1996

Object-Oriented Design and Literate Programming

Glen D. Finston
San Jose State University

OBJECT-ORIENTED DESIGN AND LITERATE PROGRAMMING

A Thesis

Presented to

The Faculty of the Department of Mathematics and Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Glen D. Finston

December 1996

APPROVED FOR THE DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

[illegible]

Dr. Cay Horstmann

[illegible]

Dr. Kenneth Louden

[illegible]

Dr. Rudy Rucker

APPROVED FOR THE UNIVERSITY

Serena W. Stanford

ABSTRACT

OBJECT-ORIENTED DESIGN AND LITERATE PROGRAMMING

by Glen D. Finston

In the first part of this thesis, principles of Object-Oriented Design in C++ are reviewed with a focus on the properties of classes and objects, the Booch Object-Oriented Design methodology, and the complexity of systems. The second part of this thesis discusses formal Literate Programming, first proposed by Donald Knuth of Stanford University in 1982. A Microsoft Windows C++ Application Design Tool, known as Cloud9 and created by San Jose State students, is enhanced to include the capability for Literate Programming. The main ideas behind Knuth's Literate Programming system, known as WEB, are presented in the application enhancement. Furthermore, the application enhancement moves the concepts of formal Literate Programming into current standards of documentation by allowing the user of Cloud9 to create online hyper-linked documentation for his/her in-process design. The obstacles encountered in performing this enhancement constitute the primary focus of this part of the thesis.

TABLE OF CONTENTS

PART I: OBJECT-ORIENTED DESIGN

LIST OF FIGURES

PART I: OBJECT-ORIENTED DESIGN

1 OBJECT-ORIENTED DESIGN AND COMPLEXITY OF SYSTEMS

1.1 Object-Oriented Design: Introduction and Comparison to Structured Design

Object-Oriented Design can be used to model the real world. Just as the real world contains physical entities with certain behaviors. Object-Oriented Design employs the use of objects with certain behaviors. Similarly, just as objects interact. communicate and are grouped together to perform some higher level function in the real world. the same is true for objects in Object-Oriented Design. In fact. this natural relationship is one of the foundations that facilitates the use of Object-Oriented Design by human beings. We are animals who think in terms of objects and interactions between objects and therefore the design of software by a similar methodology is perhaps the easiest path to understanding a complex system.

Object-Oriented Design's predecessor. Structured Design, follows an algorithmic decomposition of the problem into its component parts. In general. structured design places emphasis on data flow. not on objects, and how data is input. manipulated and output. "Structured design should be chosen for architectural design when no more specialized design methodology exists for the application, and when the flow of data can reasonably be used as the unifying principle of the software."[1]

Structured Design has no built-in mechanism for encapsulation. i.e. hiding the implementation of data types. All encapsulation must be made explicit by the programmer. Object-Oriented Design is by default encapsulated because class methods and member data are private unless stated otherwise.

Using Object-Oriented Design, one can break a problem down into the vocabulary of its inherent domains, modeling the physical entities and their behavior which leads to simplification of the problem. This is necessary as software systems are becoming increasingly complex. In order for a single individual to understand a complex, monolithic software system, one must remember simultaneously vast amounts of information related to the system. The human brain struggles to understand the essential elements of a single complex level of abstraction, and a "complete" understanding of a complex layer of abstraction can take days.

1.2 The Structure of Complex Systems - Human Biological Systems

Human biological systems provide a good example of complex systems and levels of abstraction. The human being can be understood in terms of systems, i.e. the circulatory system, the digestive system, the nervous system, etc. Each system unto itself has a known responsibility in maintaining the human life, but life only exists within the homeostatic interaction and equilibrium of these systems (strong allegory with software). The sciences of medicine and biology are still in their infancy in terms of mastery of the understanding of these complex systems and their inter-relatedness and intra-relatedness.

Doctors fall into the categories of general practitioner or specialist. A general practitioner is trained in an overview of the human body, while a specialist can spend his/her working life focused on a single complex system (domain) or even organ in the human body. The entire human body is too vast an organism for an individual to understand all its various components and systems completely. The general practitioner generally only needs sufficient understanding of the micro issues involved in key levels of abstraction of the human body to be what is considered a competent doctor in our society.

