EXPERIENCES WITH OBJECT ORIENTED PROGRAMMING

Karl-Heinz Alws, Ingrid Glasner-Schapeler
Standard Elektrik Lorenz AG, ZT/FZSD
Postfach 40 07 49
D-7000 Stuttgart 40

Abstract

The object oriented programming paradigm has been applied to the development of syntax-directed structure editors. The principal features of our implementation are presented, which is derived from a grammar for structured control flow documents. Experiences we gained show that object oriented programming is a valuable programming paradigm, in particular with respect to software reusability and maintainability.

1.0 Introduction

Due to the ever increasing demand for software systems and to their increasing complexity, the process of software development has received a lot of attention.
Novel approaches to programming promise new ways to cope with the problems involved. One of these approaches currently under research is object oriented programming. It is claimed that it provides advances towards the following goals:

- o to improve reusability of code
- o to improve maintainability of software systems
- o to support rapid prototyping

In order to evaluate the object oriented programming technique, we decided to apply it to a particular task, namely the development of syntax-directed structure editors. We first developed a graphical structure editor for a variant of Nassi-Shneiderman diagrams. Then we tried to reuse as much code as possible for a pseudo code editor.

This paper is to communicate the experiences we gained. As the vehicle for our implementation, we used the programming language OOPC (Object Oriented Precompiler to C) [Cox, 1983] .

In the following we first give an overview of the object oriented programming paradigm. Then we briefly describe the behavior required from the structure editors, and provide some detail about their object oriented implementation. Finally, we discuss problems encountered, and derive conclusions related to the claims mentioned above.

2.0 Object Oriented Programming

The definition of "object" used in this paper is adopted from Smalltalk 80 [BYTE, 1981] resp. from OOPC [Cox, 1983]. An object consists of both data and the procedures necessary to handle that data. Objects know how to manipulate themselves. This is quite different from the conventional programming approach with its distinction between passive data and active programs.

An object corresponds to some real-world entity. E.g. considering a CAD system for automatic layout computation of printed board assemblies, any particular IC would be represented by an object. In a traditional approach to software development an overall software system would have to know how to draw the shapes of the various device types after having placed them. In the object oriented approach all knowledge about drawing is contained in the objects themselves, and the overall system only has to know that each object can be drawn, not how the drawing is performed.

Communication among objects is established by means of messages. Access to and modification of an object is possible only by sending it a message. All processing takes place inside objects and is triggered by message sending. Messages are represented by expressions composed from an identification of the receiver, a selector string, and optionally some arguments. The receiving object responds to a message by performing the appropriate actions via a routine determined by the message selector. The set of message selectors to which an object can react provides a clean and simple interface for the object. Nothing else has to be known to other objects in order to communicate with it.

Every object is an instance of some class. New objects, i.e. new instances of a class, are created by sending a message to the class itself. The class

defines the methods, i.e. the routines available for manipulation of its instances, and thus determines the repertoire of messages that its instances understand. The class also defines names for "instance variables" which occur in all its instances and capture the status information specific to a certain object.

Classes are organized in an inheritance hierarchy. When a class is defined as subclass of an existing one, the new class automatically inherits the procedural and structural information associated with the superclass. Some of this may be not defined in the superclass itself, but inherited from a still more general class, i.e. information is actually inherited from a chain of superclasses. The inherited knowledge may be augmented and specialized by the subclass by definition of additional variables and/or routines. A method inherited from the superclass can also be overridden in its subclasses by locally defining a method with the same name.
An example for an inheritance tree is given in Figure 1.
The inheritance mechanism allows for class libraries from which applications can be built by defining specific subclasses.

Message sending as implemented in OOPC differs from a conventional subroutine call in that the binding of the message to the routine to be executed is dynamic, performed at runtime instead of by the compiler. Methods are identified by a method selector string. When an object receives a message, the message selector is matched against the set of method selectors associated with the class of the receiver; if necessary, search continues along the superclass chain, thus implementing inheritance of methods. Dynamic message binding has the advantage that the class of the message receiver does not have to be known in advance.

