Visualizing Visual Prolog

Chan Bok, Axon Research
http://web.singnet.com.sg/~axon2000

Abstract: This paper looks at some aspects of visualization of Visual Prolog at various levels of detail. At the predicate level, a combination of Ferguson Diagram and Nassi-Schneiderman Diagram is suggested. At a more conceptual level, a special form of Class Diagram is suggested. The latter is further illustrated by three case studies involving COM.

Keywords: *Class Diagram, Ferguson Diagram, Nassi-Schneiderman (N-S) chart, Prolog Foundation Class (PFC), COM, ActiveX.*

Introduction

The author's interest in prolog visualization began in the early nineties when he first learned Visual Prolog [1]. Being able to represent prolog codes diagrammatically has shown to be invaluable in understanding the language. Subsequently the use of Class Diagrams has helped in understanding and using the packages found in the Prolog Foundation Class (PFC).

This paper does not consider dynamic visualization of algorithm or data. Such visualization is best done using animated debuggers and profilers.

What constitutes a Good Diagram

Firstly it is worthwhile to highlight some factors that make a "good" diagram. In particular, a good representation should:

- Complement and "play to the strength of human perception, memory, comprehension and reasoning while avoiding their weaknesses" [4].
- Represent many useful attributes such as entities, its types, properties, and relations between entities.
- Exploit the power of human vision to perceive shape, size, texture, proximity, boundary, depth, opacity, symmetry, etc.
- Be clear and readable even at a distance.
- Can be easily modified to reflect better understanding and changing needs.

Visualizing Predicate Logic

Visualization at the predicate level is particularly useful for a beginner to Prolog. A diagram at the predicate level can denote clauses, facts, unification, recursions, and related details at this level.

At this level, predicate logic can best be represented using Ferguson diagrams characterized by semi circles. The ◠ symbol represents logical negation or goal, and the ◡ symbol represents assertion, fact or rule. Unification is thus represented by a matched pair forming a full circle (see Figure 1 below).

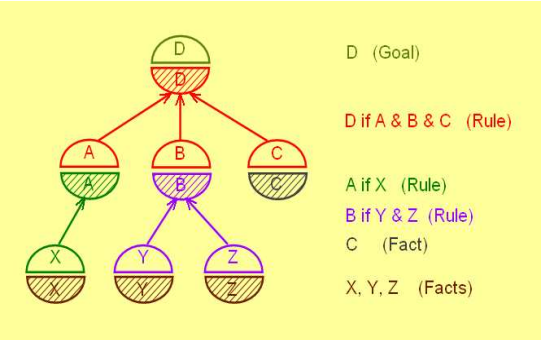


Figure 1 – A simple Ferguson Diagram for representing predicate logic.

A Ferguson diagram can be extended to show variables and its direction of flow. (As the Chinese saying, by "drawing the intestines"). We illustrate with a more complex program below. This program (adapted from a PDC publication, see Appendix A) solves the critical path problem declaratively.

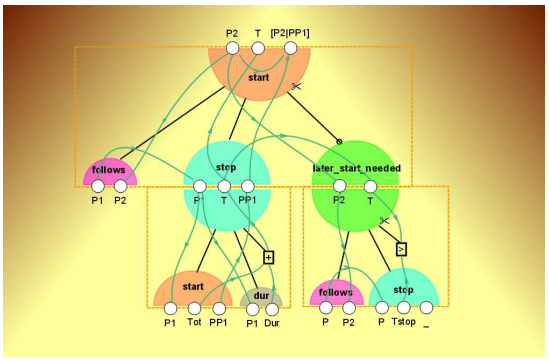
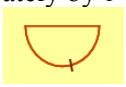


Figure 2 - Ferguson diagram showing variables and flow direction. See Appendix A for larger diagram.

Various types of predicates (viz. procedure, determ, nondeterm and failure) can be shown using some annotations or variations of a semicircle. A cut can be indicated by a short bar (or more elaborately by ✂) at the edge of the semicircle. 

Since VIP7.0, the *if-then-else, foreach*, and other constructs were introduced and these can be elegantly diagrammed by incorporating the goto-less Nassi-Schneiderman [5] flowchart (see Figure 3 and 4).

