1993

Object-Oriented Modeling and Design Using DELTA, an Incremental Design Language.

Nancy Kay Gautier
Louisiana State University and Agricultural & Mechanical College

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9419887

Object-oriented modeling and design using DELTA, an incremental design language

Gautier, Nancy Kay, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1993

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

OBJECT-ORIENTED MODELING AND DESIGN USING DELTA, AN INCREMENTAL DESIGN LANGUAGE

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Nancy K. Gautier
B.S., Southeastern Louisiana University, 1980
M.N.S, Louisiana State University, 1983
December 1993

ACKNOWLEDGMENTS

Without the wisdom and guidance of my major professor, Dr. Doris L. Carver, this dissertation would not be possible. Dr. Carver has been my role model, my inspiration in believing that it is possible for me to realize my dreams. Little did I realize when she interviewed me for my first job and asked that fateful question, "Would you be willing to work on a PhD?", that she would one day become my mentor. Dr. Carver and I have established a history that has become ever present in my life and will remain with me throughout my career. I would like to thank Dr. Carver for listening to my concerns, for comforting me in times of stress, for giving me sound advice, for bringing in sunshine to moments of despair, and for giving me a sense of continuity in an ever changing discipline.

I would like to recognize the contributions of each of my committee members. Dr. Donald H. Kraft led me through my first computer science graduate course and has shown a genuine interest in my degree. Dr. J. Bush Jones made me aware that my love of mathematics could be applied to the realm of computer science. Dr. Harriet Taylor provided me with insight into programming languages. Dr. J. Robert Dorroh has nurtured my mathematical background without which I would not have attained my present status.

I would like to acknowledge the inspiration of my undergraduate advisor, the late Dr. Newton Steele Andrews, who often came to mind because of his dreams for my academic pursuits.

I would like to thank my father, the late Emile E. Gautier, Jr, who directed me toward a career in mathematics and computer science during my high school years. I would like to thank my mother, Catherine M. Gautier, for the sacrifices she made in order for me to work on a degree.

I would like to thank my sister, Ellen, for listening to my troubles, for reminding me to nourish myself, and for caring when deadlines were upon me.

I would like to thank my fiance, Rob, for realizing that my career is important to me and for not asking for more time than I could give.

Last, but not least, I would like to thank my many friends for all their words of encouragement, pats on the back, and many praises which were little stars of light in the darkest moments of my quest.

TABLE OF CONTENTS

ABSTRACT

Object-oriented technology has opened the doors for many new ideas in system development. The object-oriented paradigm has produced many new object-oriented programming languages. As with any new methodology, a need for formalism arises to remove ambiguities and inconsistencies and to bring a sense of continuity to software design. Formal languages provide a sound basis for software development throughout the software life cycle. This work presents a set of characteristic features for object-oriented design languages and defines a formal object-oriented design language, **DELTA**.

The rapidly changing face of software has led to an ever increasing need to update out-of-date methods and user interfaces. Software developers want to be able to use the same type of visual interfaces available in application software. The introduction of windowing environments has led to a market for methodologies which incorporate graphical features to supplement textual components of software. The present genre of formal languages must evolve in the same direction to be considered as effective in the design process. **DELTA** meets this need by providing a modern development environment with graphical features to complement the text that is necessary in any design specification.

Researchers and prominent software engineers have provided a litany of object-oriented methodologies. The

commonality of these methods is the step-by-step approach to software development. Software engineers agree in theory that the best approach to designing software which will stand the test of time is one which has a sound established discipline. Such a discipline produces a design in incrementations. **DELTA** supports this theory by providing established levels of incremental design representation.

The advent of computer-aided design has led to the evolution of rapid-prototyping. Changes in system requirements, detection of errors, competition in the market, and the ongoing maintenance of software systems can be addressed by the development of system prototypes. **DELTA** responds to this challenge by establishing a design specification representation which can be easily mapped to an object-oriented programming language. This transition from design to prototype can be enhanced by formal annotations to the chosen implementation language. Annotations have been developed for **DELTA** software designs prototyped in the object-oriented language Actor.

1. INTRODUCTION

We have much more powerful technology to build systems of today as compared to that of the procedural language era. The systems of today have also increased in size, complexity, and volatility from those of a decade or two ago. Batch-oriented systems have given way to today's on-line systems. These interactive systems are more concerned with developing a user interface--window capabilities, pull-down menus, manipulating icons, controlling position of a mouse--that takes up to 75 percent of the code. Building today's data-oriented systems has given the modeling of the data a higher priority than functional complexity. Object-oriented approaches allow a more natural development of such modern systems [Coa90].

Formal languages pervade all phases of software development. Object-oriented techniques have been applied to the various phases of software development as well. As a result, numerous object-oriented design methods have been proposed. Since design languages are formal languages used to enhance the design process, a natural progression is to incorporate the concepts of object-oriented design into a design language. This research defines a formal object-oriented design language, **DELTA**, which is an incremental detailed design representation.

Ivar Jacobsen, who has more than 25 years of experience in the industry and is the developer of the Objectory process, has recently stated the following as trends in future

object technology [Jac93]. Developers will work with graphical development environments, because describing objects with text alone is ineffective. In order for the development process to be effective, the modeling language for analysis and design must be formalized. The modeling language will have formal, intuitively intelligible semantics.

The trend toward Graphical User Interfaces, GUI's, has led to a demand for graphical interfaces at all phases of design. Many graphical query languages for databases have evolved. Among them are CUPID [McD75], GUIDE [Won82], PICASSO [Kim88]. Wu, developer of GLAD (Graphics LAnguage for Database [Wu87]), describes his experience in developing an easy-to-learn and easy-to-use visual interface for databases using object-oriented design techniques [Pin90]. The employment of object-oriented design improved the quality and quantity of the code which is easily modified and very reusable. It is not inconceivable to realize the same potential benefits for a design language.

The modes of expression of our languages must evolve in order for the architecture of object-oriented systems to evolve substantially and support an effective development process [Jac93]. **DELTA** advances the technology by providing graphical views and interfaces in the design-code transition.

A recent workshop assessed the role of computer-aided prototyping in software development [Luq92]. Among the benefits cited were improvements in communication which

trigger inevitable requirements changes early in development, reductions in risk by providing a basis for assessing the feasibility and performance of alternative designs, and the most feasible way to validate specifications.

Prototyping was performed manually prior to the advent of comptuer-aided design. It was slow and expensive, and resulted in prototypes that were not easily modified. Two main types of prototyping have evolved today, throwaway and evolutionary [Luq92].

Throwaway prototypes are used primarily to gain insight into the behavior of a system and the feasibility of a design. This type of prototyping is also known as rapid-prototyping. The programming language chosen may or may not be the final programming language used in the system.

Evolutionary prototypes are used to produce a series of prototypes which converge to become an acceptable version of the system. A system is modeled and feedback from customers is used to address modifications to the requirements specifications. Caution must be taken to avoid diverging in the transformation from prototype to production code. Maintenance is usually performed first on the flexible prototypes to see how changes affect the behavior of the system.

Rapid-prototyping can be applied to various types of systems. Recently, formal description techniques were developed to provide rapid-prototyping of communication protocols and concurrent systems from LOTOS (language of

temporal ordering specification) specifications [Val93]. **DELTA** provides a formal means of rapid-prototyping object-oriented specifications.

The need for an object-oriented formal design language which combines graphics with text in an incremental development environment that allows prototyping is the motivation for this research.

2. RELATED WORK

2.1 Formal Languages

The importance of formal languages was realized by Peter Naur in 1959, when John Backus presented a description of Algol in a formal syntactic notation that did not agree with Naur's interpretation of the Algol-58 report. Thus a precise method of describing syntax, Backus-Naur Form (BNF), was developed. Descriptive languages, such as BNF, are called metalanguages. A metalanguage is used to describe another language. Simultaneously, Noam Chomsky developed a mathematical description of four different classes of languages. The class of languages described by BNF is Chomsky's context-free grammars. The relationship between BNF and Chomsky's hierarchy led to the mathematical analysis of the syntax and grammar of programming languages. Among the realizable benefits developed from this analysis are automatic parser-generators, automating a previously difficult aspect of compiler writing.

Formal methods are used in systems development. A sound, well-defined mathematical basis can be provided for a formal method in terms of a formal specification language [Win90]. Formal specification languages arose from a need to complement natural language descriptions of requirements specifications. In writing technical documents, the same concept should consistently be denoted by the same words. The concepts underlying formal specification languages are basically the same. These concepts include well-known

mathematical notions like sets, functions, relations, and sequences [Mey85]. A formal specification language can be mathematically defined in terms of its syntactic domain, its semantic domain, and a relation defining the semantic interpretation of its syntactic elements [Gut82]. Specification languages represent formal methods in terms of a well-defined logical inference system that can be completely mechanized. This inference system can be used to support or challenge the validity of a requirements specification. Specification languages target particular types of users (customers and specifiers vs. specifiers and implementors) and a particular application domain (sequential vs. parallel, procedural vs. object-oriented, and others). Formal methods are not restricted to requirements analysis. All phases of system development can benefit from applying formal methods. As a result formal specification languages can be developed to represent the formal methods applied at each stage of system development [Win90].

2.1.1 Requirements Analysis

There are several specification languages which can be used in requirements analysis to transform informal requirements to formal specifications, as well as, in system design to show decomposition and refinement. A requirements specification language should reveal ambiguities, contradictions, and incompleteness in the customer's requirements.

A well-known example is the language PAISLey, a Process-oriented, Applicative, and Interpretable Specification

Language, used to formally specify requirements for embedded systems [Zav82]. The motivation in developing PAISLey was to produce requirements specifications that are precise, unambiguous, internally consistent, and complete. An LALR grammar for PAISLey in BNF was developed. Using the operational approach to development, which emphasizes the construction of an operating model of the system functioning in its environment, the requirements specifications are executable. The specification becomes a simulation model generating behaviors of the specified system. An executable specification can be tested and debugged. Demands of executability impose a coherence and discipline. PAISLey is noted as being capable of specifying the results of design decisions. A logical extension would be to investigate its properties as a design specification language. A uniform language for requirements and design would make it feasible to greatly improve the traceability and automatability of design. This might also lead to a better theoretical understanding of design. At the time PAISLey was introduced, it was noted that the potential of formal languages had not yet fully been exploited for expressing real-world concepts.

2.1.2 System Design

Formal methods can be applied to transforming a requirements specification into a system design. Using a top-down approach, this involves functional decomposition, stepwise refinement, and identifying interfaces between modules.

Using an object-oriented approach, this involves identifying objects, operations, and the interfaces between them. Representative languages include PSL/PSA, Larch, Anna, and A++.

PSL/PSA is a Problem Statement Language developed by D. Teichrow at the University of Michigan. The Problem Statement Analyzer is the PSL processor. PSL is a model that describes a system as a set of objects, where each object may have properties, and each property may have property values. Objects may be interconnected through relationships. PSL/PSA supports not only requirements analysis, but also design [Tei77].

Each language developed in the Larch family of specification languages has two components. The first is a Larch Shared Language which is common to all languages and is primarily algebraic. Its equations define relations among operators which are then used by the interface specifications. The second component is a Larch Interface Language which is particular to a specific programming language. It is used to specify program modules and to provide information needed to write programs that use these modules. Only two such languages had been developed at the time the Larch family was introduced--Larch/Pascal and Larch/Clu [Gut85].

Luckham describes two approaches to design specification languages [Luc85]. One approach is referred to as a "fresh start" because it does not have to accommodate the peculiarities of a given programming language design. The

other is referred to as an "evolutionary approach" because it extends an existing high-level programming language which people will be more likely to use. Anna, ANNotated Ada, is an extension of Ada which provides additional facilities for formal specifications. Anna's extensions are generalizations of explanatory constructs already in Ada, additions of new constructs, and additions of new specification constructs.

A++, an annotated C++, follows in Anna's footsteps by providing an annotation formalism to support object-oriented concepts [Cli90]. A++ has made an attempt to improve both code safety and efficiency. Semantic information can be expressed in behavioral specifications of objects in A++. The syntax of A++ is intended to be more natural to C++ programmers, and therefore accommodates the specific constructs of C++.

Although several design languages have been developed, there are certain features provided by **DELTA** which are lacking in its forerunners. One such feature is the formal graphical definition which **DELTA** provides of the object hierarchy. The textual counterpart can be modified and the graphical definition is simultaneously updated. Other design languages do not provide these graphical features. Another feature is **DELTA**'s language independent notation which allows greater applicability. A limitation of the existing object-oriented design languages is that they must be used in conjunction with a specific programming

language. **DELTA** addresses each of the problems noted above and thus encourages the use of object-oriented technology.

2.1.3 System Verification and Validation

Formal verification of a system is the process of showing that a design correctly implements its specification. Although complete correctness may be impossible to verify, the application of formal methods to systems has uncovered errors that would have otherwise gone undetected. Among these are flaws discovered in published algorithms and circuit designs that had been accepted as correct for years.

Kemmerer uses a variant of Ina Jo [R] to demonstrate the need for testing formal specifications in order to detect design errors. Ina Jo is a nonprocedural assertion language that is an extension of first-order predicate calculus. As Kemmerer notes, most specification languages are nonprocedural. One advantage is that no commitment is made to the order in which parts of an operation are to be performed. Another advantage is that this allows the implementor to choose the order that is most efficient in terms of time or space. A disadvantage is that generally only one possible implementation is considered when converting to an executable procedural form. A second disadvantage is that there is no guarantee that all implementations will provide the desired functionality [Kem85].

ASLAN is a specification language for sequential systems that is based on 1st order predicate calculus. RT-ASLAN was developed as an extension of ASLAN that provides methods of inserting Real-Time verification [Aue86]. ASLAN was chosen as a starting point because its specification language and processor provide top level and interlevel correctness mechanisms. RT-ASLAN was developed to address the problem of developing reliable software that meets critical correctness requirements and critical performance deadlines.

2.1.4 Taxonomy of Formal Methods

Specification languages permit concise statement and automated analysis. Some specification languages are graphical, and others are textual. Some are manually applied and others have automated processors [Fai85].

Wing [Win90] gives a taxonomy of formal methods and the corresponding specification languages. This taxonomy includes two broad classes of formal methods: model-oriented vs. property-oriented. Model-oriented methods construct the system in terms of mathematical structures. PAISLey is a model-oriented language. Property-oriented methods specify the system's behavior indirectly by a set of axioms. Anna and Larch are specification languages that employ an axiomatic method. These languages state no more than the necessary minimal system constraints making it possible for more implementations to satisfy the specification. **DELTA** is such a property-oriented method.

more implementations to satisfy the specification. **DELTA** is such a property-oriented method.

Another category in Wing's taxonomy is visual methods which contain graphical elements in the syntactic domain of the language. Any language which incorporates visual methods such as Petri nets, Harel's state charts, HIPO charts, or Booch diagrams falls into this category. The Miro visual languages were based on Harel's higraph notation. **DELTA** incorporates a visual method and thus is a part of this category.

Formal methods which support executable specifications are more restricted than non-executable languages and may suffer from "implementation bias". PAISLey is an executable specification language.

2.2 Object-Oriented Development

The object-oriented paradigm to software development is based on a data-driven approach. It permeates all phases of the software development life cycle. The decomposition of a problem begins with object- oriented techniques in the analysis phase and continues into the design phase.

The typical software life cycle with a procedural approach is based on the "Waterfall" model. Problems noted by Korson and McGregor [Kor90] are that the traditional life cycle does not include iteration, emphasize reuse of existing software, or provide a model to integrate the phases. Each system is developed from the ground up. A

large portion of the total system costs, as much as 60%, are attributed to maintenance. The object-oriented paradigm addresses each of these problems.

The object-oriented software life cycle has three general phases: analysis, design, and implementation. The boundaries between these phases are not distinct. There is a high degree of overlap and iteration. Henderson-Sellers and Edwards [Hen90] suggest a Fountain model which places requirements analysis and specification at its base. The succeeding phases bubble upward towards implementation. Maintenance plays a decreased role in the life cycle. Object-oriented programming languages adjust readily to this model. In fact, Meyer [Mey88] promotes Eiffel as both a design and implementation language.

The move towards object-oriented development has been gradual. This can be attributed to the existing programming languages. Functional decomposition is a natural way to develop systems that use procedural languages. Procedures and algorithms are the base for development. However, these systems cannot be easily maintained when new data structures or new functions are warranted.

"Object-based" techniques make a move towards viewing the system as a set of objects rather than procedures. More attention is devoted to data specifications than the procedural approach, but the architecture is still developed using functional decomposition. Jackson Structured Development (JSD) [Jac83] is one such methodology.

Analysis uses data structures to assist in the functional decomposition of a system.

A completely object-oriented approach to software development views the system as a collection of objects. This uses bottom-up techniques and is the antithesis to top-down functional decomposition. The basic components of a system are objects grouped into classes. Inheritance between classes allows one to express specializations and generalizations of concepts represented by the classes. The object-oriented paradigm arose from object-oriented languages. Since the term object-oriented brings different interpretations to mind, some of the basic concepts of object-oriented programming are defined before discussing object-oriented analysis and design.

2.2.1 Object-Oriented Concepts

1) Object

- the basic run-time entity which occupies memory
- has an associated address (like a record in Pascal)
- the bits in its allocated memory space determine its state
- has a set of procedures which define meaningful operations on that object

Example: Rectangle, Pentagon, Triangle and Circle are objects of a system

2) Class

- is a description of 1 or more similar objects (like a type in a procedural programming language)
- an object is an instance of a class

Example: the object Rectangle is an instance of the class Polygon

3) Attribute

- characteristics of a class

Example: an attribute of the class Polygon is the number_of_sides

4) Message

- actions are performed in the system by sending a message to an object
  a) the object receives a message
  b) a selector specifies what kind of action to perform
  c) the object uses its methods to perform the action

Example: Rectangle.create is a method for Rectangle which dynamically allocates space for the object Rectangle when it receives the message create

5) Inheritance

- a relation between classes that allows one to define a new class based on an existing class
- most promising concept to realize the goal of

constructing software systems from reusable parts

a) Hierarchical Inheritance

- a class is defined in terms of a single superclass

Example: A Rectangle is a Polygon which is a Closed_Figure which is a Figure.

Class	Attribute
Figure	pixel_width
Closed_Figure	perimeter
Polygon	number_of_sides

Rectangle can inherit all of these attributes and specialize them

b) Multiple Inheritance

- this increases data sharing by making it possible to combine descriptions from several classes
- a class precedence list can be used to resolve conflicts (like dominant and recessive genes that are inherited)

Example: A Rectangle is a Polygon and also a Screen_Image.

c) Benefits of Inheritance

- a class can be reused that is almost, but not exactly, what is needed
- if higher-level classes are kept in a software repository, eventually a generalization

of almost any desired class would already exist

6) Polymorphism

- the ability to take more than one form

Example: the message "draw" will take on different methods for each type of polygon

7) Dynamic Binding

- the code associated with a message is not known until run-time

Example:

Current_Figures.item(i).draw

- the code matching the dynamic type of Current Figures.item(i) will be executed
- this code remains unchanged even if additional kinds of shapes are added to the system

2.2.2 Object-Oriented Analysis

Most computer professionals know several programming languages and diagramming techniques to represent design. However, most of them know only one approach to analyzing and developing a system because learning a new development requires a fundamental change in their way of thinking [Kor90]. If one follows a procedural paradigm, then one takes a task-oriented viewpoint which produces procedures that perform tasks. Suppose a system developer is given a statement of requirements for a software system which would control the traffic lights at the intersection of two

streets. A procedural paradigm approach would propose a functional decomposition that would include these functions:

- -Control the Lights of the Intersection
  - -Set Initial Sequence of Lights
  - -Check Clock
  - -Read Sensors
    - -Poll each Sensor
    - -Set each Sensor Bit
  - -Decide on Next State
  - -Change Status

If one follows an object-oriented paradigm, then one takes a modeling viewpoint which produces class descriptions that model the problem domain. An object-oriented paradigm approach to the above requirements specification would identify the entities that are present in the problem:

- -Intersection
  - -Direction
    - -Lane
      - -Sensor
        - -Trip Sensor
        - -Pressure Sensor
      - -Traffic Light
        - -Turn Light
        - -Straight Light
- -Controller
  - -Clock

This approach would also determine messages between objects:

-Controller controls TrafficLight

-Controller reads Sensor

Object-Oriented Analysis (OOA) involves modeling the problem domain which consists of real-world objects and operations. The analysis phase examines the Requirements Specification and determines what is needed to model the problem. This is determining what the system must do to satisfy the customer. OOA is based on the uniform application of methods of organization, communication with messages, and behavior classification. OOA [Coa90] consists of five major steps:

1) Identifying Objects

2) Identifying Structures

3) Defining Subjects

4) Defining Attributes (and Instance Connections)

5) Defining Services (and Message Connections)

Various techniques have been developed to analyze the Requirements Specification and to develop object-based models that can be used in the design phase [Lee90], [Car90], [Lee91], and [Cor92].

2.2.3 The Design Phase of Software Development

Design takes the What? of the Requirements Analysis Specification and translates it to the How? of a Design Specification. The goal of the designer is to produce a model or representation of a system in sufficient detail so

that its physical implementation in a programming language can be realized.

2.2.3.1 Software Design Fundamentals

Software design methodologies are still in their infancy. They began with refining the architecture of software in a top-down manner [Wir71]. A set of criteria was introduced for the development of modular programs [Den73]. The philosophy of structured programming came about from the procedural aspects of design definition [Dah72]. Continued efforts produced methods for the translation of data flow [Ste74] and data structure [Jac75] into a design definition. Latest efforts have proposed an object-oriented approach to design [Cox86]. Software design continues to evolve as new architectures, new design methodologies, better design analysis techniques, and a broader based understanding of the effects of a good design are developed. Common characteristics found among the various design methodologies are [Was80]:

1) a mechanism for the translation of information domain representation into design representation
2) a notation for representing functional components and their interfaces
3) heuristics for refinement and partitioning
4) guidelines for quality assessment.

Although design disciplines are often lacking flexibility, object-oriented techniques can provide a quality design that can lend itself more readily to change.

The first step in the design phase is to apply the concepts of data abstraction and information hiding to develop a data design. Data abstraction allows one to select data structures which are logical representations of the data objects which have been identified in the requirements definition and specification phase [Was80]. By reducing interdependencies between software components, information hiding is important for ensuring reliability and modifiability of software systems [Pas86]. All of the various design methodologies address the issues of data design in some fashion.

The purpose of architectural design is to represent the relationships among the major structural elements of a system. Architectural design also brings together program and data structure. Interfaces are defined as well which enable data to flow throughout the system. Various design methods treat architectural design as a holistic view of software. Just as an architect does not concern himself with the type of nails, the brand of paint, or the type of flooring that goes into his design, designers must not concern themselves with coding details in architectural design.

Once the preliminary design concerns of transforming requirements into data and software architecture have been met, the detail design phase refines the architectural representation to develop detailed data structure and algorithmic representations of the software. Because there are

ambiguities involved in using a natural language to specify a design, a more constrained form of expression needs to be used.

There are various design notations. Among them are graphical design tools, such as the commonly known flow-chart and the box diagram. These are mainly for procedural approaches to design. Booch diagrams [Boo83] and modular design charts as proposed by Wiener and Sincovec [Wie84] are developed for object-oriented design approaches. Tabular design tools also exist, such as decision tables which translate actions and conditions into a table which can be used as machine-readable input to a table driven algorithm. Yet another design notation is a **P**rogram **D**esign **L**anguage (PDL). A PDL is a pidgin language that uses the vocabulary of one language, like English, and the overall syntax of another, like a structured programming language [Cai75]. Any of these design notations, if used properly, can be an invaluable aid to developing a design representation. No matter how good a design notation is however; if it's not used as it was intended, then it does nothing to clarify the design representation and may even introduce error. The design notation should provide a design representation that is easy to understand and review. There should be a natural transition from design to actual code. To assure correctness, the design representation should be easy to maintain [Pre92]. **DELTA** is an object-oriented

design representation which combines graphical methods with PDL-like text.

Software metrics can be used to assist in proper development methods applied in the design phase. A framework for object-based measurement [Car88] has been developed. It measures characteristics of the initial requirements document from an object-oriented point of view. The primary goal of applying these metrics is to reduce subjectivity. Added benefits derived are support for the determination of suitability of an object-oriented paradigm, and identification of potentially complex parts of the system.