OOPC is based upon the Smalltalk object concept. It translates object oriented programs containing class definitions and message passing expressions into code of the C programming language. OOPC also supplies a library of classes implementing several basic kinds of data structure (e.g. linked lists). A very important feature provided by this library is a general object save/restore capability which is made possible by dynamic message binding. It allows any collection of objects to be written to a file and read in again in structured form, so that no application-specific parsing and unparsing routines are required.

3.0 User's View of the Structure Editors

The graphical structure editor (GSE) and pseudo code editor (PCE) support software engineers during the detailed design phase of software development. They allow for interactive construction of structured control flow documents.
The control flow is represented graphically by a variant of Nassi-Shneiderman diagrams [Nassi and Shneiderman, 1973]. A diagram is composed of graphical constructs (blocks), each of them representing an activity, which is described by pieces of text inside the block.
There are two categories of blocks: simple blocks and composite blocks. Simple blocks only contain text, whereas composite blocks contain other blocks in analogy to the structured statements of a programming language, e.g. conditional (if), selection (case), iteration (loop).
Diagrams are constructed according to two basic principles of structured programming: sequencing of blocks, and nesting of blocks.
A sample diagram is given in Figure 2.

The graphical structure editor (GSE) is a specialized syntax-directed editor possessing knowledge about the structure of Nassi-Shneiderman diagrams. A syntax-directed editor knows about the syntactic structure of the objects which it maintains and manipulates. It uses this knowledge to prevent modifications that would introduce syntactic errors.

The pseudo code editor (PCE) is syntax-directed like the GSE, and provides the same command interface. It is required to deal with documents written in a pseudo code notation which is logically equivalent to the graphical notation of Nassi-Shneiderman diagrams. I.e. the pseudo code constructs correspond to the building blocks of diagrams and are combined according to the abovementioned rules (sequencing, nesting).
Figure 3 shows the pseudo code document corresponding to the Nassi-Shneiderman diagram in Figure 2.

A common disk file format for documents capturing only the structural information of a design document is used by both editors. Since each of the editors can generate the appropriate display representation, a document can be created as diagram by the GSE, saved, and, in a subsequent session, manipulated as pseudo code by the PCE (and vice versa).

4.0 Implementation of the Structure Editors

The operation of both GSE and PCE is directed by the syntactical structure of the diagram or pseudo code, respectively, to be edited. This is achieved by internally representing documents in both editors as abstract syntax trees. These syntax trees are constructed according to the following context-free grammar which is common to GSE and PCE:

```
diagram       ::= {block}
block         ::= primitblock | emptyblock |
                  caseblock   | ifblock    |
                  loopblock
primitblock   ::= Text
loopblock     ::= Text {block}
ifblock       ::= Text alternative alternative
caseblock     ::= Text {alternative}
emptyblock    ::= "empty"
alternative   ::= Text {block}
```

The notation {x} denotes an arbitrary number of occurences of x (at least one). The symbol Text is a terminal symbol of the grammar and represents some character string.

Each node of a syntax tree, i.e. each structural unit of a design document, is implemented as a separate object. The objects modelling a design document are instances of classes roughly corresponding to the syntactic categories of the underlying grammar.

E.g., each if-block occuring in a diagram is an instance of class IF_BLOCK. An if-block object knows where it is placed in the syntax tree, and which diagram components (i.e., which other objects) are nested within the if-block. It also contains knowledge about its display representation.
A certain set of message selectors is understood by all if-block objects, e.g. "delete" and "draw".
The corresponding instance variables and methods are in part defined locally within class IF_BLOCK, in part inherited from its superclass chain.

The class hierarchy in the object oriented implementation of GSE was quite naturally derived from the syntax rules for Nassi-Shneiderman diagrams.

