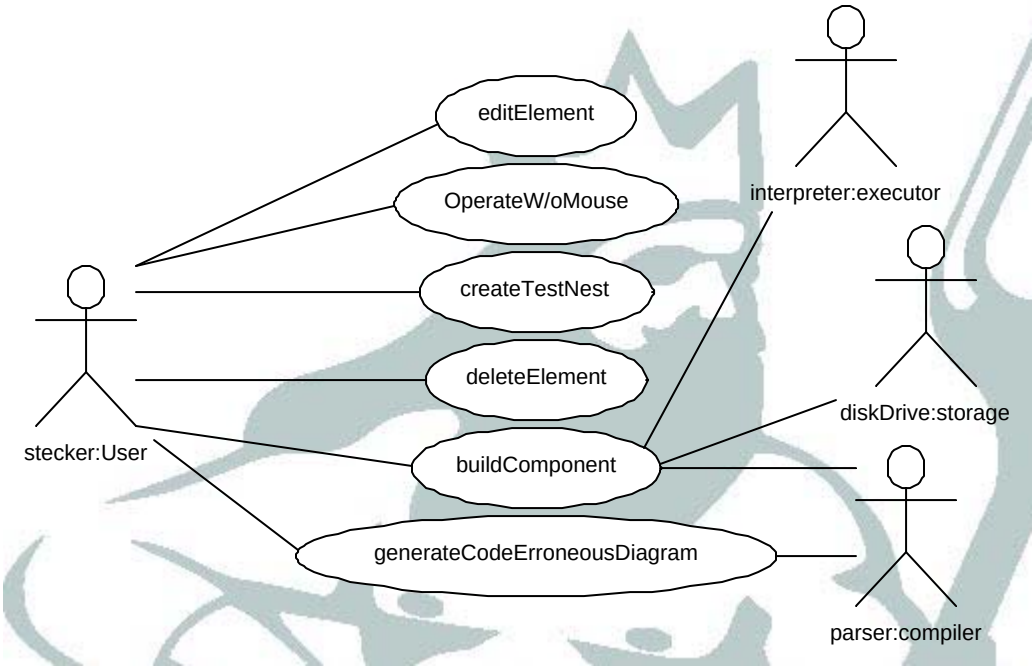
NONFUNCTIONAL REQUIREMENTS

4. Implementation

"For the sake of portability to different platforms, the editor should be implemented in Java using Java Swing framework." [1]

SYSTEM MODELS

5. Scenarios



editElement

Scenario name	editElement
Participating actor instances	stecker:user
Flow of events	1. Stecker determines item (element) to edit 2. Stecker selects element (only 1 click) 3. Application highlights element 4. On *double-click,* application opens "edit element" dialogue box 5. Stecker changes the element characteristics (eg. number of input ports, number of output ports, etc.) 6. Application warns Stecker if ports are currently in use 7. Stecker clicks "OK" button to verify corrections 8. Application verifies validity of corrections (i.e. can't have 1.5 ports) 9. Application makes changes to view 10. Stecker determines validity of changes 11. If changes are not as desired, Stecker makes corrections 12. If changes are as desired, Stecker compiles and runs the model, or saves the model for later use

Step 6: Customer input: Parser delivers error messages and GUI shows state of ports. No further development of this function.

Step 8: Lead developer: Limit input to valid conditions by using a list of choices.

operateW/oMouse

Scenario name	operateW/oMouse
Participating actor instances	stecker:user
Flow of events	1. Stecker is working within application 2. Stecker determines mouse cursor or "click" function do not work, or Stecker does not wish to use mouse 3. Stecker uses keyboard 4. Stecker uses "F1" to call help function 5. Stecker types keyboard/keystrokes/mouse/etc. to determine default keystrokes in order to use application 6. Default keystrokes are displayed for all major functions of application (e.g. move, resize, add element, etc.) 7. Stecker prints keystrokes 8. Streker creates a model using only keystrokes 9. If changes are not as desired, Stecker makes corrections 10. If changes are as desired, Stecker compiles and runs the model, or saves the model for later use 11. Stecker exits application

27 May 04: Customer desires no further development of this scenario.

Step 1: Lead developer: Understood, does not need elaboration.

Customer question: How will a user navigate on the diagram in this scenario? Answer: By using tab key, forward and back, in tab order. Note: All visible items must be contained in the tab order.

createTestNest

Scenario name	createTestNest
Participating actor instances	bruce: User
Flow of events	1. Bruce clicks on the Data button in the toolbar and then clicks on the position within an existing Data Node on the canvas area where he wants the center of the outer data node. 2. Bruce names the node "TestInner1", indicates there will be 3 input ports, and selects ellipses ('…') in the properties window. 3. Bruce moves the inner node to the lower right corner of the outer node. 4. Bruce resizes the outer node to make room for a second inner node. 5. Bruce again clicks on the Data button in the toolbar and clicks on the position within the outer data node where he wants the second inner data node. 6. Bruce names the node "TestInner2", and indicates there will

	be 1 input in the properties window. 7. Bruce moves the second inner node to the lower left corner of the outer node. 8. Bruce right-clicks on the "TestInner1" node and changes the number of input ports to 2 in the properties window.

Step 1: Further elaboration has determined that the nesting state will be determined exclusively by the graphical depiction of one element inside the boundaries of another. The parser will check for illegal conditions. This direction is chosen to reconcile apparent visual logic with underlying reality.

Step 11: Under current development plan, the move button must be selected prior to selecting and moving this node.

deleteElement

Scenario name	deleteElement
Participating actor instances	stecker:user
Flow of events	1. Stecker is working within application 2. Stecker selects the delete function from the roolbar (or with a keystroke or menu item) 3. Application waits for "element click" 4. Application changes cursor upon encountering an element 5. Stecker selects element 6. Application request confirmation to delete element 7. Application gives warning if element is connected to another object 8. Connections are detached 9. Element is removed 10. If changes are not as desired, Stecker makes corrections 11. If changes are as desired, Stecker compiles and runs the model, or saves the model for later use 12. Stecker exits application

Step 1: Lead developer: Understood, does not need elaboration.

Step 7: Customer input: GUI shows relationships. No further development of this function.

Step 8: Customer input: Connections are deleted

Step 10: Further editing is allowed.

buildComponent

Scenario name	buildComponent
Participating	stecker:user, hardDrive:storage, parser:compiler:,

actor instances	interpreter:executor
Flow of events	1. Stecker starts the application 2. Presented with a blank diagram, Stecker changes the diagram signature and title to D-component and Integer Mixer. 3. He changes to three input ports for the diagram. 4. Stecker selects a switch for his use. 5. Stecker positions the switch on the diagram area. 6. He turns the switch into a 2 port input. 7. Next he labels the diagram ports and the switch ports. 8. He labels the switch to provide his intended functionality. 9. He resizes the switches to make the text readable. 10. Next he copies this switch twice to end up with three switches all with the same formatting and positions the two new switches. 11. After adding a fork, he connects appropriate ports together. 12. Stecker has added a second, unneeded fork on the diagram, so he deletes it from the diagram. 13. Satisfied with this version of the diagram he saves a copy to his hard drive for future use. 14. He then opens his main project so he can add this component into his design. 15. Stecker now compiles and runs his main application, observing the results.