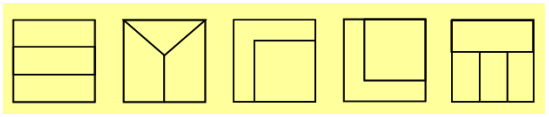


Figure 3 – Nassi-Schneiderman chart for sequence, branch, do-while, do-until and case.

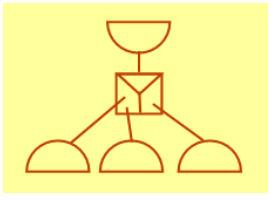
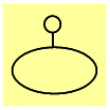


Figure 4 – Example of how N-S chart can be added into a Ferguson Diagram.

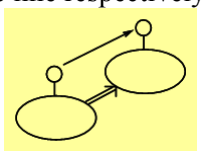
Class Diagrams for Visual Prolog

Next we look into the higher level of diagramming using Class Diagrams, which show object-oriented features such as classes, interfaces, inheritance, delegation, and packaging.

Conventional UML Class Diagram notation is not suitable for various reasons. In UML notation, a class is represented as a circle and interfaces are not represented. A more appropriate symbol for a class is an ellipse which is more elegant and economical on space. The "lollipop" symbol  which is a standard symbol for a COM interface can be adopted, whether COM or otherwise.

The combined symbol  for a class-interface pair is thus compact, aesthetic, and accurately reflects the (.pro) file as an ellipse and the (.i) file as a small circle. However the circular symbol  can be used to indicate a "support" class that does not create any object.

Support and Inheritance relationships can thus be shown as upward arrows with single and double line respectively, as follows:



Delegation can be shown as downwards dotted lines from an interface to another class.

Using the above convention, a Class Diagram for the PFC/GUI Package can be drawn as shown in Figure 5.

The GUI package is an amazing design masterpiece and much can be learned by studying it. Using a Class Diagram (see Figure 5), one is in a better position to observe and learn about:

- The relationship between Controls and Dialogs.
- What is a Container window and a Container control.
- The differences between a Form window and a Document window.
- The differences between Standard and User controls.
- How the Form window is a special case of a Dialog.
- What are the Common controls and Standard controls.
- How the OOGUI is built on top of the native Windows APIs.

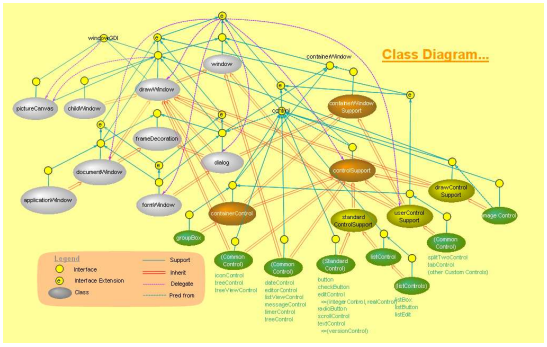


Figure 5 - A Class Diagram of the PFC/GUI package – an amazing design masterpiece. For enlarged diagram see Appendix B and also http://discuss.visual-prolog.com/viewtopic.php?t=6006

To further demonstrate the diagramming methodology and assess its usefulness, we look at three test cases related to COM, a typically complex subject:

A. COM Server and User
B. PFC/COM Package
C. Webbrowser Control

Case A - COM Server and User

The *COM Server and User* example (Figure 6) is based on the "Visual Prolog Examples" that come with installation of VIP6 or 7. Learning how to interface with a simple component introduces one to the complex COM technology involving binary/native interfaces, components, glue codes, the PFC-COM packages, etc.

Although most of the complex glue codes can be auto-generated (as can be seen in the later Case C), it is good to have an understanding of the technique in order to resolve any resultant problems.

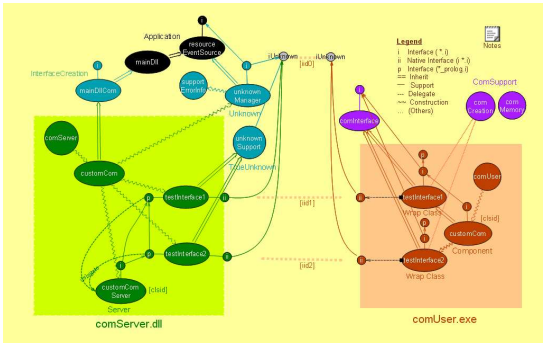


Figure 6 – The COM Server and User example covers a large number of concepts that go into COM. See Appendix C for larger diagram.

