Computer Science and Systems Analysis

Computer Science and Systems Analysis

Technical Reports

Miami University *Year* 1991

A Comprehensive Description and Critical Analysis of Object-Oriented Software Development

Charles III
Miami University, commons-admin@lib.muohio.edu

MIAMI UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & SYSTEMS ANALYSIS

TECHNICAL REPORT: MU-SEAS-CSA-1991-003

A Comprehensive Description and Critical Analysis of Object-Oriented Software Development

Charles K. Ames III

School of Engineering & Applied Science | Oxford, Ohio 45056 | 513-529-5928

A Comprehensive Description and Critical Analysis of Object-Oriented Software Development

by

Charles K. Ames III
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #91-003 August 1991

A Comprehensive Description and Critical Analysis of Object-Oriented Software Development

FINAL REPORT

Presented in Partial Fulfillment of the Requirements
for the Degree of
Master of Systems Analysis
in the
Graduate School of Miami University

BY

Charles K. Ames III

Miami University

1991

Reading Committee:

Dr. James D. Kiper, Advisor

Dr. Alton F. Sanders

Prof. Douglas A. Troy

A Comprehensive Description and Critical Analysis of Object-Oriented Software Development

Charles K Ames III
Systems Analysis, Miami University
August 21, 1991

Object-orientation and the object model underlie a simple, intuitive, and useful approach to software development that has great potential for significantly improving the software development process. Object-orientation unifies activities, such as analysis and design, that currently largely independent and somewhat incompatible. Despite its simplicity, a shroud of mystery surrounds this approach. The difficulty often encountered when learning and understanding object-oriented methods stems partly from the capricious and improper use of object-oriented jargon in conversation and in the literature. This paper explores the foundations of the object model, defines the associated terminology in concrete terms, and gives an overview of various object-oriented methods as they relate to the software lifecycle.

1 Introduction

Object-orientation is an increasingly popular paradigm which has great promise for improving software development throughout its life cycle. There are many advantages and potential benefits to be gained from shifting to an object-oriented viewpoint. Unfortunately, the popularity of the adjective "object-oriented" is not without a downside. The widespread and capricious use of this term has resulted in a somewhat ambiguous and jargonistic terminology. It is difficult to know what is meant when someone says "this <thing> is object-oriented." In addition, discussions in the literature often describe the object model in terms of itself, with the reader left to "catch-on" on his or her own.

This paper is intended to provide a reasonably complete and understandable description of the object model and its associated terminology, and to explore a variety of object-oriented methods. Section 1 discusses the foundations of object-oriented modelling, and section 2 introduces and defines terms commonly used to describe the object model. Section 3 attempts to sharpen the reader's understanding of several of the concepts introduced in section 2 by comparing and contrasting those that are often confused. Section 4 discusses the potential benefits of adopting object-oriented methods, and section 5 discusses the impact of object-oriented methods on the software lifecycle.

1.1 The Object-Oriented Mindset

Object-orientation is more than a software development method or a programming style. It is a way of looking at systems that is fundamentally different from that which underlies more traditional (i.e.

structured) approaches to software development. Some traditional methods focus almost exclusively on what the system is to do, with minimal consideration given to associated data until the system architecture is mostly complete. Other methods focus on the data that is to be manipulated, subsequently deriving functionality to transform this data into the data that is needed. The object-oriented mindset allows a developer to see systems in terms of active components made up of data fused together with associated functionality. These components often correspond closely to the entities actually found in the problem domain.

Consider a student registration system. Functional decomposition would lead to a process driven view of the system based on a set of procedures, such as add_student_to_course and drop_student_from_course, with data being considered later. A data driven approach would lead to a view of the system based on important data elements, such as student_rec and course_list, with the necessary functionality being of secondary importance. An object-oriented view of the system would contain active entities, such as student and registrar. These entities would have state (data) and behavior (functionality) that is similar to their real world counterparts.

Clearly, a student registration system built using any of these three approaches will accomplish the same thing. The difference is in the way the system is modelled. The key to understanding and making use of object-orientation is in seeing systems the same way we see the real world: in terms of active, interacting entities. Process driven and data driven approaches each place heavier emphasis on either processes or data. Object-oriented approaches facilitate a more balanced view of systems. Every process or action is explicitly associated with some entity and targeted at some entity. We say that "at the student's request, the registrar enrolls students in courses" rather than "enrollments happen," or "a student plus a course makes an enrollment." This more natural notion (that entities have an active role rather than a passive one) is directly reflected in the software and its design when systems are modeled and developed in an object-oriented manner.

1.2 Natural Methods of Organization

According to Encyclopedia Britannica, people constantly undertake three activities in organizing their thinking [12]:

1. The differentiation of experience into particular objects and their attributes - e.g., when they distinguish between a tree and its size or spatial relations to other objects.
2. The distinction between whole objects and their component parts - e.g., when they contrast a tree and its component branches.
3. The formation of and distinction between different classes of objects - e.g., when they form the class of all trees and the class of all stones and distinguish between them.

Object-oriented methods are based on applying these same principles in organizing software systems. Novices often express that they see nothing new about object-orientation. This perception seems to follow given that the foundation of object-orientation rests on thought processes that are inherent in all people. Perhaps all the hype about object-orientation has created the expectation that it is highly complex, esoteric, and difficult to understand. This simply is not the case. Perhaps the reason for the "nothing new" feeling is that while object-orientation is relatively new to software development, it is anything but new to people and their everyday thinking.

1.3 Programming Paradigms

One may wonder "What precisely is object-oriented, versus what is not object-oriented." A related question is "What is object-oriented *programming*, and what makes it different from other programming paradigms?" To answer these questions, it is useful to explore a continuum of programming paradigms

and their characteristics in terms of *encapsulation*, *abstract data typing*, and *inheritance* [65]. Each paradigm in the following discussion adds one of these characteristics resulting in an improvement over the previous paradigms.

Procedural programming is probably still the most commonly used approach to programming. The fundamental idea is to decide which procedures are needed to solve the problem and use the best algorithms you can find or create. The focus is on procedure design, with data design being driven by the requirements of the procedures. The only language features necessary to support this style of programming the abilities to pass arguments to and return values from functions.

Data hiding is a principal used to manage the complexity of programs by grouping related data and procedures together. The fundamental idea is to decide which modules are needed in order to partition the program so that data is hidden in modules with related procedures. The focus is on module design. A language must include some kind of module mechanism (e.g. the file in C, or the module in Modula 2) in order to support data hiding. A slightly stronger version of data hiding is *encapsulation,* in which module boundaries are strictly enforced. Encapsulation is discussed in detail in section 2.3.

Data abstraction occurs when an appropriately defined module may be considered to be a new data type in the language. Such types are called *abstract data types*, in contrast to the built-in primitive types and simple data structures. The fundamental idea is to decide which new types are needed to solve the problem, then provide a full set of operations for each new type. The focus is on the design of abstract data types, which often model entities observed in the problem domain. A language must allow the definition of user-defined types along with operators for these types, that is, it must have an extendable type system, in order to support data abstraction programming. ADA supports abstract data typing via its package mechanism.

Object-oriented programming is distinguished from data abstraction by the addition of *inheritance* to the formula. The fundamental idea is to decide what classes (abstract data types) are needed, provide a full set of operations for each class, and make commonalty among classes explicit by using inheritance. The focus is on designing a hierarchy of classes such that classes lower in the hierarchy are specializations of more general classes that are higher in the hierarchy. For example, the class "boy" is a specialization of the more general class "child". A language must provide syntax for expressing classes and a mechanism for inheritance in order to support object-oriented programming.

One way, then, to answer the question "What is and is not object-oriented?" is to respond: "Anything that allows the expression of, or is expressed in terms of, abstract data types using inheritance is object-oriented; anything that fails to meet either of those criteria is not."

1.4 The Diagram Editor

Part of the research supporting this paper was devoted to the application of object-oriented methods to the development of a simple diagram editor. Portions of the design and implementation of this system are used for illustration throughout the paper. Complete documentation of the development project is included in the appendix. This description is intended only to introduce some of the objects and classes that will be used in illustrations, therefore, some terms used here will not be defined until later sections.

The Diagram Editor is a graphical, mouse driven application intended to assist designers in constructing diagrams representing object-oriented designs. The notation supported by the editor is derived from the notation used in Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) [12],[13]. The user may create and manipulate symbols representing classes and objects by using a mouse to select choices from floating pop-up menus. Lists of properties (i.e. attributes and services) may be added to each class and object symbol. Connections linking symbols together can be created to represent various kinds of relationships among classes and objects. Each symbol, property, and connection may be annotated by typing text into a pop-up edit window.

Several key objects from implementation of the diagram editor are referenced frequently throughout the paper. These include the diagram and dispatcher objects, and the classes list, selectable, class, property, and connection. The diagram object maintains a list of all symbols and connections currently in the diagram and controls the order in which symbols and connections display themselves to the screen and save themselves to disk. The dispatcher object takes care of routing events (mouse clicks, keyboard input) to the appropriate screen objects.

The list class defines a general purpose "container" for storing references (pointers) to other objects. The dispatcher and diagram objects use list objects to keep track of the screen objects they manipulate. Symbol objects (class and object objects) use list objects to store property objects that represent the symbol's attributes and services. Figures 3 (page 10) and 4 (page 12) illustrate this information graphically.

2 Concepts and Terminology

Object-oriented methods are based on the object model (rather than the Object Model: there is no formally defined model with this title.) The object model underlies how one views the world (problem domain, solution space, etc...) when developing software. Many terms have become quite popular in referring to elements of the object model, some being more important and meaningful than others. Some terms refer to essential ideas of object-orientation; others refer to by-products. Some refer to useful mechanisms; others to intellectually pleasing intricacies that are otherwise of little value. The following paragraphs attempt to make these distinctions clear.

2.1 Objects

Terms: *object, attribute, service, message, method, operation, client*

The notion of an object in object-oriented analysis and design is the same as it is in everyday life. An object is a "thing." All things, such as the Empire State building or the Queen Mary, have certain qualities. Things have boundaries. There is a distinction between what is part of and what is not part of a particular thing. Things have traits, or *attributes*, such as color, size, and maximum capacity. Things can perform actions, or *services*, e.g. the Queen Mary can carry travelers across the ocean. Actions can be performed on things, e.g. the Empire State building can be painted. Objects can be very large and complex, made up of many parts, like the Queen Mary, or much smaller in scope, like the spark plug in cylinder #1 of a motorcycle's engine.

In the domain of software development, an *object* is something that has identity, state, behavior, and a boundary. The identity of objects in software systems comes in many forms [31]. Objects are often identified by location (i.e. their address in memory), by the value of one or more of their attributes, or by a unique identifier, sometimes referred to as a *handle*. To illustrate the appropriateness of each of these methods, consider three analogous identifiers for the Queen Mary:

Methods of Object Identification

- Location: "Berth 17, Port of Long Beach, Long Beach, California."
- Value: "Large luxury cruise ship with a black and red hull that is now used as a hotel, restaurant, and tourist attraction. "
- Id: "00125379108" (Arbitrary, but unique, integer identifier)

An object's state is represented by the values of its attributes. The Queen Mary's state at some point in time could be status="underway", capacity="2500" and that of my spark plug could be sparking="true" or sparking="false." These values may change over time, but only in a controlled manner. The set of values these attributes may assume, and thus the set of states the object may assume, is part of the definition of the object. For example, upon arrival at her destination, the Queen Mary's state could change to status="at port", or status="under repair", but not to status="Mary had a little lamb."

The behavior of an object is defined by the services, also referred to as *operations*, it can perform. For example, the Queen Mary can carry travelers across the ocean and my spark plug can spark. Objects

follow some *method* to carry out these operations. The term *method* denotes the implementation of an operation, and its use is intended to create the sense that the object somehow "knows" what it is doing. In software systems, operations are named. For example, the Queen Mary's "carry travelers across the ocean" operation might be named "go", and a spark plug's "spark" operation might be cleverly named "spark". In any case, every object has a set of operations with associated methods that define its behavior.

The assertion that objects have a boundary is meant to convey the idea that objects are finite and well defined. At least in software systems, there is no ambiguity in deciding what is and is not part of a particular object, and what a particular object can do. It is in this sense that objects are bounded.

Objects offer their services for use by other objects. An object which uses a service of another object is referred to as a *client* of that object. A client requests the services of another object by sending it a *message*, which consists of the requested service's name and any necessary parameters. A message denotes a service which corresponds to a method. The reasons for this separation of concepts is explained further in section 2.5.

One might notice the similarity between an object and a record in a database system. Each has a set of values which contribute to its state. Unlike records, however, functionality is part of an object. Semantics (meaning, behavior) are external to records, but internal to objects.

2.2 Classes

Terms: *class, abstract data type, instance, instantiate*

Earlier we established that the Queen Mary is an object, as are the Titanic and the S.S. Minnow. Although these objects share similar behavior, similar state spaces, and similar boundaries, they do not share identity, nor do they share state. For example, the Queen Mary is much larger than the S.S. Minnow, and the Titanic is neither "underway" nor "at port". Although different in specifics, each of these objects can still be described as a "ship". An object's *class* embodies all that the object has in common with all similar objects. The class definition describes the form and behavior of all objects of that class. All objects have a class, even if they are unique. Objects bear an "is-a" relationship to their classes, for example, as shown in figure 1, the Titanic has an "is-a" relationship to the class ship.

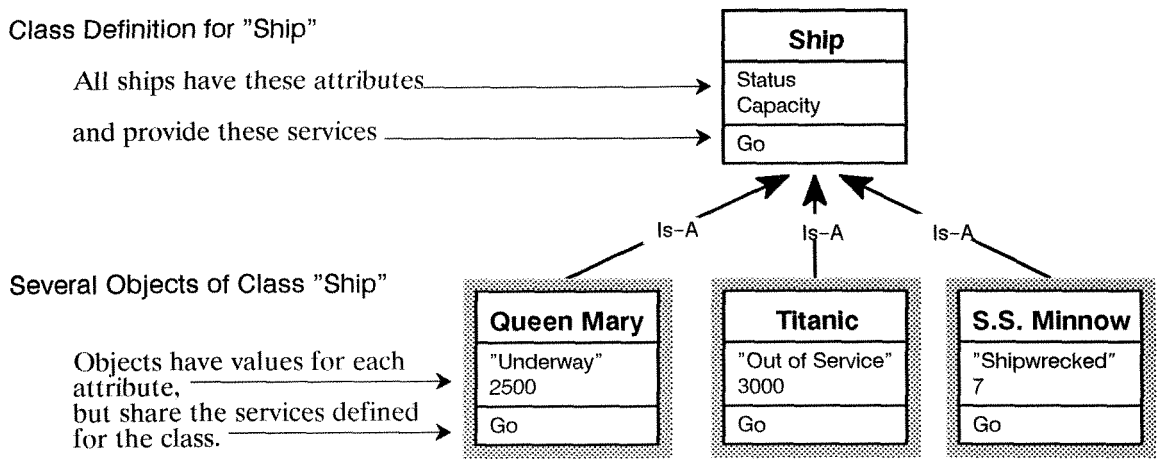


Figure 1: Class, Instance, and the "is-a" Relationship

A class may be thought of as a template, or pattern, for forming an object, or as an object factory (i.e. an object that creates other objects.) In any case, a class defines the structure and function of a potentially infinite set of individual objects. An object is an *instance* of its class, e.g. the Titanic is an instance of class ship. The act of creating a new object of a certain class is referred to as *instantiating* that class.

A class definition may also be thought of as a type definition. *Data abstraction* is the process of creating *abstract data types*, also referred to as user-defined types. This involves grouping data together with the operations that operate on, or otherwise involve that data.

One kind of object that occurs repeatedly in many applications is the list. Consider the following class definition for list:

Listing 1

```
// C++ Class definition for a linked list abstract data type
class list  {
    public:
        int insert(object *obj);    // add obj to beginning of list
        int remove(object *obj);    // delete obj from the list
        object *first();            // return first object in list
        object *next();             // return successive objects
    private:
        node_t head, *cursor;
    }
```

By virtue of this class definition, list is now a type that can be used in the same way more familiar types are used, such as integer and real. Variables (instances) of type (class) list can be declared and used just as variables of type integer can be. Abstract data types, such as list, can be distinguished from other data structures by the presence of a well defined set of operations to manipulate the state of objects of that type. Without these operations, abstract data types would be nothing more than records or other passive data structures. Classes are the vehicle of data abstraction in object-oriented analysis, design, and programming.

2.3 Encapsulation

Terms: *public, private, interface, implementation*

Restricting access to certain items of data via scoping mechanisms is called data hiding. This is useful in preventing unwanted dependencies from arising on data that is semantically internal to some unit (e.g. a module, object, or class.) Encapsulation supports data abstraction by helping to enforce rules associated with a particular abstract data type, and by establishing the "boundary" of the abstract data type. Encapsulation gives a sense of closure to a program unit, and can be thought of as "protection from outside manipulation."

To illustrate the importance and potential value of encapsulation, the following discussion relates some experiences from the development of the diagram editor system. The class list was written as part of the diagram editor. Many of the objects in the diagram editor have an object of type list as one of their components: the dispatcher maintains a list of all selectable objects that currently appear on the screen. The diagram object contains a list of all the class and object symbols and various connections that make up the diagram, and each class and object symbol has a list of attributes and a list of services. Many parts of the system depend on the list class.

The job of the dispatcher is to take the (x,y) position of a mouse click and search its list of objects in order, from first to last, for one whose representation on-screen contains this point. The list class definition guarantees that the list is ordered with the most recently inserted object being first in the list. If two objects overlap the most recently created one will be visually on top, and will appear in the dispatcher's list before the overlapped object . Thus, the dispatcher will always check the top object first and the correct object will be selected.

Later in the development it became apparent that a change would be necessary. One of the diagram object's jobs is to maintain the display. Occasionally it is necessary to re-display the entire diagram, one object at a time. List objects of the class defined above may only be traversed one way: from most recent to oldest, but displaying the objects in this order would cause stacks of overlapping objects to appear in reverse order. Despite all the dependencies on the list class, the solution was actually quite easy to implement. The list class was redefined (internally) to be a doubly linked list and two new operations, list::last

and list::prev were added. This allowed the list to be traversed in reverse order so that screen objects could be displayed correctly. *No* other changes to the system were required! The following code fragment shows the revised class definition.

Listing 2

```
// C++ Class definition for a doubly linked list abstract data type
class list_t  {
    public:
        int insert(object *obj);    // add obj to beginning of list
        int remove(object *obj);    // delete obj from the list
        int append(object *obj);    // add obj to end of list
        object *first();            // return first object in list
        object *next();             // return successive objects
        object *last();             // return last object in list
        object *prev();             // return previous object
    private:
        node_t head, tail, *cursor;
    }
```

Notice the addition of the append operation. Still later in the development it became necessary to be able to add objects to the end of the list to maintain proper ordering. None of these revisions to the list class required any changes to be made to any other existing code. This is because other objects rely only on the *interface* to the list class, and are independent of its *implementation*. The interface to a class consists of its *public* attributes and operations, that is, those that are visible to other objects. Data and operations used to implement the services of the class are *private*, or invisible to other objects. Since the old parts of the list interface remained constant, no other object was affected by the changes, even though the internal representation of the list changed significantly.

2.4 Inheritance

Terms: *base class, derived class, super-class, abstract class, ancestor, descendant*

Earlier we said that an object has an "is-a" relationship to its class. Classes can also have "is-a" relationships to other classes. In Zoology, animals are grouped into species. An object is to its class as a particular animal is to its species. Species are then grouped into genus, genus into families, families into orders, and so on, up to kingdoms (see figure 2). Classes are often generalized to form *super-classes*, and these classes can be further generalized to form still higher level classes. The terms *ancestor* and *descendant* are often used in the obvious way to refer to classes in multilevel inheritance hierarchies, such as that shown in figure 2. Classes that are lower in the hierarchy are more detailed, more specialized, and have more strict criteria for object membership.

In the previous discussion, we pointed out that the Queen Mary, the Titanic, and the S.S. Minnow were all ships. The U.S.S. Vincines and the U.S.S. New Jersey are also ships, but in addition to the similarities that these vessels bear to the others, there are some rather drastic differences. The Queen Mary, the Titanic, and the S.S. Minnow are all pleasure ships, whereas the Vincines and the New Jersey are war ships. In object-oriented terminology we say that pleasure-ship and war-ship are *specializations* of the more general class ship. Conversely, ship is a *generalization* of both pleasure-ship and war-ship. We can make use of this commonalty by defining the general class ship to have attributes and services common to all ships, then defining the classes war-ship and pleasure-ship in terms of what detail they add to the general class ship. For example, a war ship is a ship that has guns; a pleasure ship is a ship that does not allow guns. Pleasure-ship and war-ship *inherit* all the attributes and methods of ship: they appear to belong to pleasure-ship and war-ship, even though they were not directly defined for these classes. We say that pleasure-ship and war-ship are *derived* from, or are *derived classes* of, class ship, and ship is the *base class* of pleasure-ship and war-ship.

The diagram editor system allows a user to create various kinds of symbols on the screen, add various kinds of properties to these symbols, and link these symbols together using various kinds of connections. Each symbol, property, and connection is represented internally as an individual object. All of these objects share certain behaviors: they all may be displayed, they all may be selected (focused on)

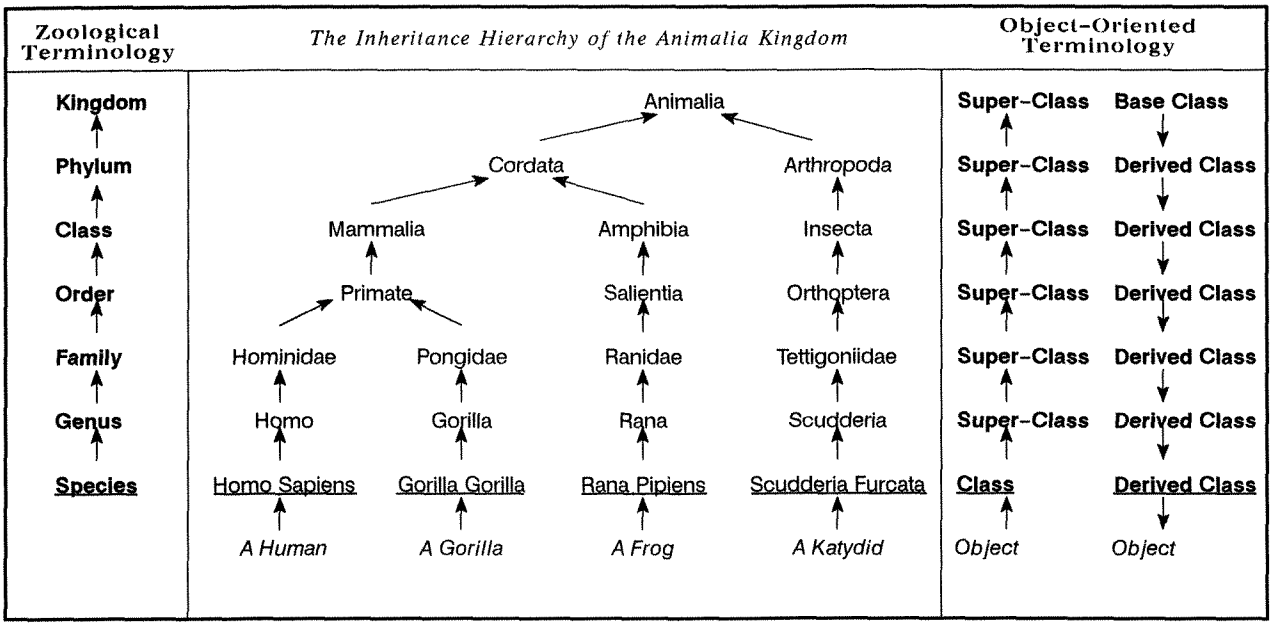


Figure 2: An Example of Generalization

with a mouse click, and they all may be acted upon (display a menu of operations that may be performed on that object, such as "move" or "delete"). They also share certain elements of state: they each have an (x,y) position on screen, and they each have a name. These are the essential characteristics of a "selectable" screen item in this system and are represented in the abstract class selectable. An *abstract* class is one that may not be instantiated directly, but serves only as a base class for one or more derived classes. The inheritance structure from the diagram editor system is shown in figure 3.

Modeling in this way is advantageous for several reasons. First it provides for a certain level of cognitive economy: once you have understood the base class, you then need only understand how each derived class differs from the more general base class. In essence, you say to yourself "OK. I already know what a ship is. A pleasure ship is just a ship that..." – already you know a lot about pleasure-ship – "...that is configured for entertaining travelers while at sea."