2.2.3.2 Object-Oriented Design Methods

Object-Oriented Design (OOD) is a methodology that combines data design, architectural design, and procedural design. Data design is developed by identifying objects. Booch [Boo91] defines an object to be a model of a real-world entity that combines both data and operations on that data. OOD is used to show the hierarchy of "objects" in a system. These objects are the basic modules in a software architecture. OOD is a set of interconnected objects [Was89]. Identifying operations that can be applied to objects and developing interfaces to send messages results in establishing an architectural structure. Defining operations in detail provides procedural design. A goal of OOD is to reduce the total life cycle software cost by increasing programmer productivity and reducing maintenance

costs. Another is to implement software systems that resist both accidental and malicious corruption attempts.

The following steps have been proposed by Booch [Boo83] for OOD:

(1) Define the problem

(2) Develop an informal strategy for the software realization of the real-world problem domain

(3) Formalize the strategy using the following substeps:

   a) Identify objects and their attributes

   b) Identify operations that may be applied to objects

   c) Establish interfaces by showing the relationship between objects and operations

   d) Decide on detailed design issues that will provide an implementation description for objects

(4) Reapply steps 2, 3, and 4 recursively until a complete design is created

It has been noted that the above method is directed towards realization in languages like ADA which do not take advantage of some of the more powerful object-oriented concepts such as inheritance and message sending. William Lorensen [Lor86] has developed the following OOD method:

(1) Identify the data abstractions for each subsystem

(2) Identify the attributes for each abstraction

(3) Identify the operations for each abstraction

(4) Identify the communication between objects

(5) Test the design with scenarios

(6) Apply inheritance where appropriate

The designer is to repeat steps 1-6 for each level of abstraction.

General Object-Oriented Software Development (GOOD) was developed by Seidewitz and Stark at NASA Goddard Space Flight Center [Sei86]. This method handles the issues of requirements specification and design phases of a software development life cycle that is Ada-oriented. The specification phase employs dataflow diagrams used to identify abstract entities. "Abstraction analysis" allows those entities to be transformed into objects in the design phase. Operations are identified by mapping these objects back to the requirements. The design phase uses object diagrams to show communication among the objects. Detailed design is performed by decomposing and annotating objects with an object description.

Hierarchical Object-Oriented Design (HOOD) is a combination of Matra's work on abstract machines and Cisi-Ingeniere's work on OOD. One of HOOD's major goals is to map its features directly to Ada concepts. A "uses hierarchy" shows how abstract objects use one another. Active objects, which interact directly with a control flow, and passive objects are both supported by HOOD [Hei87].

Multiple-view Object-Oriented Design (MOOD) is a structured OOD methodology developed by Kerth which applies concurrency. This method uses an analysis model developed with Ward/Mellor's Structured Analysis with Real-Time Extensions [War85] to support program construction.

Even though MOOD supports OOD, it allows concurrent processes to be expressed as tasks, not objects. Objects and tasks are identified, and their influence on each other is noted. Implementation of objects is addressed as well as sequential execution of a routine [Ker88].

Wasserman [Was89] observed that several of the above methods, (GOOD, HOOD, and MOOD), address an architectural design method, but have limitations. These methods do not make distinctions between the definition and use of objects that are adequate enough to develop a library of reusable objects. Support for inheritance seems to be limited. The methods seem to be more suited for Ada's interpretation of an object (a package or task), as opposed to addressing more general ideas of OOD. The concepts of structured design are lacking as well. Wasserman tries to overcome these limitations with a method called **O**bject-**O**riented **S**tructured **D**esign (OOSD) that attempts to provide a single architectural design notation that can support every software design. OOSD attempts to support generation of code from its notation for various object-oriented languages such as Ada, C^{++}, Eiffel, and others.

Bertrand Meyer presents techniques for OOD [Mey88]. He addresses several design issues such as finding classes, defining interfaces, and incorporating inheritance.

In summary, methods have been developed for OOD. These methods identify data, develop an architectural structure, and establish a basis for a detailed design representation.

DELTA can be used to represent an architectural design. **DELTA** refines this architectural design incrementally by developing detailed data representations, both visual and textual, in a formal object-oriented design language. **DELTA** is language independent which allows new data structures and constructs to be easily represented. By bringing formalism to the design phase in a visual method, **DELTA** provides an environment in which to easily maintain design specifications. This supports a means to assure correctness and consistency while removing ambiguity.

3. LANGUAGE FEATURE FOUNDATIONS OF DELTA

Formal specification languages were originally developed to satisfy a need to complement natural language descriptions of requirements specifications. However, formal specification languages have permeated all stages of system development. The common basis for formal specification languages is a firm mathematical foundation for a formal method. It is difficult to accommodate the diverse technology and complex systems of today with one formal method. As discussed in 2.1.4, various areas have been addressed by formal methods and Wing [Win90] has developed a taxonomy of formal methods.

Since **DELTA** is a formal language with visual and textual representations, preliminary work to designing its syntax involved establishing a basic set of characteristics for an object-oriented design language [Gau91]. These characteristics established a basis on which to build the constructs of **DELTA.**

Object-oriented techniques are touted as a more natural way of developing today's data-oriented, interactive, online systems. Top-down design methods are not conducive to tapping the potential of object-oriented programming languages of today, namely inheritance, which promotes reusability of code [Mey87]. To meet this demand, numerous object-oriented design methodologies have been proposed as discussed in section 2.2.3.2. Various design notations have also been developed and take many different forms.

One such design notation is a Program Design Language (PDL). A PDL may be either an extension of an existing language, adding new constructs to it, or it may be developed specifically for use as a design representation. Besides combining several qualities of a good design notation, a PDL can be used to provide documentation for ease of ongoing maintenance of software. Depending on the final choice of the implementation language, there is a possibility for automatic code generation.

Most existing PDL's have been developed for procedural language design methods, namely top-down design. A set of characteristics for a basic PDL syntax has been provided by Pressman [Pre92]. These characteristics were derived from procedural languages. Modern systems can be developed more rapidly by adapting existing software to meet the needs of the current application domain. In order for an Object-Oriented Design Language (OODL) to accommodate this goal, one must determine how an existing library of classes can be best utilized. This utilization is dependent upon the type of inheritance that an implementation language has to offer, whether it be single or multiple inheritance. Certain domain applications lend themselves more readily to a design that is best implemented using single inheritance, and others would benefit most from a design which has incorporated multiple inheritance. As a result of examining the different object-oriented design methodologies that have been developed in recent years, a basic set of

characteristics for an OODL was defined. These characteristics provided a foundation for the development of **DELTA.**

Since most existing design languages have been based on a procedural paradigm, the characteristics of those languages were used as a basis to establish characteristics for an OODL. A comparison of the procedural and the object-oriented paradigms gives insight as to some of the benefits to be derived from using an object-oriented approach over a procedural approach. One of the key differences in the two paradigms is inheritance which can enhance a design [Mey87]. However, inheritance presents a multitude of problems in software development, if not used correctly or to its full potential [Hal87], [Ste86]. An OODL should present a way to incorporate inheritance to overcome these problems without introducing additional development costs. The incorporating of inheritance into an OODL is discussed and a summary of inheritance traits is given. A survey of various object-oriented design methods has shown some of the benefits as well as some of the inadequacies of existing object-oriented design representations. Paralleling the characteristics of a basic procedural design language syntax, the characteristic features of an OODL which should be incorporated into its syntax are presented. The benefits to be derived from using an OODL are stated.

3.1 Importance of Formal Languages

The importance of formal languages was realized by the developers of BNF notation which provided a precise method of describing syntax. As Wing notes [Win90], there are several factors which contribute to the importance of formal languages, among which are:

-- Establish a sound mathematical basis for a formal method
-- Enhance every phase of system development
-- Support or challenge the validity of a formal method
-- Add new constructs to a language
-- Detect flaws in existing formal methods

As stated previously, formal languages can be used to enhance every phase of system development. Object-oriented techniques have been applied to various phases of software development as well. As a result, numerous object-oriented design methods have been proposed. Since design languages are formal languages used to enhance the design process, a natural progression is to incorporate the concepts of object-oriented design into a design language.

3.2 Characteristics of a Design Language

Because there are ambiguities involved in using a natural language to specify a design, a more constrained form of expression needs to be used. One such design notation is a PDL as described in section 2.2.3.1. New constructs or extensions to languages have evolved as a result of PDL's. A PDL may resemble a programming language such as

PASCAL or ADA, or it may be developed specifically to use as a design notation. In either case, the following characteristics are considered to be desirable for a design language [Pre92]:

(1) a fixed syntax of keywords that provide for all structured constructs, data declarations, and modularity characteristics

(2) a free syntax of natural language that describes processing features

(3) data declaration facilities that should include both simple and complex data structures

(4) subprogram definition and calling techniques that support various modes of interface description

Other characteristics which are deemed appropriate are:

-- provide documentation

-- automatic code generation

A PDL can use familiar control constructs found in high-level languages, while allowing descriptions that are less restrictive of functional operations. Abstract data types or new concepts can be described without the compiling requirements of the underlying programming language. Creative thinking is not as influenced in a PDL as in the rigid constructs of a programming language [Som92].

3.3 Characteristics for a Basic PDL Syntax

Existing PDL's have been developed mostly for procedural languages. As a result, they have been modeled after commonly used "structured" programming languages. Pressman

[Pre92] provides a list of characteristics of a basic PDL syntax:

-- subprogram definition

-- interface description

-- data declaration and typing

-- techniques for block structuring

-- condition constructs

-- repetition constructs

-- I/O constructs

3.4 Procedural vs. Object-Oriented Paradigm

In order to gain insight into the differences between a PDL used mainly for procedural languages and an OODL for object-oriented languages, a comparative study was made on the two paradigms. The distinguishing traits are summarized in the following table.

PROCEDURAL PARADIGM	VS.	OBJECT-ORIENTED PARADIGM
--Top-down functional decomposition		--Bottom-up data modeling approach
--Does not promote reusability		--Promotes reusability
--Difficulty in maintaining a system that warrants new data structures or taps new technology		--Allows ease of maintaining volatile systems

3.5 Inheritance -- Single vs. Multiple

Just as genetic engineering attempts to capitalize on certain existing traits and remove undesirable traits in an organism, inheritance allows software engineers to create specializations of existing classes and objects by inheriting desirable attributes, variables, and methods and overriding unwanted descriptions with substitutions. Because Object-Oriented Programming (OOP) languages offer a library of existing classes, inheritance should be used to allow reusability of that code. This will not only reduce total lines of code, but also increase programmer productivity. Inheritance will help to eliminate duplication of code, increase flexibility, and consequently help to ease ongoing software maintenance. In order to capitalize on the benefits of inheritance, Object-Oriented Design (OOD) is preferred over procedure-oriented design as Bertrand Meyer states: "Top-down design goes against reusability" [Mey87]. Reasons Meyer gives why reusability is not more common are:

1) economic--programmer is not needed on next job
2) not-invented-here
3) libraries of reusable modules are needed
4) design with reusability isn't easy
5) too many variants
6) tough to capture commonality within a conceptual subset

Besides these problems, dynamic binding which makes inheritance possible also increases run-time costs.

Just as some organisms have only one parent, there are some classes, types, or objects which may be defined in terms of a single superclass, supertype, or superobject. This is called single or hierarchical inheritance. Specialization describes the ability to inherit all or some existing properties and to define local properties, which may be additional or substitutional, thereby overriding existing properties. An OODL should be able to allow for the incorporation of these characteristics of inheritance.

Just as we are products of a combination of our parents' genes, there are certain subclasses, subtypes, or subobjects that have multiple superclasses, supertypes, or superobjects. Multiple inheritance increases the potential for code sharing, as well as, the complexity of software design. Halbert and O'Brien [Hal87] discuss the various ways one may use multiple inheritance to combine multiple supertypes. Supertypes of relatively equal importance may be combined to create a new subtype. Both contribute major attributes that are inherited by the subtype. However, sometimes there is a primary supertype combined with auxiliary supertypes. One supertype contributes the major attributes to the subtype and the other supertypes contribute attributes which refine or add to the major ones. Common misuse of supertyping occurs when one who is inexperienced in OOD defines supertypes as various parts of an object and the subtype as the object composed of these parts. An OODL should help to avoid this misconception. Another problem

to be addressed by an OODL, discussed by Stefik and Bobrow [Ste86], is how one may resolve conflicts of multiple definitions through the use of a class precedence list. In Smalltalk, no precedence is assumed. The user explicitly states a dominant class. Flavors and Loops use a fixed precedence relationship. The precedence relation for any given class is determined by its metaclass in CommonLoops.

The summary of inheritance characteristics mentioned above follows:

INHERITANCE

--Reusability of Code

--Increased flexibility

--Ease of ongoing maintenance

--Relies on dynamic binding

SINGLE	VS.	MULTIPLE
--inherits properties from one superclass		--inherits properties from more than one superclass
--library of classes		--two types
--specialization		--equal inheritance
--selecting desired properties		--primary superclass and auxiliary superclasses
--omitting undesired properties		--common misuse of supertyping
--redefining existing properties locally		--conflicts resolved by a class precedence list

3.6 Language Features

A survey of various object-oriented languages produced common traits which led to the determination of the following characteristics. These characteristics are presented as features which should be included in an OODL.

-- Definition of objects

-- Class Definition

- a description of 1 or more similar objects

-- Method Definition

- condition constructs
- repetition constructs
- I/O constructs

-- Message Passing

- establishing a protocol, a set of related methods to perform an action

-- Encapsulation

- all operations defined for an object must be protected from unauthorized access

-- Polymorphism

- allows different classes of objects to respond to same protocols thus enabling interchangeable pieces of code

-- Inheritance

- allows commonality of existing objects or classes to be exploited
  - Single

- Multiple
  - definition of a class precedence list

-- Garbage Collection
- allows dynamic memory management

3.7 Summary

If software designers have an OODL which makes implementation of the design easier, then they are more likely to tap the benefits of OOD and OOP languages. Coupling is a measure of interconnection among software modules. OOD encourages information-hiding and data abstraction which increase reliability and minimize coupling. OOD also makes the design more flexible and resilient to change which will reduce the cost of software maintenance. Inheritance promotes reusability and code factoring which reduce time and total lines of code, thereby increasing programmer productivity.

A summary of the above mentioned benefits of an OODL is:

-- Basis for development of a formal language in the design phase

-- Emphasis on inheritance

-- A means of tapping the benefits of OOD and OOP languages

--Encapsulation

--allows polymorphism

--increases reliability

--minimizes coupling

--Inheritance which promotes reusability and code factoring

--increases productivity

--reduces chance of error

-- Flexibility of OOD makes it more resilient to change

--eases ongoing maintenance

4. DELTA

Graphical and textual notations which reflect and support object-oriented concepts at all stages of system development are needed to realize the emergence of object-oriented technologies [Edw93]. Graphical notations to represent a design, such as Booch diagrams [Boo91] can only be used to a certain point of abstraction. A design description language, or PDL, can best describe the lowest level of a software design. **DELTA** is such a design description language which incorporates the language features of an OODL presented in section 3.6. **DELTA** provides a high-level graphical/textual representation of object-oriented design concepts and has at its foundation a formal specification language. **DELTA** aids designers of object-oriented software systems and object-oriented databases by providing an online facility to develop the design incrementally. **DELTA** is a language independent design representation which allows for a wide range of adaptability. **DELTA** also provides a basis for prototyping.

The goals of **DELTA** were established with realizable benefits such as documentation and rapid-prototyping. Levels of **DELTA** were created to provide a graphical development environment which allowed incremental design representations. A formal syntax was then established which held the principles of object-oriented design. Finally, a prototype of **DELTA** was written to ensure its feasibility as a

graphical design language environment which embodied the concepts of object-oriented design.

4.1 Definition of the Formal Design Language DELTA

The foundations of design languages are the characteristic features of the paradigm which they employ. These features provide a base upon which the language constructs are built. **DELTA** contains features which are inherent to the object-oriented paradigm. A formal description of **DELTA** is presented as follows. The goals of **DELTA** are stated. The definition of the grammar for **DELTA** is given. The graphical features and corresponding textual views of **DELTA** are shown.

4.1.1 Overview of DELTA

DELTA is a design specification language which supports the following goals:

1) The user should be able to access a graphical view of text at all levels of abstraction.
2) The notation, both graphical and textual, should support the standard object-oriented concepts in object-oriented programming languages.
3) Simplicity should be maintained throughout the language.
4) Documentation should be readily available for each phase.
5) Rapid-prototyping should be possible.

4.1.1.1 Graphical View of DELTA Text

DELTA provides the developer with an environment which allows two views, a graphical and a textual view, of each level of the design-code transition. All object-oriented design specifications contain diagrammatic illustrations of the object hierarchy. However not all design environments provide the capability to represent these concepts. At any level in detailed design development in **DELTA**, a graphical and textual view are accessible. As changes are made in the design-code transition phase, the graphical and textual views are simultaneously updated.

4.1.1.2 Notation of DELTA

DELTA supports existing object-oriented terminology and concepts. A developer who is already familiar with those terms and concepts will be able to quickly take advantage of **DELTA** notation. **DELTA** can be used incrementally. This allows a developer who is not familiar with a particular object-oriented programming language to develop a system that begins as language independent, but leads to a detailed design which is language dependent.

Language independent strategies are needed in dealing with persistent objects in databases. Database repositories which are shared by many departments and divisions of a large organization often contain persistent objects that are not written in the same language. A language independent design representation is more likely to find universal

support in projects which access persistent objects developed in various languages [Dan93].

4.1.1.3 Simplicity

By keeping simplicity in **DELTA**'s notation, an experienced object-oriented software developer will have a relatively small learning curve and will be more apt to use formalism in the design code transition. A developer without the familiarity of object-oriented terms and concepts can incrementally become familiar with **DELTA**. **DELTA** provides the basic set of characteristics for the language features of an OODL as presented in section 3.6.

4.1.1.4 Documentation

The turnover rate of developers and programmers is high. Documentation is an important aspect of compensating for the loss of knowledgeable employees. **DELTA**'s syntax promotes a clear, well-documented version of the design, both graphical and textual, at all levels. This enables modification of an existing system to be performed by those who are familiar or unfamiliar with the project.

4.1.1.5 Rapid-Prototyping

One of the major advantages of object-oriented systems over procedural systems often cited is rapid-prototyping. **DELTA** supports this characteristic by using a formalism which can be easily mapped to an object-oriented programming language. One language implementation of a **DELTA** design used in rapid-prototyping has been done using Actor [Gau93a]. By using the existing syntax of the

implementation language's comments, **DELTA** provides a method to incorporate detailed design documentation which allows compilation and immediate execution. As time is a large factor in producing quality software which is marketable, and change is a definite in any software development process, rapid-prototyping provides a means to reduce time and allow change.

4.1.2 Graphical and Textual Features of DELTA

DELTA is composed of four distinctive levels which promote top down decomposition. The first level displays the hierarchy of classes in the system with a textual counterpart. The second level displays one class with its class and instance variables with a textual listing of variables. The third level displays one class with its class or object methods with a textual listing of methods. The fourth level displays one method with its arguments, local variables and return type with a textual counterpart.

4.1.2.1 Formal DELTA Grammar

The philosophy in the current **DELTA** design has been to provide the necessary minimal system constraints for representing an object-oriented design. This makes it possible for **DELTA** to be implemented in more object-oriented programming languages. The formal **DELTA** syntax is defined with the following lexical conventions. Terminals or keywords in **DELTA** are indicated in capital letters. Formal **DELTA** comments which can be used in the **DELTA**/Actor prototyping combination are indicated in the delimiters /*@

and @*/. As experience with applying **DELTA** to various implementation domains is gained, **DELTA** will be revised and expanded.

The formal **DELTA** grammar is defined as follows:

```
ClassHierarchy : OBJECT DescendantClasses
DescendantClasses : /* empty */ | HierarchicalList
HierarchicalList : /* empty */ | LEVEL [LevelNumber]
                                 CLASS ClassName
                                 ANCESTOR ClassName
                                 HierarchicalList
LevelNumber : NonZeroDigit SucceedingDigits
SucceedingDigits : /* empty */
                   | NonZeroDigit SucceedingDigits
                   | Zero SucceedingDigits
NonZeroDigit : 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Zero : 0
ClassDeclaration : CLASS ClassName;
                   ClassDescription
                   ENDCLASS;
ClassDescription : AncestorStatement
                   ClassVariableStatement
                   InstanceVariableStatement
                   ClassMethodStatement
                   ObjectMethodStatement
AncestorStatement : ANCESTOR
                         ClassName;
```

```
ClassVariableStatement : CLASSVARIABLES
                                ClassVariableList
ClassVariableList : NONE | CVNameAncestorList
                           CVNameList
CVNameAncestorList : /* empty */
                   | $CVName INHERITEDFROM Ancestor;
                     CVNameAncestorList
CVNameList : /* empty */ | $CVName;
                           CVNameList
InstanceVariableStatement : INSTANCEVARIABLES
                                    InstanceVariableList
InstanceVariableList : NONE | IVNameAncestorList
                              IVNameList
IVNameAncestorList : /* empty */
                   | $IVName INHERITEDFROM Ancestor;
                     IVNameAncestorList
IVNameList : /* empty */ | IVName;
                           IVNameList
ClassMethodStatement : CLASSMETHODS
                               ClassMethodList
ClassMethodList : NONE | CMNameList
CMNameList : /* empty */ | CMName (ParameterList)
                           RETURNTYPE type;
                           CMNameList
ObjectMethodStatement : OBJECTMETHODS
ObjectMethodList : NONE | OMNameList
```

```
OMNameList : /* empty */ | OMName (ParameterList)
                           RETURNTYPE type;
                           OMNameList
ClassMethodDefinition : DEFINE CLASSMETHOD
                        CMName (ParameterList)
                        RETURNTYPE type;
                        StatementBlock
ObjectMethodDefinition : DEFINE OBJECTMETHOD
                         OMName (ParameterList)
                         RETURNTYPE type;
                         StatementBlock
ParameterList : self | self,
                       ArgumentList;
                       LocalVariableList
ArgumentList : ARGUMENTS ArgNameList
ArgNameList : NONE | ArgName ANList
ANList : /* empty */ | , ArgName ANList
LocalVariableList : LOCALVARIABLES LVNameList
LVNameList : NONE | LVName LVNList
LVNList : /* empty */ | ,LVName LVNList
StatementBlock : { StatementList }
StatementList : /* empty */ | Statement; StatementList
Statement : Assignment | Conditional | Iteration | Message
            | Return
Assignment : Object /*@ ISASSIGNED @*/ AssignStmt;
AssignStmt : Assignment | Expression | StatementBlock
Expression : Literal | Message
```

```
Conditional : IfStmt | IfElseStmt | SelectStmt

Iteration : Enumeration | Loop

Enumeration : DoStmt | CollectStmt | ExtractStmt

Loop : StatementList /*@ ENDCONDITION @*/ Condition |
      /*@ INITIALCONDITION @*/ Condition StatementList

Return : /*@ RETURN @*/ ^Statement

IfStmt : /*@ IFSTATEMENT @*/
         IF Condition THEN StatementList ENDIF;

IfElseStmt : /*@ IFELSESTATEMENT @*/
             IF Condition THEN
                StatementList
             ELSE
                StatementList
             ENDIF;

SelectStmt : /*@ SELECTSTATEMENT @*/
             SELECT CaseStmtList
             DEFAULT StatementList
             ENDSELECT;

CaseStmt : /*@ CASESTATEMENT @*/
           CASE Condition IS
                StatementList
           ENDCASE;

Message : /*@ SELECTOR @*/ MessageName
          (/*@ RECEIVER @*/ Receiver
           /*@ ARGUMENTS @*/ ArgumentList);

Receiver : /*@ NONE @*/ | ObjectName

MessageArgumentList : /*@ NONE @*/ | ,MsgArgName MANList

MANList : /* empty */ | ,MsgArgName MANList
```

Since **DELTA** can be used incrementally to represent a design, the following sections describe this process.

