University of Waterloo
Department of Computer Science
Waterloo, Ontario, Canada

Technical Report Series

CS-93-26

ENHANCING SOFTWARE DESIGN REUSE: NESTING IN OBJECT-ORIENTED DESIGN

by

D.D. Cowan C.J.P. Lucena

May, 1993

ENHANCING SOFTWARE DESIGN REUSE: NESTING IN OBJECT-ORIENTED DESIGN

D.D. Cowan
Computer Science Department & Computer Systems Group
University of Waterloo
Waterloo, Ontario
Canada
N2L 3G1
dcowan@csg.uwaterloo.ca

C.J.P. Lucena
Departamento de Informática
Pontifícia Universidade Católica
22453-900, Rio de Janeiro-RJ,
Brazil
lucena@inf.puc-rio.br

May 7, 1993

Abstract

It has been observed that design of complex objects such as software requires both decomposition by form (atomic objects) and decomposition by function (nesting) in order to reduce the design to a set of manageable components. However, the object-oriented design paradigm mostly supports decomposition by form. This paper uses a simple example to motivate the need for nesting (decomposition by function) and illustrates how nesting might be incorporated into a design language. We conclude that the introduction of nesting into software specification and design significantly increases reusability.

Key Words: programming languages, program specification, software design and implementation, software engineering

1 Introduction

Authors such as Maher [Mah90] have observed that designers in various engineering disciplines use both decomposition by function and decomposition by form to reduce their projects to manageable components. Similarly, software designers should use both design strategies since they also build complex objects. Decomposition by form follows the object-oriented paradigm and object-oriented programming languages [GR83, BS83, Str86, CN91] and design methodologies [Boo91, R^+91] support decomposition by form through such techniques as creating subclasses (inheritance) and encapsulation. Decomposition by function requires that an object be divided into smaller components to which a small set of actions can be applied. The relationship among the larger component and its constituents is expressed through nesting, a concept that some authors claim is not properly supported by object-oriented languages [BZ88] and is not supported at all by strictly object-oriented design methodologies [Jal89].

Although there have been arguments made in favour of nesting in object-oriented specification and design, we came to the conclusion that most of the arguments used so far are not very satisfactory. Some of the arguments sound like a nostalgic defense of structured design/programming [Ala88, Jal89], and some authors even show how to convert a structured design into an object-oriented design [Ala88]. Other authors [BZ88, Ass92, Mad87] have examined a related issue, namely, the implementation of nesting in object-oriented programming languages. We believe a common concern at both the design and programming language levels is nesting encapsulation. That is, the semantics of nesting should allow reference to definitions from outside the containing block without violating encapsulation [BZ88, Ass92, Mad87].

We feel there is a need for an appropriate illustration of the "form versus function dilemma" that every designer needs to face. In other words, a discussion about when to use decomposition by form (inheritance) and when to use decomposition by function (nesting) should be presented in the context of a software design activity. Since problem solving at the design and implementation levels can always take place using only one of the two kinds of decomposition, a "metric" is necessary to justify decisions that combine both approaches to design. The metric we propose in this paper is enhanced design reuse.

Our motivation for the combination of inheritance with nesting at the design level comes from our work on Abstract Data Views (ADVs) [CILS93a, CILS93b]. At first the concept of ADVs was used only for the design of user interfaces. Later this concept was generalized to deal with module interconnection in general and the design of concurrent and distributed systems [PLC93]. The justification for the combined use of nesting and inheritance can be naturally explained in the case of user interfaces. Nesting models the issue of "locus of association" in human interfaces. Nested objects know "where they are" with respect to other objects on the screen, therefore minimizing the so-called constraint problem [Lel88, Car92]. Inheritance is normally used to specialize interface objects.

A justification for the combined use of the two kinds of decomposition is less obvious in other application domains. We discuss this issue in this paper using a simple software design situation. What we have done was to "simulate" the locus of association situation in our example to try to convince the reader that at least in this situation (which occurs very often in software designs), a combined use of the two decomposition styles is justified because design reuse is clearly improved.

It should be noted that we discuss specification and design issues in this paper, not implementation issues. One contribution of this paper is to illustrate the importance of nesting to those researchers who are extending formal design notations to encompass object-oriented design concepts [S^+90, CDD^+90, CHB92, Fit91]. We also use the design example to introduce the notions of maximization of reuse as a design metric, and the properties of locus of association, object-set browsing and nesting encapsulation. These are all properties which are introduced when nesting is used as a design notion. In our work on ADVs we expressed nesting using the extensions of VDM proposed by Ierusalimschy [Ier91, Rob93], although we use an informal notation in this paper.