Second, using inheritance also provides for a certain level of code and design reuse. Inherited methods usually need only slight changes or may need no change at all. Thus the programmer writes fewer lines of code. Also, once a class is designed, derived classes are described only in terms of how they differ from (i.e what they add to or modify of) the base class; thus, the design of the base class is reused.

2.5 Message Passing

Terms: *message, method, binding, static binding, dynamic binding*

Earlier we said that an object requests the services of another object by sending (passing) a message to it. The message denotes a service offered by the receiving object, which corresponds to a method that is internal to the object.

Although passing a message is equated with calling a function or invoking a procedure in most languages, there are some differences in semantics. First, the term *message passing* implies some level of concurrency. One ends a message, then one goes about one's business, which may or may not include waiting idle for a reply. While this is not the case in most languages, the name *message* does allow for that possibility and further fosters the notion that objects are independent, active entities, which is central to object-orientation.

Second, message passing relies the possibility of *dynamic binding:* without dynamic binding, sending a message is no different than calling a function. When a module calls a function, control is passed to

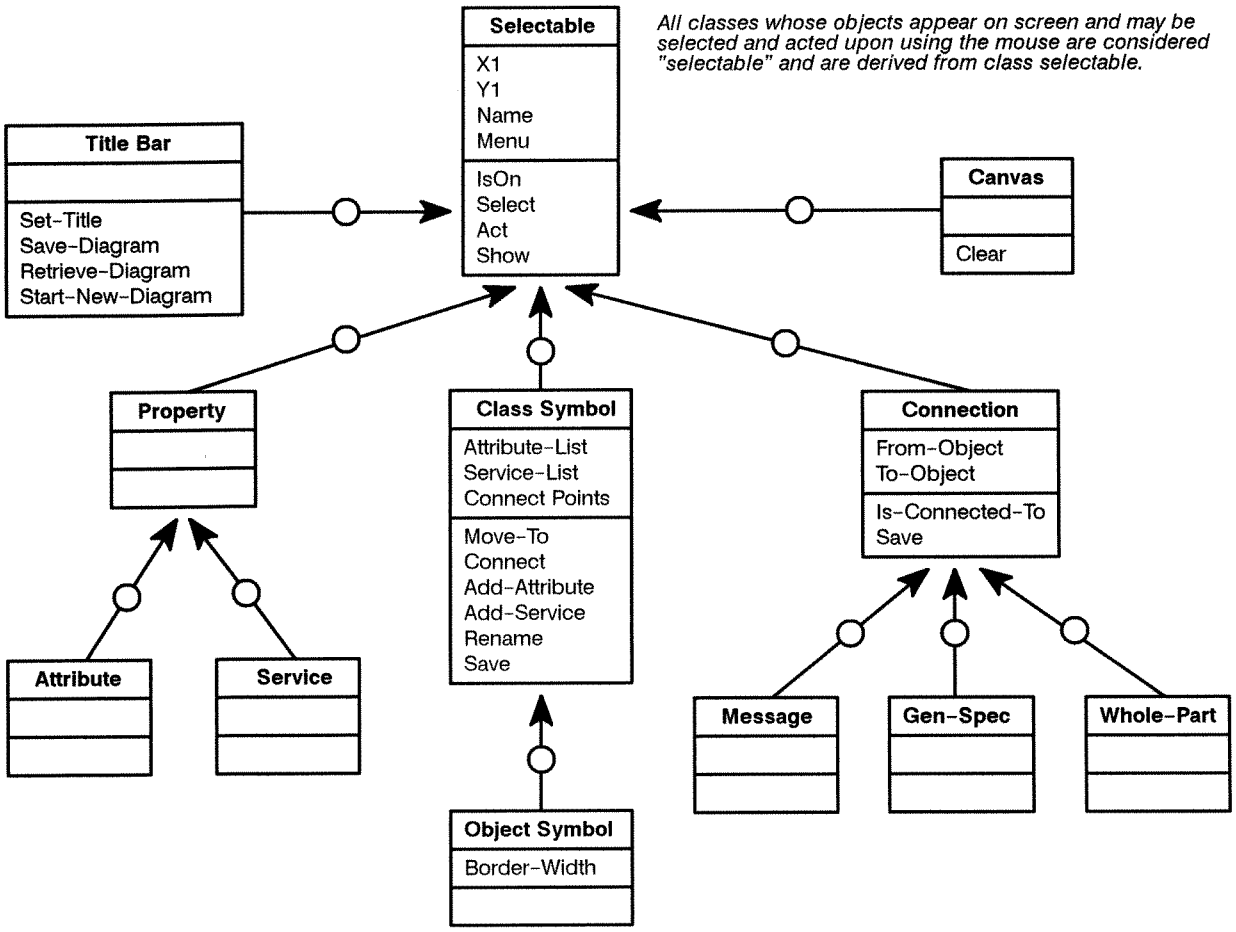


Figure 3: Inheritance Hierarchy for the Diagram Editor System

a statically defined point somewhere else in the program. Such functions are said to be *statically bound* because calls to them may be resolved at link time. When an object sends a message to another object, control may or may not be passed directly to a function. A message is a request for action, not a function call. It may be the case that any one of several different responses are possible, and furthermore, the appropriate response may not be determinable at compiler/link time. The process of determining which of the possible responses is appropriate then finally invoking the appropriate function is called *dynamic binding*.

In the diagram editor system, diagrams consist of one or more class/object symbols and zero or more connections of various types. Each symbol and each link in a diagram is represented internally as an object. The inheritance hierarchy of figure 3 shows that symbol objects (class, object) and connection objects (message, gen-spec, whole-part) are all descendants of class selectable, and thus, are type compatible with selectable. That is, they may be used in any context in which a selectable may be used in, and may receive and respond to any message that a selectable may receive.

Each of these classes provides the service show() which draws objects of that class on the screen. However, because objects of each class appear differently on the screen, each class has a different method which corresponds to the show() service. The list that the diagram object uses to store symbols and connections only contains objects of type selectable. Since the symbol and connection types are type compatible with selectable, the list will store them, but it interprets them as having type selectable. One of the functions of the diagram object is to refresh the screen image, which it does by first clearing the screen by sending the message clear() to the canvas object, then sending the message show() to each object in its

list. Since the diagram object knows these objects in the list only as selectables, it cannot know which method to invoke for a particular object. Luckily, it does not need to. It need only send the message show() to each object in the list and let the object itself decide how to respond, which will always be by invoking its internal method that corresponds to the message show(). The call to show() is dynamically bound.

Note that dynamic binding and inheritance are closely related: in otherwise statically bound languages, dynamic binding is only required in the presence of inheritance. Also note that without inheritance, or more precisely, without type compatibility, there is no need for dynamic binding. All function calls could be statically bound, hence the notion of message passing would be unnecessary.

2.6 Polymorphism

Terms: *parametric polymorphism, inclusion polymorphism, overloading, coercion*

Polymorphism is the "capability of assuming different forms" [47]. In the software domain, polymorphism is related to the notion of type. Typed languages, such as Pascal, are based on the idea that every value and every expression has a unique type, and hence, a single interpretation. Such languages are said to be *monomorphic*. In contrast, languages such as C++ and Smalltalk are *polymorphic*, meaning that some values and expressions may have more than one type.

There are at least four distinct forms of polymorphism: parametric, inclusion, overloading, and coercion [10]. *Parametric polymorphism* is exhibited by procedures that work uniformly for a range of types. The function list::insert() from the diagram editor system exhibits parametric polymorphism since it will produce the same result given an object of any class derived from class *selectable*. *Inclusion polymorphism* is exhibited by derived classes in that they may be interpreted as belonging to any of their superclasses. Thus an object of class service may also be interpreted as being of class property or of class selectable. Type compatibility is an example of inclusion polymorphism. Parametric and inclusion polymorphism are examples of *universal polymorphism*.

Overloading is a form of polymorphism exhibited when a single operator or function name may be applied to multiple types. For example, list::find is overloaded. It may be called with either a pair of integers (x,y screen coordinates) or with a character string as its argument, but performs the same function in both cases: locates an object and returns a pointer to it. One can think of an overloaded function as a function whose name includes the types of its parameters. The expressions '1 + 1' and '1.0 + 1.0' suggest that the + operator is overloaded to perform both integer and real addition. This gives the appearance of a polymorphic function or operator, but in fact, there are two functions called find and two + operators. The type of the argument is used to decide which function or operator to actually apply.

Coercion is used when values of different types are used in the same expression. In the expression '1 + 1.0' neither of the two overloaded + operators is applicable, so one or the other operands is converted, or coerced, into the type of the other. Coercion is an operation used to convert the type of an argument to the type expected by an operator or function.

Overloading and coercion are forms of *ad hoc polymorphism*. Ad hoc polymorphism is a weaker form than universal polymorphism. Ad hoc polymorphism could also be called "apparent" polymorphism. Overloading requires multiple bodies of code that are applied according to the type of the operand, whereas parametrically polymorphic functions execute the same body of code for all valid argument types. Coercion causes the representation of some object to change, whereas types that are inclusion polymorphic are simply considered "compatible" -- no change of representation takes place.

Dynamic binding and type compatibility are what allow objects to exhibit polymorphism in object oriented systems. Polymorphism is not a defining quality of object-oriented systems, rather it is a by-product of it. Overloading and coercion do contribute, but not to the point that one is justified in saying "X is object-oriented because it is polymorphic." Polymorphism is not as central an issue as it would seem from the literature.

2.7 Composition

Terms: *component, aggregation, nested object, decomposition, delegation*

Objects are composed of attributes and associated methods. An attribute may be a simple value, such as status="Underway", or it may be some complex structure, such as another object. Such contained

objects are referred to as *component* objects. If we were to build an object-oriented model of the Queen Mary, we would probably have one object called Queen-Mary that is composed of several smaller component objects, such as Engine, Cargo Bay, and Helm. These objects are also occasionally referred to as *nested objects*. Care must be taken here: while the object Engine is nested within the Queen-Mary object, Engine's class may or may not be nested within Queen-Mary's class. Many other objects of many other classes may also contain engines. A truly nested object is one whose class is visible only to the containing object. This rarely occurs.

There are two classes in the diagram editor system whose objects serve as components to several classes: list and menu. The diagram and dispatcher objects both contain lists, and all the symbol objects contain two lists (one for attributes, one for services.) Every object of class selectable contains a menu object. Figure 4 illustrates this graphically. Arrows in this diagram denote *instance connections,* which

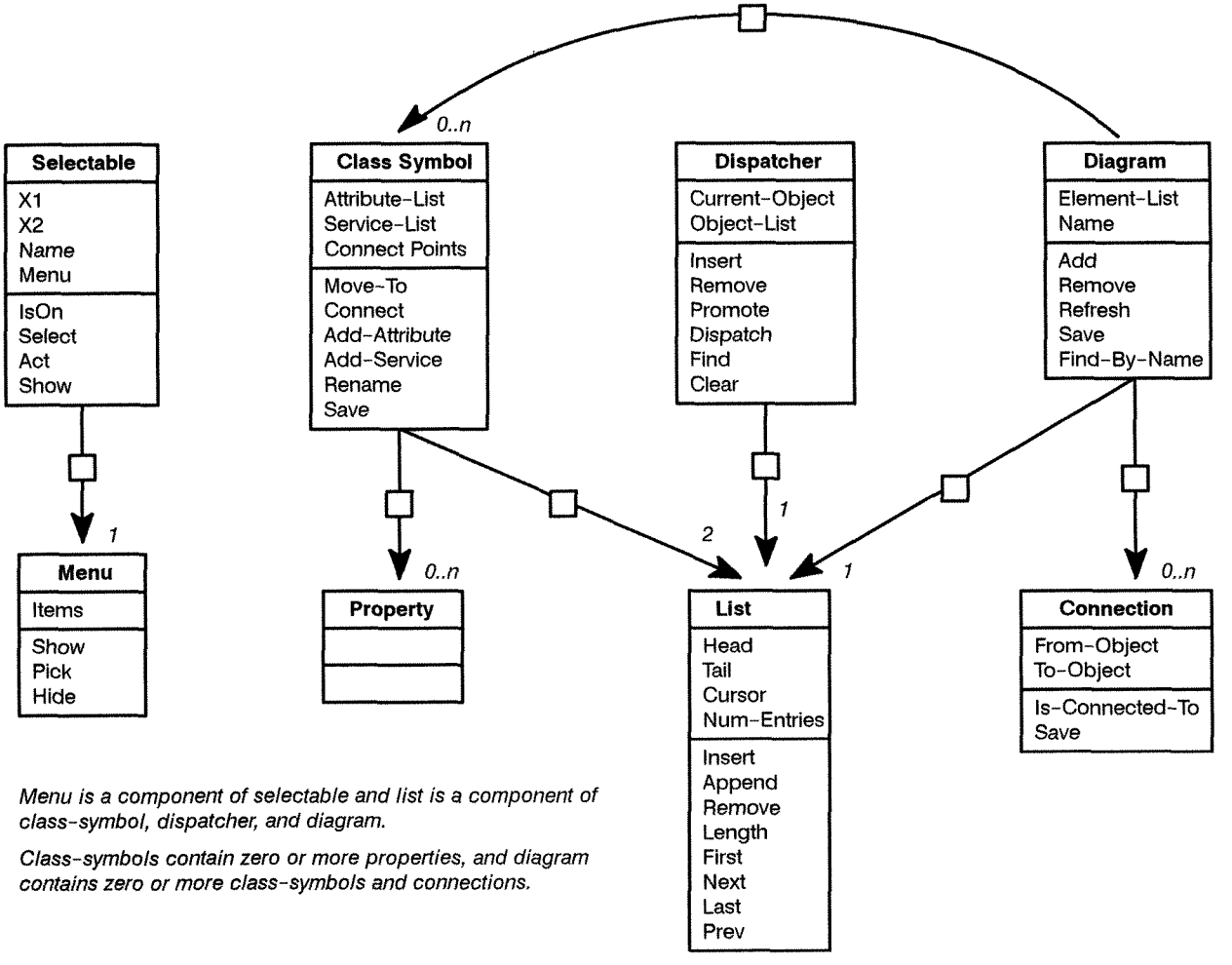


Figure 4: Composition and Containment in the Diagram Editor

mean that objects of the origin (of the arrow) class contains some number of objects of the destination class. For example, dispatcher contains exactly one list and class-symbol contains 0..n properties.

Notice that the class definitions for selectable, class-symbol, dispatcher, and diagram in figure 4 all contain attributes naming the menu and list objects they contain, e.g. class-symbol.attribute-list names a list object that is a component of objects of type class-symbol. However, there are no attributes in class-symbol or diagram which correspond to the property and connection objects they may contain. That is because no object of type property or connection is actually a component of any class-symbol or diagram object. Class-symbol objects act only as containers for property objects. The definition of

class class-symbol is independent of the definition of class property. A class-symbol object may exist whether or not any property object exists, however it cannot exist without at least two list objects. This is analogous to the fact that a lid is part of a cookie jar, but a cookie is not.

Composition and inheritance are often confused, partially because the term *sub-class* is used to refer to both the class of a component object and to a derived class. Component objects, along with other data items, are combined with methods to form *aggregate* objects. Complex objects are *decomposed* into smaller objects. However, each object - the aggregate and each component - maintains its identity and has existence. When an object of class B, which is a derived from class A, is created, no object of class A necessarily exists. Only an object of class B must exist, although the B object may be interpreted as having type A. B is a special case of A; A is a generalized form of B; B is a transformation of A; B is not a component of A.

Inheritance may be simulated using a method called *delegation* [64], which is based on composition. One way of looking at inheritance is that the derived class is an instance of the base class with some behavior modification and possibly some additional elements of state. In this way, one can form a specialization class without inheritance by defining a new class to have an object of the base class as one of its components. The data and methods of the base class are now available to the new class as if they were inherited. Additional data, and modified or new methods, are supplied by the aggregate class. When a message is received requesting an "inherited" method, the request is simply delegated to the component "base" object. The result of delegation is similar to the result if true inheritance were used, although type compatibility is lost.

Note that although we often refer to components as component classes or sub-classes, components are always objects. One might say "one component of class A is an object of class B" but never "class B is a component of class A".

2.8 Generic Typing

Terms: *generic type, genericity, template, parameterized type, type parameter*

Genericity is a trait exhibited by *generic types*, also referred to as *parameterized types*. A parameterized type is one whose methods contain code that operates on objects of a type that remains unknown until an object of the generic type is declared with a *type parameter*. A type parameter is a argument which denotes a type, as opposed to denoting a value. Genericity is most useful with respect to container objects [15], such as arrays and lists. To illustrate the usefulness of generic typing, consider the following definition of a linked list to store floating point numbers:

Listing 3

```
// C++ class definition for a linked list of floats
class list_of_float {
    public:
        int insert(float obj);          // insert obj at the head of the list
        int remove(float obj);          // remove obj from the list
        float first(void);              // return the first object in the list
        float next(void);               // return the next object in the list

    private:
        struct node_t {                 // node_t is a struct for list nodes
            float obj;                  // the object to be stored
            struct node_t *next;        // pointer to the next node in the list
            };
        struct node_t head, *cursor;    // list head, and a pointer for iterating
    };
```

The operational characteristics of a linked list are identical no matter what type of object the list is intended to contain. One could create a new list class to contain integers, or strings, or any other type of object simply by changing every occurrence of float in the class definition above to the desired type name, such as int or char *. An easier way is to design a parameterized list type. The type of the object

to be contained is then an argument in the declaration of each list object. The following is a parameterized list class in C++ syntax[1]:

Listing 4

```
// C++ class definition for a general purpose linked list type
template <class T> class list {          // T is the type arg, list is the class name
    public:
        int insert(T obj);               // insert obj at the head of the list
        int remove(T obj);               // remove obj from the list
        T first(void);                   // return the first object in the list
        T next(void);                    // return the next object in the list

    private:
        struct node_t {                  // node_t is a struct for list nodes
            T obj;                       // the object to be stored
            struct node_t *next;         // pointer to the next node in the list
            };
        struct node_t head, *cursor;     // list head, and a pointer for iterating
    };
```

This parameterized list class, or *template* in C++ terminology, is generic in the sense that it may be instantiated to form list objects that can store objects of any single type. The following code fragment shows how one would create several lists to contain different types of objects.

Listing 5

```
// Instantiate lists to contain integers, symbols, and boats
//   -- note that symbol_t and boat_t are defined elsewhere

list<int>       integer_list;
list<symbol_t>  symbol_list;
list<boat_t>    boat_list;
```

A parameterized list class is useful when the actual list object is to contain objects of any single type. However, if the list must contain objects of varying type, some other scheme must be used, such as using inheritance to create compatible types.

Functions and procedures may also be type parameterized, as in ADA. Such mechanisms make it easy to create overloaded operators and functions. Consider the following ADA subprogram declaration:[2]

Listing 6

```
-- a generic ADA subprogram to swap two objects of type T
generic
    type T is private;
procedure swap (x, y: in out T) is
    temp: T
begin
    temp := x; x := y; y := t;
end swap

procedure int_swap is new swap(INTEGER);
procedure sym_swap is new swap(SYMBOL);
procedure boat_swap is new swap(BOAT);
```

---

1. Parameterized types in C++ are experimental and not yet supported by most C++ compilers. The syntax shown here is a slightly modified version of that described in [19] which is to serve as the base document for an ANSI standard C++.

---

2. The syntax of this example is not correct ADA: the header and body are combined here for ease of presentation.

The procedure definition here is similar to a class definition in the sense that it defines a pattern for a class of procedures that swap things. It is like a type, but a procedure type rather than a data type. The last three lines show how special purpose swap procedures could be declared.

3 Comparative Analysis of the Elements of the Object Model

People often find object-oriented thinking and methods difficult to learn. Part of the problem is that several elements of the object model seem very similar to each other. Despite the apparent similarities, each of the elements described section 2 is distinct, and each is essential in supporting some facet of the object model. The following sections attempt to bring out some of the more subtle aspects of the object model and resolve some common confusions.

3.1 Inheritance versus Generic Typing

Generic typing and inheritance are two mechanisms often found in object-oriented languages. Both support the production of highly extendible and reusable software components. Some of the effects of each may be simulated using the other, but both features must be supported by a language in order to gain full advantage [43].

One major effect of the use of inheritance is polymorphism - the ability to send the same message to a range of types with different outcomes. This effect is easily simulated with overloading, which is supported by generic typing. The problem with overloading is that there must be a separate procedure defined for each type to be operated on, and a new procedure must be created in order to add a new type to the set of possibilities. With inheritance, however, one need only create the new compatible type and the existing procedures will work unchanged. Any number of new types may be derived from some existing class to extend the range of possible valid types.

In creating the list class in the diagram editor system, several options were explored. Either generic typing or inheritance can be used to create a generic list class. In either case, only one list class need be written. The approach that should be taken depends on how the list objects will be employed.

Using inheritance, one can define a list class that is to store objects of class selectable. This list can then contain not only selectable objects, but objects of any class derived from class selectable, and do so simultaneously. The drawback is that the implementation of the list class only has access to the features offered by the lowest common denominator, class selectable. The added features that an object of some class derived from selectable might have are invisible to the list object. If generic typing is used to create a parameterized list class instead, a list object can be created to contain objects of a particular class. This list would then have access to all the public attributes and services offered by objects of that class. The drawback is that the list may contain objects of that class only. Actually, it may contain objects of any class derived from that of the type argument, but as far as the list is concerned, they are objects of the class of the type parameter.

Inheritance is the mechanism of choice when a heterogeneous container is needed. This is the more flexible solution and provides for easier extensibility. Generic typing is more appropriate when a homogeneous container is needed. This allows for stronger type checking (fewer compatibility rules apply) and results in a more highly reusable software component.

3.2 Inheritance versus Composition

These two mechanisms form distinct, but easily, confused hierarchies that often appear together in object-oriented systems. The confusion arises partly because the ambiguous usage of the term *sub-class* to refer to both a derived class and a component class. For example, consider a class called vehicle. Engine and automobile might both be called sub-classes of vehicle. The difference is that engine is part of a decomposition of vehicle, whereas automobile is a specialization of vehicle. Forsaking the use of

the term *sub-class* in favor of the less ambiguous and more accurate terms *component class* and *derived class* would do much to eliminate this confusion.

3.3 Inheritance and Encapsulation

A subtle point of interest is the fact that the use of inheritance may compromise the level of encapsulation that a particular class exhibits [63]. Languages support encapsulation by allowing access to an object only through its external interface, i.e. other objects may only access those attributes and services that are defined in the public section of the object's class definition. This interface also serves as a contract between the class with any potential client (class which uses the resources of the first class). By declaring certain attributes and services to be public, the class is in effect guaranteeing that these attributes and services will remain constant.

The use of inheritance introduces a new category of client which can breach the shell of encapsulation. In most languages, derived classes have access to parts of their base class that other clients do not. This introduces the possibility that the implementation of a derived class may be affected if some otherwise encapsulated part of the base class is changed, which defeats the purpose of encapsulation in the first place.

One solution is to use delegation instead of inheritance to specialize classes. Derived classes become nothing more than regular clients of base classes and, thus, no longer have access to non-public attributes and services of the base class. One disadvantage is that delegation does not encompass type compatibility, so much flexibility is lost. Another disadvantage is that derived classes formed using delegation do not automatically inherit the attributes and services of the base class, but must provide their own method (even if all it does is call the base class's method) for every method it wishes to inherit.

Occasionally it is appropriate for a derived class to have access to more of the base class than a normal client would. A better solution is to create a another level of access control between public and private. Attributes and services at this level would be accessible to derived classes but not to other clients, while private members of a class are not accessible at all outside the class. C++ implements this solution by offering three sections in class definitions: private (visible only to the class itself), protected (visible to the class and all descendants of the class), and public (visible to all classes).

One could also be of the opinion that because of the "is-a" relationship between a derived class and its base class, it is appropriate for the derived class to make use of all facilities of the base class and to change with the base class. This is a rather extreme position and any added conceptual integrity is surely offset by the compromise of proven principles of software engineering.

3.4 Classification of Services

Several object-oriented design methods include a step in which relationships between objects and services are identified and cross-referenced to determine the completeness of the model. Types of relationships for this purpose include "provides", "suffers", and "uses". An object may either provide, suffer, or use a service, or have no relationship to it at all. For every service that an object uses, some other object must provide that service. Enforcing the converse may reduce the complexity of the system by limiting the total number of services to precisely those that are needed, but this could hamper the production of general purpose, reusable software components.