4.1.2.2 Level I

The first phase of any object-oriented design includes identification of the classes to be used in the system. Level I allows the user to represent the selected classes graphically showing their hierarchical nature. For example suppose the diagram in Figure 1 illustrates the classes and their hierarchical relation. All classes are descendants

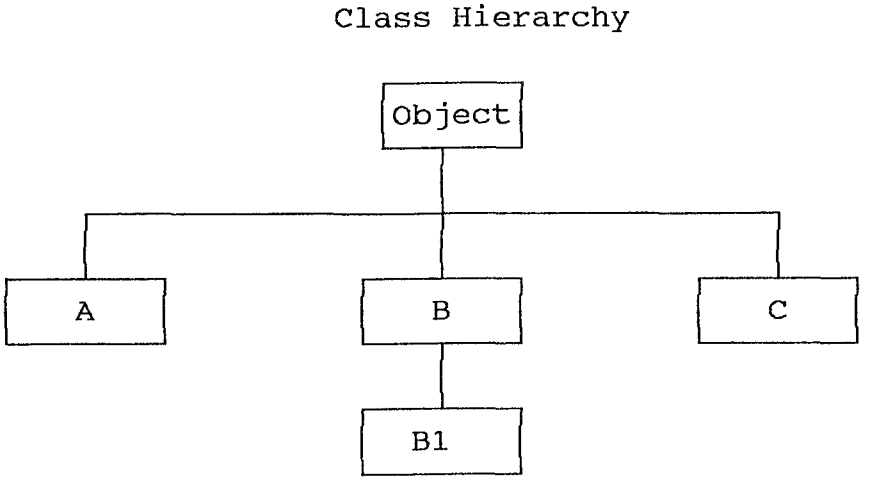

Figure 1 - Sample Class Hierarchy

of class **Object.** The corresponding textual display of **DELTA** shows these objects listed hierarchically with annotations indicating the level and ancestor of each class.

```
                Annotated Class Hierarchy

        Object

             LEVEL [1] CLASS A ANCESTOR Object

             LEVEL [1] CLASS B ANCESTOR Object

                  LEVEL [2] CLASS B1 ANCESTOR B

             LEVEL [1] CLASS C ANCESTOR Object
```

Section 4.1.3.1 presents the grammar which represents the **DELTA** representation of this level. Whenever the basic hierarchy needs to be modified, the textual counterpart of Level I can be accessed and the graphical view is updated.

4.1.2.3 Level II

The second phase of design representation is the class definition. **DELTA** provides annotations to indicate the class and ancestor, as well as to distinguish between class and instance variables. **DELTA** dialog prompts the user for information which creates the following text:

```
CLASS B1;
ANCESTOR B;
CLASSVARIABLES
     $CVName INHERITEDFROM Ancestor;
     ...
     $CVName;
     ...
INSTANCEVARIABLES
     IVName INHERITEDFROM Ancestor;
     ...
     IVName;
     ...
```

Section 4.1.3.2.1 presents the grammar which represents the **DELTA** representation of this level. The user may select to view the graphical display shown in Figure 2. Each class box contains icons to represent Class Variables and Instance Variables. Upon selection the user is able to

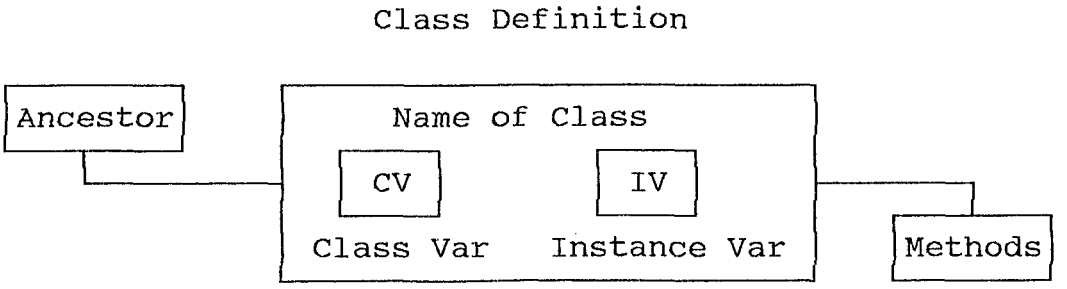

Figure 2 - **DELTA** Class Definition Graph

view a listing of variables. The corresponding textual selection can be displayed. Any changes made to the system are automatically updated and are reflected here.

4.1.2.4 Level III

The next incremental phase is the design process is to define the methods for a class. Methods provide the functionality of classes. **DELTA** provides annotations to distinguish CLASS from OBJECT methods and to indicate the RETURN type. **DELTA** dialog prompts the user for information which creates the the following text.

```
CLASSMETHODS
     CMName (ParameterList) RETURNTYPE type;
     ...
OBJECTMETHODS
     OMName (ParameterList) RETURNTYPE type;
     ...
```

Section 4.1.3.2.2 presents the grammar which represents the **DELTA** representation of this level. The graphical view of Level III is shown in Figure 3. When either **Class Methods** or **Object Methods** is selected, a listing of the corresponding methods is displayed. Any changes made to

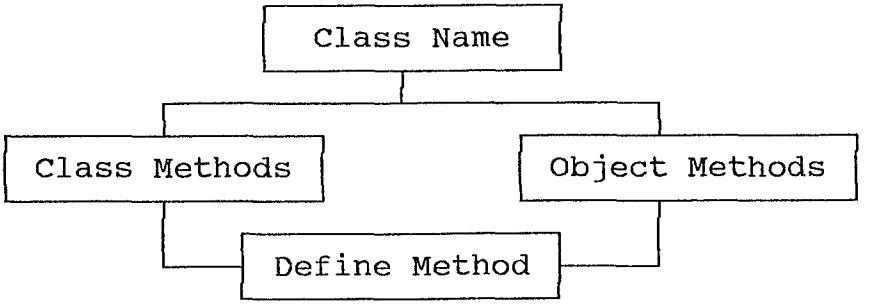

Figure 3 - **DELTA** Level III Methods Graph

the system are automatically updated and are reflected here.

4.1.2.5 Level IV

The fourth phase of design representation is a more detailed representation of a method which provides the functionality to a class of objects. **DELTA** dialog prompts the user for information which creates the following annotated text.

```
DEFINE CLASSMETHOD MethodName
     ( ARGUMENTS Argument, Argument, ...
     | LOCALVARIABLES LocalVariable, ...)
       RETURNTYPE type;
{
  StatementList
}
```

Section 4.1.3.3 presents the grammar which represents the **DELTA** representation of this level. The graphical view shown in Figure 4 is displayed. The name of the class and method are displayed in the boxes on the left. One of the boxes on the right may be selected for creating, or

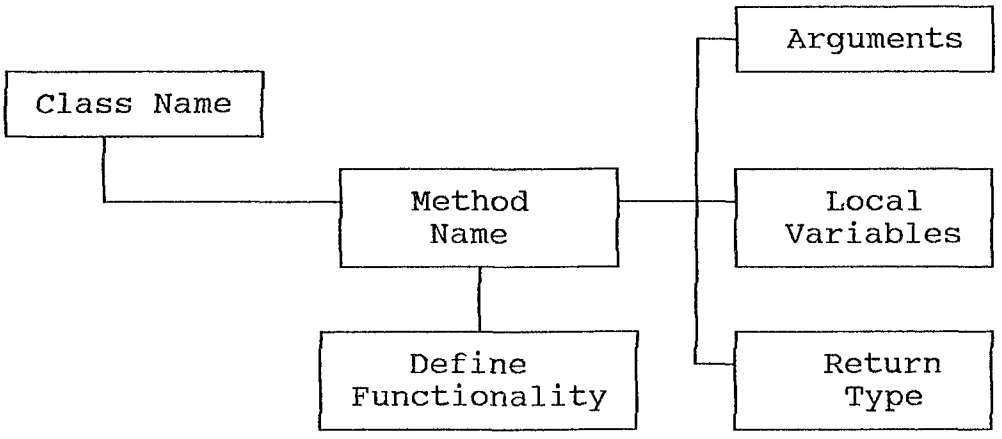

Figure 4 - **DELTA** Level IV Method Definition Graph

viewing/modifying existing attributes. Once again, the capability to view and/or modify any of these parameters is available.

Section 4.1.3 presents the formal grammar of **DELTA** which provides a consistent and unambiguous way of representing the design for each of the above levels. This provides a software developer the means with which to express a design to the programmer. **DELTA** also provides a documented form of the system design. The documentation **DELTA** provides will prove invaluable in the modification and maintenance of the system, or in the use of actual system development when prototyping.

4.1.3 Formal Definition of DELTA

The grammar is presented separately for each level as discussed in section 4.1.2. The first part of the grammar corresponding to Level I provides a formal way to represent the class hierarchy. A basic outline of the grammar which

corresponds to Level II and III is presented and discussed. This is followed by a more in depth discussion of each level. Next the formal grammar which corresponds to Level IV is given.

4.1.3.1 Class Hierarchy

The class hierarchy of the classes to be used in a system is a fundamental aspect of the design. A formal way to represent ancestor and descendant classes which use single inheritance is included in **DELTA**. The grammar which corresponds to Level I of Section 4.1.2.2 follows.

```
ClassHierarchy : OBJECT DescendantClasses
DescendantClasses : /* empty */ | HierarchicalList
HierarchicalList : /* empty */ | LEVEL [LevelNumber]
                                 CLASS ClassName
                                 ANCESTOR ClassName
                                 HierarchicalList
LevelNumber : NonZeroDigit SucceedingDigits
SucceedingDigits : /* empty */
                 | NonZeroDigit SucceedingDigits
                 | Zero SucceedingDigits
NonZeroDigit : 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Zero : 0
```

4.1.3.2 Basic Outline

The basic outline of a **DELTA** class is composed of two parts. The first part is a textual representation for the first increment in defining classes. The second part of the outline is a top-level representation for the next

```
PART I     CLASS ObjName;
           ANCESTOR
                ObjName;
           CLASSVARIABLES
                $CVName INHERITEDFROM Ancestor;
                ...
                $CVName;
                ...
           INSTANCEVARIABLES
                IVName INHERITEDFROM Ancestor;
                ...
                IVName;
                ...
PART II    CLASSMETHODS
                CMName (ParameterList)
                       RETURNTYPE type;
                ...
           OBJECTMETHODS
                OMName (ParameterList)
                       RETURNTYPE type;
                ...
           ENDCLASS;
```

Figure 5 -- Outline of a **DELTA** class

increment in defining Class and Object Methods. The outline is presented in Figure 5.

4.1.3.2.1 Class Definition

A **DELTA** class is defined in increments. The first increment is the textual representation shown in Part I of Figure 5. **DELTA** provides a textual version of the class definition in Level II as described in section 4.1.2.3. **DELTA** provides information, such as from which ancestor class a variable is inherited, which supports a well-documented class.

The formal grammar for **DELTA**'s textual version of the class definition is:

```
ClassDeclaration : CLASS ClassName;
                   ClassDescription
                   ENDCLASS;
ClassDescription : AncestorStatement
                   ClassVariableStatement
                   InstanceVariableStatement
                   ClassMethodStatement
                   ObjectMethodStatement
AncestorStatement : ANCESTOR
                        ClassName;
ClassVariableStatement : CLASSVARIABLES
                              ClassVariableList
ClassVariableList : NONE | CVNameAncestorList
                           CVNameList
CVNameAncestorList : /* empty */
                   | $CVName INHERITEDFROM Ancestor;
                     CVNameAncestorList
CVNameList : /* empty */ | $CVName;
                           CVNameList
InstanceVariableStatement : INSTANCEVARIABLES
                                 InstanceVariableList
InstanceVariableList : NONE | IVNameAncestorList
                              IVNameList
IVNameAncestorList : /* empty */
                   | $IVName INHERITEDFROM Ancestor;
                     IVNameAncestorList
```

```
IVNameList : /* empty */ | IVName;
                           IVNameList
```

4.1.3.2.2 Type of Method

The next increment in **DELTA** provides an annotated representation for the textural view of the Class and Object Methods. Part II of the basic outline (Figure 5) corresponds to Level III of **DELTA** described in section 4.1.2.4. The grammar of **DELTA** for Level III follows:

```
ClassMethodStatement : CLASSMETHODS
                           ClassMethodList
ClassMethodList : NONE | CMNameList
CMNameList : /* empty */ | CMName (ParameterList)
                           RETURNTYPE type;
                           CMNameList
ObjectMethodStatement : OBJECTMETHODS
ObjectMethodList : NONE | OMNameList
OMNameList : /* empty */ | OMName (ParameterList)
                           RETURNTYPE type;
                           OMNameList
```

This part of the grammar represents a top-down approach to defining methods. The next increment of **DELTA** allows a developer to use a textual representation to distinguish between Class and Object methods and provides a documented form of the Class and Object methods.

4.1.3.3 Method Declaration

The next part of **DELTA**'s grammar can be used as the next increment in the top-down approach to defining a

method. This part of the grammar corresponds to Level IV as described in section 4.1.2.5. By distinguishing between a class or object method, and between arguments or local variables, verification can be performed on the information.

```
ClassMethodDefinition : DEFINE CLASSMETHOD
                        CMName (ParameterList)
                        RETURNTYPE type;
                        StatementBlock
ObjectMethodDefinition : DEFINE OBJECTMETHOD
                         OMName (ParameterList)
                         RETURNTYPE type;
                         StatementBlock
ParameterList : self | self,
                       ArgumentList;
                       LocalVariableList
ArgumentList : ARGUMENTS ArgNameList
ArgNameList : NONE | ArgName ANList
ANList : /* empty */ | , ArgName ANList
LocalVariableList : LOCALVARIABLES LVNameList
LVNameList : NONE | LVName LVNList
LVNList : /* empty */ | ,LVName LVNList
```

Note: StatementBlock is defined in Section 5.1.1.

4.2 Implementation of DELTA

4.2.1 DELTA System Requirements

A prototype of **DELTA** is written in Actor 4.1 and runs under Microsoft Windows version 3.0 or above. **DELTA** consists of 2 files, DELTAWIN.EXE and DELTAWIN.IMA. DELTAWIN.EXE has a file size of 391626 bytes and can be run by choosing **File Run** in the Windows Program Manager. DELTAWIN.IMA has a file size of 482806 bytes and provides the **DELTA** source code.

4.2.2 DELTA System Testing

System testing of **DELTA** was performed at each of the Levels, I through IV. Each method was tested as it was implemented and added to the **DELTA** system. White box testing, which assures internal operations perform according to specification, was performed on each method in a level. Black box testing, which demonstrates that the functionality of the software is operational, was performed on each level before the next level was implemented. White and black box testing was then performed in the same manner on each succeeding level. Finally, black box testing was performed on the entire system.

Test cases for **DELTA** were selected from published object-oriented development examples. The first case was an ATM system [Wir90]. The second case was a visual database interface in which Actor was the implementation language used for the project [Pin90]. The third case was a home heating system implemented in Smalltalk [Boo91]. The

fourth case was a digital circuit simulation also implemented in Smalltalk [Bou92]. The fifth case was a business bar chart system implemented in Actor [Whi92a].

4.2.3 An Example DELTA Design Specification

An illustration of an application of **DELTA** as a design representation language is given below. The system engineering and software requirements analysis which precede the design phase are assumed to have been completed.

The following analysis taken from [Whi92a] demonstrates the application of object-oriented logic to program design. The task is to create an application that can draw business charts. The specifications of the system are given as follows:

- A user should be able to choose between two chart types --horizontal bar or vertical bar--and switch the chart type at any time.
- The program's main window must have standard menus for creating a chart and choosing the chart type. For new charts, the user must be able to use dialogs for entering data and label values.
- Each chart must automatically adjust its size to fit the data and the window size.

The design must produce an application that has a main window, two different kinds of charts, and a dialog box for entering data and label values. The objects, their functionality, and data are identified as follows:

OBJECT	FUNCTIONALITY	DATA
main window	act as a standard window	chart
	display the chart	
	redraw as necessary	
	provide menu choices and dialogs	
	select different charts	
chart (vertical or horizontal)	draw itself	data
	scale if necessary	labels
		scale
		area
		space

A representation of the design of this system using **DELTA** is illustrated in the following sections.

4.2.3.1 Initialization

DELTA initially displays a window with a popup menu as shown in Figure 6. The user selects **New...** to start a new **DELTA** application. The dialog box shown in Figure 7 prompts the user for a filename. **DELTA** then automatically goes to Level I.

4.2.3.2 Level I

Level I in **DELTA** displays the window shown in Figure 8 with the given popup menu. The user selects **New...** and the dialog box shown in Figure 9 appears. The user enters the number of classes in the system, for example 6. The dialog box shown in Figure 10 appears which prompts the user for the hierarchy level of the first class. The user

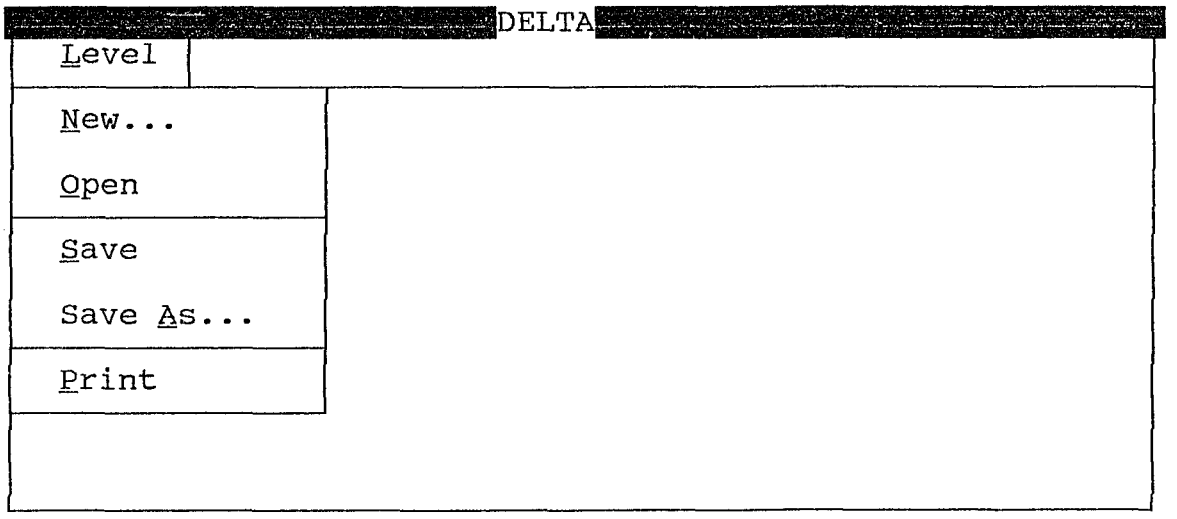

Figure 6 - **DELTA** initialization window

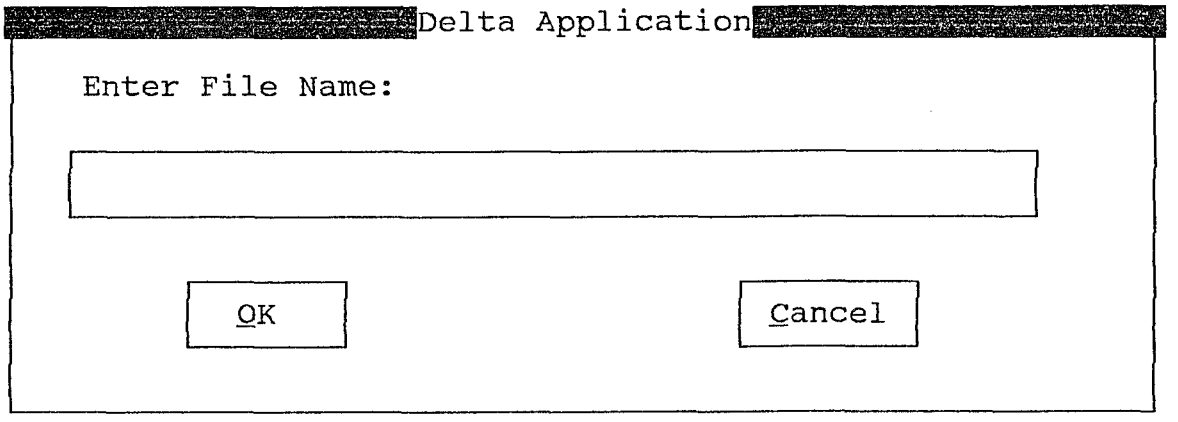

Figure 7 - **DELTA** file name dialog box

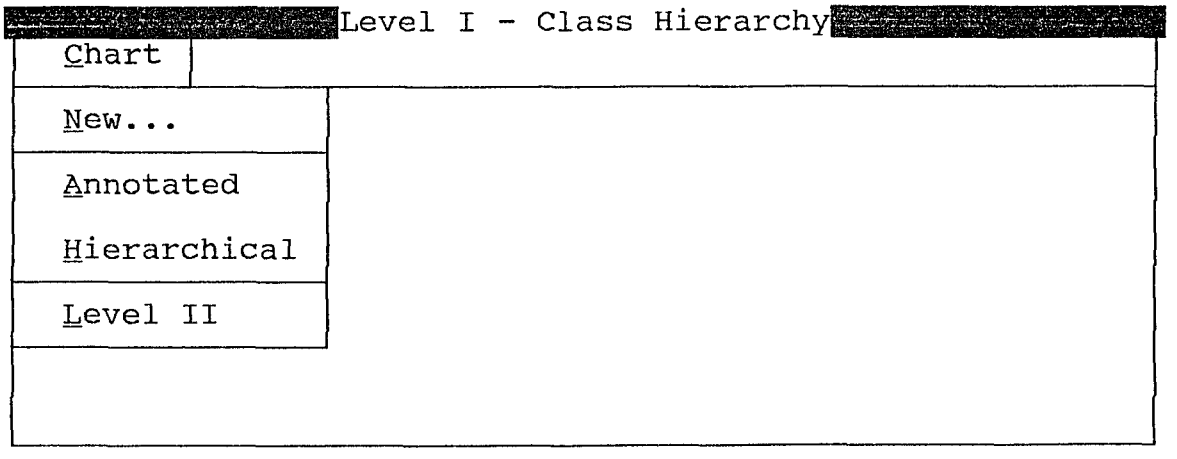

Figure 8 - **DELTA** Level I Window

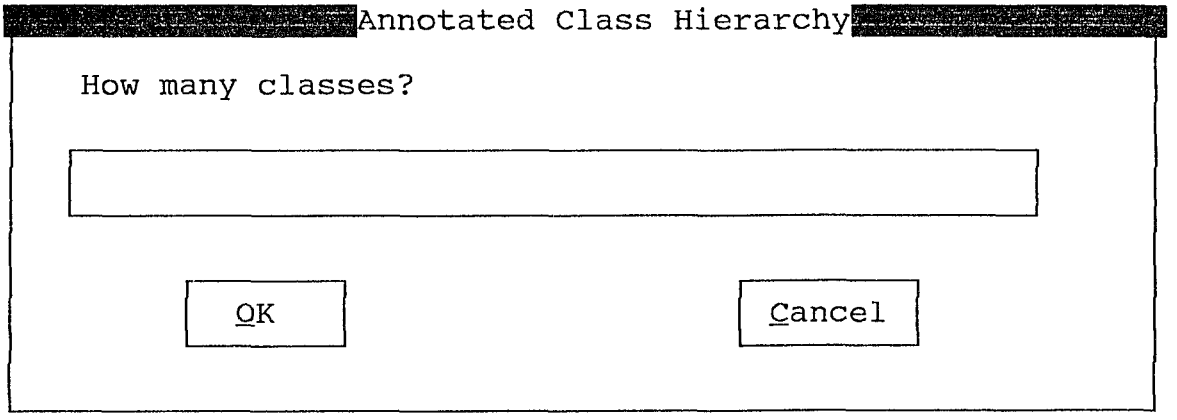

Figure 9 - **DELTA** Level I dialog box

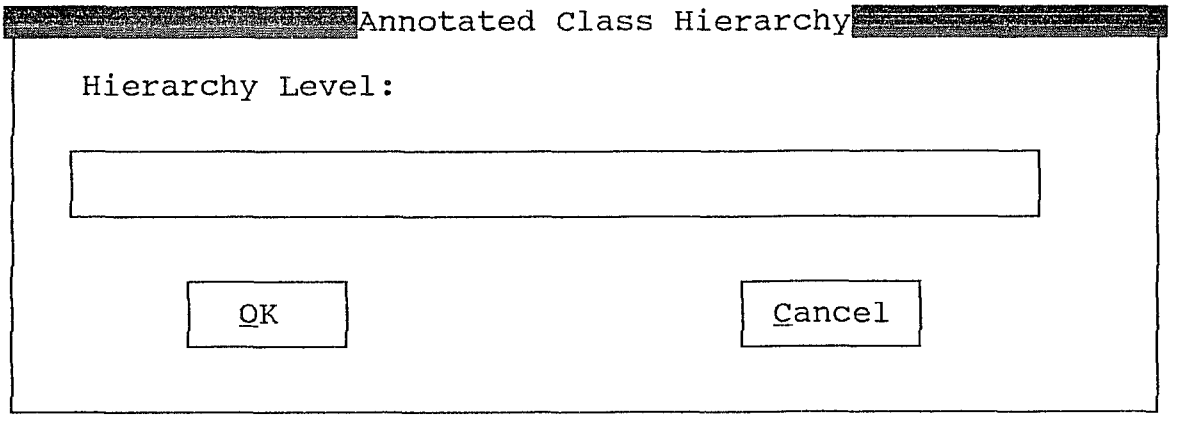

Figure 10 - **DELTA** Level I dialog box

enters the level in the class hierarchy for the first class, for example 1. The dialog box shown in Figure 11 appears which prompts the user for the class name. The user enters the name of the first class, for example Windows-Object. The dialog box shown in Figure 12 appears which prompts the user for the ancestor of the first class. The user enters the ancestor of the first class, for example Object. The dialog then repeats the last 3 boxes for the second through sixth classes. Upon entry of the last class, the annotated class hierarchy is constructed from the responses entered by the user and is displayed.