The methods and the instance variables realizing the manipulations of the internal representation, like deleting nodes from or inserting nodes into a tree, are the same for all different kinds of syntactical units that may occur as nodes in a syntax tree. Therefore, such methods are contained in the upper, more general classes of the inheritance hierarchy. These classes also define the instance variables realizing the links between tree nodes.
On the other hand, methods and instance variables related to the external (graphical) representation as displayed to the user of the GSE are specific to the various kinds of blocks. Such methods are responsible for computing the layout information (height, width, position on the screen) associated with a block, and for generating its shape (as an ensemble of graphical symbols). Being different for different kinds of syntactical units, these methods and instance variables are defined in the lower classes of the inheritance tree.
Figure 4 shows the resulting class organization. (Alternatives occur as components of if-blocks and case-blocks.)

The grammar underlying the internal representation of documents is the same for GSE and PCE. Therefore, the class hierarchy looks the same for both editors, i.e. there is no difference concerning the class names and the subclass relationships between classes.
Furthermore, the editors have the same command interface and thus perform the same operations on the internal document representation (syntax tree). Because the upper classes are concerned with this internal representation only, they are identical for GSE and PCE. The corresponding code, developed during implementation of the GSE, could thus be reused without any modifications in the PCE implementation.
The classes in the lower part of the class hierarchy, however, had to be developed anew, since they deal with the external representation of design documents which, for PCE, is in the form of pseudo code, as opposed to the diagrammatic representation used by the GSE.
Figure 4 may be misleading in that it suggests that only a small portion of the GSE code could be reused for the PCE. However, the small number of upper level classes contain the bulk of code. The methods to be added by the lower level classes are rather simple. The OOPC source code for each of the editors consists of about 5700 lines of code (not including comment lines). About 4300 lines of code are common to both GSE and PCE. Thus, only about 1400 lines of specialized code had to be written in order to implement the PCE, which means that 3/4 of the GSE could be reused.
When developing the new specialized classes of the PCE, we had to take care that they present the same message interface as the corresponding GSE classes, in order to guarantee compatibility with the reused general classes. Instances

of the new classes must be able to react to all messages that may be sent to them from within some method defined in an upper level class, e.g. to a "draw" message issued by the "delete" method defined in class NODE.

Apart from the distinction between upper level and lower level classes dictated by the distinction between display representation and internal representation, introduction of additional subclass relationships was motivated by our desire to exploit the inheritance mechanism wherever possible. E.g., class EMPTY_BLOCK is defined as subclass of class PRIMIT_BLOCK. Thus, the fact that empty blocks are displayed in the same fashion as primitive blocks is simply reflected by not defining in class EMPTYBLOCK specific methods dealing with the display representation, but making use of appropriate methods inherited from class PRIMIT_BLOCK.

A commonly used programming technique is to define a quite general "default method" on a high level in the hierarchy, and to override this method in a descendant class by defining a specialized method with the same selector. A general method "textinsert", for example, is defined in class FIXNODE, which performs all necessary actions (linkage and layout update and redisplay) when a new text string is inserted into a block. The text insertion operation is not admissible for empty blocks; their text must always be the string "empty". The implementation of this restriction is by defining a special method "textinsert" for class EMPTY_BLOCK, which does nothing but issue an error message.

The strong correspondence between class hierarchy and grammar for structured control flow documents is crucial for the simple implementation of syntax checking, the most important feature of the syntax-directed structure editors.
A possible document modification, e.g. replacing a construct by a different one, is syntactically admissible only if the new construct is an instance of the type of construct required at this position. E.g., a case-block in the body of a loop may be replaced by some other kind of block, but not by an alternative (since alternatives occur as components of if-blocks and case-blocks only).
Using object oriented programming, the implementation of the necessary syntax checks was straightforward. In OOPC, each object can reply to a message asking whether it is an instance of (some subclass of) a certain class. E.g., a primitive block knows that it is an instance of a subclass of BLOCK and therefore can be placed at any position in a document where blocks are admissable.
By choosing a hierarchy of classes that reflects the syntactical relationships as defined by the grammar, the inheritance mechanism can thus be exploited in

order to guarantee, in an easy way, that design documents constructed with GSE or PCE are always syntactically correct.

5.0 Problems and Suggestions

In implementing the structure editors as collections of OOPC classes, we have encountered several problems. Some of these problems just indicate a lack of certain language features in OOPC, or of tools to support the object oriented design methodology, some of them, however, are of a more fundamental nature.