The "suffers" relationship from object to service, which was introduced in [7], is difficult to understand. It could be interpreted as denoting that the service is performed on the object, that is to say, the object is a passive operand of some service provided by another object. The intended meaning of this term is not known to this author, nor was it discernable from the cited text.

3.5 Static versus Dynamic Binding

Dynamic binding is used in the presence of inheritance to ensure that the correct response to a given message (the correct method, function call) is given. Dynamic binding adds a great deal of flexibility to a programming language but also introduces a great deal of overhead. When a message is sent to a

particular object, a table look-up is required to determine which of the possible functions to invoke in response. If performance is a factor, this overhead can be intolerable when it is not necessary. Not all function calls require dynamic binding. Those that can be resolved at compile time should be in order to minimize overhead. Some mechanism is needed to identify those functions which will be dynamically bound and those which will not.

Every message in Smalltalk is dynamically bound [21], which is one reason for its notoriously slow performance. In contrast, languages such as Pascal and C define all functions calls to be statically bound, that is, resolved at compile/link time. C++ provides a keyword, *virtual*, for explicitly identifying those functions that will require dynamic binding. Other languages accomplish this automatically by syntax analysis.

To summarize, dynamic binding adds much flexibility to a language and supports the development of highly extendable software components, but there is a definite trade-off with efficiency. If efficiency is a factor, static binding is preferable.

4 Advantages of the Object-Orientation

This section discusses some of the potential benefits of applying object-oriented thinking and methods to system modeling and software production. Each of these advantages is achievable with conventional languages and methods if discipline is exercised. Object-oriented languages, methods, and thinking support these advantages and make them easier to achieve.

4.1 Reusability

The class is the unit of modularity in object-oriented software, just as the integrated circuit (IC) is the unit of modularity in electronic devices. ICs are generally encased in resin, so hardware designers have no choice but to utilize and trust the external interface provided by the IC and ignore the details of its internal workings.

Encapsulation lends a similar sense of closure to software modules. Encapsulated modules, or software ICs [15], may be reused, added to a system other than the one they were developed for, with confidence in the fact that they will perform as expected. If multiple applications are developed in a single problem domain, a greater number of more highly specialized software ICs are likely to be developed. Over time, the degree of reuse increases, and the focus shifts from development of new software ICs to refinement of existing ones. This will be discussed further in the context of frameworks.

Inheritance also supports reusability in that an existing software IC can be used as a base class for a specialized, tweaked and tuned, version of the class to fill a special need. Very little new code may need to be written.

The bottom line is that the greater degree of reuse you can achieve, the fewer lines of code you will write to perform a given task. If this higher level of reuse results in less time and effort being spent on a task, then productivity is increased for that task. Object-orientation supports a high degree of reuse.

4.2 Managing Complexity

An object-oriented approach helps control complexity in all aspects of software development. Initial modeling phases benefit from the fact that object-oriented methods tend to produce models that draw directly on the vocabulary of the problem domain. Such models are inherently easier for users to understand and promote a higher level of communication between developers and users. Design phases benefit in several ways. First, earlier models are often directly reusable. Second, using classes as the basic unit of modularity, combined with a high degree of encapsulation, promotes well organized systems that are structured around many modules of relatively small scope. In addition, this structure is likely to have a high degree of correspondence to the real world. As a result, systems may be apprehended in the same way we apprehend the real world: in terms of entities that do things.

4.3 Stability

Object-orientation can give conceptual integrity to a software development effort from beginning to the always elusive end. Classes identified as part of the problem domain during domain analysis are likely to end up as part of the implemented system. Models developed in the initial phases of a project can be built upon and expanded to accomplish the goals of later stages. Because the initial objects are taken directly from the problem domain, the overall structure of a given system remains largely intact throughout the entire life of the system.

Another characteristic of object-oriented systems is that perceived complexity tends to increase less dramatically as development progresses. Objects provide a solid foundation on which to build. Each stage of development tends to result in a relatively solid platform for the next stage to begin, even though stages may not be well defined. This idea can be illustrated by imagining the difference between climbing a ladder and bouncing progressively higher on a trampoline to reach the top a tall building.

Due to encapsulation and typically small granularity, it is more feasible to test classes exhaustively in isolation, and to have confidence that the results will hold when the class is placed into new contexts. Systems built from stable components are likely to be easier to test and more stable than systems built from scratch.

4.4 Maintenance

Maintenance activities can be divided into three major categories: debugging, modifying, and extending. The key to debugging, i.e. correcting an error, is locating the source of the error. The design of an object-oriented system is reflected in the code itself, in the class definitions, which can act as a system road map in searching for the source of errors. This is more effective than using conventional design documents for several reasons. First, these documents are rarely current, especially after a system has been released. Using the code itself as the design reference guarantees that all changes will be apparent.

Because of encapsulation, refinements and other modifications can be made without having to worry about side-effects, i.e. unexpected effects to unrelated parts of the system. In addition, extensions to the system are made easier via inheritance.

5 Impact of the Object Model on Software Lifecycle

The object model can serve as a foundation, a mindset, or an orientation, that underlies one's approach to software development. Activities throughout the lifecycle may be undertaken in terms of objects. Object-oriented development relies heavily on flexible shifting back and forth among various activities, and especially on prototyping to gain insight to feed back into higher level analysis and design activities.

Object-orientation is compatible with existing life cycle models. Life cycle models, such as the waterfall model [59] and the spiral model [5], prescribe how development activities are to be organized. Object-orientation states only how these activities are to be conducted, not necessarily how they are ordered. Regardless of the life cycle model used, certain kinds of activities will take place at some point during a development effort. These activities include analysis, design, implementation (coding), testing, and maintenance. The object model serves as a consistent underlying theme that smooths the interaction between types of activities and fosters a higher level cohesiveness and conceptual integrity throughout the entire development process. A single "orientation" or mindset is employed throughout the life of a system. The following paragraphs describe the impact of object-orientation on each of these activities.

Analysis Objects and operations from the problem domain comprise the initial model. These high-level objects can be used to organize the exploration for requirements. The domain model itself is directly reusable as a starting point for design.

Design Design activity is greatly aided by the fact that the one does not have to start with a clean slate. Design efforts build directly on the results of problem domain analysis by expanding the model to to encompass such design concerns as user interface and data management. More emphasis is placed

on designing highly reusable software components. The overall system structure is largely established by the domain model.

Implementation Encapsulation, generic typing, and inheritance promote a high degree of reuse in the construction of object-oriented systems which results in fewer lines of code being written, and thus higher productivity if reusable classes are readily available. There is a high start-up cost, however. These reusable components must be developed initially, which may actually require more effort than developing specialized components for the system at hand. The real payoff comes with a long term commitment to object-orientation.

Testing Encapsulation and a high degree of modularity ease the organization and implementation of testing efforts. The unit of test is the class. Object-oriented systems are typically composed of a few high level objects that are themselves composed of objects of successively smaller scope. The structure of the system itself suggests an plan of attack for testing.

Maintenance Encapsulation and the typically small granularity due to the use of classes as the unit of modularity limit both the extent of errors and the scope of their effect. The fact that the design of a system is explicit in its code means that maintenance personnel will always have a current reference to guide them in their search for errors. Encapsulation also limits the effect that modifications and extensions to the system may have. Inheritance greatly aids in extending a system. New functionality may be added to a range of types with little effort using inheritance. New types may be added without requiring changes to existing code via inheritance (type compatibility) and dynamic binding.

The effect of adopting an object-oriented view of software development is the potential unification of the entire development process. Many methods have been published prescribing how to conduct most of the activities discussed above in an object-oriented manner, particularly design [6],[13] and, more recently, analysis [12][62]. Several methods attempt to merge object-oriented and structures techniques [68],[69], possibly to provide an avenue for steadfast structure-oriented developers to wean themselves away from their atiquated ways. In any case, there is much interest to in the object-oriented paradigm both in industry and in the literature, and with good cause, as this paper has tried to show.

6 Summary

Object-orientation is a mindset for software development based in part on the natural methods of organization that all people employ in organizing their thinking. It involves viewing systems as collections of active entities, having both behavior and state, and organizing these entities and entity types into hierarchies to explicitly represent and take advantage of their commonalty. This paper has attempted to provide a basis for truly understanding and appreciating the potential value of this paradigm.

6.1 Lessons Learned

One major lesson learned from the reading and reflecting which underlies this paper was the need to distinguish between the intellectually pleasing and the truly important. There are many concepts associated with object-orientation that are fascinating to puzzle over, such as the subtle differences among forms of polymorphism, or the meaning of multiple-inheritance in real world terms. However, these concepts are not of central importance to making use of, and benefitting from, object-orientation.

The development of the diagram editor gave rise to several insights. First, object-oriented programming is more an act of arrangement than of synthesis. One tends to spend a little time initially specifying classes, then a lot of time shifting functionality around until a satisfying solution is obtained. A related observation is that no amount of foresight can replace prototyping. Some aspects of a system simply do not become apparent until their absence begins to inflict pain, which is invariably well into implementation.

Enhanced reusability and extensibility due to the use of object methods is a recurring theme in the literature. Unfortunately, these statements are rather difficult to identify with simply from reading and talking. One almost cannot understand the full impact of these words without first-hand experience.

To relate one more short example from the diagram editor system, the systems requirements called for the ability to add descriptive text to any or all symbols, connections, attributes, and services in a diagram. Implementation of this was left until the end, although the system was designed with this requirement in mind from the beginning. A simple text editor object was developed and tested independently. Integration of the text editor object into the diagram editor required about 15 minutes and 10 lines of code. This incredibly easy integration was made possible by the presence of encapsulation and inheritance. Encapsulation guaranteed that the introducing the editor object into the system would have no unseen effects, and inheritance allowed the code that called on the editor to be grafted in at the top of the inheritance hierarchy and simply inherited by every other object that needed text editing capability.

There is real value in assuming an object-oriented point of view in software development. Developers need only be willing to change their ways and move ahead.

6.2 Future Work

The next logical step is to expand this paper to include a detailed discussion of several popular object-oriented analysis and design methods, and and overview of object-oriented languages. With these addition, this paper could serve as a complete introduction to object-orientation.

Many of the assertions about the relative importance and potential advantage of various aspects of object-orientation are derived from the limited experiences of the author. A study conducted to collect more extensive experimental evidence to support these claims would be valuable.

Several enhancements could be made to the diagram editor to make it a more useful tool. Some of the more interesting enhancements are listed here:

- Add scroll bars to allow the creation of diagrams that are bigger than the screen. Allow different views of diagrams, such as overview (with attributes and services toggled off), and inheritance (with all other types of connections toggled off,) and so on.
- Implement four objects: screen, printer, keyboard, and mouse, in such a way that the diagram editor can rely exclusively on these objects for console and hardcopy I/O. The diagram editor could then be easily ported to other systems by simply re-implementing these four objects. If the screen and printer objects offered similar interfaces, WYSIWYG hardcopy output would also be easy to obtain.
- Provide for the specification of type information for attributes, and argument and return types for services. Given this type information, it would be relatively easy to extend the diagram editor to generate compilable class definitions and method stubs in, say, C++ or possibly ADA syntax.
- Generalize the system to support user-defined notations. This would include the development of a language for the specification of notations.

Ultimately, the diagram editor could be expanded into the realm of CASE for object-oriented development. This would be a fascinating, although time consuming project.

References

1. Bailin, S. C., "An Object-Oriented Requirements Specification Method", *Communications of the ACM*, 32(5), May 1989, Page 608

2. Barnes, J. G. P., "An Overview of ADA," *Software - Practice and Experience*, Vol. 10, 1980, Page 851

3. Beck, K; Cunningham, W., "A Laboratory for Teaching Object-Oriented Thinking", *ACM SIGPLAN Notices*, 24(10), 1989, p. 1-6

4. Bobrow, D. G; et. al., "CommonLoops: Merging Lisp and Object-Oriented Programming," *ACM SIGPLAN Notices*, 21(11), November 1986, Page 17

5. Boehm, B. W., "A Spiral Model of Software Development and Enhancement", *ACM SIGSOFT Software Engineering Notes*, 11(4), August 1986, p. 14-24

6. Booch, G., *Object Oriented Design with Applications*, Benjamin/Cummings, 1991

7. Booch, G., "Object-Oriented Developement", *IEEE Transactions on Software Engineering*, 12(2), February 1986, Page 211

8. Borgida, A; Greenspan, S; Mylopoulos, J., "Knowledge Representation as the Basis for Requirements Specifications", *IEEE Computer*, April 1985, Page 82

9. Bulman, D. M., "An Object-Based Development Model", *Computer Language*, 6(8), August 1989, p. 49-59

10. Cardelli, L; Wegner, P., "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Surveys*, 17(4), December 1985, Page 471

    Explores the interactions among the notions of type, data abstraction, and polymorphism using a lambda calculus-based model. Distinguishes static and strong typing, and describes four forms of polymorphism that are often exhibited in modern programming languages.

11. Canning, P. S; Cook, W. R; Hill, W. L; Olthoff, W. G., "Interfaces for Strongly-Typed Object-Oriented Programming", *ACM SIGPLAN Notices*, 24(10), 1989, p. 457-467

12. Coad, P; Yourdon E., *Object-Oriented Analysis - Second Edition*, Prentice-Hall, 1991

    Describes a method and associated notation for domain analysis and problem space modelling in terms of classes/objects and inheritance, composition, message, and association relationships. A

very practical book. The authors relate numorous experiences with the method and give many heuristics and bits of wisdom.

13. Coad, P; Yourdon E., *Object-Oriented Design*, Prentice-Hall, 1991

Describes a design method that is an extension of their analysis method, OOA. In addition to the five layers of the model introduced in OOA, OOD adds another dimension to the model in he form of four components: problem domain, user interface, task management, and data management.

14. Coad, P., "Object-Oriented Analysis", *American Programmer*, special issue on object-orientation, 2(7,8), Summer 1989

Introduces OOA, the method and notation later described in their 1991 book, noted above.

15. Cox, B. J., Object Oriented Programming - An Evolutionary Approach, Addison-Wesley, 1986

One of the first widely read books on object programming. Introduces Objective-C, an extension of C which includes object definition and message expression facilities similar to those found in Smalltalk-80. Represents reusability as the most important benefit of object oriented programming and discusses the notion of the "software IC" and how it promotes reuse.

16. Cunningham, W; Beck, K., "A Diagram for Object-Oriented Programs", *ACM SIGPLAN Notices*, 21(11), November 1986, Page 361

Introduces a notation for representing message sending dialogs between objects in object-oriented systems. Describes a mechanism for automatically generating these diagrams from Smalltalk-80 code.

17. Dahl, O. J; Nygaard, K., "SIMULA - An ALGOL-Based Simulation Language," *Communications of the ACM*, 9(9), September 1966, Page 671

18. Danforth, S; Tomlinson, C., "Type Theories and Object-Oriented Programming",

19. Ellis, M; Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley, 1990

Comprehensive description and definition of C++. Includes descriptions of some yet to be released features, including parameterized types (true genericity,) and exception handling mechanisms.

20. Gibbs, S; et. al., "Class Management for Software Communities", *Communications of the ACM*, 33(9), September 1990, Page 90

21. Goldberg, A; Robinson, D., *Smalltalk-80 - The Language and its Implementation*, Addison-Wesley, 1983

22. Halbert, D. C; O'Brien, P. D., "Using Types and Inheritance in Object-Oriented Programming", *IEEE Software*, September 1987, Page 71

Describes how to identify types (classes) and organize them into a tyoe hierarchy with the cominality factored out into the higher levels. Gives a clear description of inheritance and multiple

inheritance and gives some insight as to when to apply them. Explores the criteria to consider when adding functionality and how decide among various options, such as creating a new type, modifing an existing type, and so on.

23. Henderson-Sellers, B.; Edwards, J.M., "The Object Oriented Systems Life Cycle", *Communications of the ACM*, 33(9), September 1990, Page 142

24. Hewitt, C. E; Atkinson, R., "Synchronization in Actor Systems", *Proceedings of the Conference on Principals of Programming Languages*, January 1977, p. 267-280

25. Hickman, C. P; Roberts, L. S; Hickman, F., *Integrated Principles of Zoology, 7th Edition*, Times Mirror/Mosby College Publishing, 1984

26. Jalote, P., "Functional Refinement and Nested Objects for OOD", *IEEE Transactions on Software Engineering*, 15(3), March 1989, Page 264

    Presents an object-oriented design method extended to include functional and object refinement. The method stresses the development of a "transformation function" (the main program) and the decomposition of objects into nested, and hidden, sub-objects.

    This method is a compromise between traditional and pure object-oriented methods in which the development of the main driver, which combines the system objects and controls overall program flow, is explicitly included in the design. It is not clear to me that the idea of nested objects is very important. While useful, it seems to me that it would inhibit reusability.

27. Jordan, D., "Implementation Benefits of C++ Language Mechanisms", *Communications of the ACM*, 33(9), September 1990, Page 61

    Gives a cursory description of the many features of C++ that make it an enhancement of C and support object-oriented programming. Features described include function name overloading, inline functions, reference parameters, constants, the class construct, operator overloading, constructors and destructors, scoping, inheritance, virtual functions, and member access control.

28. Kaehler, T; Patterson, D., *A Taste of Smalltalk*, W. W. Norton & Company, 1986

    A hands-on introduction to the Smalltalk environment. Illustrates many features of the environment, as well as some Smalltalk-80 sytnax, through the development of and a series of refinements of to a program to solve the Towers of Hanoi problem. Starts by showing equivalent solutions in C, Pascal, and Smalltalk for comparison. A good introduction to Smalltalk, but needs to be complimented with stronger material on object-oriented design and programming.

29. Keene, S., *Object-Oriented Programming in Common Lisp - A Programmer's Guide to CLOS*, Addison-Wesley, 1989

30. Kilian, M., "Trellis: Turning Designs into Programs," *Communications of the ACM*, 33(9), September 1990, Page 65

    A brief introduction to the major features of the Trellis programming environment and the Trellis/Owl language.

31. Khoshafian, S. N; Copeland, G. P., "Object Identity", *ACM SIGPLAN Notices*, 21(11), November 1986, Page 406

Describes how identity is often equated with addressability (variable names) in programming languages, and with value (e.g. a key) in database languages. Proposes an object structure that includes surrogates, globally unique identifiers that provide strong support for value, structure, and location independent identity of objects.

32. Kornson, T.; McGregor, J.D., "Understanding Object-Oriented: A Unifying Paradigm", *Communications of the ACM*, 33(9), September 1990, Page 40

33. Lang, K. J; Pearlmutter, B. A., "Oaklisp: an Object-Oriented Scheme with First Class Types," *ACM SIGPLAN Notices*, 21(11), November 86, Page 30

34. Laranjeira, L. A., "Software Size Estimation of Object-Oriented Systems", *IEEE Transactions on Software Engineering*, 16(5), May 1990, Page 510

35. Lieberherr, K. J; Riel, A. J., "Contributions to Teaching Object-Oriented Design and Programming", ACM SIGPLAN Notices, 24(10), 1989, p. 11-22

36. Liskov, B., et al, *Clu Reference Manual*, Massechusetts Institute of Technology, Technical Report no. 225, 1979

37. Liskov, Snyder, Atkinson, Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM*, 20(8), August 1977, Page 564

38. Loomis, M. E. S; Shah, A. V; Rumbaugh, J. E., "An Object Modeling Technique for Conceptual Design", *Proceedings of the European Conference on Object-Oriented Programming*, 1987

39. Madsen, O. L; Moller-Pedersen, B., "Virtual Classes: A Powerful Mechanism in Object-Oriented Programming", *ACM SIGPLAN Notices*, 24(10), 1989, p. 397-406

40. Meyer, B., *Object-Oriented Software Construction*, Prentice-Hall, 1988

41. Meyer, B., "Eiffel: A Language and Environment for Software Engineering," *The Journal of Systems and Software*, Vol. 8, 1988, Page 199

42. Meyer, B. "Reusability: The Case for Object-Oriented Design", *IEEE Software*, March 1987, Page 50

Describes the motivation for and problems associated with achieving reusability in software production. Shows how inheritance and, to a lesser degree, genericity are vital to achieving reusability. An illustration is given using "simple", procedural, and object-oriented design approaches. Includes a good discussion of waht factors inhibit the reusability of software.

43. Meyer, B., "Genericity versus Inheritance", *ACM SIGPLAN Notices*, 21(11), November 1986, Page 391

Describes the notions of genericity and inheritance and their relationship to each other in detail. Shows that a limited form of genericity may be simulated using inheritance, but that inheritance cannot be simulated using genericity. Also discusses the bearing that genericity and inheritance have on various classes of problems. Concludes that a complete object-oriented language should support both.

44. Moon, D. A., "Object-Oriented Programming with Flavors," *ACM SIGPLAN Notices*, 21(11), November 1986, Page 1

45. Mullin, M., *Object Oriented Program Design with Examples in C++*, Addison-Wesley, 1989

46. Nexpert Reference Manual, Neuron Data Corporation, 1989

47. Neilson, W. A., ed., *Webster's New International Dictionary*, G. & C. Merriam Company, 1950

48. O'Brien, P; Halbert, D; Kilian, M., "The Trellis Programming Environment", *ACM SIGPLAN Notices*, 22(12), December 1987, p. 91-102

49. Ohori, A; Buneman, P., "Static Type Inference for Parametric Classes", *ACM SIGPLAN Notices*, 24(10), 1989, p. 445-456

50. Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15(12), December 1972, p. 1053-1058

51. Page-Jones, M; Constantine, L; Weiss, S., "Modeling Object-Oriented Systems: The Uniform Object Notation", *Computer Language*, 7(10), October 1990, p. 69-80

Introduces the Uniform Object Notation (UON) for modeling the structure of large object oriented systems. The notation supports four types of diagrams: inheritance, object interface, method structure, and object communication.

52. Pascoe, G. A. "Elements of Object-Oriented Programming", *Byte*, August 1986, Page 139

A basic overview of some principals and techniques normally associated with object-oriented development. Attempts to provide a sort of decision procedure for identifying an object-oriented language: a language must support data abstraction, information hiding, dynamic binding, and inheritance in order to be considered an object-oriented language. Gives definitions for some basic terminology and discusses some advantages and disadvantages of object-oriented programming.

53. Pedersen, C. H., "Extending Ordinary Inheritance Schemes to Include Generalization", ACM SIGPLAN Notices, 24(10), 1989, p. 407-417

54. Peterson, G. E., ed. *Tutorial: Object-Oriented Computing, Volume 1: Concepts*, Washington, D.C., Computer Society Press of the IEEE, 1987

55. Peterson, G. E., ed. *Tutorial: Object-Oriented Computing, Volume 2: Implementations*, Washington, D.C., Computer Society Press of the IEEE, 1987

56. Pun, W; Winder, R., "A Design Method for Object-Oriented Programming", *Proceedings of the Third European Conference on Object-Oriented Programming*, Cambridge: Cambridge University Press, 1989

57. Raj, R. K; et al., "Emerald: A General-Purpose Programming Language," *Software - Practice and Experience*, 21(1), January 1991, Page 91

Describes Emerald as a strongly typed, general purpose programming language. Emerald supports abstract data typing, concurrency via monitors, and the development of distributed systems via self contained objects. Although Emerald does not directly support inheritance, it does provide "object constructors" which may be used to simulate inheritance.

58. Rosson, M. B; Gold, E., "Problem-Solution Mapping in Object-Oriented Design", *ACM SIGPLAN Notices*, 24(10), 1989, p. 7-10

59. Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques", *Proceedings of WESCON*, August 1970

60. Sandberg, D., "An Alternative to Subclassing", *ACM SIGPLAN Notices*, 21(11), November 1986, Page 242

Presents a mechanism for separating the type hierarchy from the class hierarchy in object-oriented systems via "descriptive" (abstract, parent-only) and parameterized classes. The point is that the design of systems using these these mechanisms, as opposed to exclusively using subclassing, is more descriptive.

Abstract and parameterized classes are certainly important constructs. However, it seems to me that the problems they solve, according to the paper, could only occur if subclassing (inheritance) was being misused.