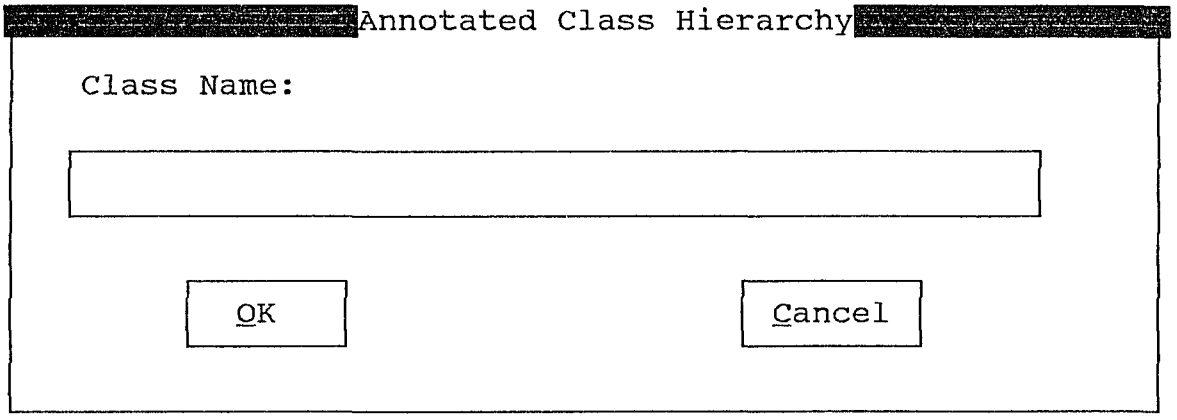

Figure 11 - **DELTA** Level I dialog box

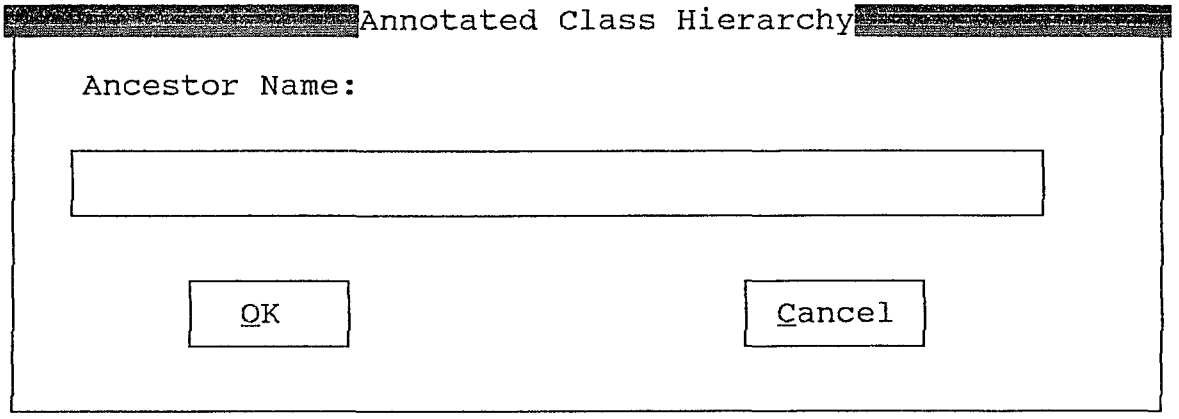

Figure 12 - **DELTA** Level I dialog box

Taking the classes selected from the bar chart exercise, the following represents the annotated text constructed from the responses entered by the user.

Annotated Class Hierarchy

Object

LEVEL [1] CLASS Windows-Object ANCESTOR Object

LEVEL [2] CLASS Window ANCESTOR Windows-Object

LEVEL [3] CLASS BCWindow ANCESTOR Window

LEVEL [1] CLASS BChart ANCESTOR Object

LEVEL [2] CLASS HBChart ANCESTOR BChart

LEVEL [2] CLASS VBChart ANCESTOR BChart

The user may now select **Chart** followed by **Hierarchical** to view the hierarchical class diagram as shown in Figure 13. The user can return to the annotated text by selecting **Chart** followed by **Annotated.** The user can enter the next incremental phase of the design by selecting **Chart** followed by **Level II.**

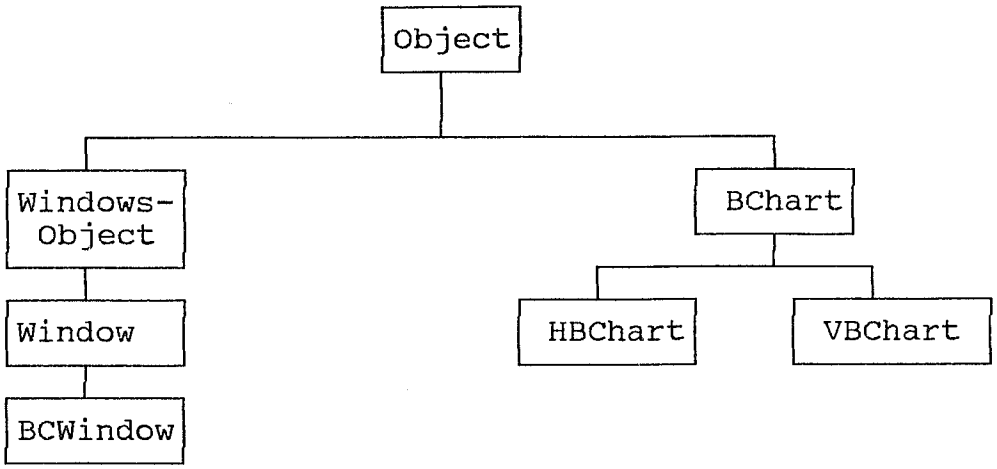

Figure 13 - **DELTA** Level I Class Hierarchy

4.2.3.3 Level II

The next increment of design representation is the class definition. Level II in **DELTA** displays the window shown in Figure 14 with the given popup menu. The user selects **New...** and the dialog box shown in Figure 15 appears. The user enters the name of a class, such as BCWindow. The dialog box shown in Figure 16 appears which prompts the user for the ancestor of the class. The user enters the class' ancestor, for example Window. The dialog box shown in Figure 17 appears which prompts the user for the number of class variables. If the user enters a number

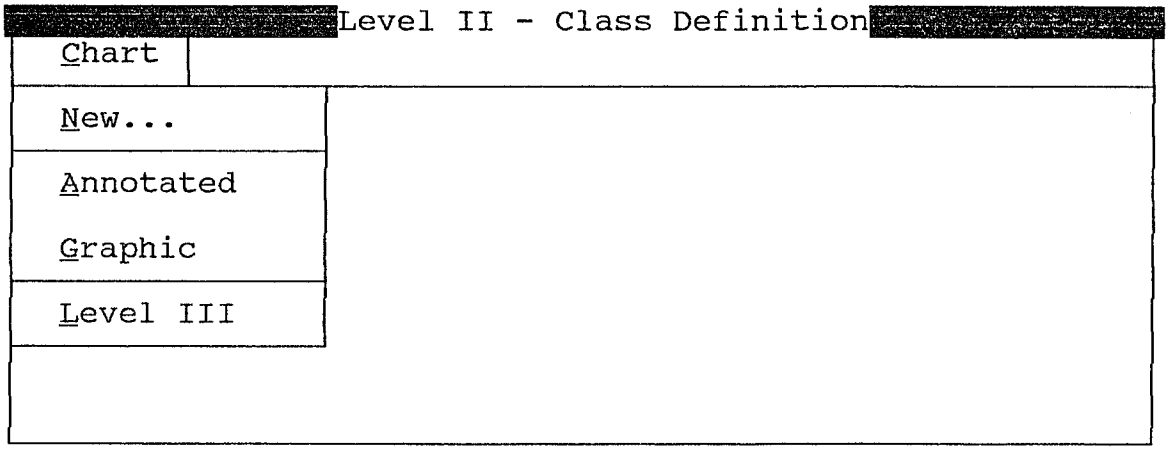

Figure 14 - **DELTA** Level II Window

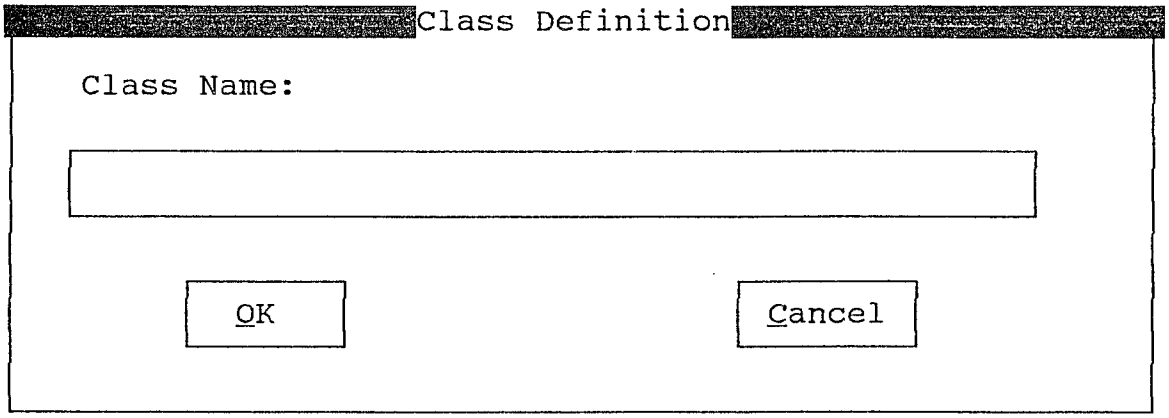

Figure 15 - **DELTA** Level II dialog box

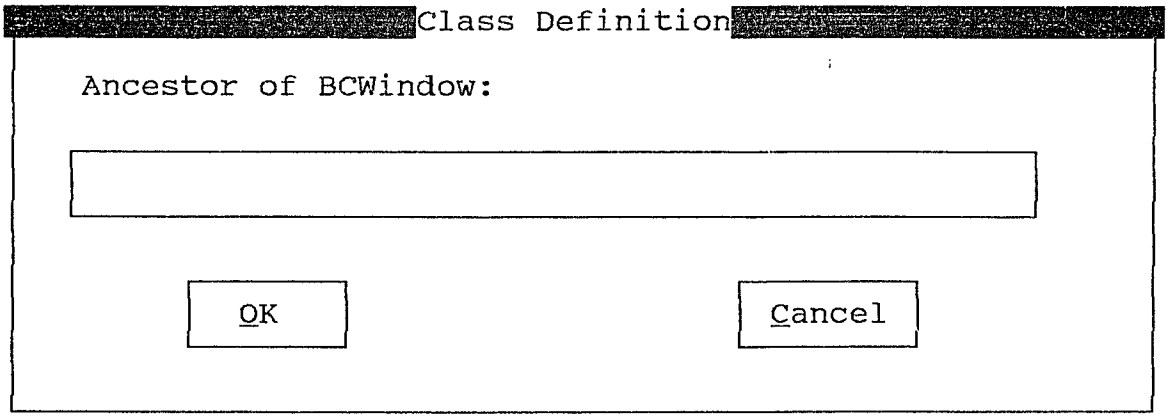

Figure 16 - **DELTA** Level II dialog box

> 0, then the dialog box shown in Figure 18 appears. After completing entry of all class variable names, the dialog

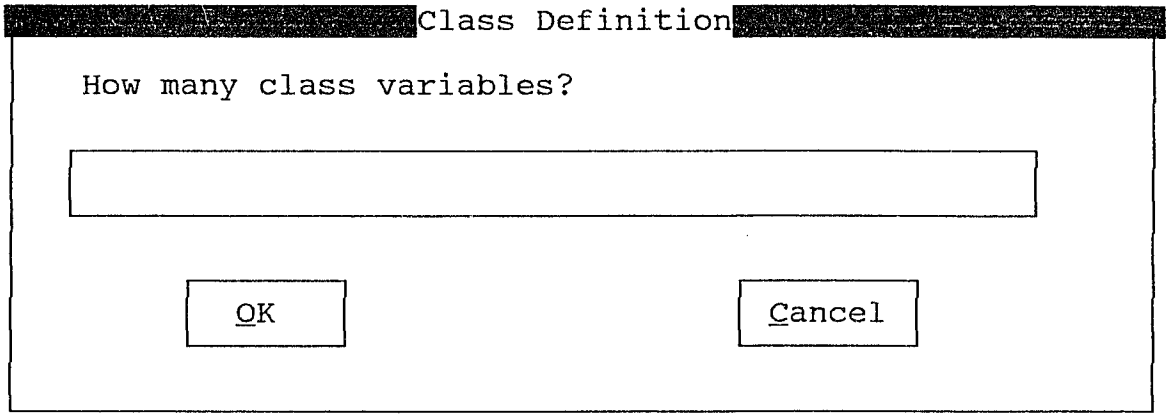

Figure 17 - **DELTA** Level II dialog box

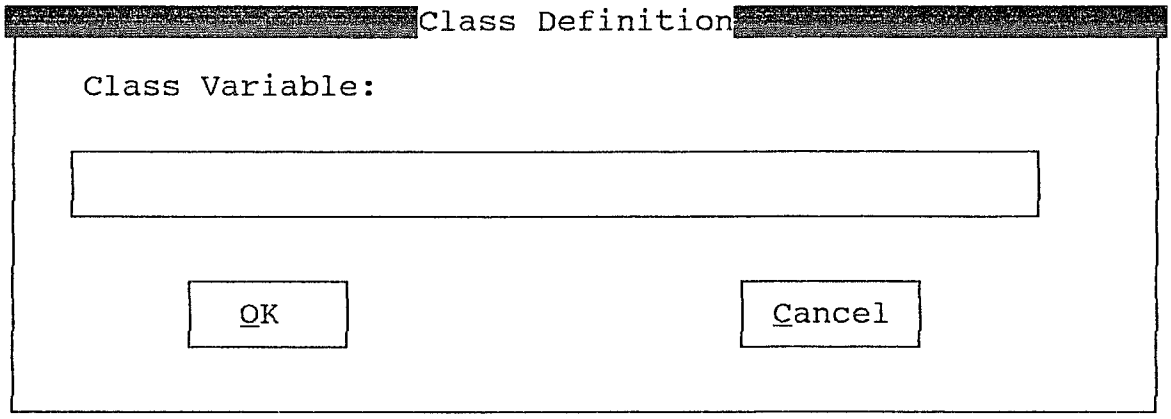

Figure 18 - **DELTA** Level II dialog box

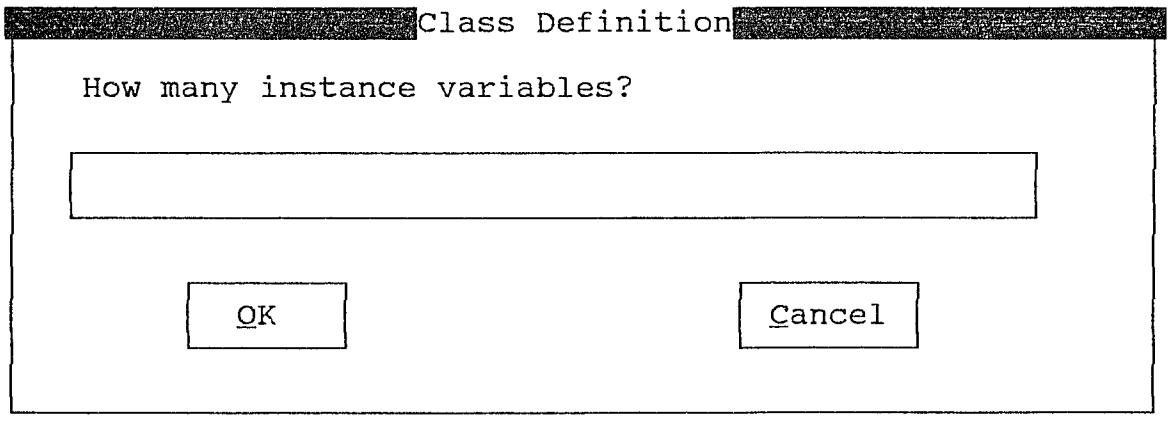

Figure 19 - **DELTA** Level II dialog box

box in Figure 19 appears which prompts the user for the number of instance variables. If the user enters a number > 0, then the dialog box shown in Figure 20 appears. After

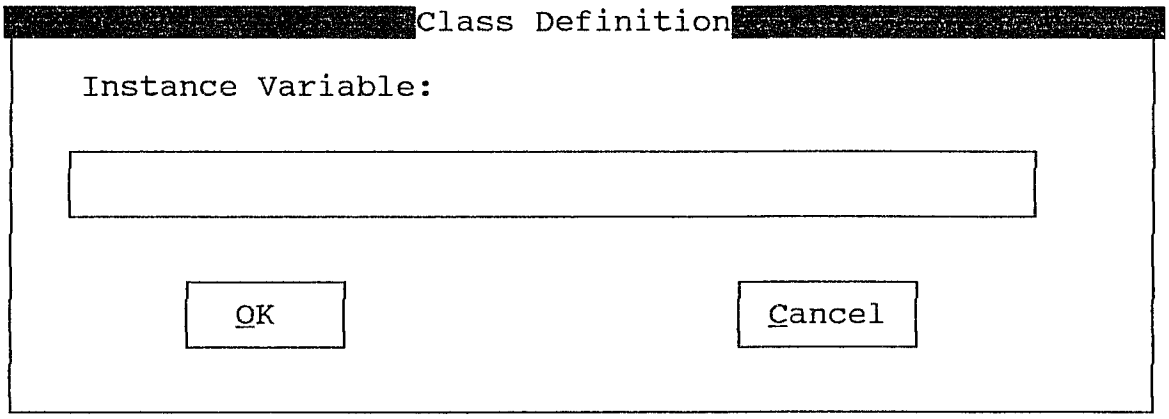

Figure 20 - **DELTA** Level II dialog box

completing entry of all instance variable names, annotated text is constructed from the responses entered by the user, for example:

```
CLASS BCWindow;
ANCESTOR Window;
INSTANCEVARIABLES
    Chart;
```

The user may now select **Chart** followed by **Graphic** to view the graphical class definition as shown in Figure 21. The user can return to the annotated text by selecting **Chart** followed by **Annotated.** The user can enter the

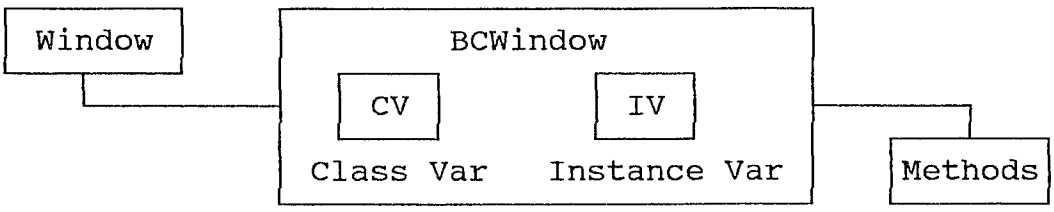

Figure 21 - **DELTA** Level II Graphic

next incremental phase of the design by selecting **Chart** followed by Level III.

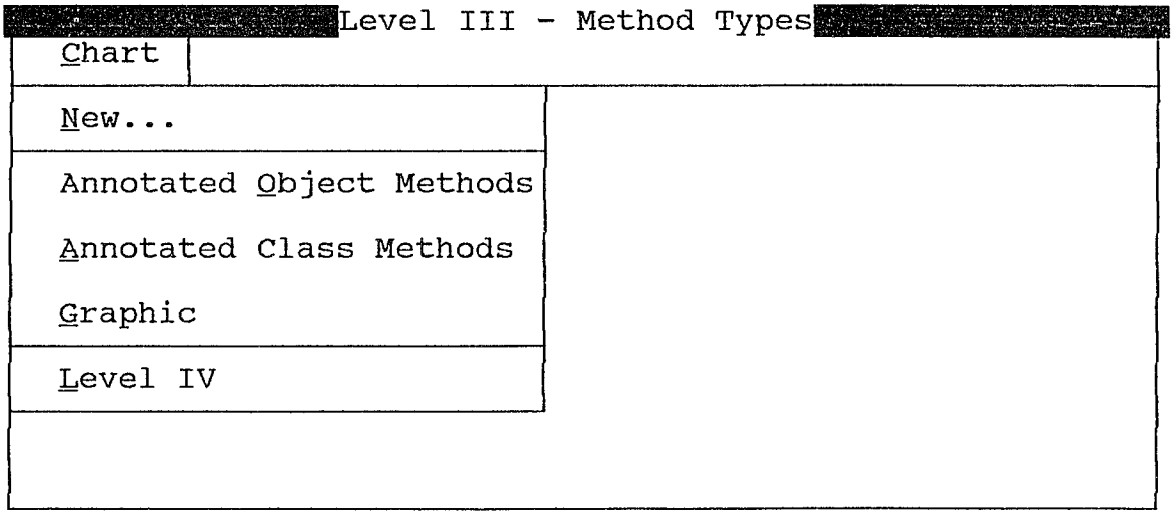

Figure 22 - **DELTA** Level III Window

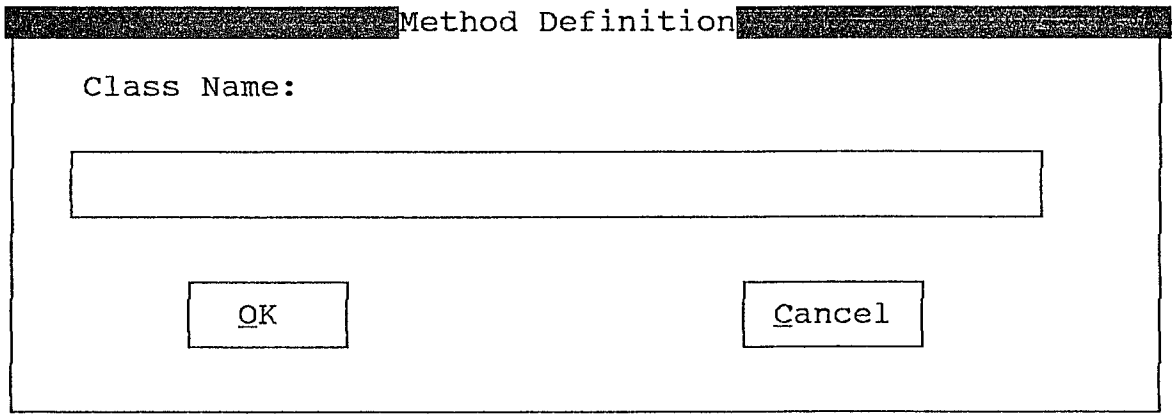

Figure 23 - **DELTA** Level III dialog box

4.2.3.4 Level III

This level accommodates the next incremental phase in the **DELTA** design process which is to name the methods for a class. Level III in **DELTA** displays the window shown in Figure 22 with the given popup menu. The user selects **New...** and the dialog box shown in Figure 23 appears. The user enters the name of a class, such as BCWindow. The dialog box shown in Figure 24 appears which prompts the user for the number of object methods. If the user enters