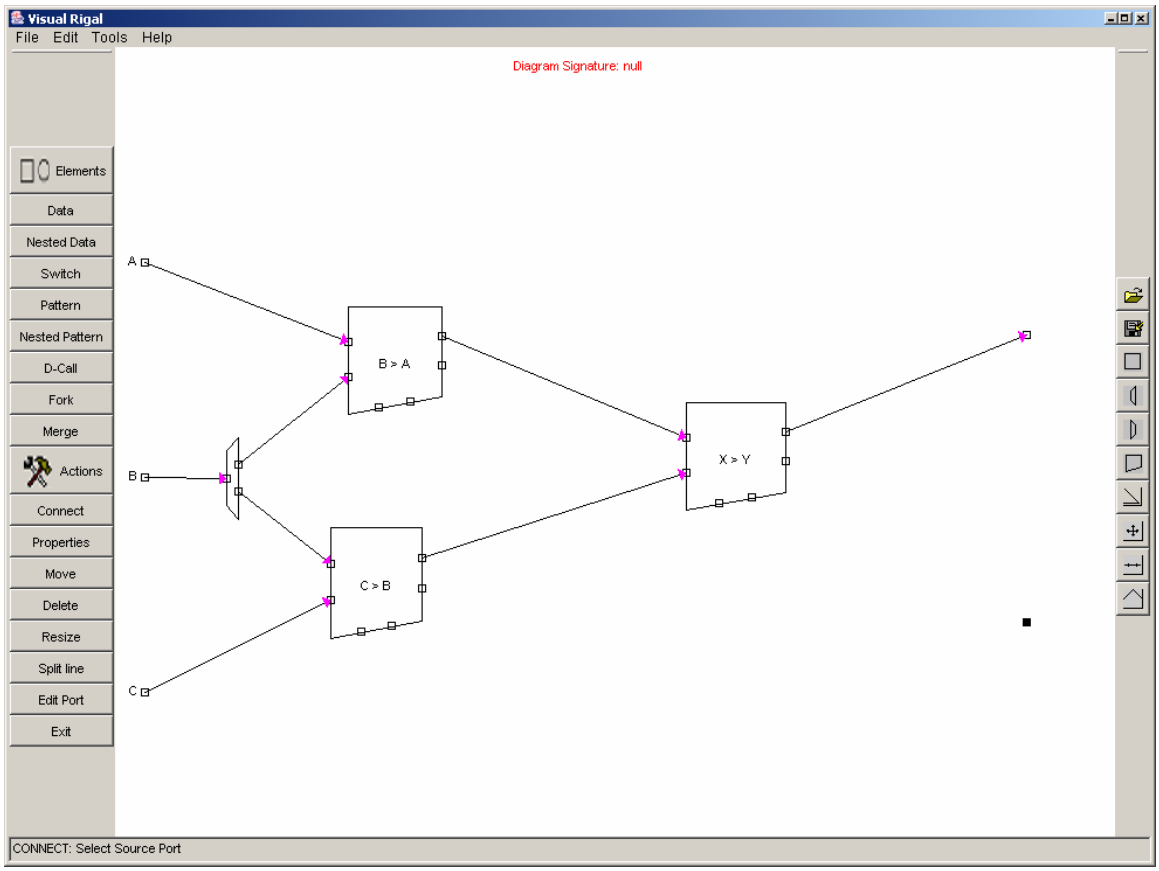
Steps 2,13 Question: What is the difference between a diagram title and signature? (In the prototype only the title appears in the text file.)

Answer: Actually, both the title (the name of program represented by the diagram) and the signature of this program are needed.

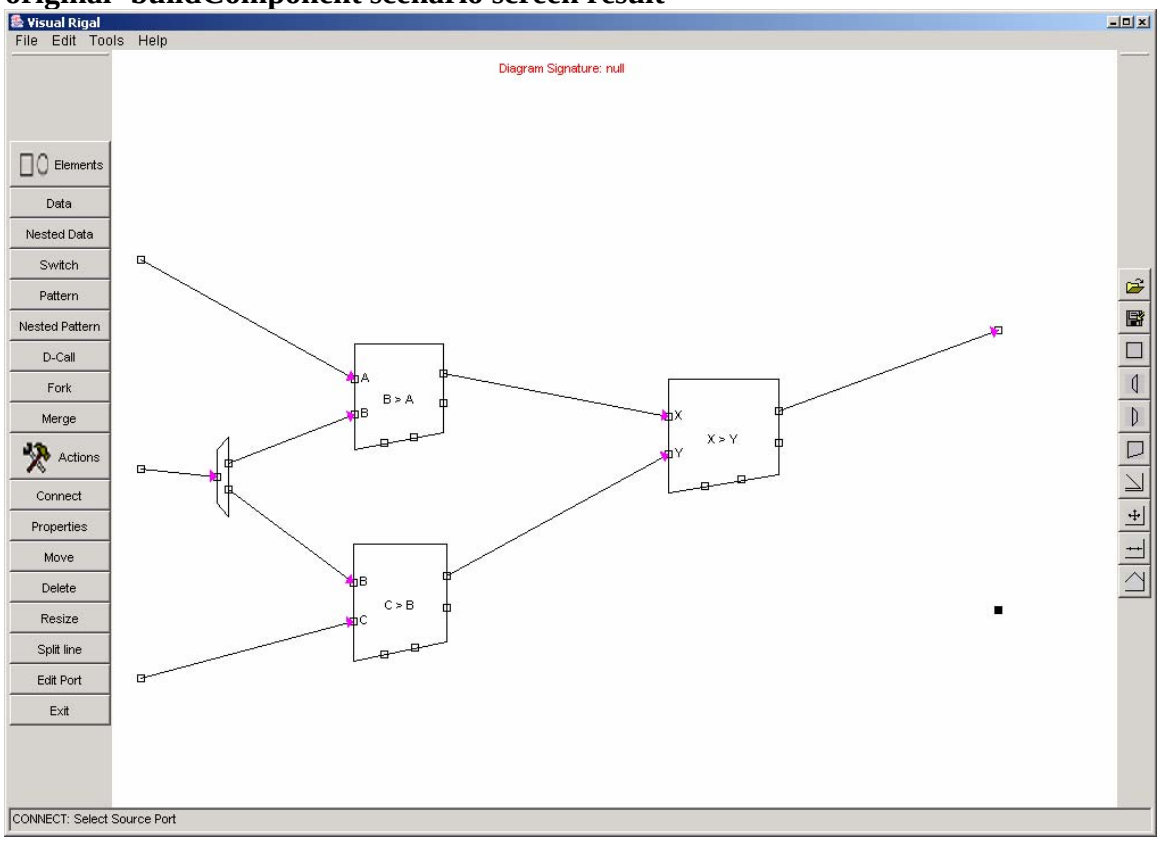
Current prototype just does not store the signature in the output text file...

Step 7: Customer input: Diagram ports may not be labeled. Note picture corrections.

Step 14: Customer question: How is this accomplished? Answer: By using the rule call node representing the just saved diagram to add this “component” to a higher level diagram.