61. Schaffert, C; et. al., "An Introduction to Trellis/Owl," *ACM SIGPLAN Notices*, 21(11), November 1986, Page 9

62. Shlaer, S; Mellor, S., *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice-Hall, 1988

63. Snyder, A., "Encapsulation and Inheritance in Object-Oriented Languages", *ACM SIGPLAN Notices*, 21(11), November 1986, Page 38

A critical analysis of the relationship between encapsulation and inheritance. Points out that the use of inheritance compromises the encapsulation of a class from the viewpoint of its children if child classes are given acces to the instance variables of the parent. The paper makes the assertion that inherited instance variable should only be accesible via operations so that the use of inheritance is invisible to child classes.

While it is true that the visibility of inheritance may lead to dependancies in child classes on the implementation of the parent class, I'm not convinced that this is bad (which seems to be presupposed in the paper.) As child classes are supposed to be specializations or refinements of their parent classes, their behavior should change if that of their parent changes.

64. Stein, L. A., "Delegation is Inheritance", *ACM SIGPLAN Notices*, 22(12), December 1987, p. 138-146

65. Stroustrup, B., "What is Object-Oriented Programming?", *IEEE Software*, 5(3), May 1988, Page 10-20

Describes a continuum of programming paradigms from procedural to data hiding to data abstraction to object-oriented and what distinguishes each. Enumerates characteristics of object orientation and describes language features which support each of these characteristics.

66. Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1986

67. Ungar, D; Smith, R. B., "Self: The Power of Simplicity", *ACM SIGPLAN Notices*, 22(12), December 1987, p. 227-242

68. Ward, P. T., "How to Integrate Object Orientation with Structures Analysis and Design", *IEEE Software*, 6(2), March 1989, p. 74-82

69. Wasserman, A; Pircher, P; Muller, R., "The Object-Oriented Structured Design Notation for Software Design Representation", *IEEE Computer*, March 1990, Page 51

70. Weiskamp, K; Heiny, L; Flamig, B., *Object-Oriented Programming with Turbo C++*, Wiley & Sons, 1991

Describes the development of a user interface toolkit in C++. The book is structured around the layers of the toolkit: the code for a layer of the toolkit is listed, explained, and critiqued in each chapter. Contains many examples and illustrates many principals of object-oriented programming.

71. Wiener, R; Pinson, L., *An Introduction to Object-Oriented Programming and C++*, Addison-Wesley, 1988

A "get up to speed quickly" book on C++. Systematically describes C++ language mechanisms and how each supports object-oriented programming. Includes extensive examples.

72. Wegner, P., "Dimensions of Object-Based Language Design", *ACM SIGPLAN Notices*, 22(12), December 1987, p. 168-182

73. Wilson, D. A., "Class Diagrams: A Tool for Design, Documentation, and Teaching", *Journal of Object-Oriented Programming*, 2(5), January/February 1990, p. 38-44

74. Winblad, A; Edwards, S; King, D., *Object-Oriented Software*, Addison-Wesley, 1990

This three part book gives a more or less comprehensive overview of object-oriented software development. It begins with an introduction to object-orientations in terms of motivation, basic mech-

anisms and concepts, and benefits. Part two is a survey of object-oriented languages, language mechanisms, and applications. Part three describes object-oriented analysis and design, maintenance of object-oriented software, and future directions.

75. Wirfs-Brock, R; Johnson, R., "Surveying Current Research in Object-Oriented Design", *Communications of the ACM*, 33(9), September 1990, Page 104

76. Wirfs-Brock, R; Wilkerson, B; Wiener, L., *Designing Object-Oriented Software*, Prentice Hall, 1990

77. Wirth, N. "Type Extensions", *ACM Transactions on Programming Languages and Systems*, 10(2), April 1988, Page 204

Wirth introduces the notion of a type extension being a mechanism, in a type system similar to that of Pascal, which may be used to create a new record type as the aggregation of one or more existing record types and one or more new fields. The type equivalence of this mechanism is defined and type tests and type guards are introduced to make up for lost type information when lower level types are referenced by pointers of higher types.

Appendix A

Request Document

Diagram Editor for Object-Oriented Design

Charles K Ames III
Miami University
June 28, 1991

This document describes a tool to support the creation of diagrams representing object-oriented designs. The notation forming the diagrams includes the following elements:

- Class: a rectangle with rounded corners, partitioned into three sections. The top section will contain the class name, the middle section will contain the list of attributes of that class, and the bottom section will contain a list of services offered by objects of that class.
- Object: similar to a class. Represents an instance od a class. The exact symbol is not yet determined. Could possibly the class symbol with a different line weight, or surround a class symbol with another line to indicate an object.
- Inheritance connection: An arrow from one class to another is used to indicate that the first class inherits from the second class.
- Composition connection: Link to indicate the one object is a part of another. (precise form is not yet determined.)
- Message connection: Link to indicate that one object sends a message to another object, i.e. the first object invokes a service of the second.
- Association connection: Link to indicate that instances of two classes that are otherwise independent may need to be associated, or grouped together. Indicates a semantic connection.

Requirements: Classes must be easily distinguished from objects. Each type of connection must be easily distinguished from the others.

The tool should have the ability to allow "apropos" text, or notes, to be specified for any or all elements of a diagram (i.e. classes, objects, connections, attributes, and services) This text should not appear on the diagram, but should be easily created and readily accessible for inspection and editing. Elements that have apropos text associated with them should be marked in some (subtle) way so that they are easily identifiable.

Diagrams must be printable. PostScript output is preferred, but not required. The tool should also be able to create a report containing each element name followed by its apropos text, if any.

The tool must be able to save diagrams for later retrievel, editing, and printing.

Creating diagrams with this tool must be easier than creating the diagrams using some more general purpose graphics editor. If it is more trouble than its worth, it is worthless.

The tools must be extensible to include new notation and reflect changes to existing notation. The tool need not be user extensible, but it must be such that a reasonably competant programmer could make any needed modifications.

Appendix B

System Requirements

Diagram Editor for Object-Oriented Design

Charles K Ames III
Miami University
July 9, 1991

The requirements for the diagram editor system are organized around the user interface, since most system functions are directly linked to some user action.

1 Menus

1.1 All menus will be pop-up floating menus. They will pop up at the location of the mouse cursor when the ACT mouse button (the rightmost button) is pressed. The most recently selected action (the first by default) will be highlighted when the menu appears. The menu's location will be determined so that the mouse cursor will be centered on the selected item. The mouse cursor will be moved only if not moving it would cause part of the menu to appear off-screen, in which case it will be moved precisely enough to allow the entire menu to appear on-screen.

1.2 Menu selections will be listed on consecutive lines, aligned on the left. The menu will remain displayed as long as the ACT button is depressed.

1.3 Menu selections (actions) are chosen by positioning the mouse cursor over the desirder selection, causing it to be highlighted (in inverse video), then releasing the mouse button. Then menu then disappears and the action is carried.

1.4 Menu operations may be cancelled by moving the mouse cursor off the menu, so that no menu option is highlighted, and releasing the mouse button. The menu then disappears and the messge "Operation cancelled." is displayed on the message line.

2 User Interface - System

2.1 The diagram editor will provide a system menu which offers the following selections:

- new start a new diagram
- open retrieve a previously saved diagram for editing
- save save the current diagram to disk
- close save the current diagram and start a new diagram
- exit terminate the diagram editor and return to the system prompt

2.2 The word "File" will appear in the upper left corner of the screen. Clicking the ACT mouse button while the mouse cursor is positioned over this word will cause the system menu to appear.

2.3 The diagram editor will display the title of the current diagram centered on the first line of the display. If there is no current diagram, then name "<untitled>" will be displayed in place of the title.

2.4 Both the system menu icon ("File") and the title will appear on the first line of the screen on what is to be called the "title bar."

2.5 The second line (in text lines) will be reserved for system messages and prompts to be used for user interaction. This is the message line.

2.6 The remainder of the screen will be surrounded by a narrow (<= 5 pixels) border. The area inside this border will serve as the canas for creating and editing diagrams.

2.7 ** Optional ** The right side and bottom borders will contain icons for scrolling windows whch are bigger than the available display area. These elevators will functin as exhibited in such packages as Microsoft Windows 3.0 and OSF Motif.

3 User Interface - Diagram

3.1 The leftmost mouse button will be the SELECT button and the rightmost button the ACT button.

3.2 Clicking the SELECT buton while the mouse cursor is positioned over a screen object causes that object to be selected, i.e. made the current object, or become the object of focus.

3.3 Selecting a non-selectable object (e.g. the message line) will cause the messge "Object not selectable." to be displayed on the message line, unless that object has a menu associated with it, in which case the message "Use the menu button." will be displayed.

3.4 Selecting a selectable object will cause the message "<object name> selected." to be displayed. The next action will be applied to that object, unless another object is selected before another action is taken.

3.5 Pressing the ACT button while the mouse cursor is over the system menu icon ("File") will cause the system menu to appear. Either an action is selected by the user and carried out by the system, or the action is cancelled by moving the cursor off the menu and releasing the mouse button.

3.6 Pressing the ACT button while in the diagram area after a screen object has been selected will cause the menu associated with that object to appear. Select/Perform or Cancel as above.

3.7 The menu for screen objects in the diagram area will offer the following choices:

- dup duplicate the current object. The new object becomes the current object and will be in move mode.
- move allow the current object to be repositioned in the diagram area. The move operation is completed by clicking the SELECT button to indicate the new position of the object

- del — delete the current object from the diagram and erase it from the screen.
- note — allow text describing the object to be entered into a pop-up text edit window.
- attr — add an attribute to the current object. User is prompted for the attribute name, then the attribute is displayed.
- serv — add a service to the current object. User is prompted for the service name. then the service is displayed.
- name — allow the name of the current diagram to be changed.

3.8 Pressing the ACT button while the mouse cursor is in the diagram area but no object has been selected will cause the diagram menu to appear, offering the following options:

- class — create and display a new class symbol
- object — crete and display a new object symbol
- message — create a new message connection between two object symbols. The message "Click on sending Class/Object" will be displayed on the message line until the user selects a class/object symbol with the mouse. Then the message "Click on receiveing object." will be displayed until the user selects another (different from the first) class/object symbol. A message onnection will then be displayed between the two symbols.
- gen-spec — create a new inheritance (generalization-specialization) connec tion between two class/object symbols. Dialog proceeds as above, except the message "Click on the derived class." will be dis played first, followed by the message "Click on the base class." A gen-spec connection is added to the diagram and displayed on the screen.
- whole-part — create a new composition (whole-part) connection between two class/object symbols. Dialog proceeds as above, except the mes sage "Click on the component class/object." will be displayed first, followed by the message "Click on the aggregate class/object." A whole-part connection is added to the diagram and displayed on the screen.
- association — create a new association connection between two class/object symbols. Dialog proceeds as above, except the message "Click on the first class/ object." will be displayed first, followed by the message "Click on the second class/ object." An association connection is added to the diagram and displayed on the screen.

3.9 Gen-spec, whole-part, and message connections are directional, and the orientation of their symbols will be determined by the order the classes were selected in the creation dialog. Association connections are non-directional.

Appendix C

Object-Oriented Analysis (OOA) Model

Diagram Editor for Object-Oriented Design

Charles K Ames III
Miami University
July 9, 1991

1 Classes and Objects

Diagram: A composite object that represents an entire diagram. Consists of a name and zero or more elements. Can be saved, retrieved, and printed.

```
Attributes:
        name
        element-list

Services:
        show
        add-element
        delete-element
        save
        retrieve
        set-name
```

Mouse: System pointer object. Acts as an interface to the physical mouse device. Creates a stream of events (x,y,button).

```
Attributes:
        cursor-location
        button-status

Services:
        show
        hide
        get-event   // wait for a click, then return (x,y,button)
        get-position
        set-position
```

```
Title-bar:
Attributes:
        title

Services:
        show
        invoke-diagram-action
```

Message-Line: display messages and get user input.

Attributes:
last-message

Services:
post
get-string
get-object

Element: A part of a diagram. Each element knows how to draw itself, how and to what it can connect itself, and how other elements may be connected to it. Has an "apropos" text description (may be empty)

Attributes:
location // (x,y) screen coordinates
name // object identifier
typename // class identifier
apropos

Services:
show
edit-apropos
edit-name

Class-Object: [Element] A diagram element representing an entity or entity type.

Attributes:
attribute-list
service-list
connection-list

Services:
add-attribute
edit-attribute
delete-attribute
add-service
edit-service
delete-service
move

Link: [Element] A general link between two class/objects representing some type of relationship.

Attributes:
endpoint-1
endpoint-2

Services:
connect

Association: [Link] A diagram element connecting two class-objects representing some arbitrary relationship (may be named -- name attribute inherited from element).

Attributes:

Services:

Whole-Part: [Link] A diagram element connecting two class-objects representing a whole-part relationship where the destination object is a component of the origin object.

Attributes:
from-object
to-object

Services:

Gen-Spec: [Link] A diagram element connecting two class-objects representing a generalization-specialization relationship where the destination object inherits from the origin object.

Attributes:
from-object
to-object

Services:

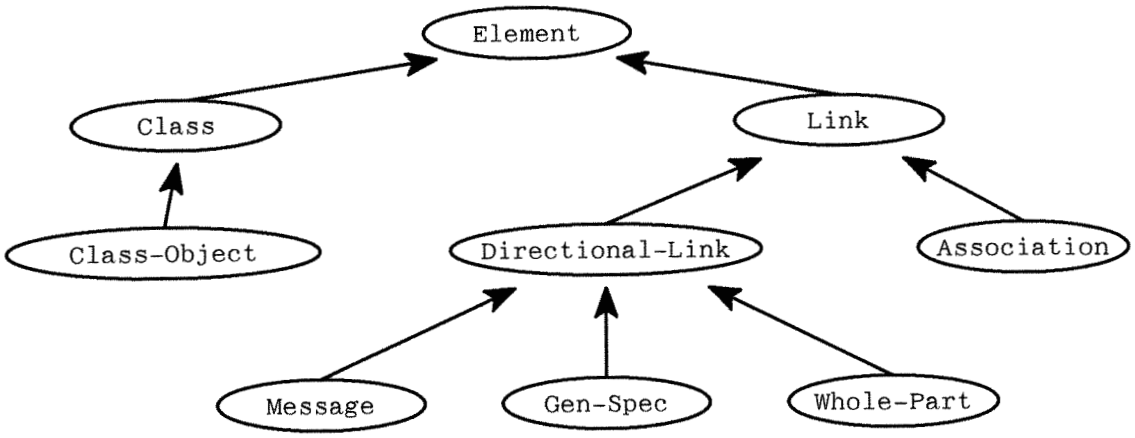
Message: [Link] A diagram element connecting two class-objects with an arrow from the sender to the receiver.

Attributes:
from-object
to-object

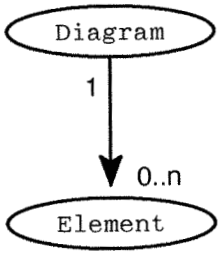
Services:

2 Structures

Generalization-Specialization Structures:



Whole-Part Structures



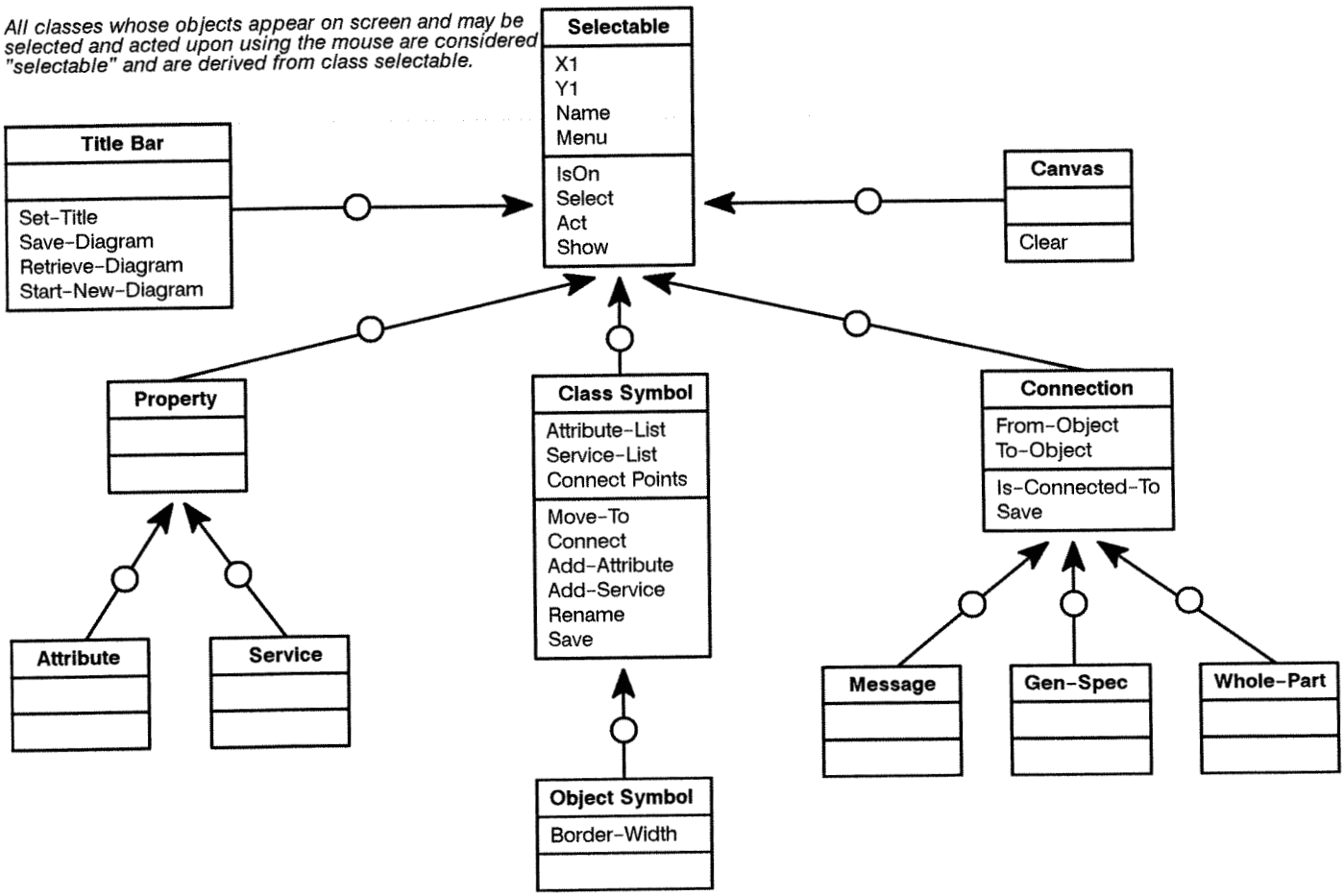
3 Subjects

There are two subjects in this system: one is composed of the element class and its derived classes. This subject defines the notation supported by the diagram editor. The remainder of the classes form the other (unnamed) subject.

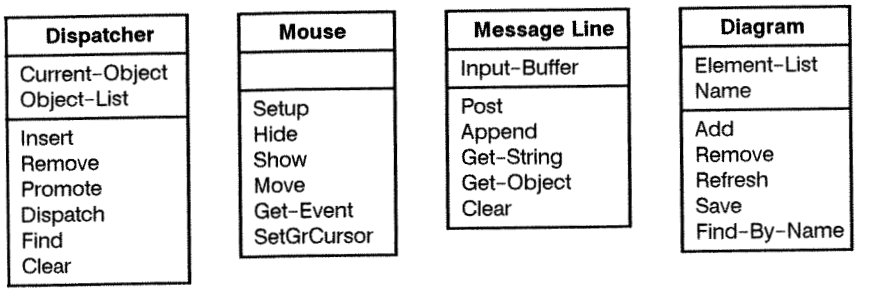
Appendix D

Inheritance Diagram

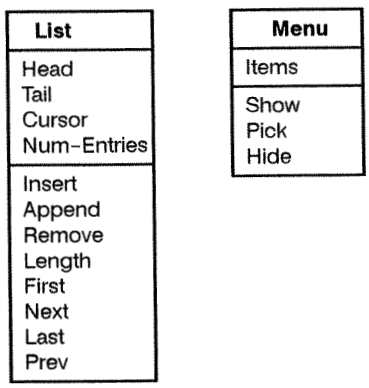
Diagram Editor for Object-Oriented Design



Standalone Objects - There is exactly one of each of these objects in the system



Frequently Reused Components



Note: There is also only one Titlebar object and one Canvas object. They are shown above since they are derived from selectable.

Appendix E

Aggregation Diagram

Diagram Editor for Object-Oriented Design

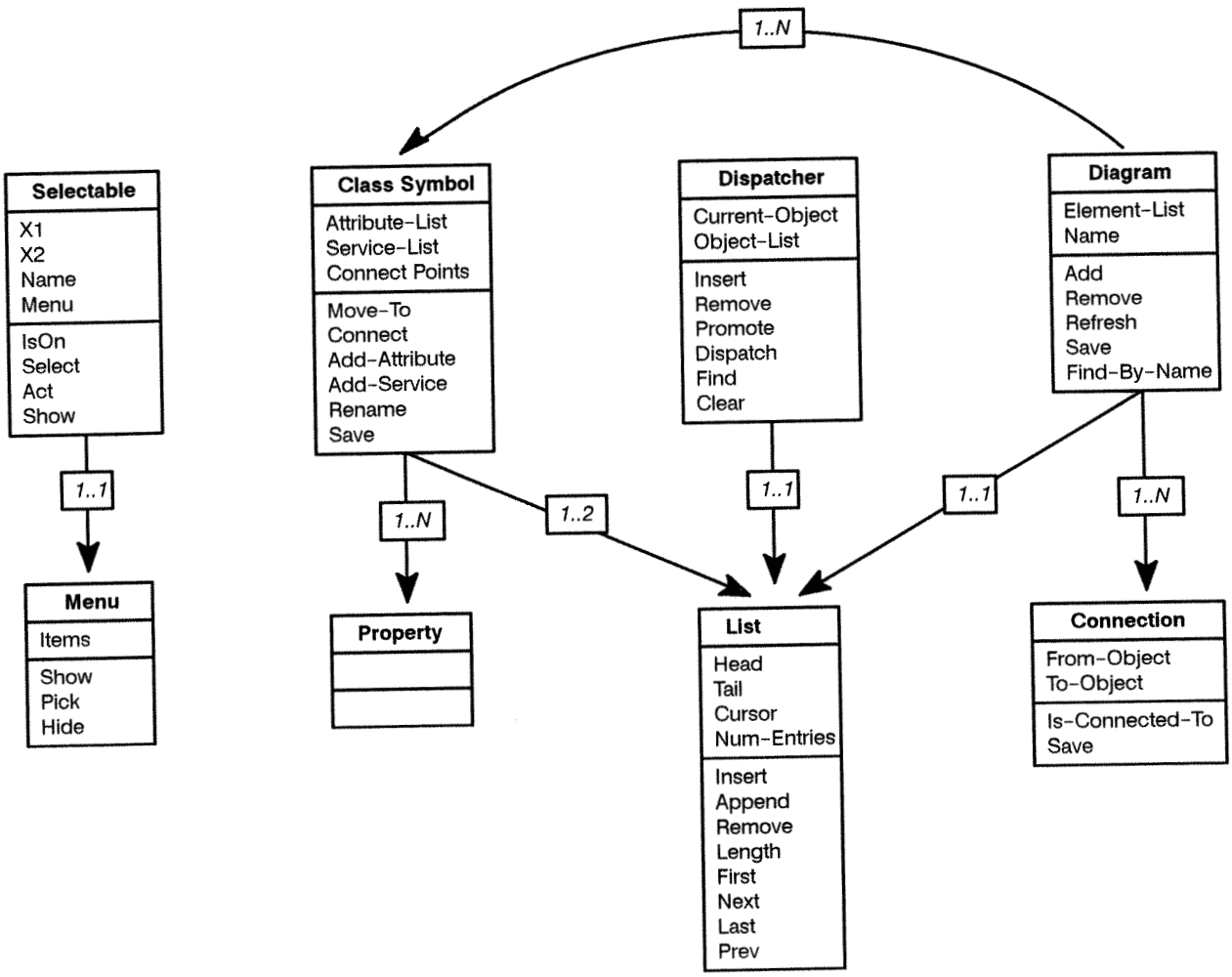
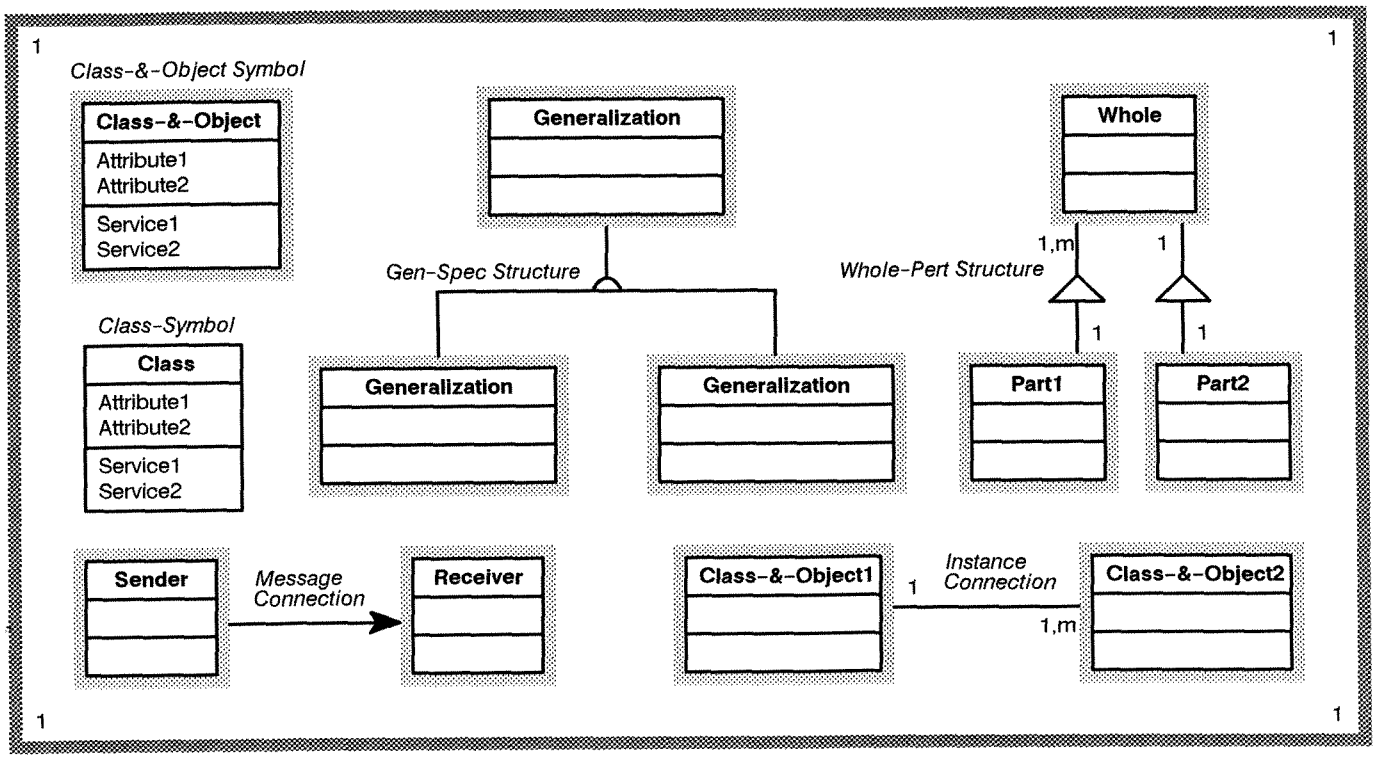