In OOPC, each class has exactly one superclass, i.e. the graph of the subclass relation for a collection of classes is always a tree. Sometimes, though, the ability to define more than one superclass for a new class, the possibility to arrange classes in a lattice, is desirable.

For example, one often needs linked lists composed of objects of a certain kind. The elements of such lists must exhibit both the behavior of list elements in general as well as the behavior of the special kind of entities from which the list is composed. It is not possible, though, to define in OOPC a class that can inherit properties from both a generic "list element" class and from another class unrelated to lists. A somewhat clumsy solution is to implement each (conceptual) list element by two objects connected via a pointer. The first object is an instance of a generic class which provides variables and methods characteristic of linked lists, the second object is an instance of the class that deals with the specific properties of the objects in the list. The disadvantages of this solution are additional storage and runtime overhead. Using OOPC, they could be avoided only by redefining data and procedures common to all kinds of singly linked lists within each of the corresponding specialized classes, which, however, means giving up the advantages of inheritance.
The availability of multiple superclasses is a language feature missing in OOPC, but provided by many object oriented programming languages, including later versions of Smalltalk.

For applications realized by some larger collections of classes, the subclass relationships between classes and the induced inheritance may become hard to keep track of. Specialized cross reference tools would be desirable to support both the design and implementation and the debugging of object oriented application software.

An "inheritance checker" should display the superclass chain for a given class. It should also list the instance variables and methods available to this class via inheritance, and indicate for each of these the superclass from which it is inherited. Such an inheritance checking tool would be particularly valuable if multiple superclasses are allowed, i.e. if there may exist several superclass chains for each class.

Conversely, given a method selector, as occuring in a message, one might want to get an overview of all classes that define a method with that name. Note that each path in the inheritance tree (or lattice, respectively) may contain more than one method definition corresponding to a given selector. An example for such a situation, occuring in the structure editor implementation, is shown in the following figure:

```
    NODE        defines   "evalWidth"
     |
    BLOCK       redefines "evalWidth"
     |
  CASE_BLOCK
```

Undesirable effects may result when, in designing class CASE_BLOCK, the software developer relies on inheritance of method "evalWidth" from class NODE, forgetting that this method was redefined in class BLOCK in a fashion not suitable for class CASE_BLOCK.

In addition to tools checking static relationships between the entities defined for an object oriented application, object-level debugging requires special runtime support in order for the programmer to see the effects of dynamic message binding. Dynamic message binding means that the method executed in response to a message is determined not at compile time, but during program execution. Therefore, objects can send messages to other objects without having to know which class the receiver is an instance of. Instance variables that are pointers to other objects need not be "typed" with a specific class, but are uniformly defined as "OBJ" in OOPC.

This, however, makes it hard for application programmers to check the soundness of their implementation. For each class, they have to make sure that its instances know how to react in each context where they may occur. This means that for each message that may be sent to an instance of some class, a corresponding method has to be available (either directly, or via inheritance). But because of dynamic binding, it is impossible to construct a cross-reference tool showing all message selectors that instances of a certain class must be able to respond to. Related design errors cannot be detected by the compiler,

but will, in general, show up during test runs of the program, i.e. rather late in the program life cycle.
Such errors can in part be prevented by implementation of default methods for all selectors occuring in some message. The definition of a default method has to be contained in a class that is a common ancestor of all classes of which instances are expected to receive a message with the corresponding selector. For example, in the structure editor implementation, a default method "draw" has been defined in class GRBOX, since we expect "draw"-messages to be sent to any kind of structural unit. The default method technique cannot, however, be applied for applications in which the only common ancestor of all relevant classes is a class taken from some given library which the application programmer cannot augment at will by additional methods.

When implementing the PCE, we tried to reuse as much code as possible from the GSE implementation of which a first version had already been completed. We expected this to be achieved in a straightforward manner, as described in the previous section, because of the conceptual subdivision of the class hierarchy into upper level classes, handling the internal representation of edited documents, and lower level classes, handling the display representation. However, in practice, this division was not so clear-cut.