original- buildComponent scenario screen result

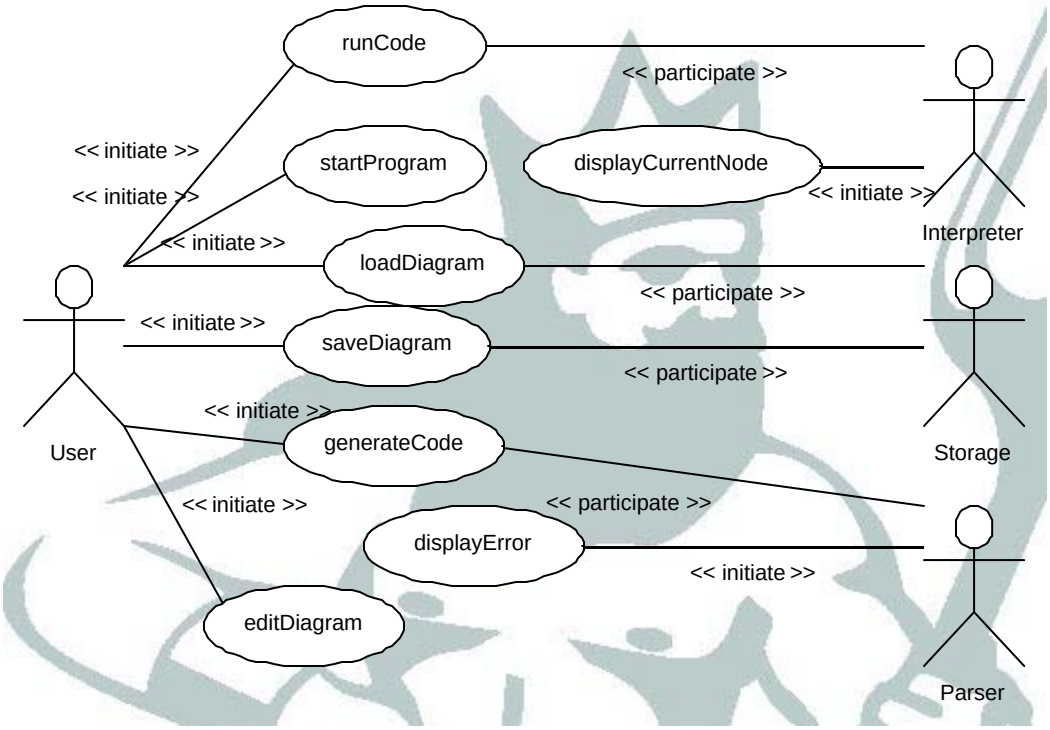


corrected- buildComponent scenario screen result

generateCodeErroneousDiagram

Scenario name	generateCodeErroneousDiagram
Participating actor instances	bruce: User
Flow of events	1. Bruce, having completed his diagram, clicks on "Tools" in the menu bar and then "Generate Code". 2. Bruce enters "FirstTry" as the filename to which to save the diagram in the 'Save As' dialog window. 3. Bruce receives acknowledgment the code is being generated. 4. Bruce receives an error message indicating a port is unbound on a node called "Data1" and on a node called "Merge2". 5. Bruce acknowledges the error to close the error message window. 6. Bruce corrects the error by connecting "Data1"s unbound ouput port to "Merge2"s unbound input port 7. Bruce right-clicks on "Tools" in the menu bar and right-clicks "Generate Code" 8. Bruce receives acknowledgment the code is being generated. 9. Bruce receives acknowledgment the code has successfully been generated.
Step 4: Customer input: Ports may be unconnected, it is not an error. Error messages are in the form of highlighted nodes.	

6. Use case model



runCode

No further detail

startProgram

No further detail

displayCurrentNode

No further detail

loadDiagram

No further detail

saveDiagram

No further detail

GenerateCode

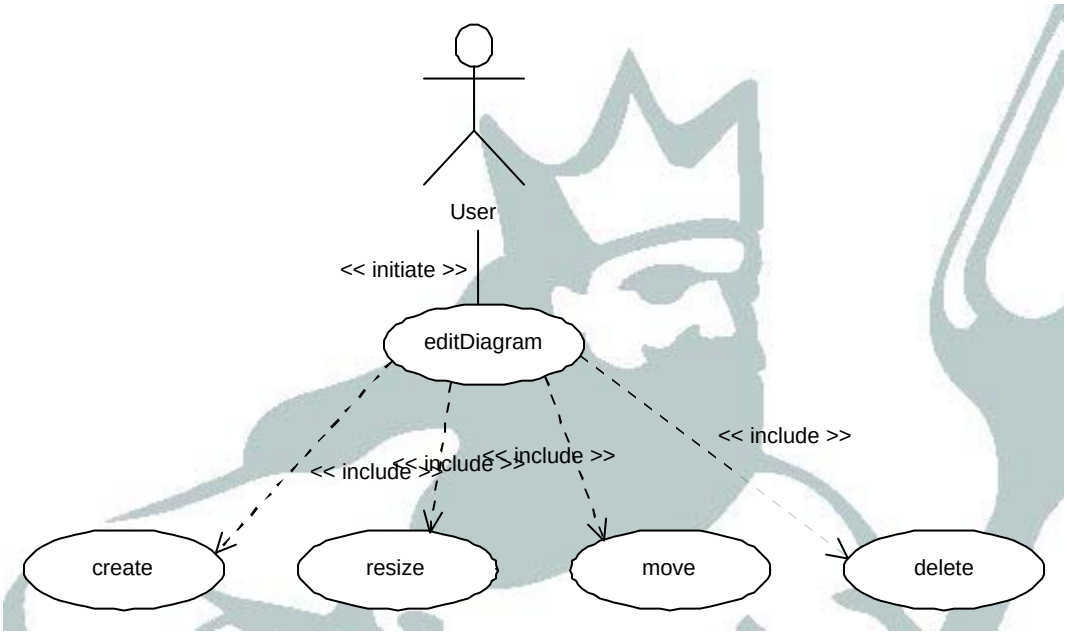
Use Case Name	GenerateCode
Participating Actors	Initiated by User Communicates with Interpreter
Flow of events	The User left-clicks on "Tools" in the menu bar and then selects "Generate Code". 1. The system responds by displaying a message dialog

	stating that the diagram is being saved and interpreted. 2. The system removes any highlighting from Nodes previously in error, saves the diagram to text, and passes the text file to the Interpreter. 3. The system receives acknowledgment that the Interpreter completed processing the file. *If the system receives an error from the Interpreter, the ReceiveInterpreterError use case is used.* 4. The system displays a message dialog, indicating that the code has been generated. 5. The User acknowledges completion by right-clicking the OK button to close the message dialog.
Entry Conditions	1. A Diagram exists.
Exit Conditions	• A text file exists representing both the placement of the elements of the diagram and the data represented by the diagram. • Code generated from the diagram exists.

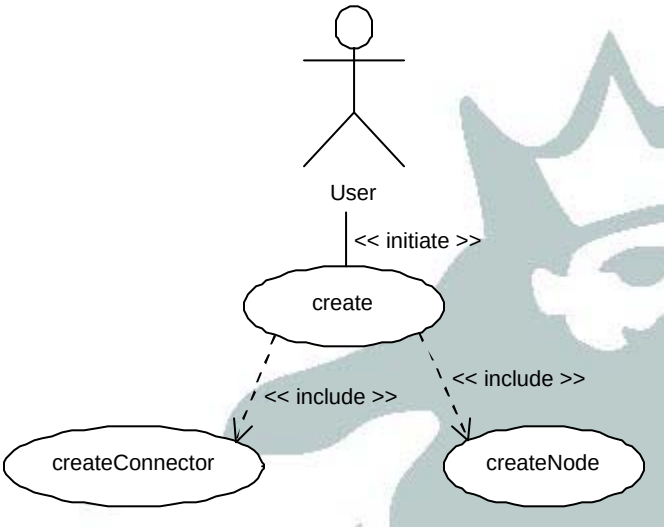
displayError

Use Case Name	displayError
Participating Actors	Initiated by Interpreter Communicates with User
Flow of events	1. The Interpreter sends an error message to the system. 2. The system displays an appropriate message dialog to the User. 3. The User acknowledges the error by right-clicking the OK button to close the message dialog. 4. The system highlights the Node or Nodes responsible for the error.
Entry Conditions	1. An attempt to Generate Code has been made. 2. An error in the diagram caused an Interpreter error.
Exit Conditions	• Code Generation is aborted. • The Node(s) most likely in error are highlighted.