2 The Problem

Consider an electronic version of a library. An electronic library is a collection of documents in machine-readable format ordered using some scheme such as the Dewey Decimal System. We wish to specify and design a program which allows a user of the library to browse all the documents in the library sequentially.

Browsing the library means that the user starts at the first document in the library and examines the cover. If the document is of interest, the user then scans the document in more detail by moving among the sections of the document in some predetermined order from front to back. The sections of the document and the order of those sections are determined by the type of the document.

3 The Structure of the Library

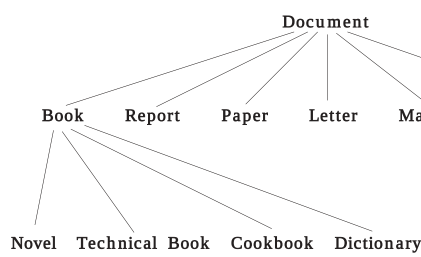

Figure 1: A Hierarchy of Document Types for a Library – An is_a Relation

The library consists of a number of documents and these documents are of many different types such as book, report, paper, letter, magazine, and newspaper. Many of these document types can be further subdivided into different classifications. For example, a book can be a novel, technical book, cookbook, or dictionary. This relationship among document types can be represented as a hierarchy and is shown in Figure 1. As we move from top to bottom in the hierarchy each document type becomes more specialized and inherits the properties of its superior entry in the hierarchy. Inheritance is often called an *is_a* relation.

Each document type in a library may have a different composition. For example, a novel has a title, author, preface and a number of chapters, while a technical book is composed of a title, author, table of contents, chapters, appendices and an index[1]. The structure or composition of a specific type of document namely, a novel and a technical book is illustrated in Figure 2 where boxes inside each other indicate composition by nesting[2] , and the left to right order of boxes indicates

[1] This description is a simplification of the structure of various kinds of books, but it is certainly adequate for the present example.

[2] Both inheritance and composition by nesting could be illustrated using a tree diagram. We have chosen two different representations to emphasize that these are different concepts.

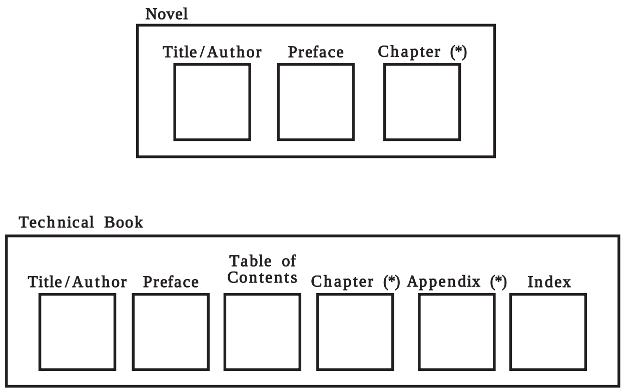

Figure 2: The Structure of two Document Types – An is_a_component_of Relation

order of appearance in the document. The asterisk (*) beside the name of a component indicates that the component may appear several times in sequence. The term *is_a_component_of* is often used to describe the nesting relationship.

4 An Object-oriented Design for the Library

In this section we consider an object-oriented design for the library in order to motivate the need for encapsulation, inheritance, and composition by nesting. Object-oriented design requires that we identify the basic objects which can act together as atomic units to produce the desired behavior. If we confine the contents of the library to novels and technical books, it is clear from Figure 2 that we need objects such as title, author, preface, chapter, table of contents, appendix and index. The library then becomes an ordered collection of documents and each document in the library such as a novel or technical book becomes an ordered composition of these basic objects.

In order to browse the library we need to define two methods or functions for each document, namely "get_next" and "examine". The method "get_next" will move to the next document and the method "examine" will allow a detailed examination of each section of a specific document. The method "get_next" can be defined for all documents as it is only necessary for the system to know how to move to the next element in the ordered collection of documents. The method "examine" is more specialized because an examination of a document requires knowledge of the specific type of document and is an example of the requirement for the "locus of association".

4.1 Encapsulation and Inheritance

Conventional structured design would specify the "examine" method for books using the pseudo-code structure shown in Figure 3. In this Figure the document type is located in a standard place in each document and is then interrogated in a case statement. Based on the value in the case statement the "examine" method can call the correct function for a specific type of document.

```
record document (typecode : integer;....)
.....
method examine(item)
    type item : document
    case item.typecode
      novel: examine_novel(item)
      technicalbook: examine_technical_book(item)
      cookbook: examine_cookbook(item)
      dictionary: examine_dictionary(item)
      report: examine_report(item)
      paper: examine_paper(item)
      .....
    esac
```

Figure 3: A Conventional Pseudo-code Specification for "examine"

In the object model of design the state of an object is encapsulated or hidden and is queried and changed through a set of associated methods or functions. Since the methods are really part of the object they can be used by naming an object and its associated method. For example, accessing the method "examine" for the object "item" can be written as

```
item.examine;
```

and replaces the pseudo-code of Figure 3.