Case B - PFC/COM Package

The *PFC/COM Package* is for developing and using COM/ActiveX components. Beginners to COM technology is faced with a vast number of new concepts and new predicates. Having an overview of the available predicates and its interdependence will be helpful (as in the similar case of the PFC/GUI Package).

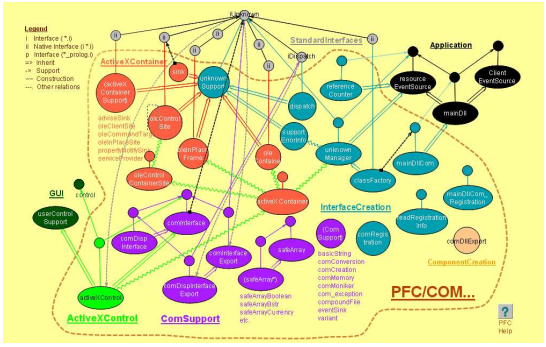


Figure 7 – The PFC/COM package is even more complex than the PFC/GUI package. See Appendix D for larger diagram.

Case C - Webbrowser Control

The *ActiveX Demo* (Figure 8) is based on the "Visual Prolog 6 Examples and Demos". This Demo shows how the Webbrowser Control (a COM/ActiveX component) can be used in Visual Prolog.

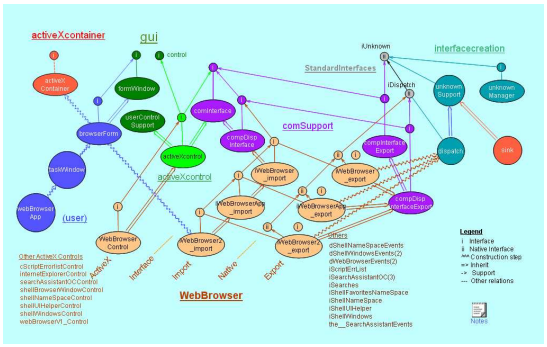


Figure 8 – The ActiveX Demo illustrates how Visual Prolog auto-generates the 5 categories of glue codes for using components. See Appendix E for larger diagram.

The Class diagram shows the structure of the generated codes organized into 5 folders (ActiveX, Import, Export, Interface, Native). We can assume that a similar pattern exists for any auto-generated codes for COM/ActiveX component integration.

The Axon Idea Processor

In this paper all diagrams were drawn using the Axon Idea Processor [2] - a visualization tool developed also in Visual Prolog.

Although not designed specifically for Prolog visualization, Axon has special features for drawing Ferguson Diagrams and browsing Visual Prolog codes.

For example, clicking on a class (ellipse) can open the underlying program files (.pro and .cl). Similarly clicking on the "lollipop" can open the interface (.i) file. In this way the tree structure of prolog files is supplemented by a more meaningful network diagram.

Another adaptation (the Idea Visualizer tool) enables progressive and selective unfolding of details (such as color, shape, links) using a slider control.

Conclusion

We have shown how Visual Prolog programs can be diagrammed at various levels of detail. In particular, Class Diagrams can provide a useful overview-cum-index and other insights to entire packages.

Visualization of Visual Prolog is a broad and complex subject. The following suggests some areas for further research:

- Automatic generation of diagrams from source codes.
- Standardization of diagramming symbols and color schemes.
- Techniques for arranging and grouping classes and minimizing long links and its criss-crossing.
- How to distribute and share such diagrams within the Visual Prolog community.