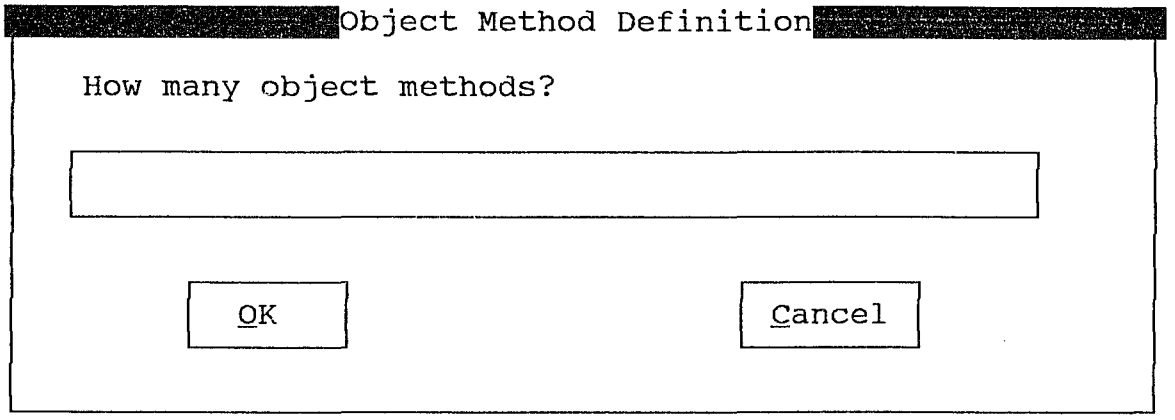

Figure 24 - **DELTA** Level III dialog box

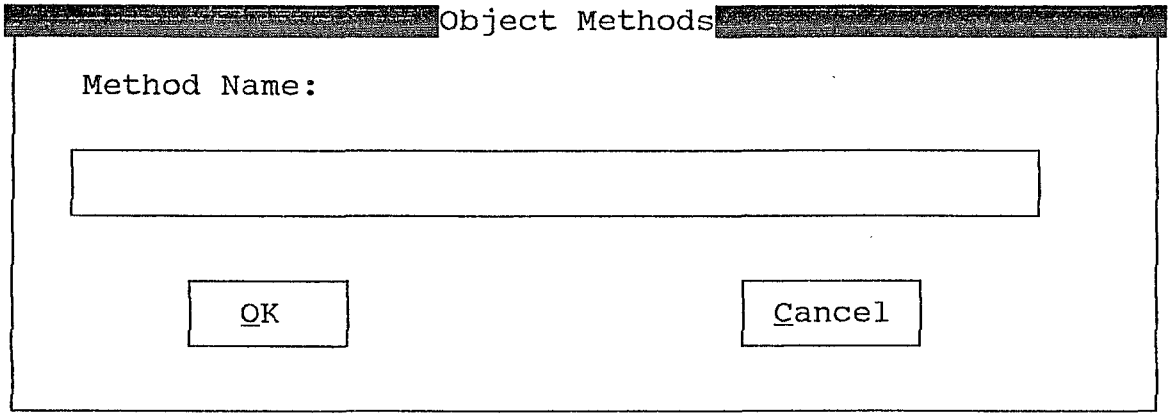

Figure 25 - **DELTA** Level III dialog box

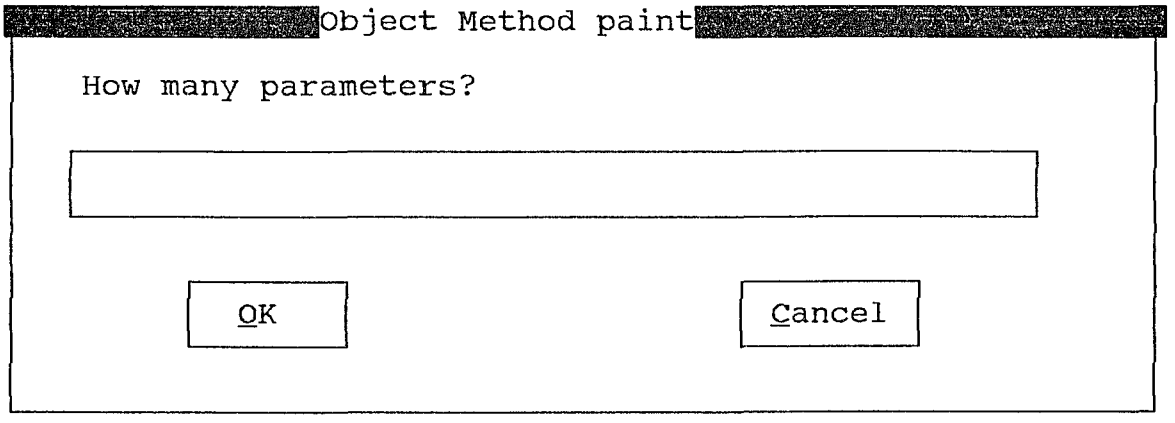

Figure 26 - **DELTA** Level III dialog box

a number > 0, then the dialog box shown in Figure 25 appears which prompts the user for the method name, for example paint. The dialog box in Figure 26 appears which

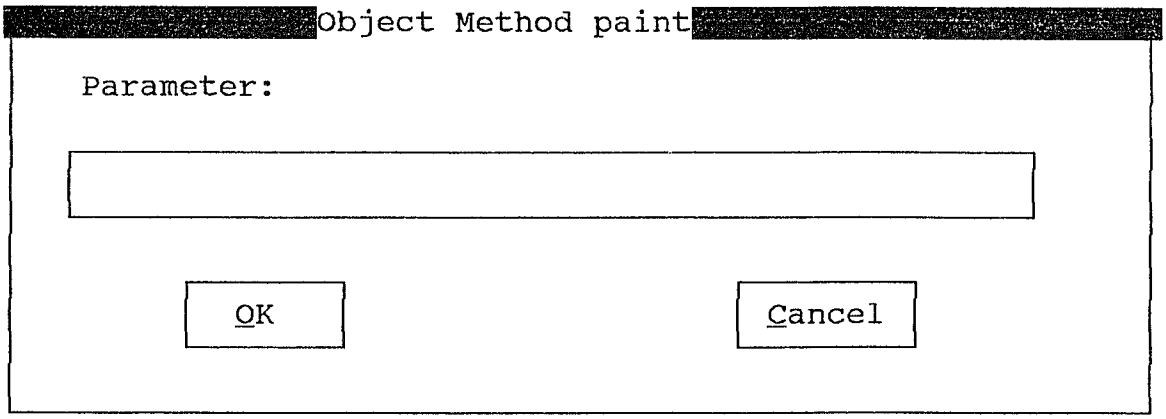

Figure 27 - **DELTA** Level III dialog box

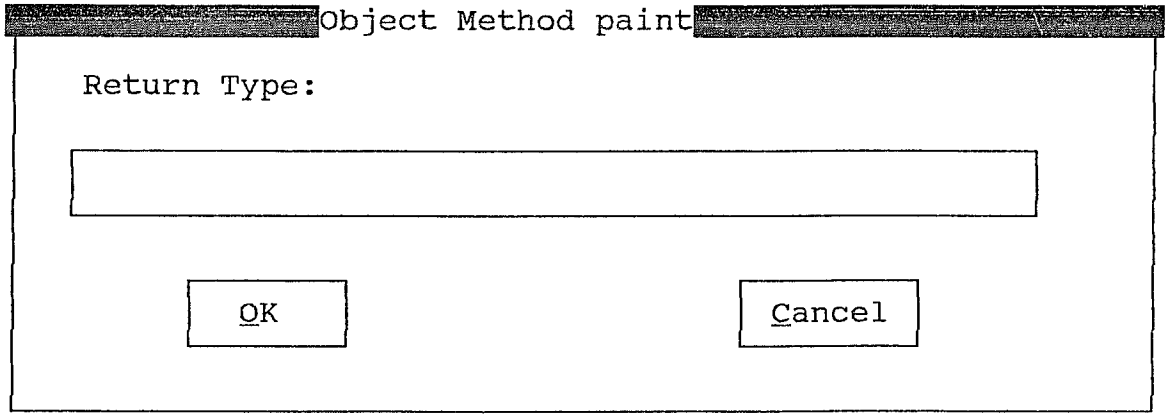

Figure 28 - **DELTA** Level III dialog box

prompts the user for the number of parameters. If the user enters a number > 0, then the dialog box shown in Figure 27 appears which prompts the user for the parameter. After entering each parameter, the dialog box in Figure 28 appears which prompts the user for the return type of the method. The user enters the return type, such as self. The dialog box shown in Figure 29 appears which prompts the user for the number of class methods. If the user enters a number > 0, then a similar set of dialog boxes appear to obtain the method name, number of parameters, each parameter's name, and the method's return type. Upon entry of the last

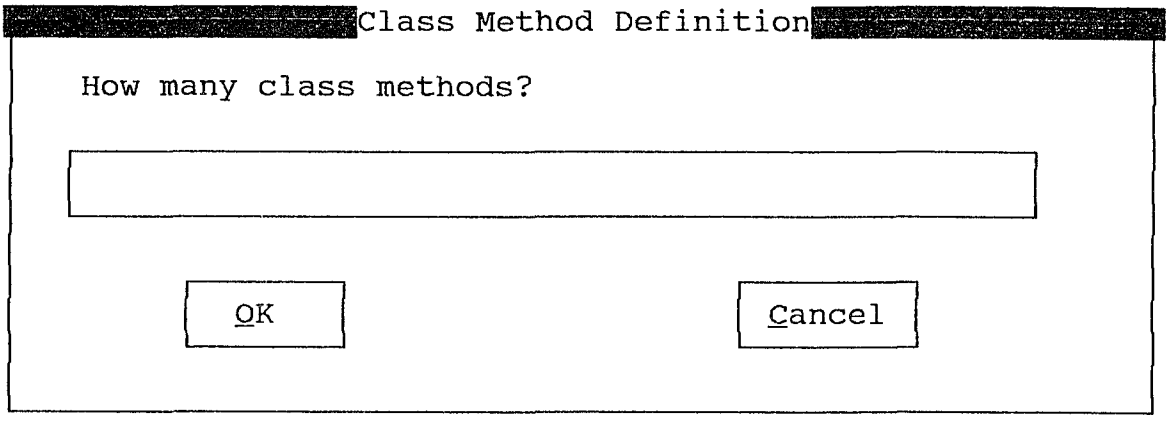

Figure 29 - **DELTA** Level III dialog box

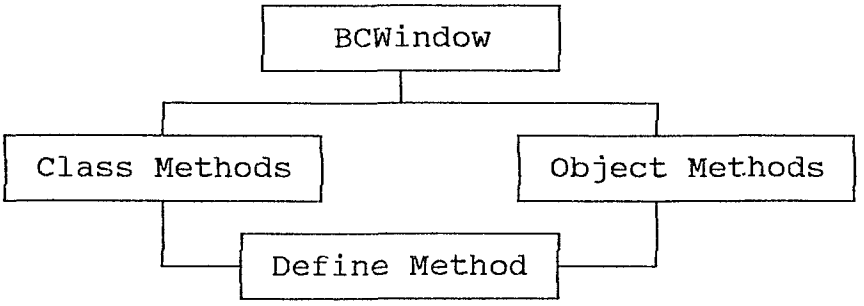

Figure 30 - **DELTA** Level III Graphic

method, the graphical view shown in Figure 30 appears. The user may now select **Chart** followed by **Annotated Object Methods** or **Annotated Class Methods** to view the corresponding annotated text. For example, after the object methods **paint** and **setChart** have been entered, **Annotated Object Methods** has the following textual display.

```
OBJECTMETHODS
    paint (self, context) RETURNTYPE self;
    setChart (self, context) RETURNTYPE self;
```

In order to define a method, the user can select **Chart** followed by **Level IV** on the menu and this action takes the user to the next incremental level.

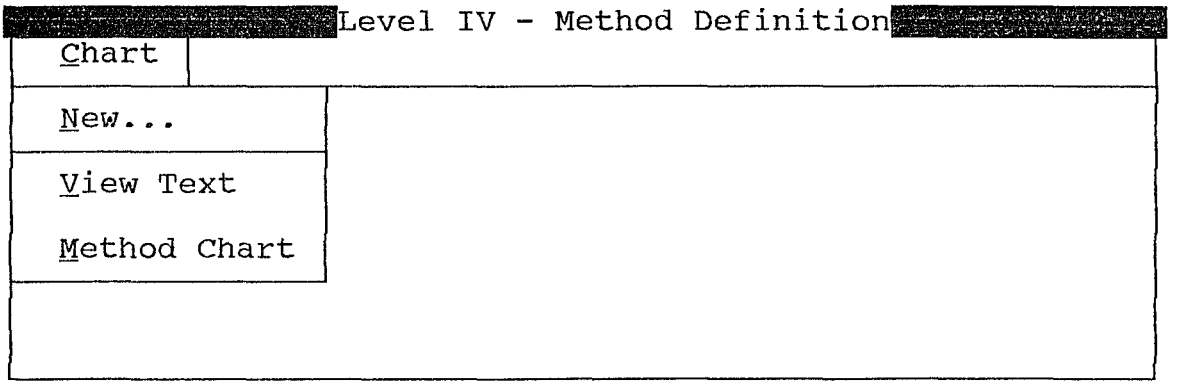

Figure 31 - **DELTA** Level IV Window

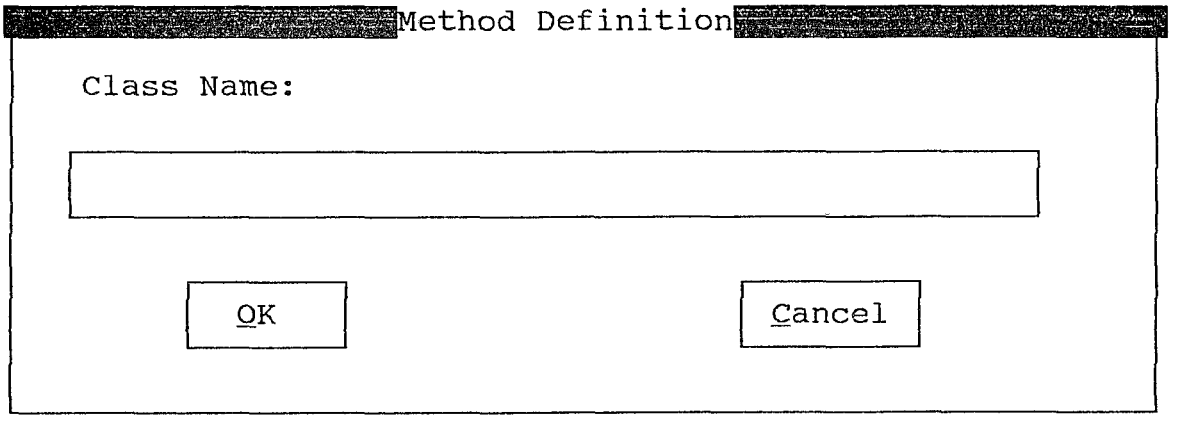

Figure 32 - **DELTA** Level IV dialog box

4.2.3.5 Level IV

The final increment of representing the design is the detailed representation of a method. Level IV in **DELTA** displays the window shown in Figure 31 with the given popup menu. the user selects **New...** and the dialog box shown in Figure 32 appears. The user enters the name of the class, such as BCWindow. the dialog box shown in Figure 33 appears which prompts the user for the name of the method. The user enters the method name, such as paint. The dialog box shown in Figure 34 appears which prompts the user for the type of method. The user enters either object or class

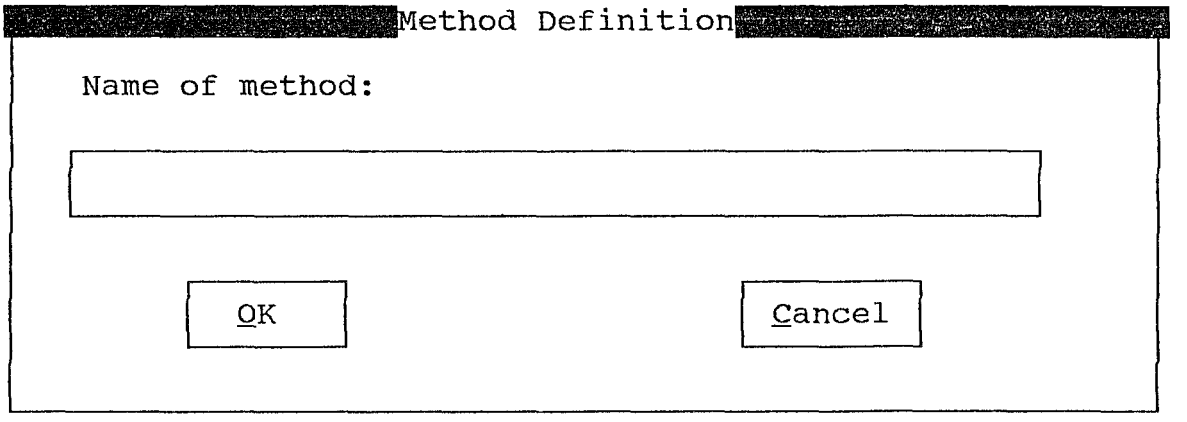

Figure 33 - **DELTA** Level IV dialog box

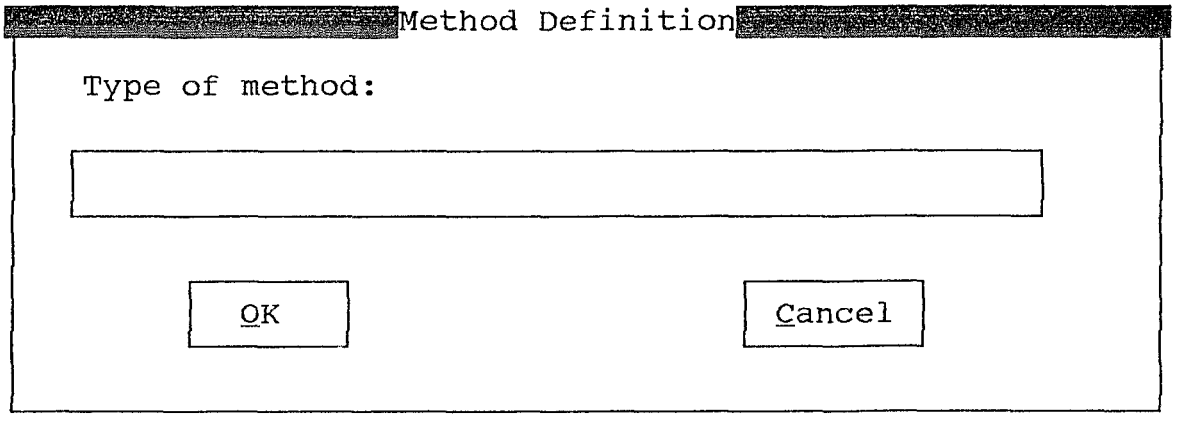

Figure 34 - **DELTA** Level IV dialog box

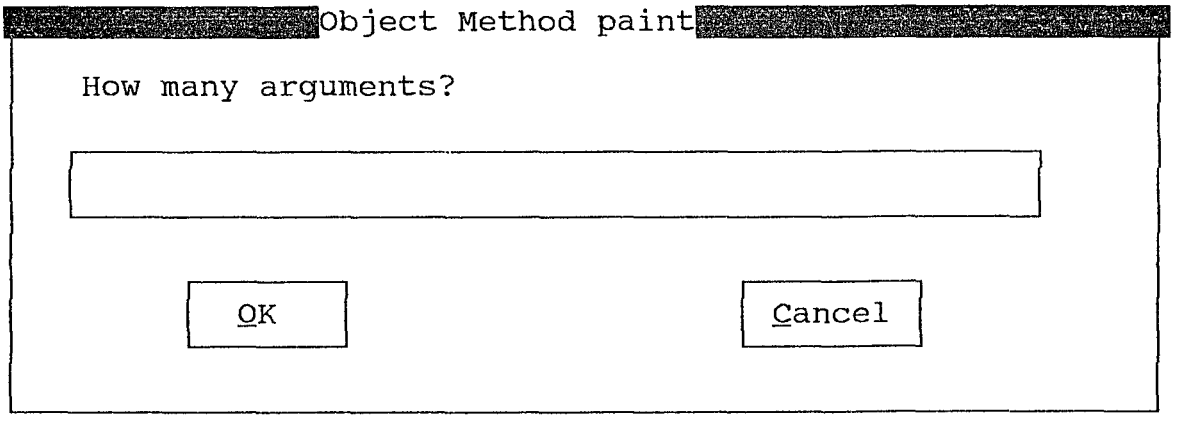

Figure 35 - **DELTA** Level IV dialog box

and the dialog box shown in Figure 35 appears which prompts the user for the number of arguments. If the user enters a number > 0, then the dialog box shown in Figure 36 appears

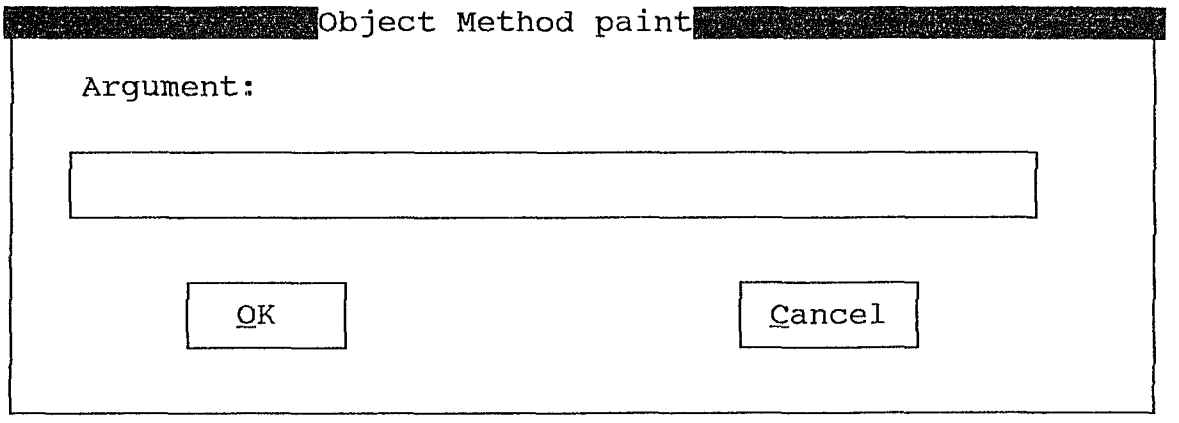

Figure 36 - **DELTA** Level IV dialog box

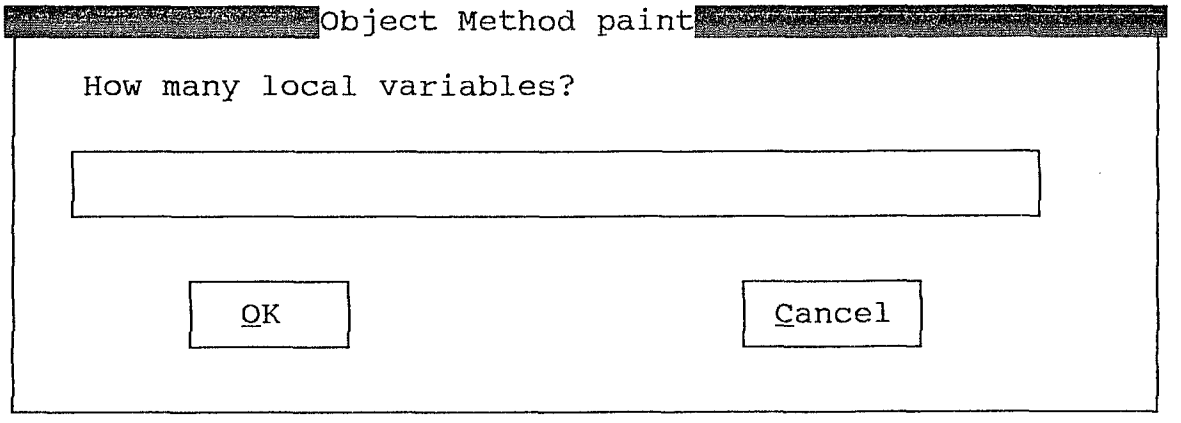

Figure 37 - **DELTA** Level IV dialog box

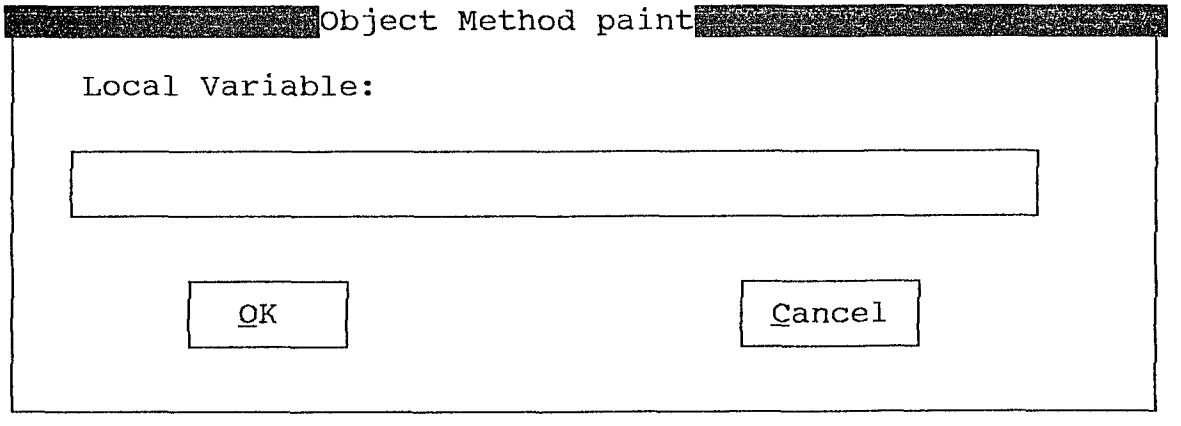

Figure 38 - **DELTA** Level IV dialog box

which prompts the user for arguments. After completing entry of all arguments, the dialog box in Figure 37 appears which prompts the user for the number of local variables.