We now must add the method "get_next" to each object so that the entire library can be browsed. Unfortunately we now must duplicate the "get_next" specification for every type of object in the library. The concept of inheritance solves this problem. Inheritance allows the definition of a type which may be specialized and thus implements the hierarchy shown in Figure 1. Since "get_next" is the same for all documents, we can now attach the specification and the corresponding state to the document type. When the newer types inherit from document they also inherit the state and all accompanying methods such as "get_next". This means the specification and state for this method are only located in one place in the program design, although it is accessible to all subtypes that inherit from the type document. A type that allows inheritance is usually called a class.

4.2 Composition by Nesting

Invoking the method "examine" for each document type requires that each component of the document be displayed in succession under user control. A simplified version of the class book containing only the components preface and chapter and their associated "display" method, might be expressed as shown in Figure 4 if we use only the concepts of encapsulation and inheritance. Inheritance is made explicit with the expression

```
novel is_a book.
```

```
class book is_a document
    where = (preface, chapter)

    function examine_preface(item)
       item.display
       where <- chapter

    function examine_chapter(item)
       item.display
       where <- preface

class novel is_a book
    where <- preface
    method examine(item)
       case where
         preface : examine_preface(item)
         chapter : examine_chapter(item)
       esac
```

Figure 4: An Object-oriented Approach to the function "examine" for the objects book and novel

This solution illustrates a strict object-oriented style of design where the designer interpreted both the relations is_a and is_a_component_of in Figures 1 and 2 as inheritance trees.

Instances of the classes book and novel maintain a variable "where" which records the next item to be examined in the document. Note the use of the case statement with the variable "where" to select the correct version of "examine". **This solution has the same problem as the one which motivated encapsulation.** Also this solution has to be created for each class because the solution must be specialized to that specific class. Such specialization limits reuse.

Note that this specification could be implemented using an array of object pointers. However, the expression of nesting would not be explicit, but would be implied by the semantics of the program.

```
class novel is_a book
    novel is_composed_of (title/author, preface, chapter)

    method examine
       next.display
       next <- succ(next)
```

Figure 5: An Object-oriented Approach to the class novel using composition by nesting

We create the concept of composition by nesting to build a class. Each class is composed of its constituent classes and their associated methods. We illustrate composition by nesting in Figure 5 by using a version of the class novel. The statement

```
novel is_composed_of (title/author, preface, chapter)
```

indicates that the class novel is composed of the classes title/author, preface and chapter, and that they appear in the order presented. In our case each of these constituent classes has a method called "display" which is invoked by naming the object of that class, and then the method. For example, "display" for the object "item" of class "chapter" would be invoked with the expression

```
item.display
```

Associated with this list of constituents in each object is a variable named "next" that is used to traverse this list. The first time the variable "next" is used its value is the first object in the list of constituents. There is also a successor method named "succ" that moves the value of the variable to the next element in the list of constituents. The method "succ" will move to the beginning of the list of constituents after accessing the last element. Thus, we have provided the design specification with an *object-set browsing* capability.

When a class such as novel is instantiated, its list of constituents is defined, but the list does not contain any instances of constituent classes. That is, the type and order of the constituents is known when the class is defined. As an object of a class such as novel "grows" and "shrinks" new instances of constituent classes are added and removed from the list. Hence, methods such as "insert" and "remove" must be defined for constituent lists and could be based on the position of the variable "next". We should also note that type and number violations are not allowed. For example, the constituent list for novel may not have an instance of an index, and if the list already contains an instance of a preface then trying to enter another preface would cause an error. We say we have achieved *locus of association* through nesting.

We observe that nesting has maintained the separation of concerns, since we first solved the problem of manipulating each individual component and then we solve the problem of composition; the two solutions proceed independently. Although the enclosing object of a class such as novel knows the identity of its constituent classes, the enclosed objects of classes such as preface and chapter have no knowledge of the state of novel. We call this property of the design *nesting encapsulation*.

Also using this design language involving composition by nesting to invoke the methods "examine" does not require any knowledge of the position in the constituent list from either of these methods. In fact we could easily change the constituent list without changing any of the specification associated with the object novel. This form of limited change makes any of these objects highly reusable.

Because the knowledge of position in the constituent list is encompassed by the variable "next" we can use inheritance to associate the method "examine" with the class document. This concept is illustrated in Figure 6. The constituent lists for document and book are empty, but this does not affect the program design. These lists become completed when the class novel is declared.