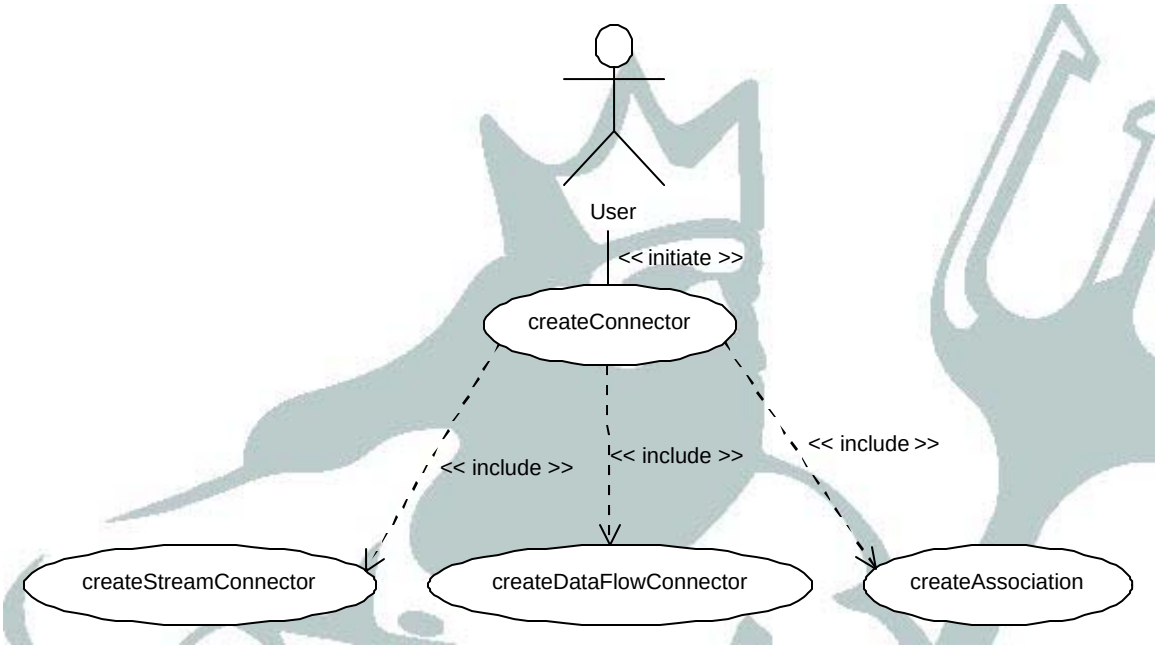
editdiagram



create



createConnector



(a) CreateStreamConnector

Use Case Name	CreateStreamConnector
Participating Actors	Initiated by User
Flow of events	1. The User left-clicks on the Connector button in the ToolBar. 2. The system indicates that it is in Connector Mode by highlighting the Connector button and displaying "Connector Mode" in the Status Bar. 3. The User left-clicks on a Port which will be one end of the Connector. 4. The system draws a Connector with one end at the selected Port and the other at the Cursor. If the selected Port is an InputPort, the head of the Connector will be at that port, otherwise, the head will be at the Cursor. 5. The User moves the Cursor to a different location in the Diagram. 6. The system continues to redraw the arrow as described in Step 4. 7. The User left-clicks on a Port which will be other end of the Connector.

	8. The system draws a Connector with one end at the first Port and the other at the newly-selected Port.
Entry Conditions	1. Two Nodes exist in the Diagram 2. There exists at least one unconnected InputPort and one unconnected OutputPort.
Exit Conditions	• System is in Connector Mode
Assumptions	• Connectors can only connect one InputPort to one OutputPort.

(b) CreateDataFlowConnector

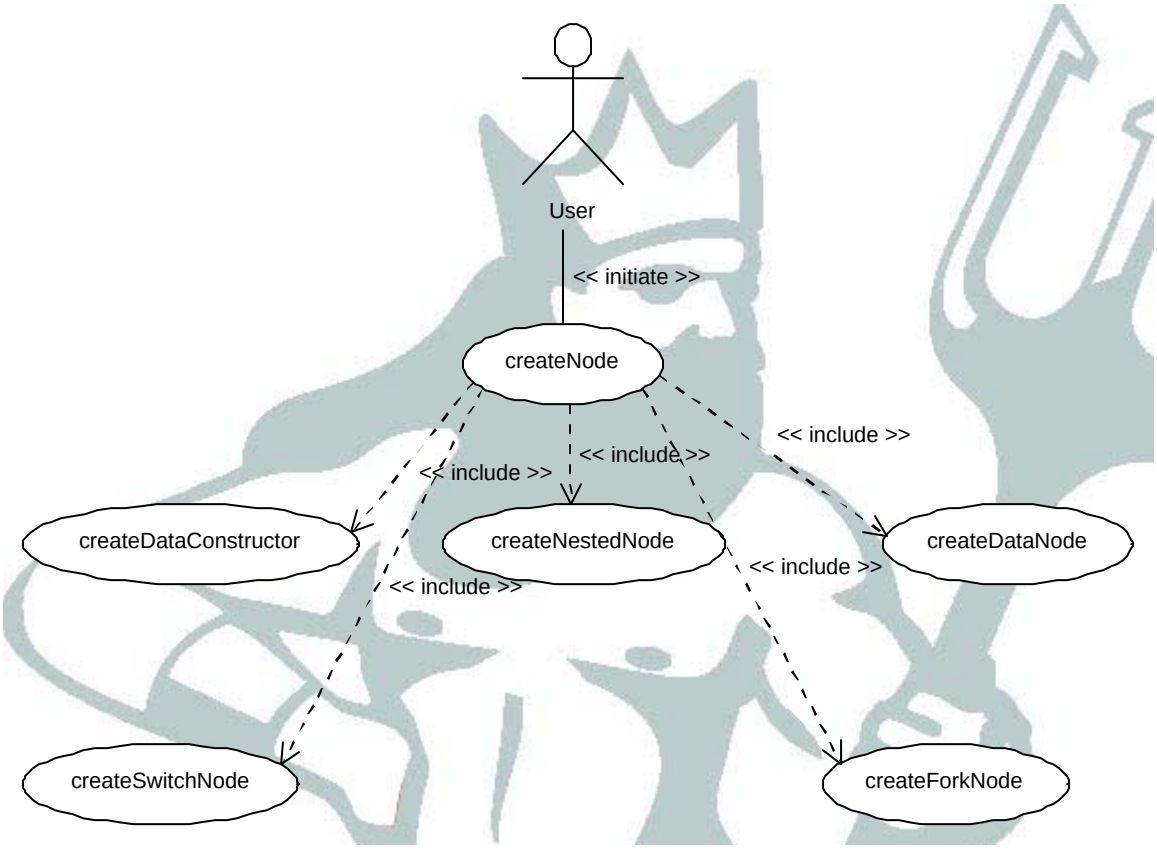
Use Case Name	CreateDataFlowConnector
Participating Actors	Initiated by User
Flow of events	1. The User left-clicks on the Dataflow Connector button in the ButtonBar. 2. The system indicates that it is in Dataflow Connector Mode by highlighting the Dataflow Connector Button and displaying "Dataflow Connector Mode" in the Status Bar. 3. The User left-clicks on the Pallet where she wants the center of the pair of Dataflow Connectors (DataFlowHead and DataFlowTail Nodes). *Customer input: the parser will check for consistency. DF-head and DF-tail can be independently created. Multiple DF-connectors can have the same name.* 4. The system displays a Dataflow Connector Properties Window. 5. The User modifies the Name field of the Dataflow Connector Properties Window. 6. The system draws a pair of Dataflow Connectors on the Pallet centered on the point the User clicked and annotates both with their name. The DataFlowHead is annotated with an InputPort and the DataFlowTail is annotated with an OutputPort. 7. The User moves the DataFlowHead Node to its appropriate

	location. *Include **MoveNode** use case*. 8. The User connects an Arrow to the InputPort of the DataFlowHead. *Include **CreateConnector** use case*. 9. The User moves the DataFlowTail Node to its appropriate location. *Include **MoveNode** use case*. 10. The User connects an Arrow to the OutputPort of the DataFlowTail. *Include **CreateConnector** use case.*
Entry Conditions	1. None
Exit Conditions	• System is in Connector Mode

(c) createAssociation

Not detailed

createNode



(d) CreateDataConstructor

Use Case Name	CreateDataConstructor
Participating Actors	Initiated by User
Flow of events	1. The User left-clicks on the Data Constructor button. 2. The system indicates that it is in Data Constructor Mode by displaying “Data Constructor Mode” in the Status Bar. 3. The User left-clicks on the Pallet where she wants the center of the outer box. 4. The system displays a Data Constructor Properties Window. 5. The User modifies the Name and Output Port Name fields of the Data Constructor Properties Window. *Customer note: Output ports may not be named.* 6. The system draws a rectangle on the Pallet centered on the point the User clicked and annotates the rectangle with the a output port labeled with its name and labels the rectangle with the Node’s Name.
Entry Conditions	2. None
Exit Conditions	• System is in Data Constructor Mode • Output ports of the Data Constructor Node are disconnected • Node appears to be a regular Data Node, but has no Input Ports.
Assumptions	• Data Constructors always have no input port and one output port • Data Constructors have at least one nested Data Node (input)

(e) CreateNestedNode

Lead designer note: Should be a mode. Development team: from 3.4.1.3, Further elaboration has determined that the nesting state will be determined exclusively by the graphical depiction of one element inside the boundaries of another. The parser will check for illegal conditions. This direction is chosen to reconcile apparent visual logic with underlying reality.