Figure 5: OOA/OOD Notation Summary



DE_Source.index


```
//---[ DE_Source.index ]-------------------------------------------------

//
//  Road map to the source code for the diagram editor system.
//


All class definitions appear in header (.h) files and all member
functions appear in .cpp files of the same name.


canvas.h, canvas.cpp:  the canvas object.  The canvas appears as the
        drawing area, and takes care of creating new symbols and connections.

connect.h, connect.cpp:  Contains the class hierarchy for the connection
        objects (message, genspec, whole-part, association)

de.cpp:  the main program for the diagram editor system.

diagram.h, diagram.cpp:  the diagram object.  Stores a pointer to each
        class and object symbol and each connection in the picture.  Takes
        care of redisplaying and saving the entire diagram.

dispatch.h, dispatch.cpp:  the dispatcher object.  maintains a list of all
        "selectable" objects on the screen and distributes events to the
        appropriate objects.

editor.h, editor.cpp:  a simple text editor to create and edit the char *text
        attribute of each symbol, property, and connection.  Used in the
        annotate() function.

effect.h, effect.cpp:  a collection of graphics special effect objects, such
        as shadow and bevel.

list.h, list.cpp:  a general purpose list class.  maintains a list of
        objects of type selectable.  objects may be inserted at either end
        and the list may be traversed in either direction.

menu.h, menu.cpp:  defines a popup floating menu.

msgline.h, msgline.cpp:  the message line.  allows messages to be posted
        (displayed) and can get user input in the form of text of clicks.

msmouse.h, msmouse.cpp, mcursor.cpp:  the mouse object.  Provides a
        relatively nice set of functions for accessing and manipulating
        the Microsoft Mouse driver.  The file mcursor.cpp contains bit
        maps for alternate mouse cursor icons.

property.h, property.cpp:  contains the definition of attribute and service
        objects.

select.h, select.cpp:  defines the abstract class selectable for deriving
        mouse selectable screen objects from.

symbol.h, symbol.cpp:  contains the class and object symbol objects.

titlebar.h, titlebar.cpp:  the titlebar object is like the "system" in this
        system.  It offers the file level services, such as save, open, close,
        name, exit.

util.h, util.cpp:  micellaneous functions that didn't really belong anywhere
        else.


// end of file
```

```
//---[ canvas.h ]-----------------------------------------------

//
// class definition for the canvas object
//

#ifndef CANVAS_H
#define CANVAS_H


#include "select.h"


class canvas_t : public selectable {
    public:
        canvas_t(void);
        ~canvas_t(void);

        void clear(void);

        char *typeof(void)  { return "canvas_t"; }
        selectable *select(void);
        int act(int x, int y);

    private:

    };

extern canvas_t canvas;


#endif


// end of file
```

```
//---[ connect.cpp ]-------------------------------------------------


//
// member functions for the connection class and its subclasses
//


#include <graphics.h>
#include <math.h>
#include <string.h>

#include "connect.h"
#include "dispatch.h"
#include "diagram.h"
#include "msmouse.h"
#include "msgline.h"
#include "util.h"


void arrowhead(int x1, int y1, int x2, int y2);


connection_t::connection_t(class_t *obj1, class_t *obj2)  {

    strcpy(name, "Connection");
    strcpy(label, "Association ");
    from = obj1;
    to = obj2;

    tolerance = 3;

    menu = &connection_menu;
    text = NULL;

    return;
    }


connection_t::connection_t(FILE *fp)  {
    char buf[40];

    strcpy(label, "Association ");

    getqs(fp, buf);
    if (buf[0] == '\0')
        strcpy(name, "Connection");
    else
        strcpy(name, buf);

    if (fscanf(fp, " %s ", buf) == 0 || strcmp(buf, "from") != 0) {
        current_diagram->error = 0;
        return;
        }
    getqs(fp,buf);
    from = (class_t *)current_diagram->find_by_name(buf);

    if (fscanf(fp, " %s ", buf) == 0 || strcmp(buf, "to") != 0) {
        current_diagram->error = 0;
        return;
        }
    getqs(fp,buf);
    to = (class_t *)current_diagram->find_by_name(buf);

    tolerance = 3;

    menu = &connection_menu;
    text = NULL;

    dispatcher.insert(this);

    return;
    }


connection_t::~connection_t(void)  {


    return;
    }


int connection_t::isa(char *s)  {

    return (strcmp(s,connection_t::typeof()) == 0) || selectable::isa(s);
    }


void connection_t::show(void)  {
    int midx, midy;  // wait until after connect to calculate

    from->connect(to, x1, y1, x2, y2);

    midx = (x1+x2)/2;
    midy = (y1+y2)/2;

    Mouse.Hide();

    setcolor(BLACK);
    moveto(x1,y1);
    lineto(x2,y2);

    if ( strcmp(typeof(), "connection_t") == 0  &&
         strcmp(name, "Connection")       != 0 )  {

        setfillstyle(SOLID_FILL, WHITE);
        bar( midx - textwidth(name)/2 - 1, midy - textheight(name) - 1,
             midx + textwidth(name)/2 + 1, midy + textheight(name) + 1);

        settextjustify(CENTER_TEXT, CENTER_TEXT);
        outtextxy( midx, midy, name );
        }

    Mouse.Show();

    return;
    }


selectable *connection_t::select(void)  {

    selectable::select();
    current_diagram->promote(this);

    return this;
    }
```

```
int connection_t::ison(int x, int y)  {
    double A = double(y2 - y1),
           B = double(x1 - x2);
    double C = double( -A * x1 - B * y1);

    double D = sqrt( A*A + B*B );
    double d = x*A/D + y*B/D + C/D;

    if ( ( ( x2 > x1 && x >= x1 && x <= x2 ) ||      // between x1 and x2
           ( x2 <= x1 && x <= x1 && x >= x2 ) ) &&  // and
         ( ( y2 > y1 && y >= y1 && y <= y2 ) ||      // between y1 and y2
           ( y2 <= y1 && y <= y1 && y >= y2 ) ) )  {
        return abs(int(d)) <= tolerance ? 1 : 0;
        }
    else
        return 0;

    }


int connection_t::act(int x, int y)  {
    int selection;

    selection = selectable::act(x,y);

    switch (selection)  {

        case 0:  // label
            strcpy(name, msgline.gets("Enter label: ") );
            if (name[0] == '\0')
                strcpy(name, "Connection");
            else
                msgline.post("Label added.");
            current_diagram->refresh();
            break;

        case 1:  // note
            annote();
            break;

        case 2:  // delete
            dispatcher.remove(this);
            current_diagram->remove(this);
            current_diagram->refresh();
            msgline.post(name);
            msgline.append(" deleted.");
            delete this;
            break;

        default:
            break;
        }

    return selection;
    }


int connection_t::is_connected_to(selectable *obj)  {

    return (obj == from) || (obj == to);
    }


void connection_t::save(FILE *fp)  {

    if (strcmp(name, "Connection") == 0)
        fprintf(fp, "%s \"\"\n", label);
    else
        fprintf(fp, "%s \"%s\"\n", label, name);

    fprintf(fp, "\tfrom \"%s\"\n", from->name);
    fprintf(fp, "\tto \"%s\"\n\n", to->name);

    return;
    }


message_t::message_t(class_t *obj1, class_t *obj2) :
          connection_t(obj1, obj2)  {

    strcpy(label, "Message ");

    return;
    }


message_t::message_t(FILE *fp) : connection_t(fp)  {

    strcpy(label, "Message ");

    return;
    }


message_t::~message_t(void)  {


    return;
    }


int message_t::isa(char *s)  {

    return (strcmp(s,typeof()) == 0) || connection_t::isa(s);
    }


void message_t::show(void)  {

    connection_t::show();
    arrowhead(x1,y1,x2,y2);

    return;
    }


gen_spec_t::gen_spec_t(class_t *obj1, class_t *obj2) :
           connection_t(obj1, obj2)  {

    strcpy(label, "Inheritance ");

    return;
    }


gen_spec_t::gen_spec_t(FILE *fp) : connection_t(fp)  {

    strcpy(label, "Inheritance ");
```

```
    return;
    }


gen_spec_t::~gen_spec_t(void)  {


    return;
    }


int gen_spec_t::isa(char *s)  {

    return (strcmp(s,typeof()) == 0) || connection_t::isa(s);
    }


void gen_spec_t::show(void)  {

    connection_t::show();
    arrowhead(x1,y1,x2,y2);
    circle( (x1+x2)/2, (y1+y2)/2, 5 );

    return;
    }


whole_part_t::whole_part_t(class_t *obj1, class_t *obj2) :
             connection_t(obj1, obj2)  {

    strcpy(label, "Whole-Part ");

    return;
    }


whole_part_t::whole_part_t(FILE *fp) : connection_t(fp)  {

    strcpy(label, "Whole-Part ");

    return;
    }


whole_part_t::~whole_part_t(void)  {


    return;
    }


int whole_part_t::isa(char *s)  {

    return (strcmp(s,typeof()) == 0) || connection_t::isa(s);
    }


void whole_part_t::show(void)  {

    connection_t::show();
    arrowhead(x1,y1,x2,y2);
    rectangle( (x1+x2)/2-4, (y1+y2)/2-4, (x1+x2)/2+4, (y1+y2)/2+4 );
    return;
    }


void arrowhead(int x1, int y1, int x2, int y2)  {
    double C1 = 15, C2 = 5;          // length and width/2 of arrowhead
    double m, den, theta;
    int poly[6], xb, yb, dx, dy;

    if (x2 == x1)  {            // special case -- x1 == x2
        poly[0] = x2;
        poly[1] = y2;
        if (y2 > y1)  {
            poly[2] = x2 - C2;
            poly[3] = y2 + C1;
            poly[4] = x2 + C2;
            poly[5] = y2 + C1;
            }
        else  {
            poly[2] = x2 - C2;
            poly[3] = y2 - C1;
            poly[4] = x2 + C2;
            poly[5] = y2 - C1;
            }
        setfillstyle(SOLID_FILL, BLACK);
        fillpoly(3,poly);
        }
    else  {
        m = double(y2-y1)/double(x2-x1);
        den = sqrt( C1*C1 / (m*m + 1) );
        }


    // calculate the point of intersection of the back of the arrow

    if (x2 > x1)
        xb = x2 - int(den);
    else
        xb = x2 + int(den);

    if (y2 > y1)
        yb = y2 - abs( int(m * den) );
    else
        yb = y2 + abs( int(m * den) );

    if ( m < 0.001 && m > -0.001 )
        theta = 1.570796327;         // arcsin(1) in rads
    else
        theta = atan( -1 / m );

    dy = abs( int(C2 * sin(theta)) );
    dx = abs( int(C2 * cos(theta)) );

    poly[0] = x2;
    poly[1] = y2;
    if ((x2 > x1 && y2 < y1) || (x2 < x1 && y2 > y1))  {  // quad 1 or 3
        poly[2] = xb - dx;
        poly[3] = yb - dy;
        poly[4] = xb + dx;
        poly[5] = yb + dy;
        }
    else  {
        poly[2] = xb - dx;
```

```
        poly[3] = yb + dy;
        poly[4] = xb + dx;
        poly[5] = yb - dy;
        }

    setfillstyle(SOLID_FILL, BLACK);
    fillpoly(3,poly);

    return;
    }


// end of file
```

```
//---[ de.cpp ]-------------------------------------------------------

//
//  Main program for diagram editor system
//



#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

#include "msmouse.h"
#include "titlebar.h"
#include "msgline.h"
#include "canvas.h"
#include "dispatch.h"
#include "menu.h"
#include "diagram.h"
#include "editor.h"
#include "util.h"


int           exit_flag = 0;

// the main players

screen_t      screen;
titlebar_t    titlebar;
messageline_t msgline;
canvas_t      canvas;
dispatcher_t  dispatcher;
editor_t      editor;
diagram_t     *current_diagram;


// menus for the screen objects

menu_t class_menu("Class", "Dup", "Move", "Delete", "Note",
                           "Attr", "Serv", "Rename", NULL),
       object_menu("Object", "Dup", "Move", "Delete", "Note",
                             "Attr", "Serv", "Rename", NULL),
       connection_menu("Connect", "Label", "Note", "Delete", NULL),
       attribute_menu("Attribute", "Rename", "Note", "Define", "Delete", NULL),
       service_menu("Service", "Rename", "Note", "Define", "Delete", NULL);


main()  {
    int x, y;
    char ch = 0;
    unsigned e;
    char msgbuf[80];

    dispatcher.insert(&titlebar);
    dispatcher.insert(&canvas);

    titlebar.show();
    msgline.show();
    canvas.clear();

    current_diagram = new diagram_t;

    msgline.post("Ready.");

    Mouse.Setup(Graphics);
    Mouse.Show();

    while (exit_flag == 0)  {
        e = get_event(ch,x,y);

        switch (e)  {
            case SELECT:
                dispatcher.dispatch(x,y,SELECT);
                break;

            case ACT:
                dispatcher.dispatch(x,y,ACT);
                break;

            case KEY:
                switch (ch)  {
                    case 27:   // <esc>
                        exit_flag = 1;
                        break;

                    default:
                        msgline.post("Use mouse, or press <ESC>");
                        break;

                    }

                break;
            }
        }

    delete current_diagram;

    msgline.post("Press any key to exit...");
    getch();

    }



int get_event(char &ch, int &x, int &y)  {
    int e, event = 0;

    while (event == 0)  {

        if ( kbhit() )  {
            ch = getch();
            event = KEY;
            }

        else  {
            e = Mouse.Event(x,y);

            if (e == LMouseDown)  {
                event = SELECT;
                }
            else if (e == RMouseDown)  {
                event = ACT;
                }
            }
        }

    return event;
    }
```

```
// end of file
```

```
//---[ diagram.h ]----------------------------------------------

#ifndef DIAGRAM_H
#define DIAGRAM_H


//
// class definition for the diagram class
//

#include <stdio.h>

#include "select.h"
#include "list.h"


class diagram_t  {

    public:
        diagram_t(void);
        diagram_t(FILE *fp);
        ~diagram_t(void);

        void add(selectable *obj);
        void remove(selectable *obj);
        void promote(selectable *obj);
        void refresh(void);
        void save(FILE *fp);
        selectable *find_by_name(char *s);
        selectable *first_connection(selectable *obj);
        selectable *next_connection(selectable *obj);

        char name[80];
        int error;

    private:
        list_t *elist;
    };


extern diagram_t *current_diagram;


#endif

// end of file
```

```
//---[ diagram.cpp ]-------------------------------------------------


//
// member functions for the diagram class
//

#include <stdio.h>
#include <string.h>

#include "diagram.h"
#include "canvas.h"
#include "connect.h"
#include "util.h"


diagram_t::diagram_t(void)  {

    elist = new list_t;
    name[0] = '\0';

    return;
    }


diagram_t::diagram_t(FILE *fp)  {
    char buf[80];

    error = 0;

    if ( fscanf(fp, " %s ", buf) == 0 || strcmp(buf, "diagram") != 0)  {
        error = 1;
        return;
        }

    getqs(fp, buf);
    strcpy(name, buf);
    elist = new list_t;

    return;
    }


diagram_t::~diagram_t(void)  {

    // need to make sure contests are deleted also (?)

    delete elist;

    return;
    }


void diagram_t::add(selectable *obj)  {

    elist->insert(obj);

    return;
    }


void diagram_t::remove(selectable *obj)  {

    elist->remove(obj);

    return;
    }


void diagram_t::promote(selectable *obj)  {

    elist->remove(obj);
    elist->insert(obj);

    return;
    }


void diagram_t::refresh(void)  {
    selectable *obj = elist->last();

    canvas.clear();

    while (obj != (selectable *)NULL)  {
        obj->show();
        obj = elist->prev();
        }

    return;
    }


void diagram_t::save(FILE *fp)  {
    selectable *obj;

    fprintf(fp, "diagram \"%s\"\n\n", name);

    obj = elist->first();                // first save class & objects
    while (obj != (selectable *)NULL)  {
        if ( obj->isa("class_t") )
            obj->save(fp);
        obj = elist->next();
        }

    obj = elist->first();                // then save connections
    while (obj != (selectable *)NULL)  {
        if ( obj->isa("connection_t") )
            obj->save(fp);
        obj = elist->next();
        }

    fprintf(fp, "end diagram \"%s\"\n", name);

    return;
    }


selectable *diagram_t::find_by_name(char *s)  {
    selectable *obj = elist->first();

    while (obj != (selectable *)NULL)  {
        if (strcmp(obj->name, s) == 0)
            return obj;
        else
            obj = elist->next();
        }

    return (selectable *)NULL;
```

```
    }


selectable *diagram_t::first_connection(selectable *obj)  {
    connection_t *con = (connection_t *)elist->first();

    while (con != NULL)   {
        if (con->isa("connection_t") && con->is_connected_to(obj) )
            return con;
        else
            con = (connection_t *)elist->next();
        }

    return NULL;
    }


selectable *diagram_t::next_connection(selectable *obj)  {
    connection_t *con = (connection_t *)elist->next();

    while (con != NULL)   {
        if (con->isa("connection_t") && con->is_connected_to(obj) )
            return con;
        else
            con = (connection_t *)elist->next();
        }

    return NULL;
    }


// end of file
```

```
//---[ dispatch.h ]----------------------------------------------------
//
// Class definition for the the dispatcher object
//
// class for object that directs mouse clicks to the appropriate
// screen objects.
//
//---------------------------------------------------------------------

#ifndef DISPATCH_H
#define DISPATCH_H


#include "select.h"
#include "list.h"


class dispatcher_t  {
  public:
      dispatcher_t(void);
      ~dispatcher_t(void);

      int insert(selectable *obj);    // add an object
      int remove(selectable *obj);    // remove an object
      int promote(selectable *obj);   // move obj to top of its priority
                                      //   category (handle overlaps)
      selectable *find(int x, int y);
      void clear(void);
      void dispatch(int x, int y, int button);

  private:
      list_t     *obj_list;
      selectable *current_obj;

  };

extern dispatcher_t dispatcher;


#endif


// end of file
```

```
//---[ dispatch.cpp ]-------------------------------------------------------
//
//  Member functions for the dispatcher object.
//
//  The dispatcher takes mouse clicks (x,y,button) and directs them to
//  the appropriate screen object.
//
//--------------------------------------------------------------------------


#include <stdio.h>

#include "dispatch.h"
#include "msgline.h"
#include "util.h"


dispatcher_t::dispatcher_t(void)  {

    obj_list = new list_t;
    current_obj = (selectable *)NULL;

    return;
    }


dispatcher_t::~dispatcher_t(void)  {

    delete obj_list;

    return;
    }


int dispatcher_t::insert(selectable *obj)  {

    obj_list->insert(obj);

    return 1;
    }


int dispatcher_t::remove(selectable *obj)  {

    obj_list->remove(obj);
    current_obj = (selectable *)NULL;

    return 1;
    }


int dispatcher_t::promote(selectable *obj)  {

    obj_list->remove(obj);
    obj_list->insert(obj);

    return 1;
    }


void dispatcher_t::clear(void)  {

    delete obj_list;
    obj_list = new list_t;

    return;
    }


selectable *dispatcher_t::find(int x, int y)  {
    selectable *obj;

    obj = obj_list->first();

    while (obj != (entry_t *)NULL && !obj->ison(x,y) )  {
        obj = obj_list->next();
        } // end while

    return obj;
    }


void dispatcher_t::dispatch(int x, int y, int button)  {
    selectable *obj;

    if (button == ACT)  {
        if (current_obj != (selectable *)NULL)
            current_obj->act(x,y);
        else
            msgline.post("No selected object to act on. Use SELECT button first.");
        }

    else  {
        obj = find(x,y);

        if (obj == (selectable *)NULL) {
            msgline.post("Not a selectable object. No object currently selected.");
            current_obj = (selectable *)NULL;
            }
        else  {
            current_obj = obj;
            current_obj->select();
            }

        } // end else

    return;
    }


// end of file
```

```
//---[ editor.h ]-----------------------------------------------------

//
// Class definition for an text edit window class
//

#ifndef EDITOR_H
#define EDITOR_H

#define MAXLINE 100
#define MAXCOL  81
#define ENDBUF  -1
#define ENDLINE 0



//
// redefine getch() to handle extended keys with a single call
//

int getkey(void);


//
// keycodes returned by getkey()
//

#define BACKSPACE     8
#define NEWLINE       13
#define UP            200
#define DN            208
#define LEFT          203
#define RIGHT         205
#define PGUP          201
#define PGDN          209
#define HOME          199
#define END           207
#define INSERT        210
#define DELETE        211
#define CTRL_Y        25
#define CTRL_Z        26
#define ESC           27
#define TAB           9
#define BACKTAB       143



class editor_t  {
    public:
        editor_t(void);
        editor_t(int left, int top, int right, int bottom);
        ~editor_t(void);

        char *edit(char *buf, char *buffer_title);

    private:
        void clearbuf(void);
        void refresh(void);
        void cls(void);
        void show(char *buffer_title);
        void hide();

        void insertline(int line);
        void deleteline(int line);
        void insertchar(int line, int col, char ch);
        void deletechar(int line, int col);

        void scrollup(int nlines);
        void scrolldn(int nlines);
        int  editbuf(void);

        char *image;                    // buffer for getimage()
        char ebuf[MAXLINE][MAXCOL];     // the edit buffer
        char *bufname;                  // name of the text being edited
        int  x1, y1, x2, y2;            // edit window size in pixels
        int  length, width;             // edit window size in characters
        int  txht, txwd;                // height and width of a character

        int  top_line, end_line, current_line, current_col;

    };


#ifndef TESTEDIT
extern editor_t editor;
#endif


#endif

// end of file
```

```
//---[ editor.cpp ]-------------------------------------------------

//
// member functions for the editor class
//

#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
#include <conio.h>
#include <string.h>

#include "effect.h"
#include "editor.h"



//
// special form of getch() to handle extended codes with a single call.
//

int getkey(void)  {
    int t = getch();

    return ( t != 0 ) ? t : getch() + 128;
    }



editor_t::editor_t(void)  {

    x1 = getmaxx() / 6;  // edit window is 2/3 the width and 1/2 height
    x2 = getmaxx() - x1; // of the screen
    y1 = getmaxy() / 4;
    y2 = getmaxy() - y1;

    txht   = textheight("H") * 4/3;
    length = (y2 - y1) / txht;
    txwd   = textwidth("H");
    width  = (x2 - x1) / txwd;

    clearbuf();

    return;
    }


editor_t::editor_t(int left, int top, int right, int bottom)  {

    x1 = left;
    x2 = right;
    y1 = top;
    y2 = bottom;

    txht   = textheight("H") * 4/3;
    length = (y2 - y1) / txht;
    txwd   = textwidth("H");
    width  = (x2 - x1) / txwd;

    clearbuf();

    return;
    }


editor_t::~editor_t(void)  {

    return;
    }


void editor_t::clearbuf(void)  {
    int i;

    for (i = 0; i < MAXLINE; ebuf[i++][0] = ENDBUF);

    top_line = 0;
    end_line = 0;
    current_line = 0;
    current_col = 0;

    return;
    }


void editor_t::refresh(void)  {
    char *lp = &ebuf[top_line][0];
    int i = 0;

    cls();

    while (*lp != ENDBUF && i < length)  {
        outtextxy(x1, y1 + i * txht, lp);    // print the line
        lp = &ebuf[top_line + ++i][0];
        }

    if (*lp == ENDBUF && i < length)
        outtextxy(x1, y1 + i * txht, "[eob]");

    return;
    }


void editor_t::cls(void)  {

    setfillstyle(SOLID_FILL, WHITE);
    bar(x1,y1,x2,y2);

    return;
    }


void editor_t::show(char *buffer_title)  {
    int msgtop    = y1-txht-5,
        titletop  = msgtop-txht-4,
        wintop    = titletop-2,
        winleft   = x1-3,
        winright  = x2+3,
        winbottom = y2+3;

    setfillstyle(SOLID_FILL, WHITE);     // blot out whatever is there now
    bar(winleft, wintop, winright, winbottom);

    setlinestyle(SOLID_LINE, 1, 1);      // draw the edit window
    setcolor(BLACK);
    rectangle(winleft, wintop, winright, winbottom);

    setfillstyle(SOLID_FILL, BLACK);
```

```
    bar(winleft+2, titletop, winright-2, titletop+txht+2);
    bar(winleft+2, msgtop, winright-2, msgtop+txht+2);

    // moveto(winleft,  msgtop);
    // lineto(winright, msgtop);
    // moveto(winleft,  texttop);
    // lineto(winright, texttop);

    shadow sh(winleft, wintop, winright, winbottom);
    sh.display();

    settextjustify(CENTER_TEXT, TOP_TEXT);
    setcolor(WHITE);
    outtextxy((winleft+winright)/2, titletop+2, buffer_title);
    settextjustify(LEFT_TEXT, TOP_TEXT);
    outtextxy(x1+1, msgtop+2, "Enter text. Use Ctrl-Z to end edit, Esc to cancel.");

    return;
    };


void editor_t::hide(void)  {

    return;
    }


void editor_t::insertline(int lineno)  {
    int i = 0, j;

    while (i < MAXLINE && ebuf[i][0] != ENDBUF) i++;

    if (i == MAXLINE) return;

    while (i > lineno)  {
        strcpy(&ebuf[i][0], &ebuf[i-1][0]);
        i--;
        }

    for (j = 0; j < MAXCOL; ebuf[i][j++] = ENDLINE);
    end_line++;

    return;
    }


void editor_t::deleteline(int lineno)  {
    int i = lineno;

    while (ebuf[i][0] != ENDBUF)  {
        strcpy(&ebuf[i][0], &ebuf[i+1][0]);
        i++;
        }

    if (end_line > 0) end_line--;

    return;
    }


void editor_t::insertchar(int line, int col, char ch)  {
    int i = col;

    while (ebuf[line][i] != ENDLINE) i++;  // find end of line

    while (i > col)  {
        ebuf[line][i] = ebuf[line][i-1];   // move text after cursor on
        i--;                               // same line to the right 1 space
        }

    ebuf[line][i] = ch;                    // insert the new character

    return;
    }


void editor_t::deletechar(int line, int col)  {
    int i = col;

    while (ebuf[line][i] != ENDLINE)  {
        ebuf[line][i] = ebuf[line][i+1];
        i++;
        }

    return;
    }


void editor_t::scrollup(int nlines)  {

    if (top_line > nlines)
        top_line -= nlines;
    else
        top_line = 0;

    refresh();

    return;
    }


void editor_t::scrolldn(int nlines)  {

    if (top_line < end_line - length)
        top_line += nlines;
    else
        top_line = end_line;

    return;
    }


int editor_t::editbuf(void)  {
    char ch[2];
    int key, x = x1, y = y1, retcode = 0, done = 0;

    ch[1] = '\0';

    settextjustify(LEFT_TEXT, TOP_TEXT);
    setcolor(BLACK);

    refresh();

    while (!done)  {

        // recalculate (x,y) text position
        x = x1 + current_col*txwd;
        y = y1 + (current_line - top_line) * txht;
```

```
// show cursor
setcolor(BLACK); moveto(x-1,y); lineto(x-1,y+txht);

key = getkey();

// hide cursor
setcolor(WHITE); moveto(x-1,y); lineto(x-1,y+txht); setcolor(BLACK);

switch (key)  {

    case NEWLINE:
        insertline(current_line + 1);
        strcpy(&ebuf[current_line+1][0], &ebuf[current_line][current_col]);
        ebuf[current_line][current_col] = ENDLINE;
        current_line++;
        current_col = 0;
        refresh();
        break;

    case BACKSPACE:
        if (current_col > 0)  {
            current_col--;
            deletechar(current_line, current_col);
            setfillstyle(SOLID_FILL, WHITE);
            bar(x1, y, x2, y+txht-1);
            outtextxy(x1, y, &ebuf[current_line][0]);
            }
        else if (current_line > 0) {
            current_col = strlen(&ebuf[current_line-1][0]);
            strcat(&ebuf[current_line-1][0], &ebuf[current_line][0]);
            deleteline(current_line);
            current_line--;
            }
        break;

    case DELETE:
        if (ebuf[current_line][current_col] != ENDLINE)  {
            deletechar(current_line, current_col);
            setfillstyle(SOLID_FILL, WHITE);
            bar(x1, y, x2, y+txht-1);
            outtextxy(x1, y, &ebuf[current_line][0]);
            }
        break;

    case CTRL_Y:
        deleteline(current_line);
        refresh();
        break;

    case CTRL_Z:   // end edit
        done = 1;
        break;

    case UP:
        if (y > y1) current_line--;
        break;

    case DN:
        if (y < y2 && ebuf[current_line+1][0] != ENDBUF)  {
            current_line++;
            while (current_col > 0 &&
                   ebuf[current_line][current_col-1] == ENDLINE)
                current_col--;
            }
        break;

    case LEFT:
        if (x > x1)
            current_col--;
        break;

    case RIGHT:
        if (x < x2 && ebuf[current_line][current_col] != ENDLINE)
            current_col++;
        break;

    case ESC:
        retcode = 1;
        done = 1;
        break;

    default:
        ch[0] = (char)key;
        if (ebuf[current_line][current_col] == ENDLINE)  {
            ebuf[current_line][current_col] = ch[0];
            ebuf[current_line][current_col+1] = ENDLINE;
            outtextxy(x, y, ch);
            }
        else  {  // insert the character
            insertchar(current_line, current_col, ch[0]);
            setfillstyle(SOLID_FILL, WHITE);
            bar(x1, y, x2, y+txht-1);
            outtextxy(x1, y, &ebuf[current_line][0]);
            }
        current_col++;
        break;

    }

  }

return retcode;
}


char *editor_t::edit(char *buf, char *buffer_title)  {
    char *p = buf, *p2;
    int i = 0, j = 0;


    clearbuf();
    insertline(0);

    while (*p != NULL)  {
        if (*p == '\n')  {
            ebuf[i++][j] = ENDLINE;
            if (*(p+1) != NULL) insertline(i);
            j = 0;
            }
        else  {
            ebuf[i][j++] = *p;
            }
        p++;
        }

    show(buffer_title);
```

```
    if (editbuf() != 0)  {
        return NULL;
        }
    else  {
        p2 = &ebuf[0][0];
        while (*p2 != ENDLINE) p2++;
        *p2++ = '\n';
        i = 1;

        p = &ebuf[i++][0];
        while (*p != ENDBUF)  {
            while (*p != ENDLINE) *p2++ = *p++;
            *p2++ = '\n';
            p = &ebuf[i++][0];
            }
        }

    hide();

    return &ebuf[0][0];
    }


// end of file
```

```
//---[ effect.h ]-----------------------------------------------------

//
// class definitions for several graphics "special effects" objects
// such as shadows and bevels.
//


#ifndef _EFFECT_DEFINED
#define _EFFECT_DEFINED


extern char fine_texture_pattern[8];


class effect {
        protected:
                int left, top, right, bottom;

        public:
                effect(int, int, int, int);
                void display( void );
                void size(int, int, int, int);

        }; // end class effect


class bevel: public effect {
        private:
                int bright, dark;

        public:
                bevel(int l, int t, int r, int b, int brt, int drk);
                void display();

        }; // end class bevel


class inner_bevel : public bevel {

        public:
                inner_bevel(int, int, int, int, int, int);

        }; // end class inner_bevel


class outer_bevel : public bevel {

        public:
                outer_bevel(int, int, int, int, int, int);

        }; // end class outer_bevel


class shadow:effect  {

        public:
                shadow(int, int, int, int);

                void display( void );

        }; // end class shadow


#endif

// end of file
```

```
/*----[ effect.h ]-------------------------------------------------*/

#include <graphics.h>
#include "effect.h"


char fine_texture_pattern[] =
    {0x55, 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55, 0xaa};


effect::effect(int l, int t, int r, int b)  {
        left = l;
        top = t;
        right = r;
        bottom = b;
        return;
        }

void effect::display( void ) {
        return;
        }

void effect::size(int l, int t, int r, int b)  {
        left = l;
        top = t;
        right = r;
        bottom = b;
        return;
        }

bevel::bevel(int l, int t, int r, int b, int brt, int drk) : effect(l,t,r,b)  {
        bright = brt;
        dark = drk;
        return;
        }

void bevel::display()  {
        setcolor(bright);
        line(left-2, top-2, right+2, top-2);            // top outer
        line(left-2, top-1, right+2, top-1);            // top inner

        line(left-2, top-2, left-2, bottom+2);          // left outer
        line(left-1, top-2, left-1, bottom+2);          // left inner

        setcolor(dark);
        line(right+2, top-1, right+2, bottom+2);        // right outer
        line(right+1, top  , right+1, bottom+2);        // right inner

        line(left-1, bottom+2, right+2, bottom+2);      // effect outer
        line(left  , bottom+1, right+2, bottom+1);      // bottom inner

        return;
        }

inner_bevel::inner_bevel(int l, int t, int r, int b, int brt, int drk) :
        bevel(l,t,r,b,drk,brt)  {

        return;
        }

outer_bevel::outer_bevel(int l, int t, int r, int b, int brt, int drk) :
        bevel(l,t,r,b,brt,drk)  {

        return;
        }

shadow::shadow(int l, int t, int r, int b) : effect(l,t,r,b)  {

        return;
        }

void shadow::display( void )  {

        setfillpattern( fine_texture_pattern, WHITE );
        setfillstyle(USER_FILL, WHITE );
        bar( left+5, bottom+1, right+6, bottom+6 );
        bar( right+1, top+5, right+6, bottom );

        return;
        }


/*----[ end of file ]----------------------------------------------*/
```

```
//---[ list.h ]-------------------------------------------------

//
//  Class definition for a list "selectable" objects.
//
//  this is a generic list class -- change the type names and reuse.
//


#ifndef LIST_H
#define LIST_H


#include "select.h"


struct entry_t  {                   // doubly linked list entry
    selectable *obj;
    struct entry_t *prev, *next;
    };


class list_t  {

    public:
        list_t(void);
        ~list_t(void);

        void insert(selectable *obj);
        void append(selectable *obj);
        void remove(selectable *obj);
        int  length(void);

        selectable *first(void);
        selectable *next(void);
        selectable *last(void);
        selectable *prev(void);

    private:
        entry_t head, tail, *cursor;
        int n_entries;

    };


#endif


// end of file
```

```
//---[ list.cpp ]-----------------------------------------------

//
//  Member functions for the list class.  This list class has the
//  property that elements are always added at the head.  This list
//  only contains objects of class "selectable".
//


#include "list.h"


list_t::list_t(void)  {

    head.obj = (selectable *)NULL;
    head.next = &tail;
    head.prev = (entry_t *)NULL;

    tail.obj = (selectable *)NULL;
    tail.prev = &head;
    tail.next = (entry_t *)NULL;

    cursor = &head;
    n_entries = 0;

    return;
    }


list_t::~list_t(void)  {
    entry_t *temp;

    cursor = head.next;

    while (cursor != &tail)  {
        temp = cursor;
        cursor = cursor->next;
        delete temp;
        }

    return;
    }


void list_t::insert(selectable *obj)  {

    cursor = head.next;
    head.next = new entry_t;
    head.next->obj = obj;
    head.next->next = cursor;
    head.next->prev = &head;
    cursor->prev = head.next;

    n_entries++;

    return;
    }


void list_t::append(selectable *obj)  {

    cursor = tail.prev;
    tail.prev = new entry_t;
    tail.prev->obj = obj;
    tail.prev->prev = cursor;
    tail.prev->next = &tail;
    cursor->next = tail.prev;

    n_entries++;

    return;
    }


void list_t::remove(selectable *obj)  {
    int deleted = 0;

    cursor = head.next;

    while (cursor != &tail && !deleted)  {
        if (cursor->obj == obj)  {
            cursor->prev->next = cursor->next;
            cursor->next->prev = cursor->prev;
            cursor->obj = (selectable *)NULL;
            delete cursor;
            deleted = 1;
            n_entries--;
            }
        else  {
            cursor = cursor->next;
            }

        }

    return;
    }


selectable *list_t::first(void)  {

    cursor = head.next;

    return cursor->obj;
    }


selectable *list_t::next(void)  {

    if (cursor != &tail) cursor = cursor->next;

    return cursor->obj;
    }


selectable *list_t::last(void)  {

    cursor = tail.prev;

    return cursor->obj;
    }


selectable *list_t::prev(void)  {

    if (cursor != &head) cursor = cursor->prev;

    return cursor->obj;
    }
```

```
int list_t::length(void)  {

    return n_entries;
    }


// end of file
```

```
//---[ menu.h ]-----------------------------------------------------

//
// Class definition for a popup floating menu class.
//


#ifndef MENU_H
#define MENU_H


typedef int (*int_fn)();


class menu_t  {
    private:
        int top, left;
        int len, wid;
        int theight, nitems;
        int last_selected_item;
        char *items[12];
    //  int_fn actions[12];
        char *image;

    public:
        menu_t(char *, ... );
        ~menu_t( void );
        void show(int x, int y);
        void hide(void);
        int pick( void );

    };


int not_impl(void);
int null_action(void);


#endif


// end of file
```