For the specialist, within each organ are levels of abstraction. At the lowest level, the problem domain of the organ consists of entities or objects such as different kinds of chemical molecules. The behavior of these molecules in the presence of others enables aggregates of them to form into entities known as cells. The cell object has a behavior such as division which involves the splitting of the cell and a copy being generated. The ability of the cell to split causes larger, as I will call them, tissue objects to emerge. Groups of tissue objects in turn support a necessary aspect of the organs function and are responsible for a new higher level of behavior. Each object with its accompanying behavior interacts with other objects with their own behaviors, and when these objects are grouped together and examined as a functioning whole, they in turn as a group exhibit a higher level behavior. Eventually, the whole picture of the organ begins to emerge. Ultimately, the organ itself is an object with its own behavior. An organ performs its higher level behaviors in concert with other organs, producing systems such as the circulatory system. Specialists and scientists can spend the greater part of their lives devoted to only a few levels of abstraction related to a specific higher level behavior problem domain within this complex system.

1.3 Corollaries Between the Medical Analogy and Complex Software Systems

Similar to medical specialists mentioned in 1.2. software engineers play different roles and work in different levels of abstraction when solving a large. complex problem. Often. due to the system's complexity. the software engineer needs to become specialized (at least temporarily) at a particular level of abstraction as design issues necessitate a more intimate familiarity. These individuals then become insulated from issues involving other levels of abstraction as they only need know what happens at the interface between their level of study and the next level above and below. This is due to a concept known as abstraction discussed below. Armed with a clear understanding of contractual obligations to other levels. engineers are free to focus on their complex task of efficiently modeling the behavior(s) exhibited.

A single system. such as the human body discussed previously. can be likened to a complex software system. The system exists as a functioning whole. yet can be decomposed into its components or objects. Different objects live at different levels of abstraction and each object has behaviors associated with it. At any given level of abstraction. an object symbiotically performs behaviors in concert with other objects which. when examined externally. are themselves higher level behaviors. Similar to the human biological system. the entire system is too complex to understand completely. simultaneously, at micro and macro levels.

1.4 Limitations of Object-Oriented Design

Many complex systems benefit from the use of Object-Oriented Design. There are limits to the amount of complexity humans can express using solely Structured Design and functional decomposition. Although Object-Oriented Design provides better mechanisms for modeling complex systems. in practice some systems are sufficiently large and complex that even with the best object ideas and use of encapsulation at the module level. the system is still quite difficult to comprehend. Errors must be reproducible but in some cases they are not. The system itself becomes a black box in which inputs generate outputs and sometimes it can be difficult to explain the results. Changing even a single line of code can take much consideration and deliberation. Although Object-Oriented Design is a vast improvement. it is not a panacea.

1.5 Attributes of a Complex System

Booch discusses five attributes of a complex system as follows:

"Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems. and so on. until some lowest level of elementary components is reached."[2] A complex system is hierarchical and decomposable.

"The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system."[3] Different observers have different needs which affect their determination of which objects or methods of a system are useful. Depending on the observer, an object may be simple or elementary, yet to another observer, the same object may be complex. An observer's focus on a particular level of abstraction affects this characterization of the object. Also, different observers view the decomposition of a system differently depending on their own paradigms and what are considered key components.

Hierarchical systems are decomposable into identifiable components that are grouped together on the basis of functionality to express some higher level behavior. "Intra-component linkages *(within components)* are generally stronger than inter-component linkages *(between components)*. This fact has the effect of separating the high-frequency dynamics of the components - involving the internal structure of the components - from the low-frequency dynamics - involving interaction among components."[4] This breaks the view of an object into two perspectives: the internal view and the external view. The internal aspects of an object can be modified in relative isolation from the effect such alterations may have on other external objects. There is, however, the issue of coupling. Strong coupling or a strong association between modules is undesirable as this makes any one module harder to understand and change because of its dependencies on other modules. However, strong coupling exists between classes because inheritance causes significant coupling. In fact, inheritance allows us to take advantage of the common behaviors and attributes of classes because of the advantage of strong coupling. There is a healthy tension between these two extremes which promotes a good design.

Hierarchical systems are composed of subsystems, and these subsystems can be categorized into distinct patterns which enable reuse of known stable designs. These patterns can be reapplied in similar circumstances with minimum effort. "Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements."[5]

"A complex system that works is invariably found to have evolved from a simple system that worked."[6] Mature complex systems are built upon simple systems. If the simple system is proven to work over time, it is worth enlargement. Note that, regarding complexity and simplicity, an object considered complex when viewed from a particular level of abstraction is considered simple when viewed from a different level of abstraction. Similarly, if viewing collections of objects that work together at a particular level of abstraction as a system, a system is simple or complex, depending on the point of view of the observer.