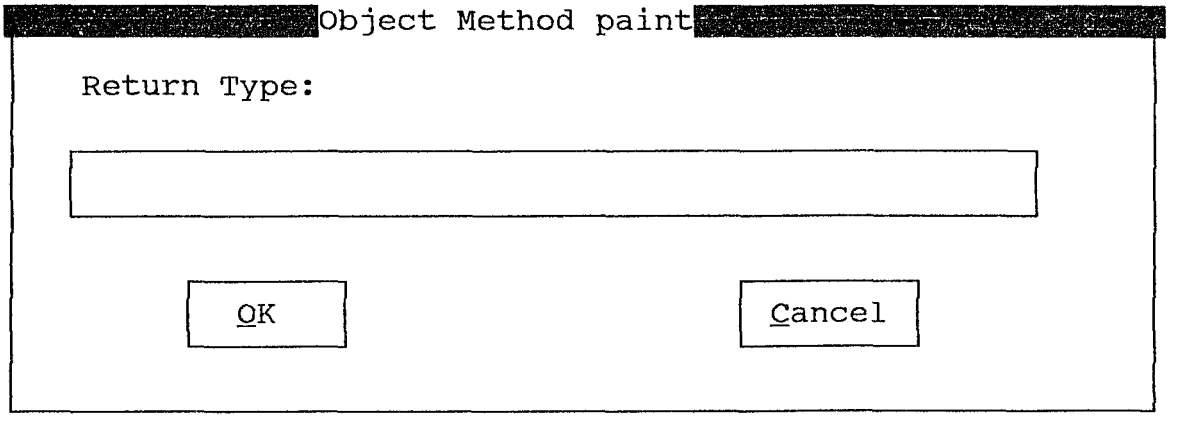

Figure 39 - **DELTA** Level IV dialog box

Object Method

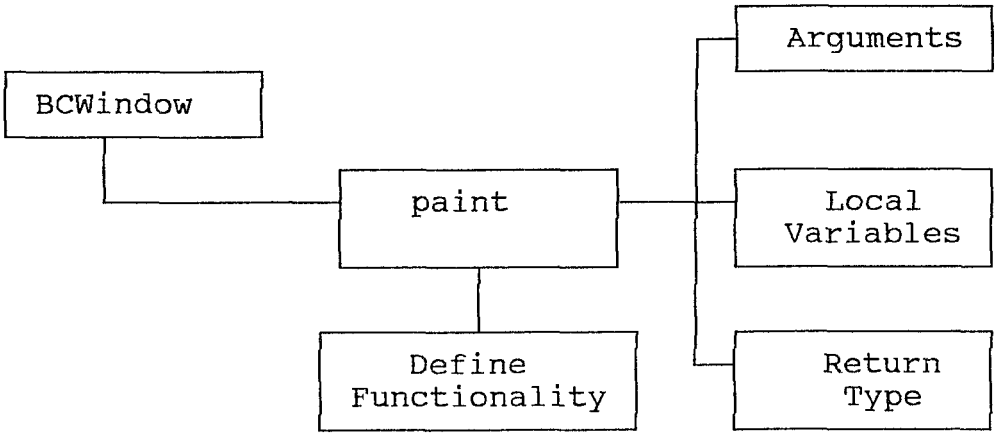

Figure 40 - **DELTA** Level IV Graphic

If the user enters a number > 0, then the dialog box in Figure 38 appears which prompts the user for local variables. After completing entry of all local variables, the dialog box in Figure 39 appears which prompts the user for the return type of the method. After the user enters a return type, the graphical view shown in Figure 40 appears. The user may now select **Chart** followed by **View Text** to view the corresponding annotated text. For example, after the object method paint has been entered, **View Text** has

the following textual display:

```
DEFINE OBJECTMETHOD paint
    ( ARGUMENTS self, context
    | LOCALVARIABLES NONE )
      RETURNTYPE self;
StatementBlock
```

The user may select one of the boxes on the right, **Arguments**, **Local Variables**, or **Return Type** for viewing/modifying existing attributes. The user can select **Define Functionality** to enter a description for the StatementBlock. A description of the **paint** method could be entered as follows:

```
/* if a chart exists, send a draw message to draw a
   chart in that context */
```

The user may now return to any level window to repeat the above steps for each class, variable, or method to be defined. A printout of the graphical and textual views of each level may be obtained by returning to the initial **DELTA** window and selecting **Level** followed by **Print.** This provides a well-documented design.

The formal design language **DELTA** provides an incremental design notation in a graphical/textual environment for representing a design specification. **DELTA**'s formal notation can aid a software developer to uncover errors, inconsistencies, or necessary changes in the design. **DELTA**'s environment accommodates the making of any modifications to the design representation in this phase, as well

as, needed design changes discovered in succeeding phases of software development. **DELTA**'s language independent design notation provides a detailed representation of a design specification which can now be mapped to the programming language in which the system will be prototyped or implemented.

5. LEVEL V: THE PROTOTYPING LAYER

The final layer of **DELTA** provides a means to support rapid-prototyping. Levels I through IV produce a language independent design representation. Level V becomes language dependent, as the design is mapped to a particular programming language. Actor was selected as the first implementation of **DELTA** to be used as a prototype. Even if a prototype in Actor is not desired, a **DELTA** design specification provides a representation that maps well to other object-oriented language implementations as the language supports basic object-oriented concepts.

The next increment of a top-down approach in the **DELTA** design process after completing Level IV is to annotate the functionality of a method. A formal notation is presented in section 5.1 for the **DELTA** annotations to Actor.

One of the design goals of **DELTA** is that it should provide support for prototyping. The grammar for **DELTA** was designed for seamless integration with the object-oriented language Actor. This integration provides for ease of prototyping. Mapping **DELTA** to Actor is illustrated in section 5.2.

Actor was chosen as the result of a comparison of pure and hybrid object-oriented languages [Gau93b]. The comparison is presented in section 5.3. The formal grammar of Actor is provided in the Appendix A.

5.1 Annotating Actor

Extending **DELTA**'s language independent layer of levels I - IV to a language dependent prototyping layer requires a notation which will allow **DELTA** annotations to be compiled in the chosen implementation language. Since Actor comment delimiters are /* and */, **DELTA** annotation delimiters are /*@ and @*/. This convention allows a developer or programmer a formal notation to quickly identify which program structures have been selected among Actor's various loops and selection statements. It is also used to clarify components of messages.

5.1.1 Method Definition

The last increment in the top-down approach of design representation is to define the methods. The next phase of system design is to develop a prototype. At this point, the user must be familiar with the chosen implementation language. Using the **DELTA** text produced from Level IV, a transition to prototyping in Actor can be performed using the corresponding annotated grammar which follows.

```
StatementBlock : { StatementList }
StatementList : /* empty */ | Statement; StatementList
Statement : Assignment | Conditional | Iteration | Message
            | Return
Assignment : Object /*@ ISASSIGNED @*/ AssignStmt;
AssignStmt : Assignment | Expression | StatementBlock
Expression : Literal | Message
Conditional : IfStmt | IfElseStmt | SelectStmt
```

```
Iteration : Enumeration | Loop
Enumeration : DoStmt | CollectStmt | ExtractStmt
Loop : StatementList /*@ ENDCONDITION @*/ Condition |
      /*@ INITIALCONDITION @*/ Condition StatementList
Return : /*@ RETURN @*/ ^Statement
SelectStmt : /*@ SELECTSTATEMENT @*/
             SELECT CaseStmtList
             DEFAULT StatementList
             ENDSELECT;
IfStmt : /*@ IFSTATEMENT @*/
         IF Condition THEN StatementList ENDIF;
IfElseStmt : /*@ IFELSESTATEMENT @*/
             IF Condition THEN
                StatementList
             ELSE
                StatementList
             ENDIF;
CaseStmt : /*@ CASESTATEMENT @*/
           CASE Condition IS
                StatementList
           ENDCASE;
```

5.1.2 Message Definition

Methods and messages provide the functionality in an object-oriented system. Actions are performed in an object-oriented system by sending a message to an object. The object selects a method to perform the corresponding action. This supports polymorphism, the ability for

different classes of objects to respond to the same messages which invoke different methods. Dynamic binding selects the code to be associated with a message at run-time. The cost and time of maintenance of adding an object to a system is reduced. The new object can respond to existing messages in the system simply by defining its own method. If an object does not have a defined method for a message, it searches the ancestors of its class. Once found, the method is invoked; otherwise, an error occurs. This is also one of the driving forces of developing reusable code. One can develop generic messages which can be matched with any existing or future method of the same name. The format of a message in Actor is

```
messageName (receiver, arg1, arg2, ...);
```

A message can appear as a statement in a method definition. The final section of **DELTA**'s grammar provides this annotated version of a message.

```
/*@ SELECTOR @*/ MessageName
                (/*@ RECEIVER @*/ Receiver
                /*@ ARGUMENTS @*/ ArgumentList);
```

The annotations can be used to improve the readability of the message. The grammar follows:

```
Message : /*@ SELECTOR @*/ MessageName
          (/*@ RECEIVER @*/ Receiver
           /*@ ARGUMENTS @*/ ArgumentList);
Receiver : /*@ NONE @*/ | ObjectName
MessageArgumentList : /*@ NONE @*/ | ,MsgArgName MANList
MANList : /* empty */ | ,MsgArgName MANList
```

Messages in Actor which have no receiver are system messages which perform functions such as memory management, obtaining information about the programming environment such as the screen size, or getting the day/time from the system clock. Sending messages to the class System can be used instead of omitting a receiver.

5.2 Mapping DELTA to Actor

Actor, developed by the Whitewater Group in 1985, runs as an MS-Windows application [Whi92b]. Windows applications, such as a database or a spreadsheet, run using MS-Windows Software Development Kit routines. These Kit routines perform various windowing functions such as displaying, resizing, or scrolling. Windows applications are more complex to develop than simple character-based systems which are driven by textual commands. Since most languages do not mask the complexity of the operating system, a great cost can be spent on training software developers to master the intricacies of developing windows applications. However, since Actor is itself a Windows application, every Kit routine can be invoked from an Actor program. Actor taps the power of encapsulation by hiding the details of these low-level Kit routines within predefined classes [Pin90]. Actor has a standard class library which a software developer can master incrementally, starting with simple window functions and working up to sophisticated graphical applications.

Actor, although similar to Smalltalk which is also a complete development environment, has a syntax unlike its predecessor. The syntax of Actor is much more similar to the common procedural languages of Pascal and C. Actor allows both static binding and the less efficient dynamic binding, which taps the potential of inheritance and helps to accommodate rapid-prototyping [Pin90].

The graphical view of a **DELTA** design supports the object hierarchy and object components available in Actor. The textual view of a **DELTA** design provides documentation which can easily be transferred to Actor class dialogs. The relationship between the Actor environment and a **DELTA** design for each of the Levels I - IV is illustrated in the following sections.

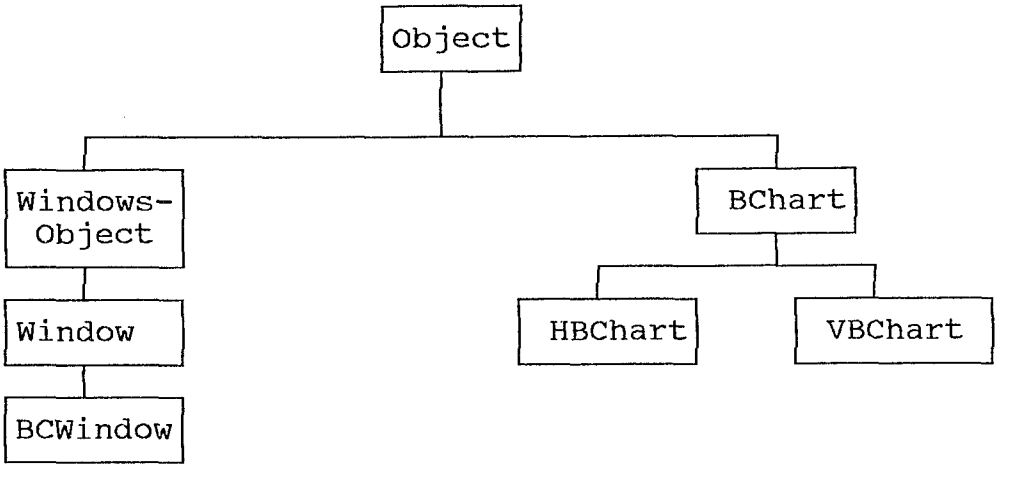

Figure 41 - **DELTA** Level I Class Hierarchy

5.2.1 Level I

The example of **DELTA**'s Level I class hierarchy produced in section 4.2.3.2, shown in Figure 41, can be used

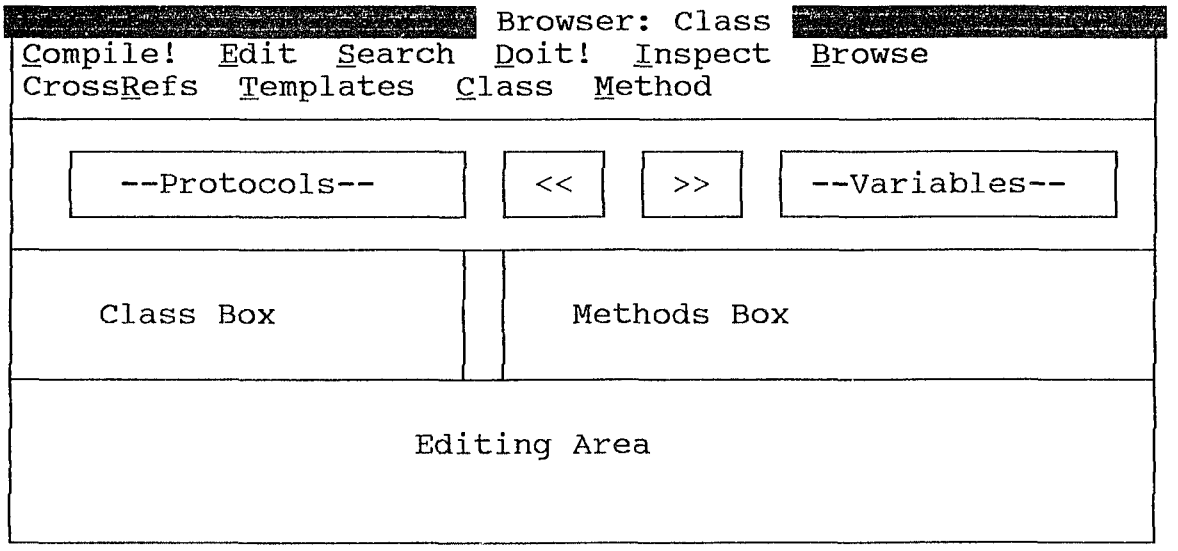

Figure 42 - Actor's **Browser** Window

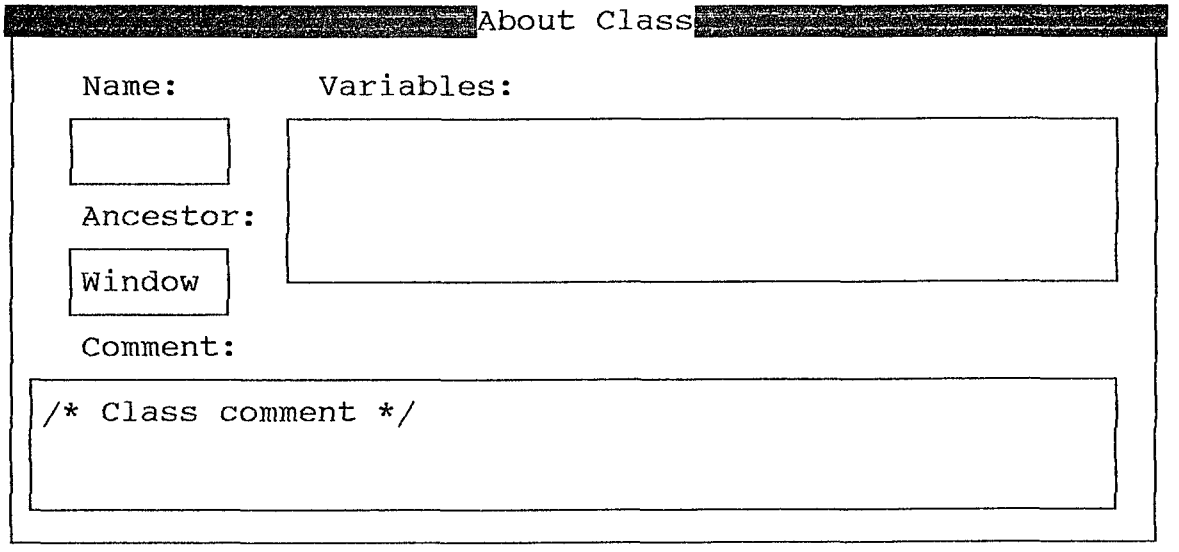

Figure 43 - Actor's About Class dialog box

to set up a hierarchy in Actor. A user first selects a class, such as **Window**, in the class box of Actor's **Browser** window shown in Figure 42. **Class** is then selected on the **Browser** menu followed by **Make descendant** to set up a class hierarchy in Actor. This action produces Actor's About Class dialog box shown in Figure 43.

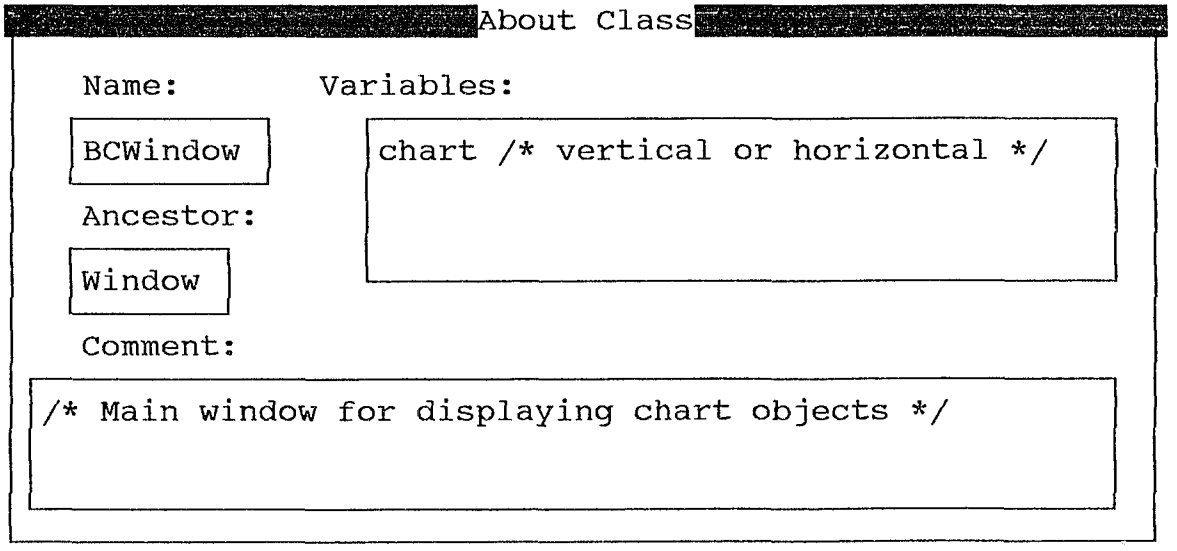

Figure 44 - Actor's About Class dialog box

5.2.2 Level II

DELTA's textual version of the class definition in Level II supports the Actor notation by including the text supplied in the About Class dialog box of Figure 43. A software developer can use the **DELTA** grammar to represent a documented form of the information provided in the Actor window.

Using the following example **DELTA** design specification text produced in 4.2.3.3:

```
CLASS BCWindow;
ANCESTOR Window;
INSTANCEVARIABLES
    Chart;
```

DELTA's text from Level II can be mapped to Actor's About Class dialog box as shown in Figure 44. The user clicks **Accept** to store the information in Actor's class hierarchy. **BCWindow** is now displayed in the Class Box of the **Browser** window (Figure 42).

5.2.3 Level III

The next increment in **DELTA**, Level III, provides a design representation for naming methods and distinguishing amongst Class and Object Methods. It also provides a documented form of the Class and Object methods which are displayed in Actor's **Browser** window's Methods Box of Figure 42.

Using the following example **DELTA** design specification text produced as part of **BCWindow** class in 4.2.3.4:

```
OBJECTMETHODS
        paint (self, context) RETURNTYPE self;
        setChart (self,context) RETURNTYPE self;
```

DELTA's text from Level III can be mapped to the Actor environment as follows. The user selects **BCWindow** in the Class Box on the **Browser** window of Figure 42. The user selects **Method** on the **Browser** menu followed by **Object Method.** This sets the default method type to Object for any new methods which are defined in Actor and displays the names of Object methods in the Methods Box of Figure 42.

5.2.4 Level IV

The final increment in a **DELTA** design specification is the method definition. The **DELTA** text produced from Level IV can be extended to include the **DELTA** annotations for Actor methods and messages as described in section 5.1. **DELTA**'s Level IV text can be used to annotate Actor's Def statement. Differences between **DELTA** and Actor are indicated in the delimiters /*@ and @*/. The **DELTA**

annotations provide clarification of methods, variables, or statements. By distinguishing between a class or object method, and between arguments or local variables, a double check can be performed on the information.

Using the following example **DELTA** design specification text produced as part of **BCWindow** class for object method **paint** in section 4.2.3.5:

```
/* if a chart exists, send a draw message
   to draw a chart in that context */
   DEFINE OBJECTMETHOD paint
     (ARGUMENTS self, context
     | LOCALVARIABLES NONE )
     RETURNTYPE self;
     StatementBlock
```

DELTA's text from Level IV can be mapped to the Actor environment as follows. Assuming **BCWindow** is the selected class and **Object** is the default method type, the user selects **Templates** on the **Browser** menu followed by **New Method.** The following template appears in the Editing Area of Actor's **Browser** window of Figure 42.

```
/* comment */
Def method (self)
{ }
```

The user can now edit the template and enter the **DELTA** text as follows:

```
/* if a chart exists, send a draw message
   to draw a chart in that context */
```

```
/*@ DEFINE OBJECTMETHOD @*/
   Def paint
      (/*@ ARGUMENTS @*/ self, context
       /*@ LOCALVARIABLES NONE @*)
       /*@ RETURNTYPE self @*/
   { /*@ IFSTATEMENT @*/
     if chart then
        /*@ SELECTOR @*/ draw
        (/*@ RECEIVER @*/ chart,
         /*@ ARGUMENTS @*/ context );
     endif;
   }
```

A **paint** method automatically defines a context for displaying objects in Actor. In this example, if a chart exists, then the **paint** message sends a **draw** message to draw a chart in that context. The user selects **Compile!** on the **Browser** menu of Figure 42 and object method **paint** is added to the Methods Box after the method is successfully compiled.