Use Case	CreateNestedNode

Name	
Participating Actors	Initiated by User
Flow of events	1. The User left-clicks on the Data Node button in the toolbar. 2. The system responds by indicating that it is in Data Node Mode by displaying “Data Node Mode” in the status bar. 3. The User left-clicks at a position on the pallet within a Data Constructor Node to indicate where the Data Node will be placed. 4. The system responds by displaying a Properties Window. 5. The User fills out the Name and Input Port Name fields, and indicates whether or not to annotate the Node with ellipses (‘…’). in the Properties Window. 6. The system draws a black rectangle within the borders of the Data Constructor Node and annotates the rectangle with ellipses, if applicable, and appropriately labeled input ports, The rectangle is also labeled with the name of the Data Node.
Entry Conditions	1. A Data Constructor Node exists on the Pallet
Exit Conditions	• The system is in Data Node Mode • The Data Node’s input ports are disconnected • The Data Node has no output ports (differs from CreateDataNode)
Assumptions	• The Nested Data Node can have 1 or more input ports. • The Nested Data Node has no output ports.

(f) CreateDataNode

Use Case Name	CreateDataNode
Participating Actors	Initiated by User
Flow of events	1. The User left-clicks on the Data Node button in the toolbar. 2. The system responds by indicating that it is in Data Node Mode by displaying “Data Node Mode” in the status bar.

	3. The User left-clicks at a position on the pallet within a Data Constructor Node to indicate where the Data Node will be placed. 4. The system responds by displaying a Properties Window. 5. The User fills out the Name, Input Port Names, and Output Port Names fields in the Properties Window. *Customer note: Output ports may not be named.* 6. The system draws a black rectangle within the borders of the Data Constructor Node and annotates the rectangle with the appropriately labeled input and output ports and labels the rectangle with the name of the Data Node. *Customer note: Output ports may not be named.*
Entry Conditions	1. The Visual Editor program is executing.
Exit Conditions	• The system is in Add Data Node mode • The Data Node's input and output ports are disconnected
Assumptions	• The Node's name also represents the mapping of inputs to outputs in the form of an expression.

(g) CreateSwitchNode

Use Case Name	CreateSwitchNode
Participating Actors	Initiated by User
Flow of events	1. The User left-clicks on the Switch Node button in the toolbar. 2. The system responds by indicating that it is in Switch Node Mode by displaying "Draw a Binary Switch" in the status bar. 3. The User left-clicks at a position on the pallet in order to indicate where the Switch Node will be placed. 4. The system responds by displaying a Properties Window. 5. The User fills out the Name, and Input Port Names in the Properties Window.

	6. The system draws a four sided polygon and annotates the polygon with the appropriately labeled input and output ports and labels the polygon with the name of the Switch Node.
Entry Conditions	1. The Visual Editor program is executing.
Exit Conditions	• The system is in Draw a Binary Switch mode • The Switch Node's input and output ports are disconnected
Assumptions	• The Node's name also represents the mapping of inputs to outputs in the form of an expression.

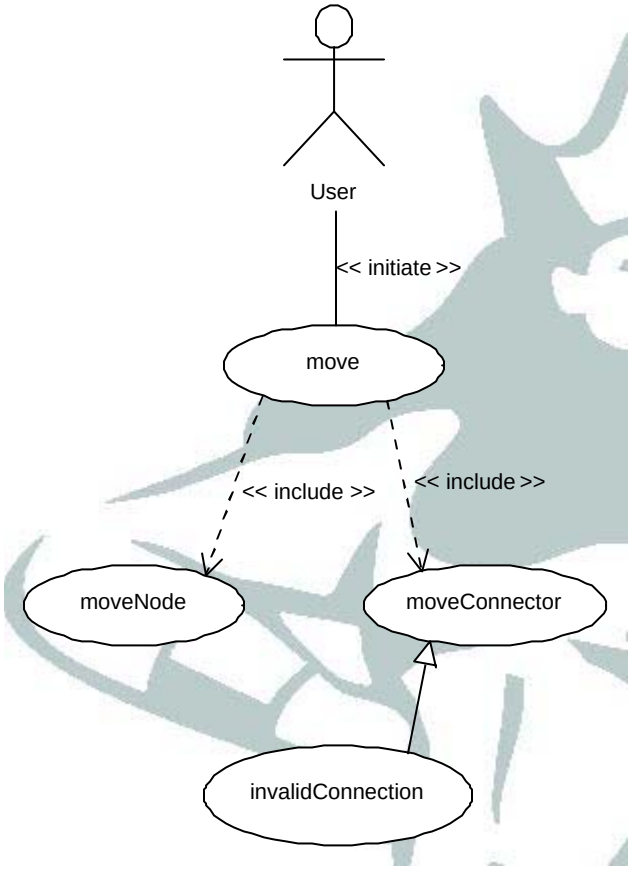
(h) CreateForkNode

Use Case Name	CreateForkNode
Participating Actors	Initiated by User
Flow of events	1. The User left-clicks on the Fork Node button in the toolbar. 2. The system responds by indicating that it is in Fork Node Mode by displaying "Add Fork Element" in the status bar. 3. The User left-clicks at a position on the pallet in order to indicate where the Fork Node should be placed. 4. The system responds by displaying a Properties Window. 5. The User sets the number of output ports in the Properties Window. 6. The system draws a trapezoid with one input port and the number of output ports specified by the user.
Entry Conditions	1. The Visual Editor program is executing.
Exit Conditions	• The system is in Add Fork Element mode • The Fork Node's input and output ports are disconnected
Assumptions	

resize

No further detail

move



MoveNode

Use Case Name	MoveNode
Participating Actors	Initiated by User
Flow of events	1. The User left-clicks on the Move button in the ToolBar. 2. The system indicates that it is in Data Constructor Mode by highlighting the Move Button and displaying “Move Mode” in the Status Bar. 3. The User left-click-and-holds on the Node she wishes to move. 4. The system redraws the selected Node with a dashed outline.

	5. The User continues to hold the left mouse button and drags the Node to a new position in the Diagram. 6. The system continues to redraw the Node, any Nested Nodes, and associated Ports and Connectors as the Node is moved. ('The Node chases the cursor') 7. The User releases the left mouse button. 8. The system redraws the Node with a solid outline and redraws any Nested Nodes, and associated Ports and Connectors
Entry Conditions	1. A Node exists in the Diagram
Exit Conditions	• System is in Move Mode • Node, its Nested Nodes, and associated Ports and Connectors are moved.

MoveConnector

Use Case Name	MoveConnector
Participating Actors	Initiated by User
Flow of events	1. The User left-clicks on the Move button in the ToolBar. 2. The system indicates that it is in Move Mode by highlighting the Connector button and displaying "Move Mode" in the Status Bar. 3. The User left-clicks on a Port to which the end of the Connector to be moved is attached. 4. The system draws a Connector with one end at the Cursor and the other at the Connector's other Port ('anchor'). If the selected Port is an InputPort, the head of the Connector will be at the Cursor, otherwise, the head will be at the anchor. 5. The User moves the Cursor to a different location in the Diagram.