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <graphics.h>

#include "msmouse.h"
#include "effect.h"
#include "menu.h"


menu_t::menu_t(char *title, ...)  {
    va_list ap;
    char *sarg;
    int_fn farg;
    int i = 1, temp;

    va_start(ap, title);

    wid = textwidth(title);
    items[0] = title;
    //  actions[0] = null_action;

    while ( (sarg = va_arg(ap, char *)) != (char *)NULL)  {
        items[i] = sarg;
        if ( (temp = textwidth(items[i])) > wid)
            wid = temp;
     // farg = va_arg(ap, int_fn );
     // actions[i] = farg;
        items[i+1] = (char *)NULL;
        i++;
        }

    nitems = i-1;
    theight = textheight("(");
    wid += (4 + textwidth("->"));
    len = theight * (nitems + 1) + 2 * nitems + 6;

    image = (char *)malloc(imagesize(left,top,left+wid+5,top+len+5));
    last_selected_item = 0;

    return;
    }  // end menu_t:::menu()


menu_t::~menu_t()  {
    free(image);
    }



void menu_t::show( int x, int y )  {
    int temp;
    left = x;
    top = y;

    if (left+wid+5 > getmaxx())
        left = getmaxx() - wid - 5;
    if (top+len+5 > getmaxy())
        top = getmaxy() - len - 5;

    Mouse.Hide();
    getimage(left, top, left+wid+5, top+len+5, image);

    setfillstyle(SOLID_FILL,WHITE);
    bar( left, top, left + wid - 1, top + len - 1 );

    setcolor(BLACK);
    rectangle( left, top, left+wid-1, top+len-1 );   // border around menu
    line( left, top + 3 + theight, left+wid-1, top + 3 + theight );

    settextjustify(CENTER_TEXT, TOP_TEXT);
    outtextxy( left + (wid/2), top + 3, items[0] );

    settextjustify(LEFT_TEXT, TOP_TEXT);
    temp = top + theight + 6;
    for (int i = 1; i <= nitems; i++)  {
        outtextxy( left+3, temp, items[i] );
        temp += theight+2;
        }

    // display shadow

    shadow sh( left, top, left+wid-1, top+len-1 );
    sh.display();

    Mouse.Show();
    return;
    }   // end menu_t::display



void menu_t::hide(void)  {

    Mouse.Hide();
    putimage(left,top,image,COPY_PUT);
    Mouse.Show();

    return;
    }



int menu_t::pick()  {
    int x, y, current = -1, newpos = last_selected_item;
    int oldx, oldy;

    int l = left+1;
    int r = left+wid-2;
    int t = top + theight + 5;
    int b = top + (2 * theight) + 6;
    int h = theight+2;                  // height of the highlight bar

    Mouse.Status(oldx,oldy);            // x, y are reference parameters

    Mouse.Move( left+wid/2, (last_selected_item+1) * h + t - 2 );

    //
    // current and newpos range from 0..n-1
    //

    while ( Mouse.Event(x,y) != RMouseUp )  {

        // update hilite position

        if (x > l && x < r && y > t && y < t + nitems * h )
            newpos = (y - t) / h;
        else
```

```
            newpos = -1;


        if (current != newpos)  {
            Mouse.Hide();

            if (current >= 0)  {           // unhighlight current
                setfillstyle(SOLID_FILL, WHITE);
                bar(l, t + current * h, r, b + current * h);
                setcolor(BLACK);
                outtextxy(l+2, t+l+current*h, items[ current+1 ] );
                }

            current = newpos;

            if (current >= 0)  {
                setfillstyle(SOLID_FILL, BLACK);
                bar(l, t + current * h, r , b + current * h);
                setcolor(WHITE);
                outtextxy( l+2, t + 1 + current * h, items[ current+1 ] );
                }

            Mouse.Show();
            }

        }  // end while

    if (current >= 0)  last_selected_item = current;

    Mouse.Move(oldx,oldy);

    return current;  // -1 == cancel, 0..n-1 == chose item i

    } // end menu_t::pick()


int not_impl()  {
    char *image = new char[imagesize(200,200,440,280)];
    getimage(200, 200, 440, 280, image);
    setcolor(WHITE);
    bar(200,200,440,280);
    settextjustify(CENTER_TEXT,CENTER_TEXT);
    outtextxy(320, 240, "This don't work!");
    getch();
    putimage(200, 200, image, COPY_PUT);
    delete image;
    return 0;
    }


int null_action()  { return 0; }
```

```
//---[ msgline.h ]-----------------------------------------------

//
// class definition for a message_line object that allows messages
// to be displayed and allows certain kinds of input to be entered.
//

#ifndef MSGLINE_H
#define MSGLINE_H


#include "symbol.h"


class messageline_t  {
    public:
        messageline_t(void);

        void show(void);
        void clear(void);
        void post(char *msg);
        void append(char *msg);
        char *gets(char *prompt);
        class_t *getobj(char *prompt);

    private:
        int miny, maxy;    // used by ison()
        int start_x;       // used for posting messages;
        char inbuf[80];
    };

extern messageline_t msgline;



#endif


// end of file
```

```
//---[ msgline.cpp ]-------------------------------------------------

//
// member functions for the message line object
//


#include <graphics.h>
#include <conio.h>
#include <string.h>

#include "msgline.h"
#include "msmouse.h"
#include "dispatch.h"
#include "util.h"



messageline_t::messageline_t(void)  {

    miny = textheight("H") * 4/3 + 6;
    maxy = (textheight("H") * 4/3 + 4) * 2 + 2;
    start_x = 4;
    inbuf[0] = '\0';

    return;
    };


void messageline_t::show(void)  {

    Mouse.Hide();

    setfillstyle(SOLID_FILL, BLACK);
    bar(1, miny, getmaxx(), maxy );
    setcolor(WHITE);
    moveto(1, maxy );
    lineto(1, miny );
    lineto(getmaxx(), miny);
    lineto(getmaxx(), maxy);

    Mouse.Show();
    return;
    }

void messageline_t::clear(void)  {

    Mouse.Hide();

    setfillstyle(SOLID_FILL, BLACK);
    bar(2, miny+1, getmaxx()-1, maxy );
    start_x = 4;

    Mouse.Show();
    return;
    }


void messageline_t::post(char *msg)  {
    int color = getcolor();

    Mouse.Hide();

    clear();

    setcolor(WHITE);
    settextjustify(LEFT_TEXT, TOP_TEXT);
    outtextxy(4, miny+3, msg);
    start_x += textwidth(msg);
    setcolor(color);

    Mouse.Show();
    return;
    }


void messageline_t::append(char *msg)  {
    int color = getcolor();

    Mouse.Hide();

    setcolor(WHITE);
    settextjustify(LEFT_TEXT, TOP_TEXT);
    outtextxy(start_x, miny+3, msg);
    start_x += textwidth(msg);
    setcolor(color);

    Mouse.Show();

    return;
    }


char *messageline_t::gets(char *prompt)  {
    char ch, *p = &inbuf[0];

    clear();
    post(prompt);

    while ( (ch = getch()) != 27 && (ch != 13) )  {
        if (ch == 8)  {  // backspace
            *(--p) = '\0';
            clear();
            post(prompt);
            append(inbuf);
            }
        else  {
            *p = ch;
            *(p+1) = '\0';
            append(p);
            p++;
            }
        }

    if (inbuf[0] == '\0' || ch == 27)  { // empty string == null string
        post("Operation Cancelled.");
        return (char *)NULL;
        }
    else  {
        clear();
        return &inbuf[0];
        }

    }  // end messageline_t::gets()


class_t *messageline_t::getobj(char *prompt)  {
```

```
    selectable *obj = (selectable *)NULL;
    char ch;
    int x, y, e;

    post(prompt);

    while (obj == (selectable *)NULL)  {

        while ( (e = get_event(ch,x,y)) != SELECT )  {
            if (e == KEY && ch == 27)  {
                post("Operation Cancelled.");
                return (class_t *)NULL;
                }
            }

        obj = dispatcher.find(x,y);

        if ( strcmp(obj->typeof(),"class_t") != 0 &&
             strcmp(obj->typeof(),"object_t") != 0 )  {
            obj = (selectable *)NULL;
            }

        }

    return (class_t *)obj;
    }


// end of file
```

```
//---[ msmouse.h ]----------------------------------------------------

// Class definiton for an object handle the mouse device.
//
// Copied from "Object Oriented Programming in Turbo C++", Wiley, 1990
//


#ifndef MSMOUSE_H
#define MSMOUSE_H


// Defines graphics mouse cursor styles

struct HotSpotStruct { int X, Y; };

struct MouseCursor {
    HotSpotStruct HotSpot;
    unsigned ScreenMask[16];
    unsigned CursorMask[16];
    };

extern const MouseCursor ArrowCursor;
extern const MouseCursor HandCursor;
extern const MouseCursor LeftRightCursor;
extern const MouseCursor UpDownCursor;
extern const MouseCursor CornerCursor;



// Mouse event codes

const unsigned Idle             = 0x0000;
const unsigned MouseDown        = 0xff01;
const unsigned LMouseDown       = 0xff01;
const unsigned RMouseDown       = 0xff02;
const unsigned MouseStillDown   = 0xff04;
const unsigned LMouseStillDown  = 0xff04;
const unsigned RMouseStillDown  = 0xff08;
const unsigned MouseUp          = 0xff10;
const unsigned LMouseUp         = 0xff10;
const unsigned RMouseUp         = 0xff20;
const unsigned MouseEnter       = 0xff40;
const unsigned MouseLeave       = 0xff80;
const unsigned MouseWithin      = 0xffc0;



// Mouse Button Masks

const unsigned LeftButton       = 0x0001;
const unsigned RightButton      = 0x0002;



// The video modes that the mouse is running under

enum VideoModeType { TextScrn, LowResGr, HerculesGr, Graphics };



// The Mouse Class

class MouseObject  {

    protected:
        int OldX, OldY;    // Used solely by Moved to keep position
        char OK;           // True if mouse initialized
        char MouseOff;     // True if mouse is disabled (Default)
        char LowRes;       // True if in 320 X 200 graphics mode
        char TextMode;     // True if in text mode

    public:
        int X, Y, Dx, Dy; // Keeps track of mouse's movement

        MouseObject(void);

        void Setup(VideoModeType VideoMode);
        int  DriverExists(void);
        int  SetupOK(void);
        void Hide(void);
        void Show(void);
        unsigned Status(int &Mx, int &My);
        unsigned ButtonStatus(void);
        int  PressCnt(unsigned ButtonMask);
        int  ReleaseCnt(unsigned ButtonMask);
        unsigned Event(int &Mx, int &My);
        unsigned WaitForAnyEvent(int &Mx, int &My);
        void WaitForEvent(unsigned E, int &Mx, int &My);
        int  Moved(void);
        void Move(int Mx, int My);
        void TurnOn(void);
        void TurnOff(void);
        int  Operating(void);
        void SetGrCursor(const MouseCursor &NewCursor);

    };  // end class MouseObject


extern MouseObject Mouse;


#endif


// end of file
```

```
// Mouse class: msmouse.cpp

#include <dos.h>
#include <stdlib.h>
#include "msmouse.h"

const int MsCall = 0x33;
const int Iret   = 0xcf;
const int False  = 0;
const int True   = 1;


MouseObject Mouse;

MouseObject::MouseObject(void)  {
    // Initializes mouse to a known state.
    // Mouse is not turned on yet.