It turned out that also in the general, upper level classes, code had to be included that is needed by only one of the editors. This was in many cases due to the fact that attached to each object representing a syntactical unit there is a second object which contains the data relevant to the external layout of the unit (height, width, etc.). Since the external representation of syntactical units is different for GSE and PCE, these "layout objects" are instances of different classes: of class GSEUNIT for GSE, and of class PCEUNIT for PCE.
This construction leads to the following problems:

1. Whenever a syntactical unit (e.g. a caseblock) is created during an editor session, a corresponding layout object has to be generated as well. In order to maximally exploit the inheritance mechanism to avoid replication of code, the method for layout object generation which is common to all kinds of syntactical units is incorporated in class GRBOX. Therefore this class, though placed very high in the inheritance hierarchy, contains code related to the external representation of documents.
   On the other hand, we wanted to use identical definitions of class GRBOX for GSE and PCE, since most of the methods of GRBOX are the same

for both editors. Therefore, the method for layout object generation defined in GRBOX had to be given the ability to determine which is the current editor, in order to decide whether a GSEUNIT or PCEUNIT has to be created.

The general problem demonstrated by this example is a conflict between the objectives of

- o code reuse within a specific application (e.g. GSE), by virtue of inheritance, which implies defining methods as high in the class hierarchy as possible
- o code reuse across applications, in the form of reusing complete class definitions, which implies keeping higher level classes as general as possible

Both objectives can be given equal consideration by introducing additional intermediate layers into the class hierarchy. This, however, increases the runtime of the resulting object oriented programs, since the search path for the method to be executed in response to a message becomes fairly long.

2. As mentioned above, a diagram can be created by the GSE and, in some later editing session, manipulated by the PCE. In our implementation we used the general save/restore mechanism provided by the OOPC system. This mechanism always stores on file the complete set of objects referenced by any object to be saved. In particular, saving a diagram created by GSE implies that all associated layout objects, i.e. instances of class GSEUNIT, are written to the file. When this file is read for subsequent use by PCE, the PCE must be able to handle these GSEUNITs, though it will immediately substitute them by instances of PCEUNIT in the restored document. Thus, the PCE must know the structure of GSEUNITs. The only way to circumvent this problem would have been to implement our own specialized save/restore mechanism instead of using the mechanism already implemented.

6.0 Conclusions

Most of the problems we had with our implementation of the structure editors were due to the fact that none of the programmers involved in this project had any previous experience with the object oriented programming paradigm. For

example, we had not sufficiently taken care of keeping the classes in the upper layers of the inheritance hierarchy as general as possible, which forced us to modify some of the GSE code in order to reuse it for PCE. But now, having gained first experiences with object oriented programming, we expect to better exploit its advantages in subsequent projects. In particular, we appreciate the advances in reusability and maintainability achieved by object oriented programming.

Object oriented programs are necessarily modular and clearly structured, as collections of classes arranged in an inheritance hierarchy. All knowledge about a certain kind of object is isolated in the corresponding class definition, with only its message interface visible from outside. Therefore, extending or modifying an existing implementation is fairly easy. Side effects introduced by changes are limited.

The inheritance mechanism provides a natural means to avoid code redundancy, and thus helps to keep consistency problems arising in the process of implementing changes to a minimum.
By factoring out common properties of several kinds of objects and associating these with fairly general "base classes", class libraries can be created which may be used by several different applications.
Specialized cross-reference tools for inheritance checking would be useful, especially during the software design phase and for debugging.

The object oriented approach seems to be particularly suitable for prototype development.
The logical subdivision of the application world into a collection of entities, which is induced by the problem to be solved, directly carries over into the object-oriented implementation: each entity is modeled by an object (or by several related objects). Therefore, once problem analysis has been performed, a first approximation of the projected software system already exhibiting most of the desired behavior can be implemented in a fairly straightforward fashion. Further improved versions will be derived from such prototype software. Since object classes are characterized by their functional interface, not by their internals, it is easy to replace these program components by functionally equivalent ones whose implementation puts more emphasis on runtime efficiency. Also, detail of program behavior can easily be added by introducing new classes as subclasses of existing ones.