The user now repeats the process of defining classes and methods in the Actor **Browser.** Methods of the prototype may be tested by typing messages in Actor's workspace and selecting **Doit!.**

DELTA supports rapid-prototyping of an object-oriented software system using formalism. This prototype can provide insight into the system's behavior and determine the design's feasibility.

The following section 5.3 is reprinted with permission (see Appendix B) from Computers in Education Journal, Vol. III, No. 3, July/September 1993, pg. 29-35.

5.3 Selection of Actor

Object-oriented languages offer a natural implementation for encapsulating data and corresponding actions, called methods, by defining objects. In a pure object-oriented language, such as Actor, everything is an object. Every action that takes place is the consequence of sending a message to an object. The object responds by executing a selected method. In a hybrid object-oriented language, such as C++, not everything is an object. Since C++ is an extension of the language C, defined data structures may not be object-oriented. Special considerations must be made to resolve selection of a method. As a result, differences arise in the development, implementation, and maintenance of pure vs. hybrid languages.

Certain characteristics contribute to the object-oriented language persona. Among these are inheritance, class libraries, method determination, polymorphism, overloading, control of attribute access, and message sending. Other characteristics that are not unique to object-oriented languages, but are critical considerations are type checking, space and time efficiency, memory management through garbage collection, and the developmental environment. These traits are incorporated into the language definition of a

pure object-oriented language. In a hybrid language, these traits are an afterthought to the original language definition and must be incorporated in such a way as to preserve a natural continuity and extensibility of existing definitions and constructs. This may lead to constructs which sacrifice some of the potential realized by pure object-oriented languages.

By developing, implementing, and maintaining the same application in a pure and a hybrid language, insight can be found as to strengths and weaknesses, as well as to advantages and disadvantages of using each. The previously mentioned characteristics of object-oriented languages for hybrid and pure languages can be compared. As a result, certain domains of applications may be more suited to using one language type over the other. Actor and C++ languages will be used as the basis for an analysis of pure vs. hybrid object-oriented languages.

5.3.1 Object-Oriented Paradigm

C++ is a multi-paradigm language. Since it is an extension of C, a language primarily associated with procedural paradigm techniques, C++ can be used in a procedural decomposition approach. However, C++ was developed to support the object-oriented paradigm. Since everything in Actor is an object, Actor forces the development of a system to be created in terms of classes. The class concept, fundamental to the object-oriented paradigm is naturally supported by Actor.

5.3.2 Defining Classes

A class definition contains the instance variables and methods shared by a set of objects. The name of the class definition can be used to create instances of objects.

In C++ **struct**, **union**, or **class** are used to define classes. For example:

```
struct Point { // defines a struct class Point
    int X;     // member data is public by default
    int Y;
};
```

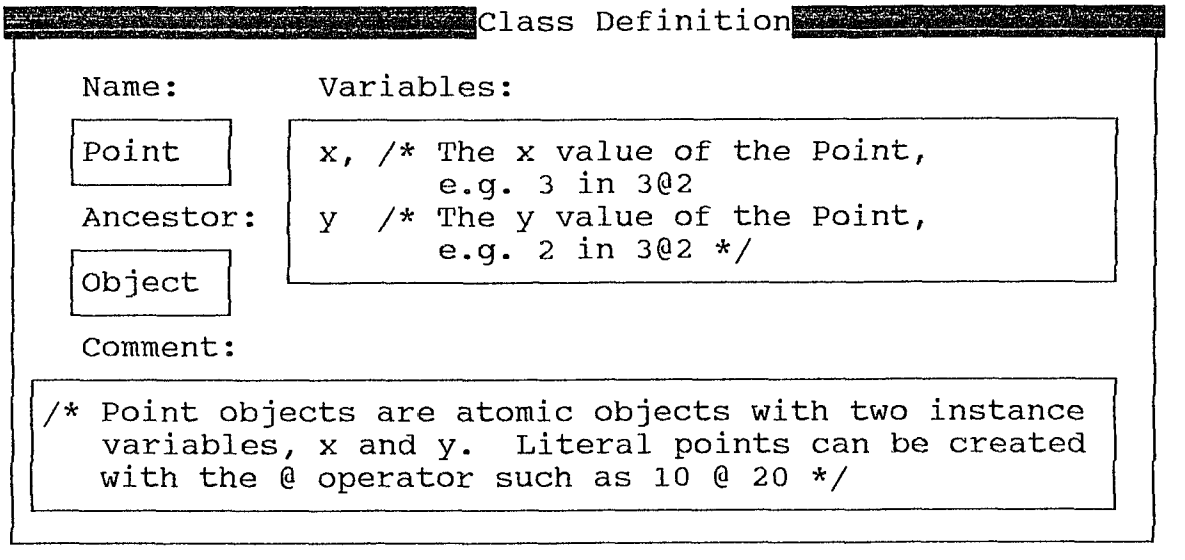

Figure 45 - Actor's Class Definition dialog box

In Actor, every class is a descendant of another class. **Point** is a descendant of class **Object.** Point is already defined as a class in the Actor hierarchy. The **Browser** is used to view existing class definitions, create new classes, and to add, modify or delete the methods in any class. To look at the class definition of **Point**,

use the **Browser** to list the Actor predefined classes, select class **Point**, and select **Class** on the **Browser** menu. The dialog box shown in Figure 45 is displayed. To open this class definition dialog box, select **Class Make Descendant** from the **Browser** menu. Fill in values for Name, Variables, and Comment. Click the **Accept** button to create the new class which is automatically placed in the **Browser.**

5.3.3 Object Instantiation

An object is an instance of a class. A variable of type **Point** would be an instance of type **Point.** An example in C++ is

```
Point Origin, Center;
```

An object in Actor can be instantiated in 3 ways:

1) Using a **New** message

```
Origin := new(Point);
```

which returns a Point initialized to nil@nil.

2) Creating a literal point by providing values

```
Origin := 0 @ 0;
```

3) Sending a **point** message to a number object

```
Origin := point(0,0);
```

5.3.4 Defining Methods

One of the differences between procedural languages and object-oriented languages is how functionality is defined. In procedural languages, functions are defined separately from the data by using functions, procedures, or subroutines. In an object-oriented language a function, called a

member function in C++ and a **method** in Actor, is declared within the class definition. Methods belong to a specific class and operate on only one type of object. Methods are invoked by sending **messages.** C++ provides 2 ways to add a member function to a class:

1) Define the function inside the class

```
struct Point {
    int X, Y;
     /* inline member function defined */
    int GetX() {return X;}
};
```

2) Declare the function inside the class, and define it outside the class

```
struct Point {
    int X, Y;
    int GetX();  /* member function declared */
};
int Point::GetX() { /* member function defined */
    return X;       /* outside the class */
}
```

The general format for an Actor method is:

```
/* Method comment */
Def methodName (self,arg1,arg2,...|loc1,loc2,...)
{ statement1;
   ...
}
```

Def denotes a method definition and is followed by the name

of the method (by convention begins with a lowercase letter). **self** refers to the object which will receive the message, since the receiver is not known at the time the method is defined. Any arguments are pass by value only. Local variables which only exist during the execution of a method are listed. The method code is enclosed in { }.

There are two categories of methods in Actor, **class methods** and **object methods.** Class methods are blocks of code which act upon the class itself, such as **new** to create a new instance of a class. Object methods are blocks of code which act upon instances of a class. Actor has less than 50 class methods and more than 1,000 object methods in the distributed Actor system.

To define a method in Actor, activate the **Browser.** Select **Class** or **Object methods.** Select **Template** on the **Method** menu. The following template appears:

```
/* comment */
Def method (self)
{}
```

Simply edit the template, and select **Accept** on the **Browser** menu to compile the method and add the method to the class definition. The following method is predefined in class **Point:**

```
/* Return the x value of the receiver point. */
Def x(self)
{^x;
}
```

5.3.5 Sending Messages

A message is used to invoke a method. Actions are performed in an object-oriented system by sending a message to an object. The object receives a message. A selector specifies what kind of action to perform. The object uses its method to perform the action. One of the strengths of object-oriented programming is **polymorphism** which is accomplished by the ability of a message to be associated with different methods.

C++ incorporates polymorphism by the use of **virtual functions.** Different versions of the same function can be used throughout a class hierarchy. Dynamic or late binding allows the specific version to be determined at run-time. The general syntax in C++ is:

```
ClassObjectName.MemberFunctionName(arg1,...)
```

For example: Origin.GetX() or Center.GetX() would invoke the Point method GetX since Origin and Center are of type Point. If MyCircle is an instance of class Circle which contains a method GetX, then MyCircle.GetX() would invoke the Circle method GetX.

Actor also incorporates dynamic or late binding to achieve polymorphism. A message is sent by:

```
messageName (receiver, arg1,...);
```

For example: x(Origin) would invoke the Point method x and x(MyCircle) would invoke the Circle method x.

5.3.6 Inheritance and Access Control

Besides the concept of combining data and functionality into an object, inheritance is one of the distinctions that sets object-oriented design and languages apart from structured design and procedural languages. Inheritance can be single or multiple. In either form, common functionality is built into the class hierarchy, allowing each subclass to specialize the inherited traits. The **base** class contains the common characteristics which are inherited by **derived** classes. With the complex systems being developed today and the ongoing software crisis, inheritance provides developers with an alternative to "reinventing the wheel".

Earlier versions of C++ only offered single inheritance, but C++ version 2.0 offers multiple inheritance as well. C++ provides access control mechanisms for its member data and functions. As was mentioned earlier, a class in C++ can be defined by **struct**, **union**, or **class**.

- **Struct** defines a class in which all the members are **public** by default, but the level of access can be changed.
- **Union** defines a class in which all the members are **public** by default, but the access level cannot be changed.
- **Class** defines a class in which all the members are **private** by default, but the level of access can be changed.

In general, member data is usually **private** and member functions are **public.** Any statement within the same scope as a class with public access control can read or change its internal data. However, this is not a desirable interface for data modification. Principles of a good design promote data hiding, i.e., data members should always be private or protected. Public member functions can then be used to access the data.

This is a drawback of C++ as a hybrid language. It adds complexity to existing structures of C to incorporate object-oriented characteristics. Since **Struct** allows public access by default, additional keywords must be used to control access of its members.

- **public** allows access to member functions within the same scope as the class definition.
- **private** restricts access only to member functions that have been declared in the same class.
- **protected** restricts access only to member functions that have been declared in the same class or by member functions in classes derived from this class.

Following the general rule to declare all data members as private and member functions as public, the **Point** structure with public access implicit in its declaration is as follows:

```
struct Point {
    int X;  /* member data is public by default */
    int Y;
    int GetX();
};
```

The same structure with private data members and public member functions explicitly declared is:

```
struct Point {
private:
    int X;
    int Y;
public:
    int GetX();
};
```

Class declares data private by default. **Class** is a construct that is found in C++, but not in C. As a result, **class** is preferred over **struct** in the object-oriented paradigm. Only public member functions are explicitly declared to allow them to be used outside of the class to access **Point** objects for initialization or data retrieval:

```
class Point {
    int X; /* member data is private by default */
    int Y;
public:    /* overrides private access control */
    int GetX();
};
```

Protected access infers the incorporation of inheritance, as it allows access to member functions in classes derived from the current class being declared. The general syntax for declaring a derived class is:

```
class DerivedClass : access_modifier BaseClass {
 ... /* private access by default */
}
```

or

```
struct DerivedClass : access_modifier BaseClass {
 ... /* public access by default */
}
```

Suppose **Point** inherits properties of a base class, **Location.** **Point** inherits all data members and member functions of **Location** and specializes the inherited class by adding data members or member functions of its own.

```
class Location {
protected:  /* allows access to derived classes */
    int X;
    int Y;
public:  /* allows access outside of this class */
    Location (int InitX, int InitY);
    int GetX();
    int GetY();
};
class Point : public Location {
/* Point is derived from class Location          */
/* public implies Point preserves protected      */
```

```
    /* access of Location's data members X and Y    */
    protected:
        Boolean Visible;
    public:
        Point (int InitX, int InitY);
        void Show ();
        void Hide ();
        Boolean IsVisible();
        void MoveTo(int NewX, int NewY);
    };
```

Multiple inheritance offer a means of a derived class in C++ to inherit from two or more base classes. This increases the potential for code sharing amongst classes. The complexity of a design is increased with multiple inheritance. If the base classes both have a method of the same name, some mechanism must be used to determine which will be selected upon receipt of a message.

Suppose the class hierarchy shown in Figure 46 is used to create a class to graph a line. Class **LineGraph** inherits from **Line** and **XYAxis**. Suppose **Line** and **XYAxis** both have a method **Show** defined, then in the body of the definition of **LineGraph::Show** would be two function calls **Line::Show();** and **XYAxis:: Show();**. The scope resolution operator, ::, overrides a member function in a derived class and provides information to the compiler. **Show()** would simply refer to the current scope's member function, **LineGraph::Show**.

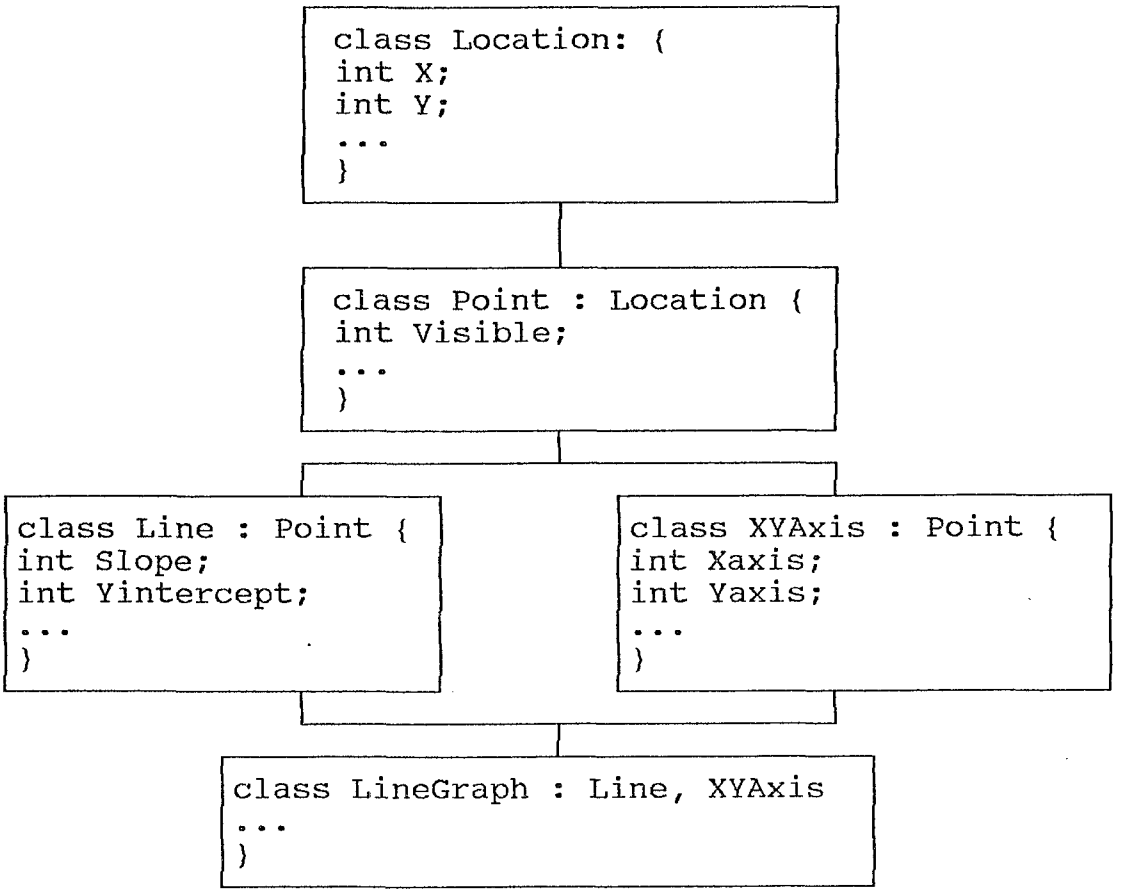

Figure 46 - Class Hierarchy of a Line Graph

Since everything in Actor is an object and Actor has single inheritance, all classes in Actor are derived or inherit from class **Object.** Every class defined in Actor must be placed into the class hierarchy and therefore has an ancestor. The Browser can be used to select the ancestor class. Upon selecting **Class Make Descendant**, the Class Definition Dialog shown previously is opened with the ancestor class automatically indicated. For example, to define a class **Line**, class **Point** would be selected as the ancestor class. Only additional variables need to be defined in the current class. Since class **Point** already

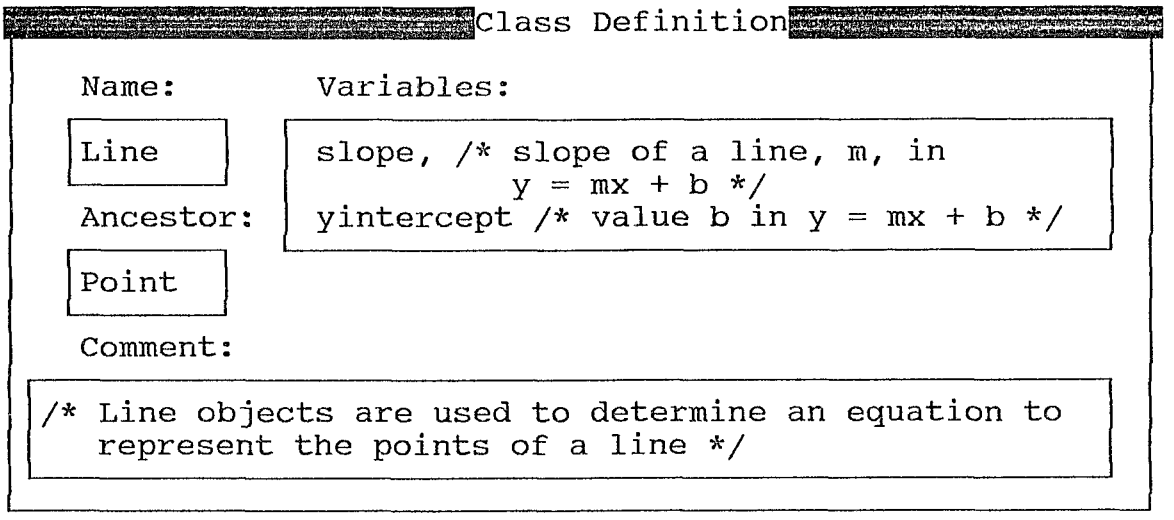

Figure 47 - Class Definition for Class Line

has variables **x** and **y** defined, only **slope** and **yintercept** will be listed in the Variables entry field. An example of the Class Definition for class **Line** is shown in Figure 47. Upon **Accepting** this Class Definition, class **Line** is now part of the class hierarchy with instance variables, **x**, **y**, **slope**, and **yintercept** and a new class source file, LINE.CLS, is added to Actor's WORK subdirectory. Now any methods defined while class **Line** is selected in **Browser** will be added to the **Line** class file in the WORK directory.

5.3.7 Garbage Collection and Memory Allocation

There are two ways to handle memory management. One way is to require the programmer to explicitly allocate and free memory, such as in C++. The other way is for the language to provide automatic memory management, such as in Actor. Not only is it disastrous if a programmer frees data before it is finished being used, manual memory management

also increases the complexity of programming. Bjarne Stroustrup, the developer of C++, felt memory management is too important not to be placed in the hands of the programmer. However, the Whitewater group felt memory management is too important for a sophisticated window programming environment to be placed in the hands of the programmer. Therefore, Actor contains a garbage collector and static memory swapping.

C++ has two special types of member functions, **constructors** and **destructors**, which play an important role in memory management. Objects can have **static** duration in which they are allocated memory from the start of program execution until it ends. All functions are static objects. All variables with file scope, globals, are also static. A variable can explicitly be declared with **static** or **extern** to be given static duration. Static objects are initialized to zero or null, if no explicit initial value is given. Objects can also have **local** duration in which they are created on a stack when a program enters a block or function and deallocated when it exits from that block or function. Objects can also have **dynamic** duration in which they are created and destroyed by specific function calls during a program's execution. Storage is allocated from a special memory reserve known as the heap. This is handled by the standard library functions, **malloc** and **free**, or by the constructor, **new**, and destructor, **delete**.

A constructor specifies how a new object of a class type will be allocated memory and initialized. A constructor can be user-defined or generated by default. The C++ compiler calls the appropriate constructor when a new object of a class is defined. This occurs in a data declaration, when an object is copied, or by using **new** to dynamically allocate a new object. Here is an example of class **Point** with a constructor:

```
class Point {
     int X;
     int Y;
public:
     int GetX();
     Point (int NewX, int NewY); /* constructor
                                    declaration */
};
Point::Point (int NewX, int NewY) /* constructor
                                     definition */
{
     X = NewX;
     Y = NewY;
};
```

The name of the constructor and the name of the class are the same, allowing the compiler to know it is a constructor. A constructor does not have a return type as other functions do. Default values can be given for function arguments:

```
Point::Point(int NewX=0, int NewY=0)
```

This statement would initialize any declared Point's X and Y values to 0. Thus, the declaration:

```
Point Center(4);
```

would initialize X to 4 and Y to 0 by default. The following statement would create a dynamic **Point** class object:

```
Point *Center = new Point(25,25);
```

However, if **new** is used to create a dynamic object, the programmer is responsible for deallocating it. C++ cannot determine when the object will no longer be needed. Therefore, the operator **delete** must be executed to deallocate memory:

```
delete Center;
```

Just as a constructor for a class can be identified by the same name, a destructor for a class is the class name preceded by '~'.

Suppose **Node** represents a linked list of **Node** records which are **Point** objects.

```
struct Node {      /* the list item can be Point */
     Point *Item; /* or any class derived from Point */
     Node *Next;
};
class List {  /* the list of objects */
     Node *Nodes;
public:
     List();  /* constructor */
     ~List(); /* destructor */
```

```
        ...
    };
```

The corresponding constructor and destructor definitions are:

```
    List::List () {    /* constructor */
        Node *N;
        N = new Node;
        N->Item = NULL;
        N->Next = NULL;
        Nodes = NULL;
    }
    List::~List() {    /* destructor */
        while (Nodes != NULL) {
            Node *N = Nodes;
            delete (N->Item);
            Nodes = N->Next;
                delete N;
        };
    }
```

Constructors and destructors cannot be inherited, but a derived class can call its base class' constructors and destructors. The compiler automatically calls constructors and destructors when defining and destroying objects. If a destructor is not explicitly defined for a class, then the compiler will generate one, as in the class **Point.** Members of an object must be deallocated before the object itself is deallocated, as in the class **List.** As one can

see, memory allocation must be a concern of a C++ programmer.

Actor has a **new** message used to create objects. There is no construct for deleting an object from memory. Actor handles this task automatically. The garbage collector, due to its careful design, takes only a fraction of a second to free memory. This does not cause any noticeable interruption in time-critical operations as can be seen in other languages which may pause for seconds or minutes while garbage collection takes place.

Actor has **static** and **dynamic** regions for its object memory. Classes, methods, symbols, and compiled methods which will remain in use for a long time are stored in static memory. Objects with a short life span, such as strings, long integers, and those created in an application are stored in dynamic memory. A **static** method can be used to make any dynamic object a static object. The garbage collector reduces its work load by focusing only on dynamic memory. It continuously copies dynamic objects that are still accessible from one memory location to another. The memory of those objects considered as "dead" is reused.

The developer can allocate the amount of kilobytes of static and dynamic memory by using the **Snapshot As** command. The developer can use the **Show Room!** command to display the amount of static space in use. The **Cleanup!** command invokes a static garbage collector that can be used

during the compile and install process. There are also two methods available for checking on dynamic and static memory. The **staticRoom()** method displays available bytes for static memory. The **checkDynamic** method displays an error box, if there is less than 5000 bytes of dynamic memory available.

A "live" or accessible object is one that can be reached through a global variable. Global variables should be kept to a minimum allowing the garbage collector to free memory quickly. Large objects no longer in use should be set to **nil**.