    OK = False;
    MouseOff = True;

    return;
    }


int MouseObject::DriverExists(void)  {
    // Returns True if a mouse driver is installed.  This function makes
    // sure the interrupt vector location serviced by the mouse is
    // pointing to something -- hopefully the mouse driver!

    void far *address;

    // Look for NULL address or IRET instruction;

    address = getvect(MsCall);

    return (address != NULL) && (*(unsigned char far *)address != Iret);
    }

void MouseObject::Setup(VideoModeType VideoMode)  {
    // Initializes the mouse object by verifying that the mouse driver
    // exists, that the mouse responds to its initialization function,
    // by moving the mouse to the top left corner of the screen, and by
    // setting various internal variables.  Call the function SetupOk
    // after calling Setup to find out if the mouse initialization was
    // successful.  The VideoMode parameter specifies which video mode
    // the mouse is being used under.

    REGS regs;

    OK = DriverExists();

    if ( OK )  {
        // Fix up Hercules mode for page 0.  Use 5 for page 1.

        if (VideoMode == HerculesGr)  {
            *(char far *)MK_FP(0x0040, 0x0049) = 6;
            }

        regs.x.ax = 0;
        int86(MsCall, &regs, &regs);
        if (regs.x.ax == 0)  {
            OK = False;
            }
        }

    if (!OK)  {        // Mouse initialization failed
        TurnOff();     // Set the mouse state to off and return;
        return;
        }

    TurnOn();          // Set the mouse state to on.

    if (VideoMode == TextScrn) TextMode = True; else TextMode = False;
    if (VideoMode == LowResGr) LowRes   = True; else LowRes   = False;

    OldX = OldY = 0;  // Initialize various variables
    X = Y = Dx = Dy = 0;

    Move(0,0);         // Move mose to the top left of the screen

    return;
    }


int MouseObject::SetupOK(void)  {
    // Returns True only if mouse initialization was successful

    return OK;
    }


void MouseObject::Hide(void)  {
    // Hides the mouse cursor.  Call this function to remove the mouse
    // cursor from the screen.

    REGS regs;

    if (!Operating()) return;

    regs.x.ax = 2;
    int86(MsCall, &regs, &regs);

    return;
    }


void MouseObject::Show(void)  {
    // Display the mouse cursor

    REGS regs;

    if (!Operating()) return;

    regs.x.ax = 1;
    int86(MsCall, &regs, &regs);

    return;
    }


unsigned MouseObject::Status(int &Mx, int &My)  {
    // This general function returns the location of the mouse in Mx, My
    // and the current status of the buttons in the return value

    REGS regs;

    if (!Operating()) return (Mx = My = 0);
```

```
    regs.x.ax = 3;
    int86(MsCall, &regs, &regs);

    Mx = regs.x.cx;  My = regs.x.dx;

    if (TextMode)  {
        Mx >>= 3;  // Adjust for text coordinates
        My >>= 3;
        }

    if (LowRes)  {
        Mx >>= 1;  // Adjust for 320 X 200 coordinates
        }

    return regs.x.bx;
    }


unsigned MouseObject::ButtonStatus(void)  {
    // Returns the status of the mouse buttons

    int Mx, My;

    if (!Operating()) return 0;

    return Status(Mx, My);
    }


int MouseObject::PressCnt(unsigned ButtonMask)  {
    // Returns number of times the button has been pressed since
    // last time called.

    REGS regs;

    if (!Operating()) return 0;

    regs.x.ax = 5;
    regs.x.bx = ButtonMask >> 1;  // Button selector
    int86(MsCall, &regs, &regs);

    return regs.x.bx;
    }


int MouseObject::ReleaseCnt(unsigned ButtonMask)  {
    // Returns number of times the button has been released since
    // last time called.

    REGS regs;

    if (!Operating()) return 0;

    regs.x.ax = 6;
    regs.x.bx = ButtonMask >> 1;  // Button selector
    int86(MsCall, &regs, &regs);

    return regs.x.bx;
    }


unsigned MouseObject::Event(int &Mx, int &My)  {
    // Gets the last mouse event.  The left mouse button has priority
    // over the mouse button

    unsigned E;

    if (!Operating())  return (Mx = My = 0);

    if ( (E = Status(Mx, My)) == 0 )  {
        // No mouse button down, but maybe there was a button press that
        // was missed.  If not, check to see whether a button release was
        // missed.  Favor the left mouse button.

        if ( PressCnt(LeftButton) > 0)
            E = LMouseDown;

        else if ( PressCnt(RightButton) > 0)
            E = RMouseDown;

        else if ( ReleaseCnt(LeftButton) > 0)
            E = LMouseUp;

        else if ( ReleaseCnt(RightButton) > 0)
            E = RMouseUp;
        }

    else {
        // A mouse button is down

        if (E & LeftButton)  {
            if (PressCnt(LeftButton) > 0)
                E = LMouseDown;           // Must have just been pressed
            else
                E = LMouseStillDown;      // already down
            }

        else if (PressCnt(RightButton) > 0)
            E = RMouseDown;

        else
            E = RMouseStillDown;
        }

    return E;
    }


unsigned MouseObject::WaitForAnyEvent(int &Mx, int &My)  {
    // Waits for a mouse event to occur and returns its code

    unsigned E;

    if (!Operating()) return( Mx = My = 0);

    while ( (E = Event(Mx, My)) == Idle);  // Loop until an event occurs

    return E;
    }


void MouseObject::WaitForEvent(unsigned E, int &Mx, int &My)  {
    // Waits for the event E to occur and returns its coordinates

    if (!Operating()) { Mx = My = 0;  return; }
```

```
    while ( Event(Mx, My) != E);  // Loop until event E occurs

    return;
    }


int MouseObject::Moved(void)  {
    // Test to see if the mouse has moved since the last time this
    // function was called.

    if (!Operating()) return False;

    OldX = X; OldY = Y;
    Status(X, Y);
    Dx = X - OldX; Dy = Y - OldY;

    return (Dx != 0) || (Dy != 0);
    }


void MouseObject::Move(int Mx, int My)  {
    // Moves the mouse cursor

    REGS regs;

    if (!Operating()) return;

    regs.x.ax = 4;
    regs.x.cx = Mx;
    regs.x.dx = My;

    if (TextMode)  {     // Adjust for text coordinates
        regs.x.cx <<= 3;
        regs.x.dx <<= 3;
        }
    if (LowRes)  {
        regs.x.cx <<= 1;
        }

    int86(MsCall, &regs, &regs);

    return;
    }


void MouseObject::TurnOn(void)  {
    // Enables the mouse code

    if (OK && MouseOff)  {
        MouseOff = False;
        Show();
        }

    return;
    }


void MouseObject::TurnOff(void)  {
    // Disables the mouse code.  This is useful when you don't want to
    // use the mouse bu the code already has mouse calls in it.

    if (OK && !MouseOff)  {
        Hide();
        MouseOff = True;
        }

    return;
    }


int MouseObject::Operating(void)  {
    // Returns a boolean falg that is true only if the mouse object has
    // ben enabled.  This is the default state.

    return !MouseOff;
    }


void MouseObject::SetGrCursor(const MouseCursor &NewCursor)  {
    // Sets the graphics mouse cursor to the type specified

    REGS regs;
    SREGS sregs;

    if (!Operating())  return;

    regs.x.ax = 9;
    regs.x.bx = NewCursor.HotSpot.X;
    regs.x.cx = NewCursor.HotSpot.Y;
    regs.x.dx = FP_OFF(NewCursor.ScreenMask);
    sregs.es  = FP_SEG(NewCursor.ScreenMask);
    int86x(MsCall, &regs, &regs, &sregs);

    return;
    }


//---[ end of file ]-------------------------------------------------
```

```
#include "msmouse.h"


const MouseCursor ArrowCursor = {
    { 0, 0 },  // Set hot spot to tip of arrow

    { 0x3fff, 0x1fff, 0x0fff, 0x07ff,   // Screen Mask
      0x03ff, 0x01ff, 0x00ff, 0x007f,
      0x003f, 0x00ff, 0x01ff, 0x10ff,
      0x30ff, 0xf87f, 0xf87f, 0xfc7f },
    { 0x0000, 0x4000, 0x6000, 0x7000,   // Screen Mask
      0x7800, 0x7c00, 0x7e00, 0x7f00,
      0x7f80, 0x7e00, 0x7c00, 0x4600,
      0x0600, 0x0300, 0x0300, 0x0000 }
    };


const MouseCursor CornerCursor = {
    { 0, 0 },  // Set hot spot to tip of arrow

    { 0x0000, 0x7fff, 0x7fff, 0x7fff,   // Screen Mask
      0x7fff, 0x7fff, 0x7fff, 0x7fff,
      0x7fff, 0x7fff, 0x7fff, 0x7fff,
      0x7fff, 0x7fff, 0x7fff, 0x7fff },
    { 0x0000, 0x0000, 0x0000, 0x0000,   // Screen Mask
      0x0000, 0x0000, 0x0000, 0x0000,
      0x0000, 0x0000, 0x0000, 0x0000,
      0x0000, 0x0000, 0x0000, 0x0000 }
    };


const MouseCursor HandCursor = {
    { 4, 0 },  // Set hot spot to tip of pointing finger

    { 0xf3ff, 0xe1ff, 0xe1ff, 0xe1ff,   // Screen Mask
      0xe001, 0xe000, 0xe000, 0xe000,
      0x8000, 0x0000, 0x0000, 0x0000,
      0x0000, 0x0000, 0x8001, 0xc003 },
    { 0x0c00, 0x1200, 0x1200, 0x1200,   // Screen Mask
      0x13fe, 0x1249, 0x1249, 0x1249,
      0x7249, 0x9001, 0x9001, 0x9001,
      0x8001, 0x8001, 0x4002, 0x3ffc }
    };


const MouseCursor LeftRightCursor = {
    { 8, 8 },  // Set hot spot to middel of arrow

    { 0xffff, 0xffff, 0xfbdf, 0xf3cf,   // Screen Mask
      0xe3c7, 0xc003, 0x8001, 0x0000,
      0x8001, 0xc003, 0x3ec7, 0xf3cf,
      0xfbdf, 0xffff, 0xffff, 0xffff },
    { 0x0000, 0x0000, 0x0420, 0x0c30,   // Screen Mask
      0x1428, 0x27e4, 0x4002, 0x8001,
      0x4002, 0x27e4, 0x1428, 0x0c30,
      0x0420, 0x0000, 0x0000, 0x0000 }
    };


const MouseCursor UpDownCursor = {
    { 8, 8 },  // Set hot spot to middle of arrow

    { 0xfeff, 0xfcff, 0xf83f, 0xf01f,   // Screen Mask
      0xe00f, 0xc007, 0xf83f, 0xf83f,
      0xf83f, 0xf83f, 0xc007, 0xe00f,
      0xf01f, 0xf83f, 0xfc7f, 0xfeff },
    { 0x0100, 0x0280, 0x0440, 0x0820,   // Screen Mask
      0x1010, 0x3c78, 0x0440, 0x0440,
      0x0440, 0x0440, 0x3c78, 0x1010,
      0x0820, 0x0440, 0x0280, 0x0100 }
    };


//---[ end of file ]---------------------------------------------------
```

```
//---[ property.h ]---------------------------------------------------

//
// class for attributes and services.  In this system, there is no
// difference except in name.
//

#ifndef PROPERTY_H
#define PROPERTY_H


#include "select.h"
#include "menu.h"


class property_t : public selectable  {

    public:
        property_t(int x, int y, char *s, class_t *object);
        ~property_t(void);

        char *typeof(void) { return "property_t"; }
        void show(void);
        void move(int dx, int dy);
        selectable *select(void);
        int act(int x, int y);

    protected:
        class_t *obj;

    private:

    };


class attribute_t : public property_t  {

    public:
        attribute_t(int x, int y, char *s, class_t *object);
        ~attribute_t(void);

    private:

    };


class service_t : public property_t  {

    public:
        service_t(int x, int y, char *s, class_t *object);
        ~service_t(void);

    private:

    };


extern menu_t attribute_menu;
extern menu_t service_menu;

#endif


// end of file
```

```
//---[ property.cpp ]-------------------------------------------------

//
// member functions for the property class (attributes and services)
//


#include <graphics.h>
#include <string.h>

#include "property.h"
#include "msgline.h"
#include "diagram.h"
#include "canvas.h"



property_t::property_t(int x, int y, char *s, class_t *object)  {

    strcpy(name,s);

    x1 = x;
    y1 = y;
    x2 = x1 + textwidth(name);
    y2 = y1 + textheight(name) * 4/3;

    text = NULL;

    obj = object;

    return;
    }


property_t::~property_t(void)  {

    return;
    }


void property_t::show(void)  {

    settextjustify(LEFT_TEXT, TOP_TEXT);
    outtextxy(x1,y1,name);

    return;
    }


void property_t::move(int dx, int dy)  {

    x1 += dx;
    y1 += dy;
    x2 += dx;
    y2 += dy;

    return;
    }


selectable *property_t::select(void)  {

    msgline.post(label);
    msgline.append(name);
    msgline.append(" selected.");

    return this;
    }


int property_t::act(int x, int y)  {
    int selection;
    char *s;

    selection = selectable::act(x,y);

    switch (selection)  {

        case 0:  // rename
            if ( (s = msgline.gets("Enter new name: ")) != NULL )  {
                strcpy(name, s);
                obj->fit(s);
                obj->show();
                current_diagram->refresh();
                }
            msgline.post(label);
            msgline.append("name changed.");
            break;

        case 1:  // note
            annote();
            break;

        case 2:  // type or proto
            msgline.post("Add type information.");
            break;

        case 3:  // delete
            if (strcmp(label,"Attribute ") == 0)
                obj->remove_attr(this);
            else
                obj->remove_serv(this);

            canvas.clear();
            obj->show();
            current_diagram->refresh();

            msgline.post(label);
            msgline.append(name);
            msgline.append(" deleted from ");
            msgline.append(obj->label);
            msgline.append(obj->name);
            msgline.append(".");
            // delete this;
            break;

        default:
            break;
        }

    return selection;
    }


attribute_t::attribute_t(int x, int y, char *s, class_t *object) :
            property_t(x,y,s, object) {
```

```
    strcpy(label, "Attribute ");

    menu = &attribute_menu;

    return;
    }


attribute_t::~attribute_t(void)  {

    return;
    }


service_t::service_t(int x, int y, char *s, class_t *object) :
          property_t(x,y,s,object) {

    strcpy(label, "Service ");

    menu = &service_menu;

    return;
    }


service_t::~service_t(void)  {

    return;
    }


// end of file
```

```
//---[ select.h ]-----------------------------------------------

//
// virtual class from which mouse selectable objects are derived
//
// abstract class to serve as base class for all mouse selectable
// screen objects.  pointers to this class are used by the dispatcher
// object.
//


#ifndef SELECT_H
#define SELECT_H


#include <stdio.h>

#include "menu.h"


class selectable  {
    public :
        virtual char *typeof(void)       { return "selectable"; }
        virtual void show(void)          { return; }
        virtual int  ison(int x, int y);
        virtual int  isa(char *s);
        virtual selectable *select(void);
        virtual int  act(int x, int y);
        virtual void annote(void);
        virtual void save(FILE *fp);

        char name[80];          // object name
        char label[20];         // class label

    protected:
        int x1, y1, x2, y2;
        menu_t *menu;
        char *text;
    };


#endif


// end of file
```

```
//---[ select.cpp ]-------------------------------------------------------

//
//  The default member functions for all subclasses of selectable.
//  Class selectable is an abstract class: it may never be instantiated,
//  but several classes are derived from it.
//


#include <string.h>

#include "select.h"
#include "diagram.h"
#include "dispatch.h"
#include "editor.h"
#include "msmouse.h"
#include "msgline.h"



int selectable::ison(int x, int y)  {

    return (x >= x1 && x <= x2 && y >= y1 && y <= y2);
    }


int selectable::isa(char *s)  {

    return strcmp(s, selectable::typeof()) == 0;
    }


selectable *selectable::select(void)  {

    msgline.post(label);
    msgline.append(name);
    msgline.append(" selected.");

    dispatcher.promote(this);

    show();

    return this;
    }


int selectable::act(int x, int y)  {
    int selection;

    menu->show(x,y);
    selection = menu->pick();
    menu->hide();

    if (selection == -1)  {
        msgline.post("Operation cancelled.");
        }

    return selection;
    }


void selectable::annote(void)  {
    char bufname[80], *p;

    Mouse.Hide();

    sprintf(bufname, "Notes for %s%s", label, name);

    if ( (p = editor.edit(text, bufname)) == NULL)  {
        msgline.post("Operation Cancelled.");
        }
    else  {
        if (text != NULL) delete text;
        text = new char[strlen(p)+1];
        strcpy(text,p);
        msgline.post("Notes added for ");
        msgline.append(label);
        msgline.append(name);
        msgline.append(".");
        }


    current_diagram->refresh();

    Mouse.Show();

    return;
    }


void selectable::save(FILE *fp)  {

    fprintf(fp,"");

    return;
    }

// end of file
```

```
//---[ symbol.h ]-----------------------------------------------------

//
// Class definitions for the Class and Object symbols
//


#ifndef SYMBOL_H
#define SYMBOL_H

#include <stdio.h>

#include "select.h"
#include "menu.h"
#include "list.h"


struct point_struct {
    int x,y;
    };
typedef struct point_struct point_t;
typedef point_t cpoints[4];



class class_t : public selectable  {
    public:
        class_t(void) { return; }
        class_t(int x, int y, char *s);
        class_t(FILE *);
        ~class_t(void);

        virtual void show(void);
        void move_to(int x, int y);
        void connect(class_t *to, int &x1, int &y1, int &x2, int &y2);
        void add_attr(char *s);
        void add_serv(char *s);
        void remove_attr(selectable *obj);
        void remove_serv(selectable *obj);
        void rename(char *s);
        void save(FILE *);
        void fit(char *s);

        char *typeof(void)  { return "class_t";  }
        int  isa(char *s);
        selectable *select(void);
        int act(int x, int y);

    protected:
        void determine_cpoints(void);

        int attr_y, serv_y;             // top of attr and serv sections
        int txht;                       // text height
        int bdwd;                       // width of border
        cpoints cpts;                   // connect points
        list_t *alist, *slist;          // attributes and services;

    };



class object_t : public class_t  {
    public:
        object_t(int x, int y, char *s);
        object_t(FILE *);
        ~object_t(void);

        int  isa(char *s);
        void show(void);

        char *typeof(void)  { return "object_t";  }

    private:

    };


extern menu_t class_menu;
extern menu_t object_menu;


#endif


// end of file
```

```
//---[ symbol.cpp ]-------------------------------------------------


#include <stdio.h>
#include <string.h>
#include <graphics.h>
#include <math.h>

#include "symbol.h"
#include "property.h"
#include "dispatch.h"
#include "msmouse.h"
#include "msgline.h"
#include "canvas.h"
#include "effect.h"
#include "diagram.h"
#include "util.h"



//
// member functions for the "class" symbol
//

class_t::class_t(int x, int y, char *s)  {

    txht = textheight("H") * 4/3;
    bdwd = 0;

    strcpy(name,s);
    strcpy(label, "Class ");

    x1 = x;
    x2 = x1 + textwidth(name) + 8;

    y1     = y;
    attr_y = y1 + txht + 4;
    serv_y = attr_y + txht + 4;
    y2     = serv_y + txht + 4;

    menu = &class_menu;
    text = NULL;

    alist = new list_t;
    slist = new list_t;

    return;
    }


class_t::class_t(FILE *fp)  {   // must scan itself
    char buf[40];
    int x, y, end_of_class = 0;

    txht = textheight("H") * 4/3;
    bdwd = 0;

    getqs(fp, buf);
    strcpy(name,buf);
    strcpy(label, "Class ");

    fscanf(fp, " (%d,%d) { ", &x, &y);

    x1 = x;
    x2 = x1 + textwidth(name) + 8;

    y1     = y;
    attr_y = y1 + txht + 4;
    serv_y = attr_y + txht + 4;
    y2     = serv_y + txht + 4;

    menu = &class_menu;
    text = NULL;

    alist = new list_t;
    slist = new list_t;

    dispatcher.insert(this);

    // now read in attributes and services

    while (!end_of_class)  {

        fscanf(fp, " %s ", buf);

        if (strcmp(buf,"Attribute") == 0)  {
            getqs(fp,buf);
            add_attr(buf);
            }

        else if (strcmp(buf,"Service") == 0)  {
            getqs(fp,buf);
            add_serv(buf);
            }

        else if (strcmp(buf, "}") == 0)  {
            end_of_class = 1;
            }

        else {
            current_diagram->error = 1;
            return;
            }
        }

    return;
    }


class_t::~class_t(void)  {

    return;
    }


void class_t::fit(char *s)  {

    if ( textwidth(s) > (x2 - x1 - 8 - 2*bdwd) ) {
        x2 = x1 + textwidth(s) + 8 + 2*bdwd;
        }

    return;
    }


int class_t::isa(char *s)  {

    return (strcmp(s,class_t::typeof()) == 0) || selectable::isa(s);
```

```
    }

void class_t::show(void)  {
    selectable *obj;

    Mouse.Hide();

    setfillstyle(SOLID_FILL, WHITE);
    bar(x1,y1,x2,y2);

    setcolor(BLACK);
    rectangle(x1, y1, x2, y2);
    moveto(x1, attr_y);
    lineto(x2, attr_y);
    moveto(x1, serv_y);
    lineto(x2, serv_y);

    settextjustify(CENTER_TEXT, TOP_TEXT);
    outtextxy( (x1+x2)/2, y1+3, name);

    obj = alist->first();
    while (obj != (selectable *)NULL)  {
        obj->show();
        obj = alist->next();
        }
    obj = slist->first();
    while (obj != (selectable *)NULL)  {
        obj->show();
        obj = slist->next();
        }