Of course it is possible to have some of the constituents in a list to be composed of lists. This can be easily handled within the constituent itself. For example, consider a class tech_chapter which consists of sections. This could be expressed as shown in Figure 7 and except for a change of name is exactly the same specification as used in Figure 6.

```
class document
    document is_composed_of ()

    method examine
       next.display
       next <- succ(next)

class book is_a document
    book is_composed_of ()

class novel is_a book
    novel is_composed_of (title/author, preface, chapter)
```

Figure 6: Associating the method "examine" with the class document

```
class chapter
    chapter is_composed_of ()

    method examine
       next.display
       next <- succ(next)

class tech_chapter is_a chapter
    tech_chapter is_composed_of (section(*))
```

Figure 7: Nested Composition

5 Conclusions

This paper has used a simple example to illustrate that design using decomposition by form and function require inheritance, encapsulation, and composition by nesting. Hence, composition by nesting has a significant role in object-oriented design. Reuse of design was used as a "metric" to demonstrate the need for composition by nesting, since reuse is more viable at the design level. Our example also clarifies the informal semantics of composition by nesting for the designers of both design and programming languages by introducing the notions of locus of association, object-set browsing and nesting encapsulation. The properties of nesting that have been illustrated, also give some indication of how to extend formal methods to incorporate this important design concept.

Most of the notions of the design approach illustrated in these examples, can be implemented more or less directly in existing object-oriented languages, although they do not use the syntactic method we have described here to produce this implementation. It would be ideal if the language used to implement our design notion of composition by nesting could exhibit some form of polymorphic behavior, since the constituent list can contain objects of any class. Also management of objects would be made easier, because the constituent list contains the names of all classes that compose a class. Because the names are easily found it should be possible to build a tool that can locate all the classes which make up a document class since they are connected in a nesting tree.

6 Acknowledgement

The authors wish to thank P.J. Bumbulis, L.M.F. Carneiro and M.H. Coffin for their many comments on the contents of an earlier version of this paper.

References

[Ala88] B. Alabiso. Transformation of data flow analysis models to object. In *Proceedings of OOPSLA, 1988*, 1988.

[Ass92] Swedish Standards Association. *Simula - Data Processing Programming Languages.* Swedish Standard SS636114SIS, 1992.

[Boo91] Grady Booch. *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[BS83] D. G. Bobrow and M. Stefik. *The LOOPS Manual.* Xerox Corporation, 1983.

[BZ88] P. A. Buhr and C. R. Zarnke. Nesting in an object oriented language is not for the birds. In *Proceedings of ECOOP'88, European Conference on Object-Oriented Programming*, 1988.

[Car92] Luiza M. F. Carneiro. *A Specification-based Approach to User-Interface Design.* PhD thesis, University of Waterloo, December 1992.

[CDD+90] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. An Object-Oriented Extension to Z. In *Formal Description Techniques (FORTE 89).* North Holland, 1990.

[CHB92] D. Coleman, F. Hayes, and S. Bear. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering*, 18(1), 1992 1992.

[CILS93a] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Abstract Data Views. *Structured Programming*, 14(1):1–13, January 1993.

[CILS93b] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Application Integration: Constructing Composite Applications from Interactive Components. *Software Practice and Experience*, 23(3):255–276, March 1993.

[CN91] B. J. Cox and A.J. Novobilski. *Object Oriented Programming.* Addison Wesley, 1991.

[Fit91] J. S. Fitzgerald. Modularity in Mode-Oriented Formal Specifications and its Interaction with Formal Reasoning. Technical report, Department of Computer Science, University of Manchester, Technical Report Series, UMCS-91-11-2, 1991.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, Palo Alto, CA, January 1983.

[Ier91] Roberto Ierusalimschy. A Method for Object-Oriented Specifications with VDM. Technical report, Monografias em Ciência da Computação, PUC-Rio, February 1991.

[Jal89] P. Jalote. Functional refinement and nested objects for object-oriented design. *IEEE Trans. on Software Engineering*, 15, 1989.

[Lel88] W. Leler. *Constraint Programming Languages.* Addison Wesley, 1988.

[Mad87] O. L. Madsen. Block structure and object oriented languages. In B.; Shiver and P. Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[Mah90] M. L. Maher. Process Models for Design Synthesis. *AI Magazine*, Winter 1990.

[PLC93] A. B. Potengy, C. J. P. Lucena, and D. D. Cowan. A Programming Approach for Parallel rendering Applications. Technical report, Monografias em Ciência da Computação, PUC-Rio, April 1993.

[R+91] James Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[Rob93] Ierusalimschy Roberto. A Formal Specification for a Hierarchy of Collections. *to appear IEE Software Engineering*, 1993.

[S+90] S. A. Schuman et al. Object-oriented process specification. In *Specification and Verification in Concurrent Systems.* Springer-Verlag, 1990.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.