5.3.8 Development Environment

It's important for an object-oriented language to have a good development environment to take full advantage of the object-oriented traits such as reusability and inheritance. C++ does not have a standard development environment, because there are several different Software vendors of C++, such as AT&T, Zortech, and Borland. Although C++ is portable, the developer may not be familiar with a different development environment.

Actor avoids these problems by being a complete development environment and programming language that runs as a Windows application. This makes it easier to develop windows applications with menus and dialog boxes. Actor's development tools are popup windows used for compilation of code, inspection of objects, and the debugging of code. Among them are:

- **Browser** used to view existing classes and methods and to create new ones which can immediately be tested
- **Workspace** used to test code that was created in a browser or on a line-by-line basis
- **Inspector** used to view or change data values
- **Debugger** used to debug run-time problems--automatically opened by Actor when a run-time error occurs

An Actor developer has a standard environment, created by The Whitewater Group, in which to work and thereby enhances reusability.

5.3.9 Class Library

The Class Library of an object-oriented language contains the building blocks for the foundation of the application. The richer the class library, the more quickly and easily a new application can be developed. Classes can be used as templates to create objects. Classes set up code organized for reusability.

C++ does not have a standard class library. Also, C++ does not provide any guidelines for organizing class libraries. This leads to incompatibility of different class libraries. If C++ is to become the dominant object-oriented language, a standard class library is a must.

Actor on the other hand includes more than 125 classes of objects. Besides the basic classes, such as those for integers, there are graphics, windowing, and data

management classes. A hierarchical chart of the classes is given in the Actor manual and is listed in the **Browser.** Actor immediately allows the developer to experience the concept of reusability as applications are built.

5.3.10 Comparison of Languages

Table 1 is a summary of the key points of the previous characteristics and implementations of C++ and Actor. If a

	C++ - 2.0	Actor - 4.1
Object-oriented Foundation	Hybrid -Based on C	Pure
Paradigm	Multi-paradigm -procedural -object-oriented	Single-paradigm -object-oriented
Defining Classes	Language Construct -struct, union, or class	Dialog Window
Defining Methods	Member Function -inline -outside class	Method -Def construct
Sending Messages -Polymorphism	Virtual Function -Object Name -Member Function -List of arguments	Message -Message Name -Receiver -List of arguments
Inheritance	Multiple	Single
Access Control	Explicit -Public, Private, or Protected	Implicit
Garbage Collection	Explicit -Constructors -Destructors	Implicit -Allows Explicit
Development Environment	Not Standard	Standard
Class Library	Not Standard	Standard

Table 1 - Comparison of C++ & Actor

choice were to be made with no previous background in object-oriented programming of using a hybrid or a pure language, then the choice is clearly a pure language. If a developer is experienced in C, then it would be a natural progression to adopt C++. Because of the widespread use of C, C++ is becoming the dominant object-oriented language. However, since Actor's syntax is similar to Pascal or C, it is easy for a C developer to learn Actor. Actor does not have any object-oriented characteristics that are more complex than its counterpart in C++. Actor provides implicit access control and garbage collection. The standard development environment and class library also provide an incentive to use Actor over C++. One feature C++ has that Actor doesn't is multiple inheritance, but problems exist. Any of a variety of ancestors can be the source of error in C++, destroying the concept of encapsulation. When multiple ancestors have methods of the same name, the first-named ancestor is selected or one is explicitly named, undermining polymorphism. Actor implements a form of multiple inheritance, termed protocols, which avoids the problem of producing code that is difficult to maintain. Another problem with a hybrid language such as C++ is, even though one may not need to know C to learn C++, most manuals and texts on C++ refer pervasively to C.

5.3.11 Summary

Users of today's systems want sophisticated Graphical User Interfaces (GUIs) such as providing windows, pull-down

menus, icons, and using a mouse. A window-based language, such as Actor, provides the standard window and menu classes, as well as, tools in the development environment to reduce the time it takes to develop a GUI. C++ does not provide a set of standard classes, so GUIs must be built based on the vendor's supplied classes. A developer has to create most of the code himself. An Actor developer would have a prototype GUI up and running much sooner than his C++ counterpart.

The strengths of a pure object-oriented language seem to be overwhelmingly in its favor. Whereas, unless one is attached to using the procedural base language for a hybrid object-oriented language, the only benefits derived are the object-oriented concepts of polymorphism and reusability that do not exist in the base language.

Actor's simplicity and consistency to the principles of object-oriented design was chosen as the first language in which to prototype a **DELTA** design. Due to C++'s growing dominance, C++ is the next logical choice for prototyping a **DELTA** design.

6 CONCLUSIONS

6.1 Summary

The importance of formal languages to software design has been noted. A formal object-oriented design language, **DELTA**, has been defined with a graphical development environment. This satisfies the demand for an easy-to-learn and easy-to-use visual interface.

Characteristic features of an OODL have been presented. **DELTA** incorporates as many of these as possible without becoming language dependent. The OODL features present in **DELTA**'s language independent grammar are:

-- Definition of objects

-- Class Definition

-- Method Definition

-- Message Passing

-- Polymorphism

-- Inheritance

- Single or Hierarchical

The following features are language dependent and are represented in **DELTA**'s formal annotations to Actor.

-- Method constructs

- condition constructs
- repetition constructs
- I/O constructs

-- Encapsulation

Since **DELTA**'s source code is written in Actor, **DELTA** provides the following feature automatically.

-- Garbage Collection

A set of goals has been established for **DELTA** and a discussion of how **DELTA** meets those goals is given in the next section.

A prototype of **DELTA** has been developed to measure its feasibility as a realizable environment.

An extensive analysis of Actor and C++ led to the selection of Actor as the first implementation language in which to prototype a **DELTA** design representation.

6.2 Contributions

The definition and realization of **DELTA**, as described in this work, advances the state of object-oriented design in the following ways:

1) **DELTA** provides access to a graphical view of text at all levels of abstraction. **DELTA** provides a visual display of the object hierarchy, class definition, and method attributes. **DELTA** can be applied independently of an implementation language as a visual design language for use in the design of object-oriented databases.
2) The graphical and textual notations of **DELTA** support the standard object-oriented concepts found in object-oriented programming languages. Among those concepts supported in **DELTA** are classes, inheritance, methods, and message passing. This makes the transition to object-oriented code more efficient and less error-prone.

3) Formalism provides a means for developers to establish consistency and remove ambiguity from a design. The simplicity of **DELTA**'s notation allows experienced developers to take advantage of its formalism.

4) A well-documented system promotes reusability and reduces the maintenance costs of software systems. Documentation is provided in **DELTA** by the graphical view of the object hierarchy and textual views of the class, method, and message definitions.

5) Studies show that Graphical User Interface (GUI) development consumes up to 65% of programming time. Since today's system call for increased GUI's as user interfaces, a formal language which taps the potential of the underlying windows classes will provide a more efficient use of development time. One certainty in the software life-cycle is change. An automated environment which accesses the design representation incrementally supports prototyping and provides an efficient means of incorporating changes to the design. A pure object-oriented language, such as Actor, provides a standard class library which promotes rapid-prototyping and realizes inheritance at its full potential. Rapid-prototyping is possible by mapping **DELTA** to Actor. Since **DELTA**'s graphical view supports Actor's object hierarchy and object components and **DELTA**'s textual view is an extension of existing Actor notation, rapid-prototyping in Actor is a smooth transition from

DELTA. Existing Actor systems can be enhanced by **DELTA**, as a visual design of the system can be generated, documentation of classes and methods can be created, and annotations can be added.

6.3 Future Research

The extension of Level IV of **DELTA**'s syntax to other object-oriented languages and environments will increase **DELTA**'s applicability and flexibility. A prototype can then be created in languages other than Actor. This will also provide insight into incorporating improvements to the existing **DELTA** syntax.

The incorporation of multiple inheritance into the syntax of **DELTA** will enhance **DELTA**'s flexibility as well. There are many critical issues to consider with multiple inheritance. The complexity of the design increases significantly. A class precedence must be established to resolve any conflicts of multiple definitions in ancestor classes.

Many software companies are creating in house class libraries for reuse. The potential exists to incorporate **DELTA** representations into those libraries. This will provide reusability of **DELTA** designs in future software projects. Prototypes created by **DELTA** can be modified and reused as well.

Maintenance of existing legacy systems is an ongoing crisis for software engineers. Demands to improve existing

systems to interface with new technology are constant. Reverse engineering tools show the complex relationships of program elements and leave the more inventive and creative thinking to software engineers. A future enhancement of the **DELTA**/Actor prototyping system is developing a means of reverse engineering.

REFERENCES

[Aue86] Auernheimer, Brent, and Richard A. Kemmerer, "RT-ASLAN: A Specification Language for Real-Time Systems", IEEE Transactions on Software Engineering, Vol. SE-12, No. 9, September 1986, pg 879-889.

[Boo83] Booch, G., Software Engineering with Ada, Benjamin-Cummings, 1983.

[Boo91] Booch, Grady, Object-Oriented Design, Redwood City, Calif.: Benjamin-Cummings, 1991.

[Bou92] Bourne, John R., Object-Oriented Engineering : Building Engineering Systems Using Smalltalk-80, Richard D. Irwin, Inc., and Aksen Associates, Inc., 1992, pg 260-269.

[Cai75] Caine, S. and K. Gordon, "PDL - A Tool for Software Design", Proc. National Computer Conference, AFIPS Press, 1975, pg 271-276.

[Car88] Carver, D. L., D. W. Cordes, and Nancy Gautier, "Object-Based Measurement in the Requirements Specification Phase", Sixth Symposium on Empirical Foundations of Information and Software Science Conference Proceedings, Atlanta, GA, October 19-21, 1988.

[Car90] Carver, D.L. and D.W. Cordes, "An Object-Oriented Framework to Support Architectural Design Development, Proc. 23rd Hawaii International Conference on System Sciences, Kaliua-Kona, Hawaii, January 1990.

[Cli90] Cline, Marshall, and D. Lea, "The Behavior of C++ Classes", Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications, Marist College, 1990.

[Coa90] Coad, Peter and Edward Yourdan, Object-Oriented Analysis, Yourdan Press/Prentice Hall, 1990, pg 5-7.

[Cor92] Cordes, D.W. and D.L. Carver, "An Object-Based Requirements Modeling Method", Journal of the American Society for Information Science, January 1992.

[Cox86] Cox, B. Object-Oriented Programming, Addison-Wesley, 1986.

[Dah72] Dahl, O., E. Dijkstra, and C. Hoare, Structured Programming, Academic Press, London, 1972.

[Dan93] Daniels, John, and Steve Cook, "Strategies for sharing objects in distributed systems", Journal of Object-Oriented Programming, January 1993, pg 27-36.

[Den73] Dennis, J., "Modularity", Advanced Course on Software Engineering, F.L. Bauer, ed. Spring-Verlag, New York, 1973, pg 128-182.

[Edw93] Edwards, J. M., and B. Henderson-Sellers, "A graphical notation for object-oriented analysis and design", Journal of Object-Oriented Programming, February 1993, pg. 53-73.

[Fai85] Fairley, Richard E., Software Engineering Concepts, McGraw Hill, 1985, pg 114-117.

[Gau91] Gautier, Nancy K., and D. L. Carver, " Language Features of an Object-Oriented Design Language (OODL)", Technical Report #91-018, Dept. of Computer Science, Louisiana State University, Baton Rouge, La., 1991.

[Gau93a] Gautier, Nancy K., and D. L. Carver, "DELTA -- A DEsign Language To Actor", Proceedings of COMPSAC '93, Phoenix, Arizona, November 1993.

[Gau93b] Gautier, Nancy K. and D. L. Carver, "Actor vs. C++ -- A Pure vs. Hybrid Object-Oriented Language", Computers in Education Journal, July/September 1993.

[Gut82] Guttag, J. V., J. J. Horning, and J. M. Wing, "Some Remarks on Putting Formal Specifications to Productive Use", Science of Computer Programming, North-Holland, Vol. 2, No. 1, October 1982, pg. 53-68.

[Gut85] Guttag, J. V., J. J. Horning, and J. M. Wing, "The Larch Family of Specification Languages", IEEE Software, Vol. 2, No. 5, Sept. 1985, pg 24-36.

[Hal87] Halbert, Daniel, and Patrick D. O'Brien, "Using Types and Inheritance in Object-Oriented Programming", IEEE Software, September 1987.

[Hei87] Heitz, M., "HOOD: Hierarchical Object Oriented Design for development of large technical and realtime software", CISI Ingenierie, Direction Midi Pyrenees, November, 1987.

[Hen90] Henderson-Sellers, and Julian M. Edwards, "Object-Oriented Systems Life Cycle", CACM, Vol. 33, No. 9, September 1990, pg 143-159.

[Jac75] Jackson, M., Principles of Program Design, Academic Press, 1975.

[Jac83] Jackson, M. A., System Development, Prentice-Hall, London, 1983.

[Jac93] Jacobsen, Ivar, "Is Object Technology Software's Industrial Platform?", IEEE Software, January 1993, pg 24-30.

[Kem85] Kemmerer, Richard A., "Testing Formal Specifications to Detect Design Errors", IEEE Transactions on Software Engineering, Vol. SE-11, No. 1, January 1985, pg 32-43.

[Ker88] Kerth, N., "MOOD: a Methodology for Structured Object-Oriented Design", Tutorial, OOPSLA '88, San Diego 1988.

[Kim88] Kim, H.J., et. al., "PICASSO: a graphical query language", Software Practice and Experience", Vol. 18., No. 3, March 1988, pg. 169-203.

[Kor90] Korson, Tim and John D. McGregor, "Understanding Object-Oriented: A Unifying Paradigm", CACM, Vol. 33, No. 9, September 1990, pg 40-60.

[Lee90] Lee, Sangbum and D.L. Carver, "The Construction of an Object-Oriented Specification", Proc. of IEEE Southeastcon '90, New Orleans, LA, April 1990.

[Lee91] Lee, Sangbum and D.L. Carver, "Object-Oriented Analysis and Specification: A Knowledge Base Approach", Journal of Object-Oriented Programming, January 1991.

[Lor86] Lorensen, W., Object-Oriented Design, CRD Software Engineering Guidelines, General Electric Co., 1986.

[Luc85] Luckham, David C., and Friedrich W. von Henke, "An Overview of Anna, a Specification Language for Ada", IEEE Software, March 1985, pg 9-22.

[Luq92] Luqi, "Status Report: Computer-aided prototyping", IEEE Software, November 1992, pg 77-81.

[McD75] McDonald, N., and M. Stonebraker, "CUPID--the friendly query language", Proceedings ACM Pacific Conference, San Francisco, 1975, pg 127-131.

[Mey85] Meyer, Bertrand, "On Formalism in Specifications", IEEE Software, January 1985, pg 6-26.

[Mey87] Meyer, Bertrand, "Reusability: The Case for Object-Oriented Design", IEEE Software, March 1987.

[Mey88] Meyer, Bertrand, Object-Oriented Software Construction, Prentice Hall, Englewood Cliffs, N.J., 1988.

[Pas86] Pascoe, Geoffrey A., "Elements of Object-Oriented Programming", BYTE, August 1986, pg 139-144.

[Pin90] Pinson, Lewis J., and Richard S. Wiener, Applications of Object-Oriented Programming, Addison-Wesley Publishing Company, Inc., 1990, pg 101-138.

[Pre92] Pressman, Roger S., Software Engineering: A Practitioner's Approach, 2nd ed., McGraw-Hill, 1987.

[Sei86] Seidewitz, E. and M. Stark, "General Object-Oriented Software Development", Report SEL-86-002, NASA Goddard Space Flight Center, 1986.

[Som92] Sommerville, Ian, Software Engineering, Addison-Wesley Publishing Company, 1989, pg 226-227.

[Ste86] Stefik, Mark and Daniel G. Bobrow, "Object-Oriented Programming: Themes and Variations", The AI Magazine, Winter 1986, pg 40-62.

[Ste74] Stevens, W., G. Myers, and L. Constantine, "Structured Design", IBM Systems Journal, vol. 13, no. 2, 1974, pg 115-139.

[Tei77] Teichroew and E. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pg 41-48.

[Val93] Valenzano, A., R. Sisto, and L. Ciminiera, "Rapid Prototyping of Protocols from LOTOS Specifications", Software --Practice and Experience, Vol. 23(1), January 193, pg 31-54.

[War85] Ward, P. T. and S. J. Mellor, Structured Development for Real-Time Systems, vol. 2: Essential Modeling Techniques, Prentice Hall (Yourdan Press), Englewood Cliffs, NJ, 1985.

[Was80] Wasserman, A., "Principles of Systematic Data Design and Implementation" in Software Design Techniques, P. Freeman and A. Wasserman, eds. 3d ed., IEEE Computer Society Press, 1980, pg 287-293.

[Was89] Wasserman, Anthony I., Peter A. Pircher, and Robert J. Muller, "An Object-Oriented Structured Design Method for Code Generation", ACM SIGSOFT Software Engineering Notes, Vol 14, No 1, Jan 1989, pg 32-55.

[Whi92a] The WhiteWater Group, Tools and Tutorials, The WhiteWater Group, Inc., 1992 pg 48-49.

[Whi92b] The WhiteWater Group, Actor Programming, The WhiteWater Group, Inc., 1992 pg xi-xii.

[Wie84] Wiener, R., and R. Sincovec, Software Engineering with Modula-2 and Ada, Wiley, 1984.

[Win90] Wing, Jeannette M., "A Specifier's Introduction to Formal Methods", Computer, September 1990, pg 8-24.

[Wir71] Wirth, N., "Program Development by Stepwise Refinement", CACM, vol. 14, no. 4, 1971, pg 221-227.

[Wir90] Wirfs-Brock, Rebecca, B. Wilkerson, and L. Wiener, Designing Object-Oriented Software, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1990.

[Won82] Wong, H.K.T., and I. Kuo, "GUIDE: graphical user interface for database exploration", Proceedings 8th International Conference on Very Large Data Bases, Mexico City, September 1982, pg 22-32.

[Wu87] Wu, C.T., "GLAD: Graphics Language for Databases", Proceedings of 11th International Conference on Computer Software and Applications, Tokyo, 1987, pg. 164-170.

[Zav82] Zave, Pamela, "An Operational Approach to Requirements Specification for Embedded Systems", IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, May 1982, pg 250-269.

APPENDIX A

Actor Language Description : Formal Grammar

```
NUMBER              : <Int> | <Long> | <IEEE Real>
LITERAL             : NUMBER '@' NUMBER
                    | "#("  [LITERAL]@ ")"
                    | "&(" Int Int Int Int ")"
                    | '"' <ascii string> '"'
                    | '#' IDENTIFIER
                    ;

KW_IF               : "if"
KW_ELSE             : "else"
KW_THEN             : "then
KW_ENDIF            : "endif"
KW_BEGIN            : "begin"
KW_WHILE            : "while"
KW_ENDLOOP          : "endLoop"
KW_LOOP             : "loop"
KW_DEF              : "Def"
KW_SELF             : "self"
KW_SELECT           : "select"
KW_ENDSELECT        : "endSelect"
KW_CASE             : "case"
KW_ENDCASE          : "endCase"
KW_USING            : "using"
KW_DEFAULT          : "default"
KW_IS               : "is"
WCALL 280           : "Call"

ASSIGN              : ":="
TYPE                : ':' IDENTIFIER
IDENTIFIER          : [a-z]+ [a-z | 0-9]@
INFIX 271           : <element of InfixOps>

------------------------------------
script              : unit
                    | script unit
                    ;

unit                : func
                    | stmtList
                    ;
```

```
rcvr            : obj
                | obj  TYPE
                ;

obj             : IDENTIFIER
                | obj '[' obj ']'
                | obj TYPE '[' obj ']'
                | ivChain
                | LITERAL
                | NUMBER
                | '-' NUMBER
                | IDENTIFIER '(' rcvr argList ')'
                | IDENTIFIER '(' ')'
                | WCALL  '(' obj argList ')'
                | WCALL  '(' ')'
                | obj '-' rcvr
                | obj INFIX rcvr
                | '(' obj ')'
                | block
                | assgn
                | ifElseStmt
                | KW_SELF
                ;

ivChain         : IDENTIFIER '.' IDENTIFIER
                | obj '.' IDENTIFIER
                | ivChain '.' IDENTIFIER
                ;

assgn           : IDENTIFIER ASSIGN obj
                | obj '[' obj ']' ASSIGN obj
                | obj TYPE '[' obj ']' ASSIGN obj
                | ivChain ASSIGN  obj
                ;

semi            : /* empty */
                | ';'
                ;

argList         : /* empty  */
                | ',' obj
                | argList ',' obj
                ;
```

```
stmt            : obj
                | ifStmt
                | loopStmt
                | caseStmt
                | '^' stmt
                ;

stmtList        : /*  empty  */
                | stmt
                | stmtList ';'
                | stmtList ';' stmt
                ;

ifStmt          : KW_IF  obj then stmtList KW_ENDIF
                ;

ifElseStmt      : KW_IF obj then stmtList KW_ELSE
                  stmtList KW_ENDIF
                ;

then            : /* nothing */
                | KW_THEN
                ;

begin           : /* nothing */
                | KW_BEGIN
                ;

is              : /* nothing */
                | KW_IS
                ;

loopStmt        : KW_LOOP stmtList KW_WHILE obj begin
                  stmtList KW_ENDLOOP
                ;

caseStmt        : KW_SELECT caseList defClause
                  KW_ENDSELECT
                ;

caseList        : caseClause
                | caseList caseClause
                ;
```

```
defClause       : /*  empty */
                | KW_DEFAULT stmtList
                ;

caseClause      : KW_CASE obj is stmtList
                    | KW_ENDCASE semi
                ;

parmList        :  /* empty */
                | IDENTIFIER
                | ',' IDENTIFIER
                | parmList IDENTIFIER
                | parmList ',' IDENTIFIER
                ;

locList         : /* empty */
                | IDENTIFIER
                | locList IDENTIFIER
                | locList ',' IDENTIFIER
                ;

locDefs         : /* empty */
                | '|' locList
                ;

fName           : IDENTIFIER
                | INFIX
                | '-'
                ;

func            : KW_DEF fName '(' KW_SELF parmList
                | locDefs ')' semi '{' stmtList '}' semi
                ;

blkHeader       : /* empty */
                | KW_USING  '(' locList locDefs ')'
                ;

block           : '{' blkHeader stmtList '}'
                ;
```

APPENDIX B : Letter of Permission

Southeastern Louisiana University
Department of Computer Science
P.O. Box 506, SLU
Hammond, LA 70402
504-549-2055

October 13th, 1993
Nancy K. Gautier
SLU 758
Hammond, LA 70402

Cheryl Fields
Computers in Education Journal
Port Royal Square
Po Box 68
Port Royal, VA 22535

Dear Ms. Fields:

I am writing to request copyright permission to include my paper, "ACTOR vs. C++ - A Pure vs. Hybrid Object-Oriented Language" which appeared in the July-September issue, as part of my published dissertation. I will need to include a copy of the request and granted permission at the beginning of the chapter which includes the above paper.

As I need to turn my work in to the graduate school no later than November 12, 1993, I would appreciate a reply as soon as possible. Thank you for all of your help this summer.

Sincerely,

Nancy K Gautier

Nancy K. Gautier

Permission is granted for you to include your paper, "ACTOR vs. C++ - A Pure vs. Hybrid Object-Oriented Language", as part of your published dissertation.

Cheryl A. Fields 10/20/93

Cheryl A. Fields
Office Manager

VITA

Nancy K. Gautier is currently a Computer Science Instructor at Southeastern Louisiana University in Hammond, Louisiana. She received an A.S. in Computer Science and a B.S. in Mathematics from Southeastern Louisiana University in 1980. She received an M.N.S. in Mathematics and Computer Science from Louisiana State University in 1983. Nancy K. Gautier has taught at the university level for 10 years. Her research interests are in Software Engineering, Object-oriented Development, and Numerical Methods. Nancy plans to be promoted to the rank of Assistant Professor in Spring of 1994 and marry Robert A. Schaerfl, Jr. in March of 1994.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Nancy K. Gautier

Major Field: Computer Science

Title of Dissertation: Object-Oriented Modeling and Design Using DELTA, an Incremental Design Language

Approved:

Boris L. Carver
Major Professor and Chairman

Daniel Reed
Dean of the Graduate School

EXAMINING COMMITTEE:

J R Dorroh

David H Kaff

J Ross Jones

Harriet G. Taylor

John M Kak

Date of Examination:

9/1/93