	6. The system continues to redraw the arrow as described in Step 4. 7. The User left-clicks on a Port which will be new location of the 'loose' end of the Connector. *8.* The system draws a Connector with one end at the anchor Port and the other at the newly-selected Port.
Entry Conditions	1. There exists at least one Connector.
Exit Conditions	• System is in Connector Mode
Assumptions	• Connectors can only connect one InputPort to one OutputPort. • Connector can be moved from a Port to the same Port (trivial)

InvalidConnection

Customer input: This should be avoided. Development team: Use of the system is envisioned beyond the immediate customer. We consider this minor functionality.

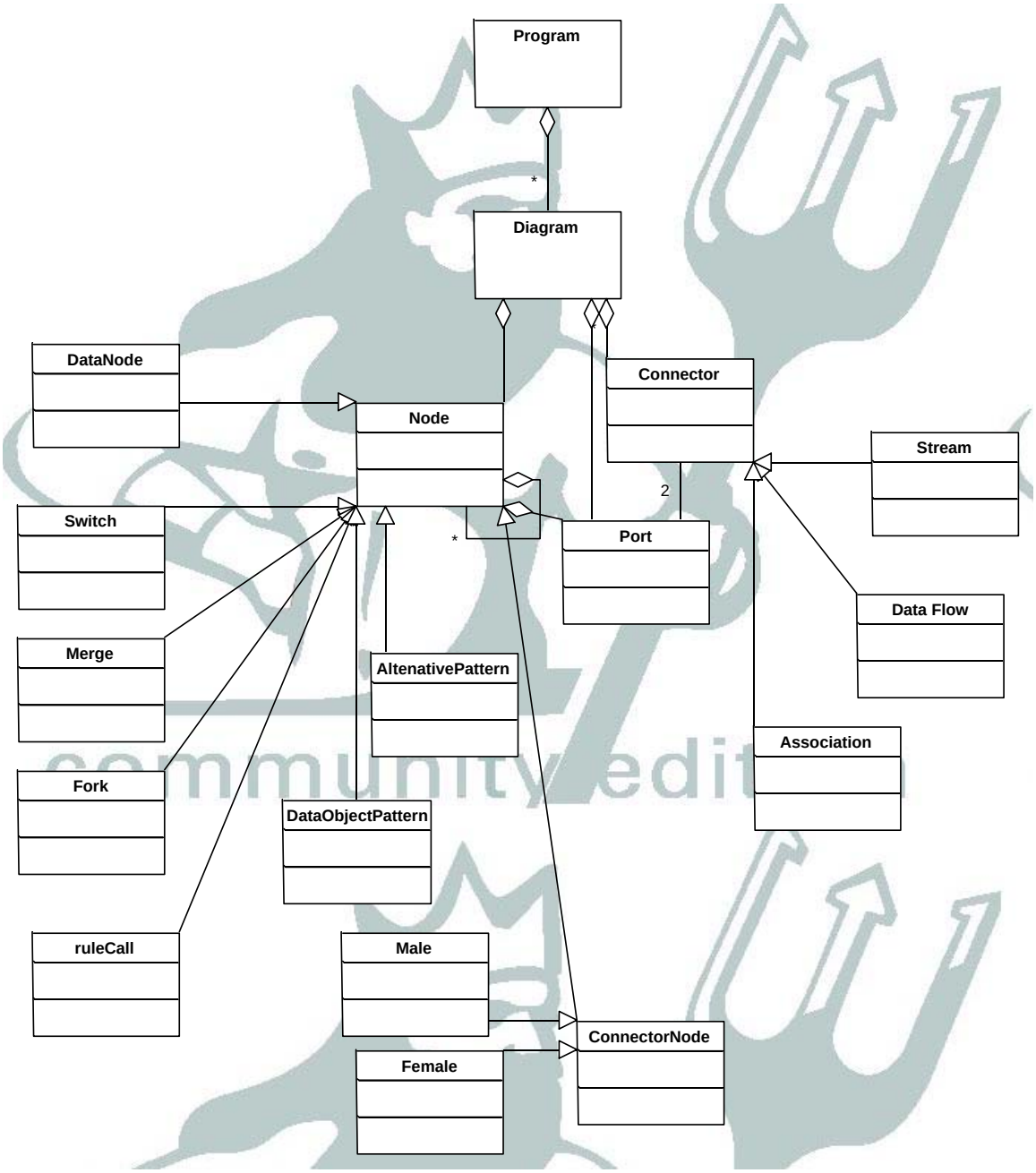
Use Case Name	InvalidConnection
Participating Actors	Communicates with User
Flow of events	*The **InvalidConnection** use case extends any use case that connects Connectors to Ports in which a Connector is attempted to connect to anything other than an appropriate port (InputPort or OutputPort).* 1. The system displays an error in the Status Bar, appended to the current message. 2. The system returns the Connector to its previous Port. If there is no previous Port (as in CreateConnector), do nothing.
Entry Conditions	1. System is in Connector Mode. 2. An attempt to connect a Connector to either nothing or an inappropriate port was made.
Exit Conditions	• System is in Connector Mode • An error message is appended to the current message in the Status Bar
Assumptions	• Connectors can only connect one InputPort to one OutputPort.

delete

No further detail.

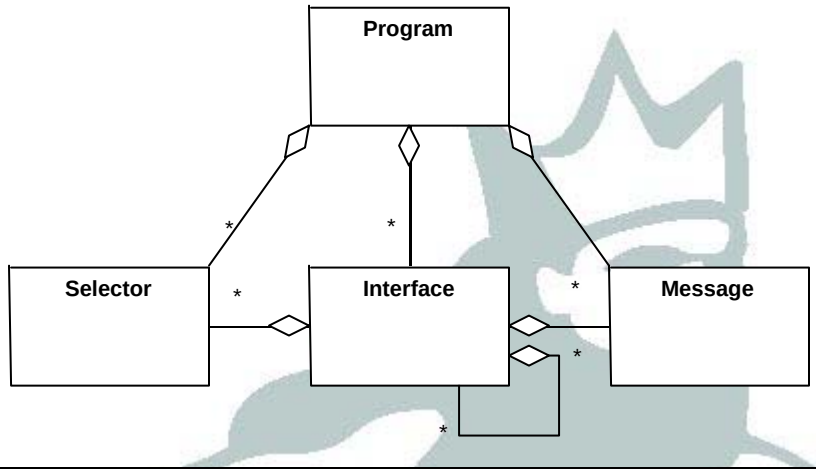
7. Object model

Entity Objects



Entity Object	Attributes & Associations	Definition
Diagram	• Name • Contained Nodes • Contained Ports • Contained Connectors	The Diagram is the canvas on which the User creates a diagram. It encompasses all the items from which code will be generated.
Node	• Unique Identifier • Label • List of Input Ports • List of Output Ports • Size • Position in Diagram • List of Nested Nodes	The Node is the basic element with which the User creates the Diagram. The node has 0 or more InputPorts and 0 or more OutputPorts. Nodes can be nested within other nodes.
Port	• Unique Identifier • Label • Associated Node • Associated Connector	The User uses Ports to connect one Node with another. A Port is, in essence, attached to a Node and is connected to one end of a Connector. A Port can be connected to only one Connector.
StreamConnector	• Unique Identifier • Input Port • Output Port	The User uses Stream Connectors to represent a directional relationships between Nodes. A Connector connects one InputPort to one OutputPort and buffers input while awaiting output.
DataFlowConnector	• Unique Identifier • Head Connector • Tail Connector	A directional Connector between a matched pair of Nodes. Data items flow one at a time through this connection.