Object oriented programming offers major advantages for finding design errors resulting from inadequate problem decomposition. Since there is such a close

correspondence between the problem structure as perceived by the software designer and the structure of the implementation, the design can be verified by testing the resulting code.
The OOPC system supplies rudimentary object level debugging facilities, in particular the option to have a message log be produced at runtime which records each message sent. If full object level debugging is provided, the software engineer can deal with his problem on a very high conceptual level only throughout the whole process of software development.

On the whole, we have made the experience that object oriented programming brought great leverage to our specific task of implementing syntax-directed structure editors, and we think that it is a valuable general programming paradigm that we would like to become widely used.

7.0 Figures

```
                    CLASS 0
                    VAR A
                    METHOD AA
                       |
                       |
         +-------------+-------------+
         |                           |
         |                           |
      CLASS 11                    CLASS 12
      VAR B                       VAR C
      METHOD BB                   METHOD CC
         |                           |
         |                           |
 +-------+--------+          +-------+-------+
 |                |          |               |
 |                |          |               |
CLASS 21       CLASS 22   CLASS 23        CLASS 24
VAR D          VAR E      VAR F           VAR G
METHOD DD      METHOD EE  METHOD FF       METHOD GG
               METHOD AA
```

Along with each class name, this figure lists the locally defined data structures (variables) and methods (routines).

CLASS 21 inherits all data structures and methods (routines) defined by its superclass chain CLASS 11 - CLASS 0 and augments this information with variable D and method DD.

CLASS 22 also inherits all data definitions and methods of its superclasses. Method AA, however, which is defined in CLASS 0, has been overridden by a new method with the same name in CLASS 22.