2 ITERATIVE AND INCREMENTAL NATURE OF OBJECT-ORIENTED DESIGN

Software design using the waterfall life cycle decomposes the process into concrete steps; each step needs to be as complete as possible before the next is allowed to proceed. Requirements had to be completed before analysis, and analysis before design, and so on. The approach could be iterative, or have the ability to wrap back to previous steps as new issues are discovered later in the process, yet this was typically discouraged as these modifications to a design produce costly propagation effects typically directly proportional to length of the gap spanned by the iteration.

The Booch method suggests a different approach for software development than the waterfall life cycle. In practice and contrary to the waterfall model philosophy, software design is ultimately an iterative process as previous work is affected by discoveries found later in the process. The Booch method breaks down the software life cycle into similar concrete steps, yet allows the engineer to switch between design and analysis with fewer negative effects. This is due to the fact that as tangible objects and abstract classes are manipulated, both design and analysis are jointly affected. This codependent relationship enables a quick appraisal of the suitability of the changes to the problem domain. The flexibility of iteration is further enhanced by Object-Oriented Design Tools that allow one to examine different structural approaches to a problem and produce code quickly.

To iterate on an entire stage encompassing the entire problem domain is inefficient. Instead, Booch suggests iterating on a distinct logical or physical view of the system. The strong intracomponent linkages that produce a separation of concerns between parts of a complex system allow the engineer to start small and iterate on a single part of the system. As these parts become more refined, the cycle is applied to more of the system incrementally. Eventually, all the different views of the system are integrated incrementally into a coherent and functional whole. "The difference (from the waterfall model) is that is this a 'mini' set of steps that is applied iteratively to pieces of a system. In practice, developers analyze a little, design a little, and code a little. Then they cycle back and do it again, only on more of the system. All of analysis, design and coding are accomplished, but in a series of cycles rather than three large leaps."[7]

3 CLASSES AND OBJECTS

3.1 Classes

Classes are factories for objects. A class represents a group of related objects. A class only exists as an abstract concept. It is given life and existence only through its instantiation into an object. Some classes will never have an instance created. Their purpose is to provide common behaviors and member data to the classes that inherit from them.

A class has both data members and methods. (The term "method" is Smalltalk terminology.) Depending on the programming language, the methods of a class may be referred to as operations or member functions. These terms will be used interchangeably. Also. invoking an object's member function can be called sending a message to an object. For simplicity. I will not use this terminology. In C++. both member data and methods of a class can be defined as public. protected or private. Member data declared as public are accessible anywhere within the program. Methods declared as public may be invoked anywhere within the program. Private member data are accessible only by methods of the class itself (or its declared friend classes. if any). Private methods can only be invoked within another method of the same class (or its declared friend classes). Similarly. at another level. protected member data are accessible only by methods of the class itself or its subclasses. Protected methods may only be invoked within another method of the same class or its subclasses. By default. all member data and methods are private unless declared otherwise. The behavior of a method is determined by the method called. the objects current state or value of its member data. and the arguments passed to the method.

3.2 Relationships between Classes

3.2.1 Association Relationship

Relationships exist between classes. An association is the weakest kind of relationship and denotes a relationship in which only a dependency between classes can be ascribed. For example. in the Object-Oriented Model for a payroll system. objects might include such entities as employee and company. The nature of the relationship between the two objects is employment. This is merely a semantic dependency. The direction of the relationship and how one class will be related to the other in the implementation is unclear in the analysis and design phase. Ascribing a semantic dependency is the first step in problem domain analysis. Once this dependency is established, decisions can be made regarding replacement of these weak associations with more binding relationships between classes. As these relationships become more refined and stronger as a result of these decisions, the responsibilities of classes becomes clearer. These more binding relationships include inheritance, aggregation and uses relationships.

3.2.2 Inheritance Relationship

Inheritance represents an "is a" relationship. Many classes are similar, have the same basic structure and exhibit the same behavior as other classes, yet they have minor differences due to their specialized purposes. These classes are related through inheritance. By stating that a subclass inherits from another class, one can efficiently explain many of the characteristic properties of an object instantiated from this subclass. There is no need to restate the properties of the enclosing superclass in the subclass. They only need be stated once.