References

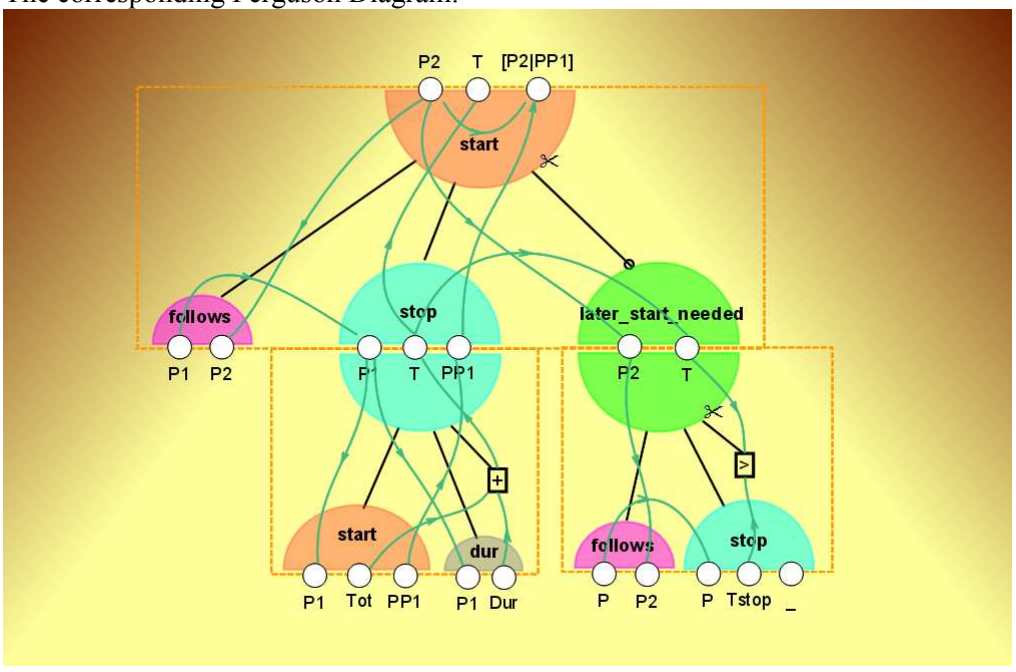
1. Visual Prolog *http://www.visual-prolog.com/*
2. The Axon Idea Processor *http://web.singnet.com.sg/~axon2000*
3. Clemens Szyperski, "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1999.
4. Christopher Chabris, Stephen Kossyln, "Representational Correspondence as a Basic Principle of Diagram Design", in *Knowledge and Information Visualization, 2005.*
5. Nassi, I. and Shneiderman, B., Flowchart Techniques for Structured Programming, SIGPLAN Notices 8, 8 (August, 1973).