Boundary Objects

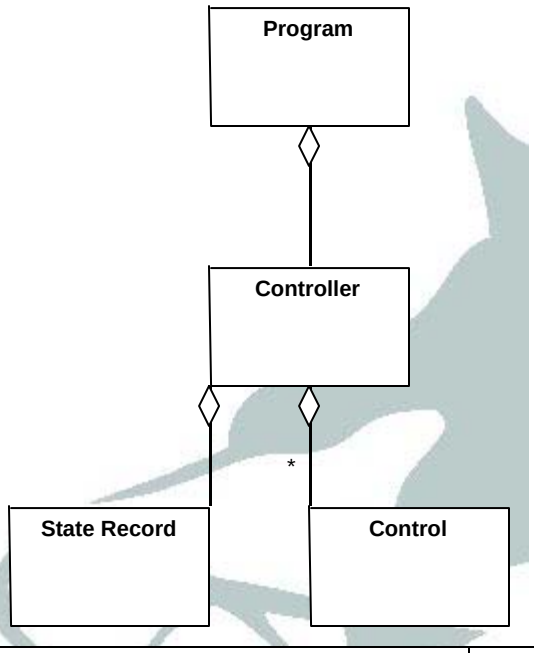


Boundary Object	Definition
PositionSelector	Inputs coordinates
UserStatusMessage	• Current Mode of the Editor • Error Messages • Text Display Field
ModeSelector	Inputs new Mode
DataNodePropertiesWindow	Interface used by User to specify the properties of a DataNode during creation or editing
SwitchNodePropertiesWindow	Interface used by User to specify the properties of a SwitchNode during creation or editing
MergeNodePropertiesWindow	Interface used by User to specify the properties of a MergeNode during creation or editing
ForkNodePropertiesWindow	Interface used by User to specify the properties of a ForkNode during creation or editing
DCallNodePropertiesWindow	Interface used by User to specify the properties of a DCallNode during creation or editing
PatternNodePropertiesWindow	Interface used by User to specify the properties of a PatternNode during creation or editing

DataFlowConnectorPropertiesWindow	Interface used by User to specify the properties of a DataFlowConnector and its components, DataFlowHeadNode and DataFlowTailNode during creation or editing
PortPropertiesWindow	Interface used by User to specify the properties of a Port during creation or editing
SavingMessageDialog	Message received by User that the Diagram is being saved to Disk
CodeGeneratedMessageDialog	Message received by User that Code has been successfully generated from the current Diagram
InterpreterErrorMessageDialog	Message received by User that an error occurred while generating code from the current Diagram
DiagramOpenDialog	Interface used by User to find and open a Diagram from Disk.
SaveDiagramAsDialog	Interface used by User to specify the name and location of the file to which the current Diagram will be saved.
Parser Interface	Bridge connecting System with Parser, allows error display and execute compilation message
Execute Compilation Message	Directs Parser to compile file(s)
Interpreter Interface	Bridge connecting System with Interpreter, allows stepped execution of code for debugging and current node display.
Interface Selector	High level selector, allows user to pick a high level interface for use.

Control Objects

The state information required to initialize the system and maintain current status of the system is in the “State Record”. Various control objects for different tasks ; file I/O, parser function, interpreter function, etc.; complete the Controller.

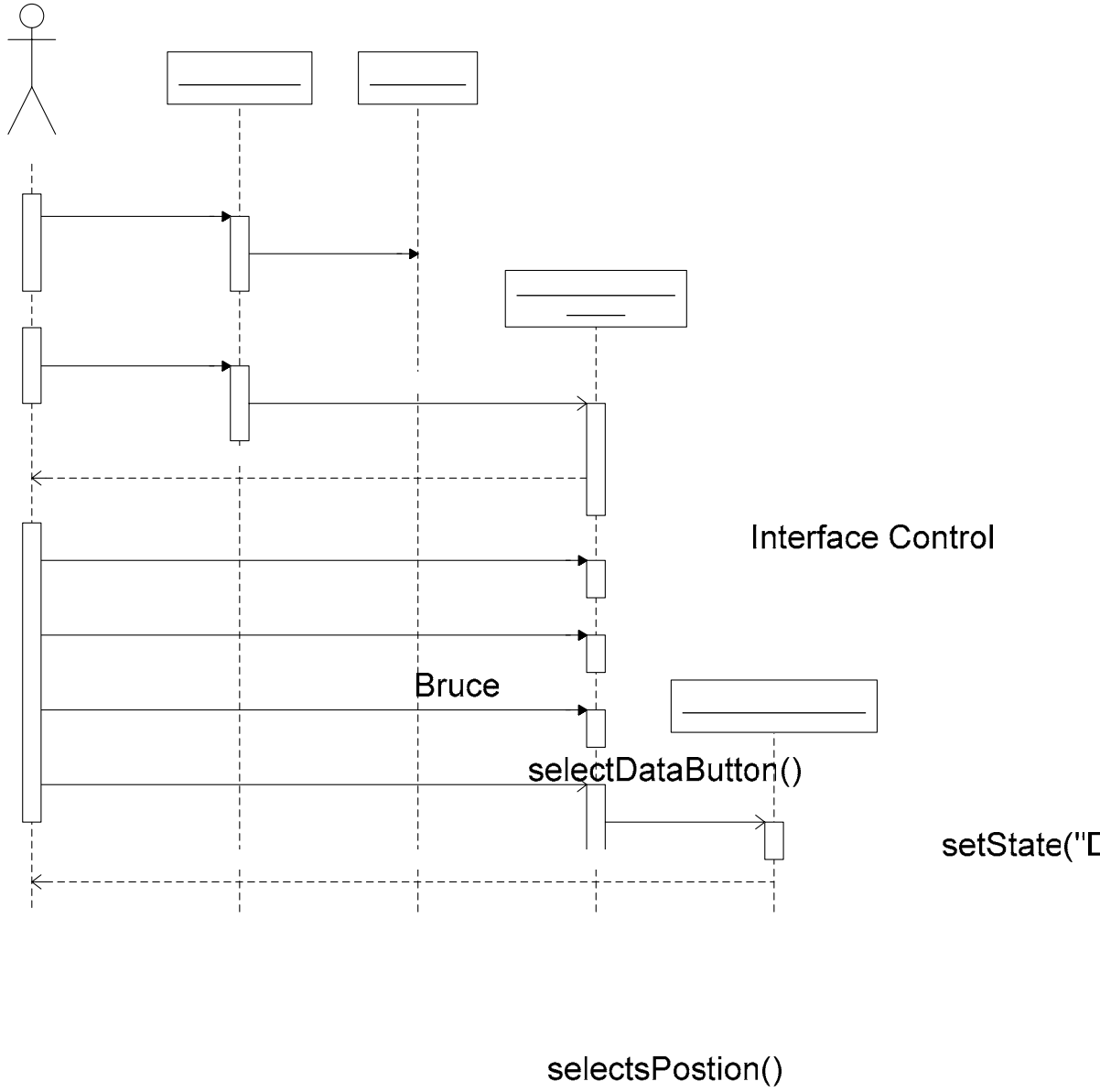


Control Objects	Definition
System Interpreter Input Control	Accepts and distributes asynchronuse messages from the interpreter for use by control objects.
Controller	High level abstraction of all control objects
State Record	Persistent data attributes of system
Diagram Selection Control	Displays the appropriate diagram given a node identification. Shows the diagram that contains that node.
Diagram Node Highlighted Control	Creates a highlighted effect over the given node. Calls attention to that node. Also used to reverse this effect.
File Input/Output Controller	Controls all file I/O
System Parser Input Control	Accepts and distributes asynchronuse messages from the interpreter for use by control objects.
Debug Control	Maintains status of interaction and intention with the Interpreter.
Compiler Control	Maintains status of interaction and intention with the Parser.
Initialization Control	Responsible for setting initial State Record values, creating the primary interface and blank initial diagram.
Diagram Save and Close Control	When closing checks to see if saved previously, if not offers to save.
Exit Application Control	Manages housecleaning functions before termination.