Figure 1: Example for Inheritance Hierarchy of Classes

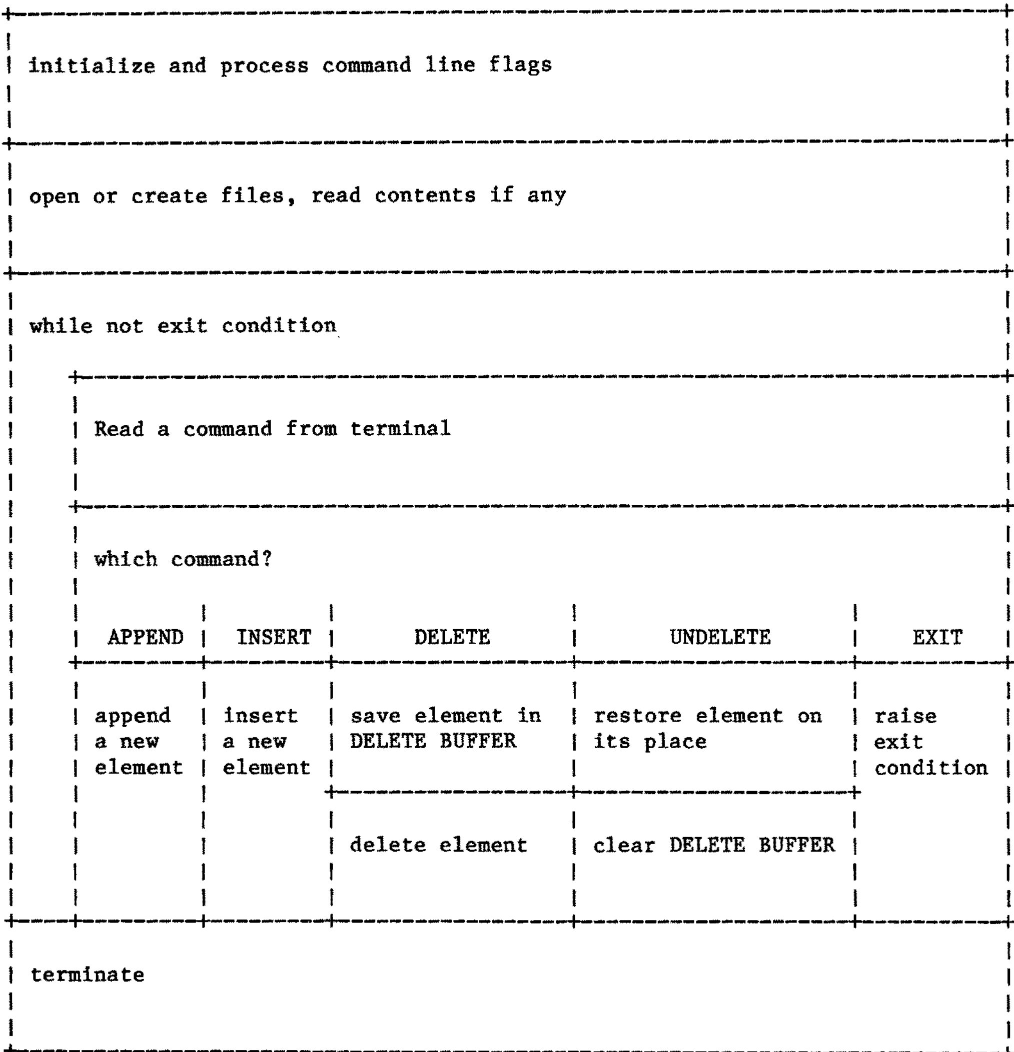

Figure 2: Example for Diagram Generated with GSE

```
b e g i n
   initialize and process command line flags
e n d
b e g i n
   open or create files, read contents if any
e n d
l o o p
w h i l e  while not exit condition
   b e g i n
      Read a command from terminal
   e n d
   c a s e  which command? i s
      w h e n  APPEND = >
         b e g i n
            append a new element
         e n d
      w h e n  INSERT = >
         b e g i n
            insert a new element
         e n d
      w h e n  DELETE = >
         b e g i n
            save element in DELETE BUFFER
         e n d
         b e g i n
            delete element
         e n d
      w h e n  UNDELETE = >
         b e g i n
            restore element on its place
         e n d
         b e g i n
            clear DELETE BUFFER
         e n d
      w h e n  EXIT = >
         b e g i n
            raise exit condition
         e n d
   e n d c a s e
e n d l o o p
b e g i n
   terminate
e n d
```

Figure 3: Example for Pseudo Code Generated with PCE

```
                        *   OBJECT (OOPC Primitive)
                        *      |
                        *      |
     +-----------+-------------+--------+
     |           |      *      |        |
     |           |      *      |        |
  PCEUNIT     GSEUNIT   *    NODE    STRING (OOPC Primitive)
                        *      |        |
                        *      |        |
                        *      |        |     ****************
                        *    GRBOX    TEXT    *Internal      *
          ***************      |              *Representation*
          *                    |              ****************
          *        +-----------+------------+
          *        |                        |
          *        |                        |
          *     FIXNODE                  VARNODE
          *        |                        |
          *********|************************|*****************
                   |                        |
   +----------+----+-------+         +------+-------+
   |          |            |         |              |
   |          |            |         |              |
DIAGRAM     ALTER-      BLOCK    SEQU_BLOCK   SEQU_ALTERNATIVE
            NATIVE         |
              |            |
              |            |                  ****************
            IFPART         |                  *External      *
              |            |                  *Representation*
              |            |                  ****************
     +--------+--+         |
     |           |         |
     |           |         |
 THENPART    ELSEPART      |
                           |
                           |
                 +---------+--------+-------+
                 |         |        |       |
                 |         |        |       |
                 |   LOOP_BLOCK     |   CASE_BLOCK
                 |                  |
                 |                  |
            PRIMIT_BLOCK        IF_BLOCK
                 |
                 |
            EMPTY_BLOCK
```

Figure 4: Class Hierarchy Implementing GSE and PCE

8.0 References

BYTE Magazine, Special Issue on SMALLTALK 80, Vol.6, No.8, August 1981.

Cox,B. The Object Oriented Precompiler. Programming SMALLTALK 80 Methods in C Language. ACM SIGPLAN Notices, Vol.18, No.1, 1983

Nassi,I. and Shneiderman,B. Flowchart Techniques for Structured Programming. ACM SIGPLAN Notices, Vol.8, No.8, 1973