Appendix A – The Critical Path Program

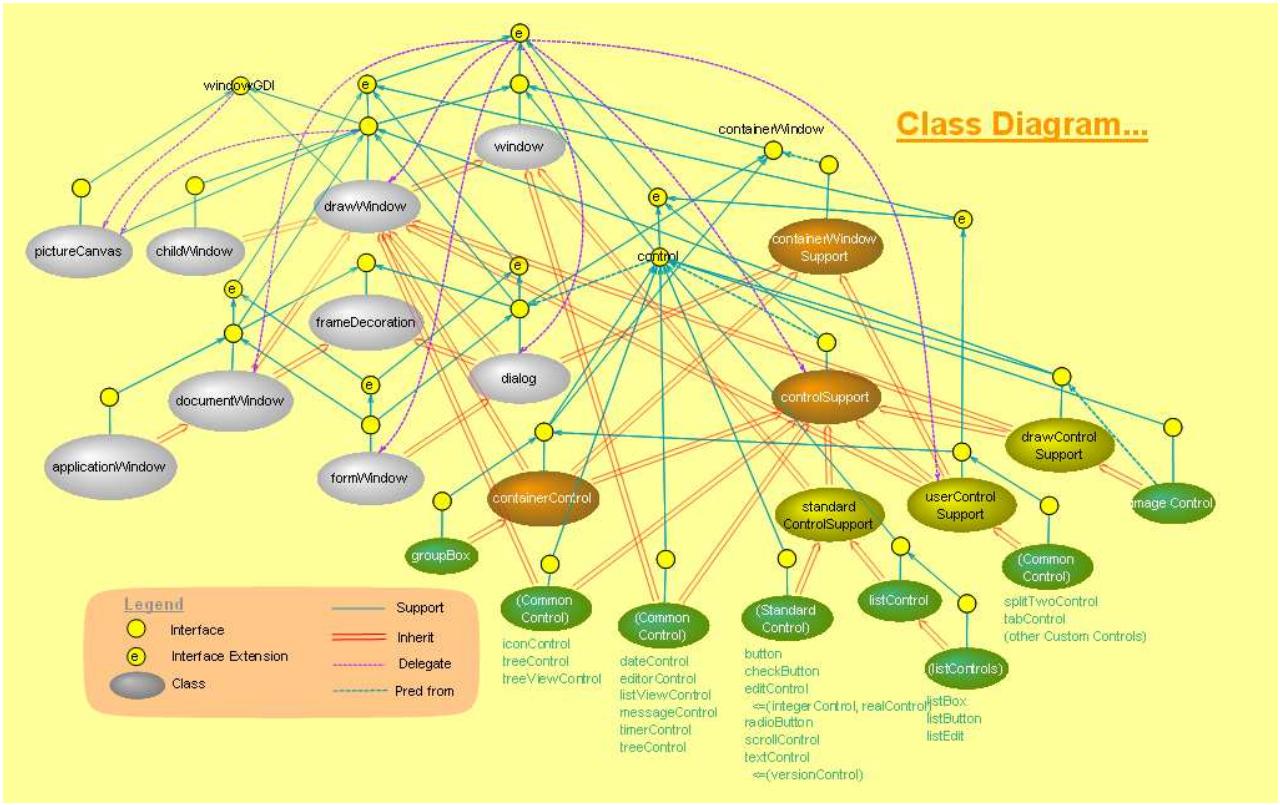
(Source: PDC publication dated June 1992)

```
domains
  process = symbol.
  path = symbol*.
  time = integer.

facts
  duration: (process,time).
  follows: (process,process).

predicates
  start: (process, time, path) procedure (i,o,o).
  stop: (process, time, path) procedure (i,o,o).
  later_start_needed: (process, time).

clauses
  start(P,0,[P]) :- not(follows(_,P)),!.
  start(P2,T,[P2,|PP1) :-
          follows(P1,P2),
          stop(P1,T,PP1),
          not(later_start_needed(P2,T)),!.

  stop(P,Tstop,Path):-
          start(P,Tstart,Path),
          duration(P,Dur),
          Tstop = Tstart + Dur.

later_start_needed(Pcheck,CheckTime):-
          follows(P,Pcheck),
          stop(P,FinishP,_),
          FinishP > CheckTime,!.

follows(make_wine,store_wine).
follows(store_wine,get_opener).
follows(get_opener,open_wine).
follows(open_wine,say_toast).
follows(say_toast,drink_wine).
follows(prepare_food,serve_food).
follows(send_invitation,wait_for_friends).
follows(serve_food,say_toast).
follows(wait_for_friends,say_toast).

duration(make_wine,100).
duration(store_wine,300).
duration(get_opener,2).
duration(open_wine,1).
duration(say_toast,1).
duration(drink_wine,2).
duration(prepare_food,10).
duration(serve_food,10).
duration(send_invitation,10).
duration(wait_for_friends,5).

goal
  start(drink_wine,Time,Path),
  (…)
```

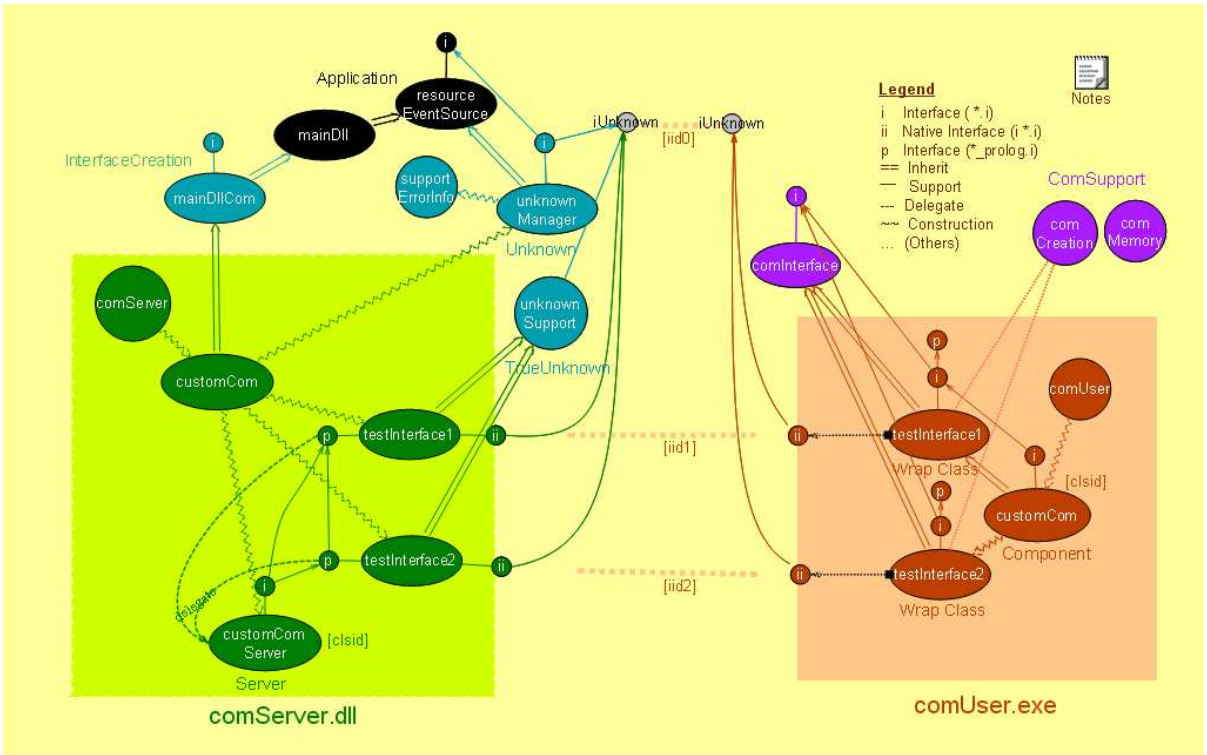
The corresponding Ferguson Diagram:



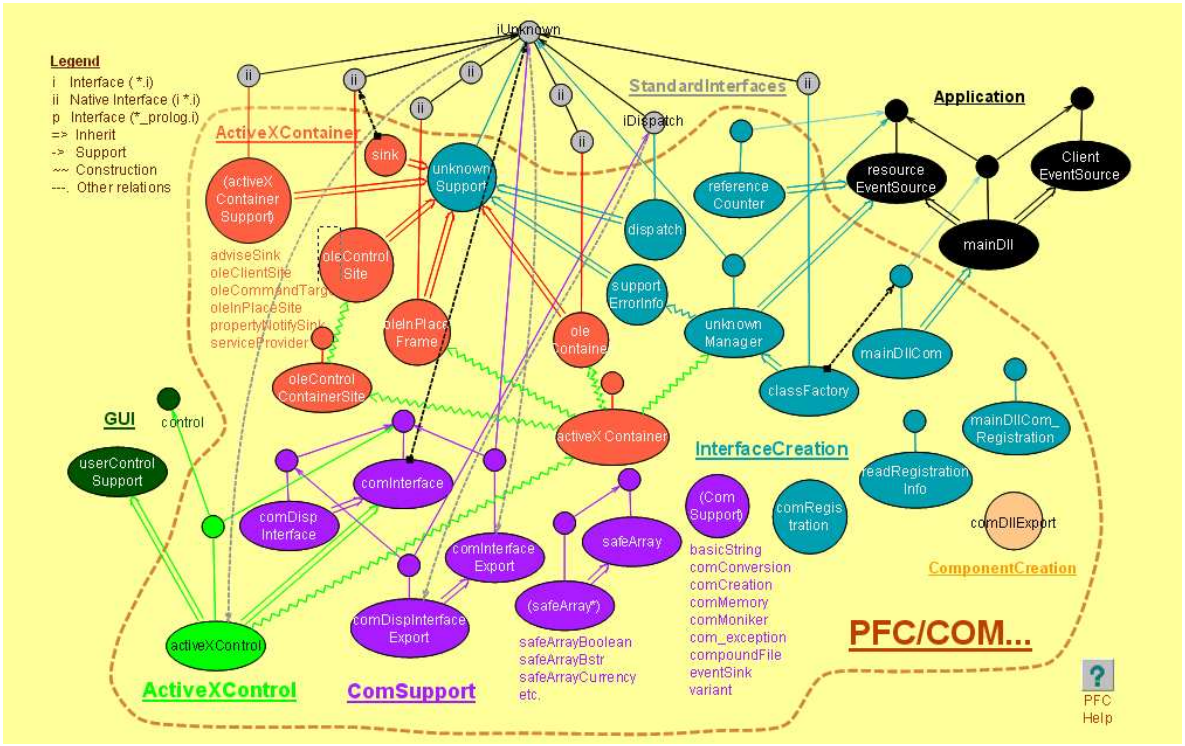
Appendix B – PFC/GUI Class Diagram



Appendix C – COM Server and User



Appendix D – PFC/COM Package



Appendix E – Webbrowser Component