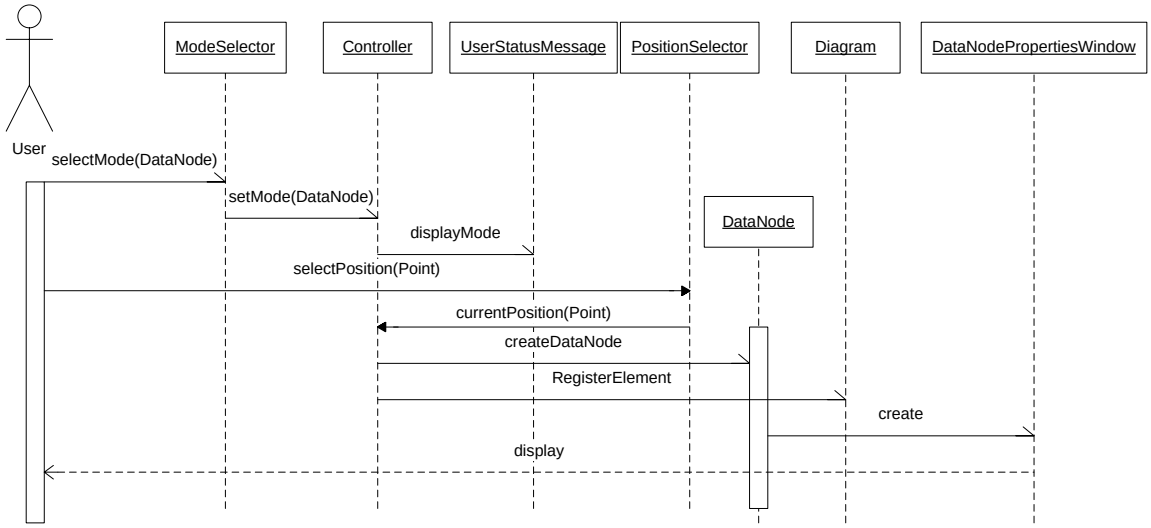
New Diagram Control	Manages creation of a fresh diagram.

8. Dynamic model

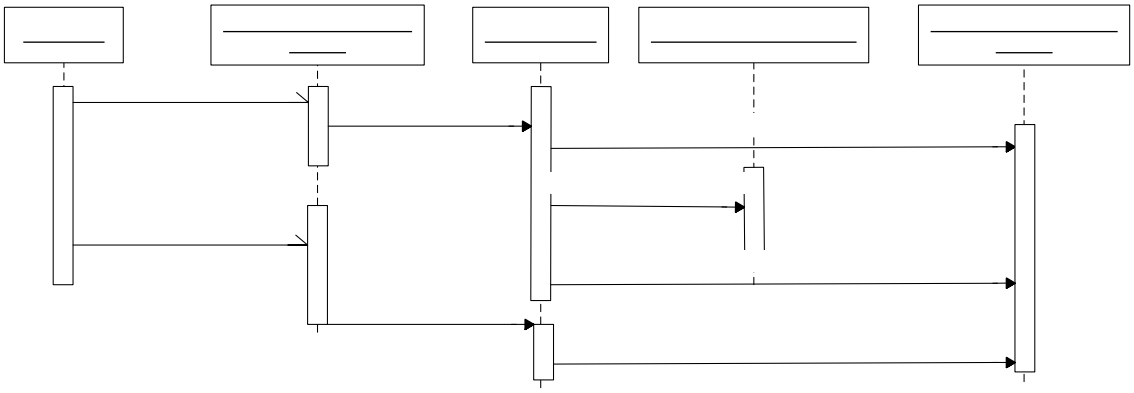
CreateTestNest Sequence Diagram



CreateDataNode Sequence Diagram

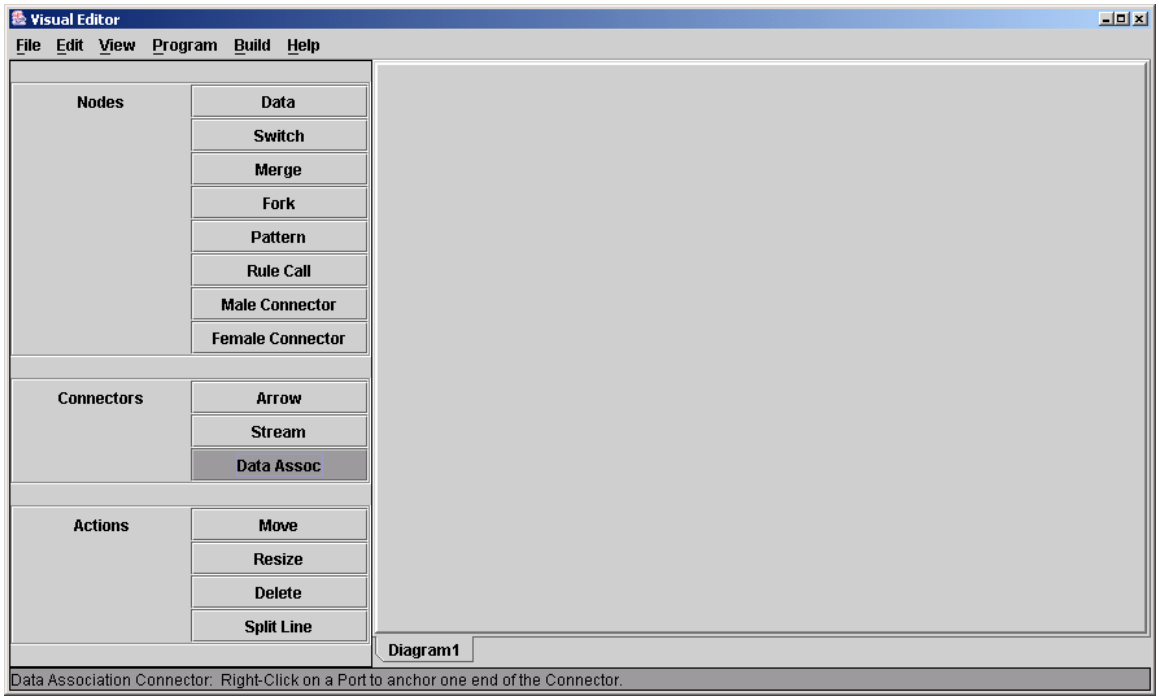


displayCurrentNode Sequence Diagram



9. User interface- navigational paths and screen mock-ups

A java jar file accompanies this development package and represents our screen mock-up and menu navigation proposal. Note the active functionality of the current mode display.



GLOSSARY

ACTIONS - User created events.

CANVAS – Portion of the user interface on which the Diagram is created and displayed. The working area of the GUI.

CONNECTORS - Represents data paths, associates ports or nodes to one another.

DIAGRAM - Displayable collection of elements, their relationships and positions. A diagram is a graphical representation of an executable program.

ELEMENT - High level abstraction for graphical elements: node, connector, port.

INTERPRETER – A separate program that takes a diagram as input and executes the program described by the diagram.

MODE - Current state of Visual Editor, defines behavior expected by user interaction. Each Node type has a unique mode, as does each Connector and Action types.

NESTED - The graphical and logical encapsulation of a Data Node or Pattern by another of the same type. The depth of nesting has no limit.

NODE - Logical and graphical element that represents structure and/or operation.

PARSER – A program that checks a diagram for syntax errors.

PATTERN -

Alternative Pattern- only continues on match
Atomic Pattern- constant to attempt match with
Rule Call- Subroutine, defined elsewhere (in another file)
Data Object Pattern- Filter
Pattern Group- contains other Icons, onscreen defined subroutine

PORT - A place connectors can be associated with, described in entity objects. Represents data input or output of a node.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Marine Corps Representative
   Naval Postgraduate School
   Monterey, California

4. Director, Training and Education, MCCDC, Code C46
   Quantico, Virginia

5. Director, Marine Corps Research Center, MCCDC, Code C40RC
   Quantico, Virginia

6. Marine Corps Tactical Systems Support Activity (Attn: Operations Officer)
   Camp Pendleton, California

7. Graham Pierson
   Naval Postgraduate School
   Monterey, California

8. Mikhail Auguston
   Naval Postgraduate School
   Monterey, California

9. Scott Coté
   Naval Postgraduate School
   Monterey, California

10. William Welch
    Naval Postgraduate School
    Monterey, California