    Mouse.Show();

    return;
    }


void class_t::move_to(int x, int y)  {
    int dx = x - x1, dy = y - y1;
    property_t *obj;

    x1 = x;
    y1 = y;
    x2 += dx;
    y2 += dy;
    attr_y += dy;
    serv_y += dy;

    determine_cpoints();

    obj = (property_t *)alist->first();
    while (obj != (selectable *)NULL)  {
        obj->move(dx,dy);
        obj = (property_t *)alist->next();
        }

    obj = (property_t *)slist->first();
    while (obj != (selectable *)NULL)  {
        obj->move(dx,dy);
        obj = (property_t *)slist->next();
        }

    return;
    }

void class_t::add_attr(char *s)  {
    int top, left;
    property_t *attr;

    top = attr_y + 3 + alist->length() * txht;
    left = x1 + bdwd + 4;

    attr = new attribute_t(left, top, s, this);
    alist->append(attr);

    if (alist->length() > 1)  {
        serv_y += txht;
        y2 += txht;
        }

    fit(attr->name);   // widen if needed

    property_t *serv = (property_t *)slist->first();
    while (serv != NULL)  {
        serv->move(0,txht);
        serv = (property_t *)slist->next();
        }

    dispatcher.insert(attr);

    return;
    }


void class_t::add_serv(char *s)  {
    int top, left;
    property_t *serv;

    top = serv_y + 3 + slist->length() * txht;
    left = x1 + bdwd + 4;

    serv = new service_t(left, top, s, this);
    slist->append(serv);

    if (slist->length() > 1)  {
        y2 += txht;
        }

    fit(serv->name);

    dispatcher.insert(serv);

    return;
    }


void class_t::remove_attr(selectable *obj)  {
    property_t *prop;

    prop = (property_t *)alist->first();

    while( prop != NULL  &&  prop != obj )
        prop = (property_t *)alist->next();

    if (prop != NULL)  {
        if (alist->length() > 1)  {
```

```
            serv_y -= txht;
            y2 -= txht;
            }

        prop = (property_t *)alist->next();
        while (prop != NULL)  {
            prop->move(0,-txht);
            prop = (property_t *)alist->next();
            }

        alist->remove(obj);

        prop = (property_t *)slist->first();
        while (prop != NULL)  {
            prop->move(0,-txht);
            prop = (property_t *)slist->next();
            }

        dispatcher.remove(obj);

        }

    return;
    }


void class_t::remove_serv(selectable *obj)  {
    property_t *prop;

    prop = (property_t *)slist->first();

    while( prop != NULL  &&  prop != obj )
        prop = (property_t *)slist->next();

    if (prop != NULL)  {
        if (slist->length() > 1)
            y2 -= txht;

        prop = (property_t *)slist->next();
        while (prop != NULL)  {
            prop->move(0,-txht);
            prop = (property_t *)slist->next();
            }

        slist->remove(obj);
        dispatcher.remove(obj);
        }

    return;
    }


void class_t::rename(char *s)  {

    strcpy(name, s);

    if ( textwidth(s) > (x2 - x1 - 8) ) {
        x2 = x1 + textwidth(s) + 8;
        }
    else if (alist->length() == 0 && slist->length() == 0)  {
        x2 = x1 + textwidth(s) + 8 + 2*bdwd;
        }

    return;
    }


selectable *class_t::select(void)  {
    selectable *obj;

    selectable::select();
    current_diagram->promote(this);

    // promote all attributes of this class/object

    obj = alist->first();
    while (obj != (selectable *)NULL)  {
        dispatcher.promote(obj);
        obj = alist->next();
        }

    // promote all services of this class/object

    obj = slist->first();
    while (obj != (selectable *)NULL)  {
        dispatcher.promote(obj);
        obj = slist->next();
        }

    // promote all connections connected to this object

    obj = current_diagram->first_connection(this);
    while (obj != NULL)  {
        dispatcher.promote(obj);
        obj->show();
        obj = current_diagram->next_connection(this);
        }

    return this;
    }


int class_t::act(int x, int y)  {
    int selection, new_x, new_y;
    char *s, ch;

    selection = selectable::act(x,y);

    switch (selection)  {

        case 0:  // dup
            msgline.post("Duplicate the current Class/Object.");
            break;

        case 1:  // move
            Mouse.Move(x1,y1);
            Mouse.SetGrCursor(CornerCursor);
            current_diagram->remove(this);
            current_diagram->refresh();
            msgline.post("Position upper left corner, then click to place Class/Object.")
;
            while ( get_event(ch, new_x, new_y) != SELECT );
            Mouse.SetGrCursor(ArrowCursor);
            move_to(new_x,new_y);
            current_diagram->add(this);
            current_diagram->refresh();
            msgline.post("Move completed.");
            break;
```

```
        case 2:  // delete
            if (current_diagram->first_connection(this) == NULL) {
                dispatcher.remove(this);
                current_diagram->remove(this);
                current_diagram->refresh();
                msgline.post(name);
                msgline.append(" deleted.");
                delete this;
                }
            else  {
                msgline.post("Connot delete class/object with connections.");
                }
            break;

        case 3:  // note
            annote();
            break;

        case 4:  // attr
            s = msgline.gets("Enter attribute name: ");
            if (s != (char *)NULL)  {
                add_attr(s);
                msgline.post("Attribute ");
                msgline.append(s);
                msgline.append(" added.");
                }
            show();
            current_diagram->refresh();
            break;

        case 5:  // serv
            s = msgline.gets("Enter service name: ");
            if (s != (char *)NULL)  {
                add_serv(s);
                msgline.post("Service ");
                msgline.append(s);
                msgline.append(" added.");
                }
            show();
            current_diagram->refresh();
            break;

        case 6:  // rename
            rename(msgline.gets("Enter new name: ") );
            show();
            current_diagram->refresh();
            msgline.post("Name changed.");
            break;

        default:
            msgline.post("Operation Cancelled.");
            break;
        }

    return selection;
    }


void class_t::determine_cpoints(void)  {

    cpts[0].x = x1;
    cpts[0].y = cpts[2].y = (y1 + y2) / 2;
    cpts[2].x = x2;

    cpts[1].y = y1;
    cpts[1].x = cpts[3].x = (x1 + x2) / 2;
    cpts[3].y = y2;

    return;
    }


void class_t::connect(class_t *to, int &x1, int &y1, int &x2, int &y2)  {

    int i = 0, j = 0, k, l;
    long d, sd = 640000;
    long X1,Y1,X2,Y2;

    //
    // find the two points, one from each vector, that are closest to each
    // other.  We use a modified distance formula:
    //
    //     d = abs(x2 - x1) + abs(y2 - y1)
    //
    // which results in the same ordering relation as the real distance
    // formula:
    //
    //     d = sqrt( (x2 - x1)^2 + (y2 - y1)^2 )
    //
    // but is easier (and faster) to compute.
    //

    determine_cpoints();
    to->determine_cpoints();

    for (k = 0; k < 4; k++)  {
        for (l = 0; l < 4; l++)  {
            X1 = cpts[k].x;      Y1 = cpts[k].y;
            X2 = to->cpts[l].x;  Y2 = to->cpts[l].y;

            d = (X2 - X1) * (X2 - X1) + (Y2 - Y1) * (Y2 - Y1);

            if (d < sd)  {
                i = k;
                j = l;
                sd = d;
                }
            }
        }

    x1 = cpts[i].x;     y1 = cpts[i].y;
    x2 = to->cpts[j].x; y2 = to->cpts[j].y;

    return;
    }


void class_t::save(FILE *fp)  {
    property_t *obj;

    fprintf(fp, "%s \"%s\" (%d,%d) {\n", label, name, x1, y1);

    obj = (property_t *)alist->first();
    while (obj != (property_t *)NULL)  {
        fprintf(fp, "\t%s \"%s\"\n", obj->label, obj->name);
        obj = (property_t *)alist->next();
```

```
        }

    obj = (property_t *)slist->first();
    while (obj != (property_t *)NULL)  {
        fprintf(fp, "\t%s \"%s\"\n", obj->label, obj->name);
        obj = (property_t *)slist->next();
        }

    fprintf(fp, "\t}\n\n");

    return;
    }



//
//  member functions for the object_t class
//


object_t::object_t(int x, int y, char *s) : class_t(x, y, s)  {

    strcpy(label, "Object ");

    bdwd = 4;

    x2 = x1 + textwidth(name) + 8 + 2 * bdwd;

    attr_y += bdwd;
    serv_y += bdwd;
    y2     += 2 * bdwd;

    menu = &object_menu;

    alist = new list_t;
    slist = new list_t;

    return;
    }


object_t::object_t(FILE *fp)  {   // must scan itself
    char buf[40];
    int x, y, end_of_class = 0;

    txht = textheight("H") * 4/3;
    bdwd = 4;

    getqs(fp,buf);
    strcpy(name,buf);
    strcpy(label, "Object ");

    fscanf(fp, " (%d,%d) { ", &x, &y );

    x1 = x;
    x2 = x1 + textwidth(name) + 8 + 2 * bdwd;

    y1     = y;
    attr_y = y1 + txht + 4 + bdwd;
    serv_y = attr_y + txht + 4 + bdwd;
    y2     = serv_y + txht + 4 + 2*bdwd;

    menu = &object_menu;

    alist = new list_t;
    slist = new list_t;

    dispatcher.insert(this);

    // now read in attributes and services

    while (!end_of_class)  {

        fscanf(fp, " %s ", buf);

        if (strcmp(buf,"Attribute") == 0)  {
            getqs(fp,buf);
            add_attr(buf);
            }

        else if (strcmp(buf,"Service") == 0)  {
            getqs(fp,buf);
            add_serv(buf);
            }

        else if (strcmp(buf, "}") == 0)  {
            end_of_class = 1;
            }

        else {
            current_diagram->error = 1;
            return;
            }
        }

    return;
    }


object_t::~object_t(void)  {

    return;
    }


int object_t::isa(char *s)  {

    return (strcmp(s,typeof()) == 0) || class_t::isa(s);
    }


void object_t::show(void)  {
    selectable *obj;

    Mouse.Hide();

    setfillpattern(fine_texture_pattern, WHITE);
    setfillstyle(USER_FILL, WHITE);
    bar(x1,y1,x2,y2);

    setfillstyle(SOLID_FILL, WHITE);
    bar(x1+bdwd-1,y1+bdwd-1,x2-bdwd+1,y2-bdwd+1);

    setcolor(BLACK);
    rectangle(x1+bdwd, y1+bdwd, x2-bdwd, y2-bdwd);
    moveto(x1+bdwd, attr_y);
    lineto(x2-bdwd, attr_y);
    moveto(x1+bdwd, serv_y);
```

```
    lineto(x2-bdwd, serv_y);

    settextjustify(CENTER_TEXT, TOP_TEXT);
    outtextxy((x1+x2)/2, y1+bdwd+3, name);

    obj = alist->first();
    while (obj != (selectable *)NULL)  {
        obj->show();
        obj = alist->next();
        }
    obj = slist->first();
    while (obj != (selectable *)NULL)  {
        obj->show();
        obj = slist->next();
        }

    Mouse.Show();

    return;
    }


// end of file
```

```
//---[ titlebar.h ]-------------------------------------------------

//
// class definition for the title bar object
//


#ifndef TITLEBAR_H
#define TITLEBAR_H


#include "select.h"


class titlebar_t : public selectable {
    public:
        titlebar_t(void);
        ~titlebar_t(void);

        void show(void);
        void set_title(char *s);
        int  save_current_diagram(void);
        void start_new_diagram(void);
        void retrieve_diagram(void);

        char *typeof(void)  { return "titlebar_t"; }
        selectable *select(void);
        int act(int x, int y);

    private:

    };

extern titlebar_t titlebar;


#endif


// end of file
```

```
//---[ titlebar.cpp ]-------------------------------------------

//
// member functions for the title bar object
//

#include <graphics.h>
#include <string.h>

#include "titlebar.h"
#include "msmouse.h"
#include "msgline.h"
#include "diagram.h"
#include "dispatch.h"
#include "canvas.h"
#include "connect.h"
#include "util.h"


titlebar_t::titlebar_t(void)  {

    strcpy(name, "<untitled>");
    strcpy(label, "Title bar");

    x1 = 1;
    y1 = 1;
    x2 = getmaxx();
    y2 = textheight("H") * 4/3 + 5;      // allow for tails (e.g. 'y')

    menu = new menu_t("File", "New", "Open", "Save", "Close", "Name", "Exit", NULL);

    return;
    }


void titlebar_t::show(void)  {

    Mouse.Hide();

    setfillstyle(SOLID_FILL, BLACK);
    bar(x1, y1, x2, y2);

    setcolor(WHITE);
    moveto(x1, y2);
    lineto(x1, y1);
    lineto(x2, y1);
    lineto(x2, y2);

    // add system menu icon and initial title
    settextjustify(LEFT_TEXT, TOP_TEXT);
    outtextxy(x1+3,y1+3,"File");
    settextjustify(CENTER_TEXT, TOP_TEXT);
    outtextxy(getmaxx()/2, 4, name);

    Mouse.Show();
    return;
    }


void titlebar_t::set_title(char *s)  {

    if ( s == (char *)NULL || *s == '\0' )
        strcpy(name,"<untitled>");
    else
        strcpy(name,s);

    return;
    }


//
//  inherited member function overridden: titlebar should never be
//  promoted.
//

selectable *titlebar_t::select(void)  {

    msgline.post("Title bar selected.");

    return this;
    }



int titlebar_t::act(int x, int y)  {
    int selection;
    char buf[80], *s;

    extern int exit_flag;

    selection = selectable::act(x,y);

    switch (selection)  {
        case 0:      // new
            start_new_diagram();
            break;

        case 1:      // open
            retrieve_diagram();
            break;

        case 2:      // save
            save_current_diagram();
            break;

        case 3:      // close
            if (save_current_diagram())
                start_new_diagram();
            break;

        case 4:      // name
            char *s = msgline.gets("Enter diagram title: ");
            if (s != NULL)  {
                strcpy(current_diagram->name, s);
                set_title(current_diagram->name);
                show();
                msgline.post("Diagram title changed.");
                }
            break;

        case 5:      // exit to dos
            exit_flag = 1;
            break;

        default:
            break;
        }
```

```
    return selection;
    }


int titlebar_t::save_current_diagram(void)  {
    char *fn = (char *)NULL;
    FILE *fp = (FILE *)NULL;
    char ch;
    int x, y;

    if (current_diagram->name[0] == '\0')  {    // diagram must be named
        strcpy(current_diagram->name,
               msgline.gets("Enter diagram title: ") );
        if (current_diagram->name[0] == '\0')
            return 0;
        else  {
            set_title(current_diagram->name);
            show();
            }
        }

    while (fp == (FILE *)NULL)  {
        fn = msgline.gets("Enter filename: ");

        if (fn == (char *)NULL)  return 0;

        if ( (fp = fopen(fn, "wt")) == (FILE *)NULL )  {
            msgline.post("Unable to open file '");
            msgline.append(fn);
            msgline.append("'.  Click to continue...");
            get_event(ch,x,y);
            }
        }

    current_diagram->save(fp);

    fclose(fp);
    msgline.post("Save operation complete.");

    return 1;
    }


void titlebar_t::start_new_diagram(void)  {

    delete current_diagram;
    current_diagram = new diagram_t;
    dispatcher.clear();
    dispatcher.insert(this);
    dispatcher.insert(&canvas);
    canvas.clear();
    set_title("");
    show();
    msgline.post("Starting new diagram.");

    return;
    }


void titlebar_t::retrieve_diagram(void)  {
    char ch, *fn = (char *)NULL, buf[80];
    int  x, y, end_of_diagram = 0;
    FILE *fp = (FILE *)NULL;

    while (fp == (FILE *)NULL)  {
        fn = msgline.gets("Enter filename: ");

        if (fn == (char *)NULL)  return;

        if ( (fp = fopen(fn, "rt")) == (FILE *)NULL )  {
            msgline.post("Unable to open file '");
            msgline.append(fn);
            msgline.append("'.  Click to continue...");
            get_event(ch,x,y);
            }
        }

    delete current_diagram;
    current_diagram = new diagram_t(fp);

    while (!end_of_diagram && !current_diagram->error)  {

        if (fscanf(fp, " %s ", buf) == 0)  {
            current_diagram->error = 1;
            return;
            }

        if (strcmp(buf, "Class") == 0)  {
            current_diagram->add( new class_t(fp) );
            }

        else if (strcmp(buf, "Object") == 0)  {
            current_diagram->add( new object_t(fp) );
            }

        else if (strcmp(buf, "Message") == 0)  {
            current_diagram->add( new message_t(fp) );

            }

        else if (strcmp(buf, "Inheritance") == 0)  {
            current_diagram->add( new gen_spec_t(fp) );
            }

        else if (strcmp(buf, "Whole-Part") == 0)  {
            current_diagram->add( new whole_part_t(fp) );
            }

        else if (strcmp(buf, "Association") == 0)  {
            current_diagram->add( new connection_t(fp) );
            }

        else if (strcmp(buf, "end") == 0)  {
            end_of_diagram = 1;
            }

        }

    if (current_diagram->error == 0)  {
        current_diagram->refresh();
        set_title(current_diagram->name);
        show();
        msgline.post("Open operation complete.");
        }
    else  {
        msgline.post("Error reading diagram in file '");
        msgline.append(fn);
        msgline.append("'.");
```

```
        delete current_diagram;
        }

    fclose(fp);

    return;
    }


titlebar_t::~titlebar_t(void)  {

    delete menu;

    return;
    }


// end of file
```

```
//---[ util.h ]-------------------------------------------------------


#ifndef UTIL_H
#define UTIL_H


#include <stdio.h>


//
// class definition for the screen object
//

class screen_t {
    public:
        screen_t(void);
        ~screen_t(void);

    private:
    };



//
//  prototype for and return codes from get_event();
//

int  get_event(char &ch, int &x, int &y);
void getqs(FILE *fp, char *buf);

#define SELECT 1
#define ACT    2
#define KEY    3



#endif


// end of file
```

```
//---[ util.cpp ]-------------------------------------------------

//
// Miscellaneous functions
//

#include <stdlib.h>
#include <graphics.h>


#include "util.h"


//
// member functions for the screen object
//


screen_t::screen_t(void)  {
    int gdriver = DETECT, gmode, errorcode;     // request auto detection

    initgraph(&gdriver, &gmode, "c:\\tcpp\\bgi\\");
    errorcode = graphresult();

    if ( (errorcode = graphresult()) != grOk) { // an error occurred
        initgraph(&gdriver, &gmode, "");        // try another path
        errorcode = graphresult();

        if ( (errorcode = graphresult()) != grOk) {
            printf("Graphics error: %s\n", grapherrormsg(errorcode));
            exit(1);
            }
        }

    return;
    }


screen_t::~screen_t(void)  {

    closegraph();

    return;
    }



void getqs(FILE *fp, char *buf)  {
    char ch, *p = buf;

    while ( (ch = fgetc(fp)) != '"' );

    while ( (ch = fgetc(fp)) != '"' )  {
        *p++ = ch;
        }

    *p = '\0';

    return;
    }


// end of file
```

```
//---[ canvas.cpp ]-------------------------------------------------

//
// Member functions for the canvas object
//


#include <graphics.h>
#include <string.h>

#include "canvas.h"
#include "msmouse.h"
#include "msgline.h"
#include "connect.h"
#include "symbol.h"
#include "dispatch.h"
#include "diagram.h"


canvas_t::canvas_t(void)  {

    x1 = 1;
    y1 = (textheight("H") * 4/3 + 4) * 2 + 3;
    x2 = getmaxx();
    y2 = getmaxy();

    menu = new menu_t("Create", "Class", "Object", "Message", "Gen-Spec",
                      "Whole-Part", "Association", NULL);
    return;
    }


void canvas_t::clear(void)  {

    Mouse.Hide();

    setfillstyle(SOLID_FILL, WHITE);
    bar(x1,y1,x2,y2);

    Mouse.Show();
    return;
    };



selectable *canvas_t::select(void)  {

    msgline.post("Canvas selected.");

    return this;
    }



int canvas_t::act(int x, int y)  {
    int selection;
    char buf[80];
    class_t *class_obj, *from, *to;
    connection_t *connection;

    selection = selectable::act(x,y);

    switch (selection)  {

    case 0:      // class
        strcpy(buf, msgline.gets("Enter class name: ") );

        if (buf[0] != '\0')  {
            class_obj = new class_t(x,y,buf);
            current_diagram->add(class_obj);
            dispatcher.insert(class_obj);
            class_obj->show();

            msgline.post("Class ");
            msgline.append(buf);
            msgline.append(" Added.");
            }

        break;

    case 1:      // object
        strcpy(buf, msgline.gets("Enter object name: ") );

        if (buf[0] != '\0')  {
            class_obj = new object_t(x,y,buf);
            current_diagram->add(class_obj);
            dispatcher.insert(class_obj);
            class_obj->show();

            msgline.post("Object ");
            msgline.append(buf);
            msgline.append(" Added.");
            }

        break;

    case 2:      // message
        from = msgline.getobj("Click on Sending Class/Object:");
        if (from != NULL)  {
            to   = msgline.getobj("Click on Receiving Class/Object:");
            if (to != NULL)  {

                connection = new message_t(from, to);
                current_diagram->add(connection);
                dispatcher.insert(connection);
                connection->show();
                msgline.post("Message connection added.");

                }
            }
        break;

    case 3:      // gen-spec
        from = msgline.getobj("Click on Derived (Child) Class/Object:");
        if (from != NULL)  {
            to   = msgline.getobj("Click on Base (Parent) Class/Object:");
            if (to != NULL)  {
                connection = new gen_spec_t(from, to);
                current_diagram->add(connection);
                dispatcher.insert(connection);
                connection->show();
                msgline.post("Inheritance connection added.");
                }
            }
        break;

    case 4:      // whole-part
        from = msgline.getobj("Click on Aggregate Class/Object:");
```

```
            if (from != NULL)  {
                to   = msgline.getobj("Click on Component Class/Object:");
                if (to != NULL)  {
                    connection = new whole_part_t(from, to);
                    current_diagram->add(connection);
                    dispatcher.insert(connection);
                    connection->show();
                    msgline.post("Composition connection added.");
                    }
                }
            break;

        case 5:      // assocation
            from = msgline.getobj("Click on First Class/Object:");
            if (from != NULL)  {
                to   = msgline.getobj("Click on Second Class/Object:");
                if (to != NULL)  {
                    connection = new connection_t(from, to);
                    current_diagram->add(connection);
                    dispatcher.insert(connection);
                    connection->show();
                    msgline.post("Association connection added.");
                    }
                }
            break;

        default:
            break;
        }

    return selection;
    }


canvas_t::~canvas_t(void)  {

    delete menu;

    return;
    }


// end of file
```

```
//---[ connect.h ]----------------------------------------------

//
// Class definitions for the various types of connection objects.
// The only difference is in their labels and their display methods.
//


#ifndef CONNECT_H
#define CONNECT_H


#include <stdio.h>

#include "select.h"
#include "symbol.h"
#include "menu.h"


//
// Base class for connections.  Also used to represent "associations"
//

class connection_t : public selectable  {
    public:
        connection_t(class_t *obj1, class_t *obj2);
        connection_t(FILE *fp);
        ~connection_t(void);

        void show(void);
        char *typeof(void)  { return "connection_t"; }
        int  isa(char *s);
        selectable *select(void);
        int ison(int x, int y);
        int act(int x, int y);
        void save(FILE *fp);

        int is_connected_to(selectable *obj);

    protected:
        class_t *from, *to;
        int tolerance;          // max distance for a click to select

    private:

    };



//
// The message connection class
//

class message_t : public connection_t  {

    public:
        message_t(class_t *obj1, class_t *obj2);
        message_t(FILE *fp);
        ~message_t(void);

        char *typeof(void)  { return "message_t"; }
        int  isa(char *s);
        void show(void);

    };



//
// The generalization-specialization (inheritance) class
//

class gen_spec_t : public connection_t  {

    public:
        gen_spec_t(class_t *obj1, class_t *obj2);
        gen_spec_t(FILE *fp);
        ~gen_spec_t(void);

        char *typeof(void)  { return "gen_spec_t"; }
        int  isa(char *s);
        void show(void);

    };



//
// The whole-part (composition) class
//

class whole_part_t : public connection_t  {

    public:
        whole_part_t(class_t *obj1, class_t *obj2);
        whole_part_t(FILE *fp);
        ~whole_part_t(void);

        char *typeof(void)  { return "whole_part_t"; }
        int  isa(char *s);
        void show(void);

    };


extern menu_t connection_menu;


#endif


// end of file
```