The Control class in a Windows application framework provides a good example of inheritance. A Control class in a Windows application framework is a class that enables the user to control the application. The Control class is an abstract class meaning that an object of this class can never be created, but it is used to give a family of objects common behaviors. There are many types of controls including push button controls. scroll box controls. check box controls. and radio button controls. Inheritance enables conservation of code as a radio button control is a Control. The radio button control subclass responds to all of the same operations as its Control superclass. The radio button Control class inherits the member data and methods of the Control class. yet it can perform its operations differently. Booch notes that, "As we evolve our inheritance hierarchy. the structure and behavior that are common for different classes will tend to migrate to common superclasses."[8] Inheritance allows one to group similar classes into one class hierarchy.

A subclass can have zero. one. or more superclasses. and classes can inherit from multiple classes. This is usually done because a class may be very similar to its superclass. yet it is convenient to provide another class that elicits specific desired behavior. The class that elicits this specific behavior is called a mixin class. A mixin class is created only to mix with another class. For example, suppose that in the Object-Oriented Model of an air traffic control system, there exists different kinds of radar used for screening different characteristics of flying airplanes. One type of radar. I will call a Velocity Radar. can detect the position, velocity and acceleration of an airplane accurately. Another type of radar. I will call an Attribute Radar. can detect the position and the type of plane. Both radar inherit data members from the Radar class such as Screen and Resolution and methods such as Report Position. yet to give unique characteristics to each radar. mixin classes are developed. For the Velocity Radar. a class called VelocityMixin is created that contains only the unique characteristics of a Velocity Radar (data structures to contain precise velocity and acceleration vectors for airplanes). Similarly, characteristics unique to an Attribute Radar are placed in a new AttributeMixin class. The class Velocity Radar would then multiply inherit from the Radar superclass, and its VelocityMixin class.

Usually the child subclass performs some specialized function. These specialized member functions may occur inside the subclass itself, or be declared virtual in the superclass. When a superclass declares a method virtual, it is essentially making a forward declaration, meaning that the method is then overridden by its individual subclasses. Overridding means the specifics of the method are determined within relationship of the subclass to itself and how the subclass would best express the operation given its own design. Because the subclass has a common superclass with other subclasses of that same superclass, each different subclass can have its own individual implementation of this common operation. The named operation may be applied to many different subclasses as long as they have the same superclass. This is known as polymorphic behavior. Using the Control class example above, all controls have a virtual "do click" method which highlights the control and performs the associated operation; however, the operation each control performs as a result of the mouse button down is different. Scroll controls scroll text while button controls toggle push buttons. Any derived Control class may override the "do click" method. When "do click" is invoked, it is determined at run-time to which class an object belongs, and the method is looked up based on this determination. and the corresponding correct function is called. Virtual methods are declared as such in the superclass and are overridden within the individual subclasses to achieve the specialized behavior. yet virtual methods express a known commonality at a higher level.

3.2.3 Aggregation Relationship

Aggregation represents a "has a" relationship. The record structure is an example of aggregation. Just as in the case of the record in which structures are contained within other structures objects may contain other objects or structures or references to other objects or structures. However. in an Object-Oriented Programming Language "the combination of aggregation with inheritance is powerful: aggregation permits the physical grouping of logically related structures. and inheritance allows these common groups to be easily reused among different abstractions."[9]

3.2.4 Using Relationship

A class which uses another class to perform its operations has a using relationship with that other class. The using relationship is a refinement of an association. The "client" class and the "server" class are stated and the relationship is no longer bi-directional. In the example of Company class and Employee class. a decision could be made to have the Company use the Employee (although logically the reverse could be stated as well). A uses relationship means that another class appears in the using classes member functions and is used by the using classes member functions.

3.3 Class Categories

Once several classes and objects have been identified, it is useful to partition the logical system. Class categories are aggregates of classes and other class categories. A class category is identified by a name which represents the primary responsibility or entities within the class category. Class categories partition the logical system into layers of abstraction. Related services meaningful to the level of abstraction under consideration are grouped into class categories. To use Booch's example of the Hydroponics Gardening System[10], the Greenhouse class category contains the abstract class Environmental Controller and Nutritionist (see Figure 1. page 63). The underlying responsibilities of groups of classes are deliberately exposed to present a higher level model or overview of the system. Different class categories can also live at the same level of abstraction.

3.4 Objects

Every object belongs to a user or library defined class. Objects are instantiated instances of a class and are created by invoking a member function of a class known as its constructor. An object contains both state and behavior information. The state of an object is determined by the values of data members. The behavior of an object is determined by the methods which can be performed upon it. The visibility of the member data and methods (public, private, or protected) is presented in the object's class. Ideally, an object's data members are declared as private to ensure that they can only be accessed or manipulated by methods of their own class. The object's methods can operate on its own data. Operations performed outside the object directly on the data can not be guaranteed to be accountable or do not have a necessary and sufficient understanding of the object to modify the member data without possible disastrous consequences. A "client" of an object is an object which invokes its "server's" methods. Clients are given the means to access the data members through member functions known as selectors, that the class grants to clients, which are the only prescribed procedures to view its instantiated object's contents. Classes selectively give clients the means to manipulate their instantiated object's contents through mutators, which are the only prescribed procedures to modify its instantiated object's contents. Because the implementation of a class is completely hidden from the outside, the implementation can be modified knowing that the change will only have an effect on the operations of its class and on no other code.

Each object has different responsibilities. Objects cooperate with each other to perform a higher level behavior. A higher level behavior is a behavior that results from the cooperative relationship of objects with their inherent individual behaviors. When examining the entire system, a behavior becomes apparent that the collection of objects produces. Each object performs a necessary and codependent function in the system to produce the higher level behavior. Therefore, the concept of an object only makes sense in terms of its relation to other objects.

3.5 Types of Operations Performed on Objects

Operations performed on or to create and destroy objects can be grouped into three general categories: constructor/destructor, selector, or mutator. A constructor and destructor creates and destroys, respectively, an instance of a class. A class may provide different constructors for the same class of object. In C++, these constructors are uniquely identified by their parameters or signature. An appropriate constructor can then be called relating to the specific instantiation information most useful to the given circumstance. In C++, defining the destructor of a class is optional. Any resources that an object creates must be released explicitly in the destructor method. An object's destructor method gives the object a means of cleaning up after itself.

A selector is the only method by which objects not granted explicit rights to directly view the member data of an object can examine the member data. Selectors are the public interface an object provides which enable other objects to view its member data. In C++, selectors are typically declared as `const`. This means that no member data may be modified within this operation.

Selectors are necessary to promote two of the major elements of the object model, abstraction and encapsulation. Prior to Object-Oriented Programming, the programmer could circumvent the weak safety mechanisms of procedural programming. A procedural program's safety mechanisms consist of documentation written by a programmer or group of programmers requesting that specific rules are respected with regard to the direct modification of data and allowing variables to be visible only where necessary (scope). But these mechanisms do little to thwart the behavior of unscrupulous programmers (such as those under strict time deadlines) from not following these decided upon rules or modifying scope as a "quick and dirty" solution to a problem. A selector provides a protocol which is the only means of getting at object member data and within its construct is the compiler enforced provision that the data can not be modified. Obviously, this stronger mechanism found in Object-Oriented Programs can be avoided by a single rogue programmer performing Object-Oriented Design without responsibility to a larger group, but in the end, it is his/her program that suffers.

Mutators, as the name implies, are the only explicit mechanisms for modifying object member data. They provide a well defined interface for modification of member data. They hide their implementation details (an abstract data type attribute) from the external program, allowing one to modify their implementation with no affect on the external program.

3.6 Lifetime of Objects

An object exists until it is destroyed. A local object created on the stack is destroyed when it goes out of scope. An object created on the heap with the new operator has to be destroyed explicitly. In C++, objects continue to exist even when all references to them have been lost and can no longer be used. Some languages provide a method known as garbage collection to reclaim this unused space.

Often if objects are reused frequently in an application, it becomes apparent that it is far less work to create them and have them live for the duration of the program instead of having to construct and destroy them on an as-needed basis. Such objects may be referred to as static objects and exist until the end of the program.

4 FOUR MAJOR ELEMENTS OF THE OBJECT MODEL

4.1 Abstraction

An abstraction is the unified presentation of an object brought about by a consistent and cohesive set of operations that can be performed upon it. Abstraction is what distinguishes the object from all other kinds of objects. The abstraction of a real-world object in a system ignores details which do not contribute to its intended use. "We (humans) have developed an exceptionally powerful technique for dealing with complexity. We abstract from it. Unable to master the entirety of a complex object, we choose to ignore its inessential details, dealing instead with the generalized, idealized model of the object."[11]

4.2 Encapsulation

Encapsulation is focused on the internal view of an object's implementation that produces its behavior. The implementation of an object's behavior is encapsulated from the outside world and is secret or hidden. This allows one to modify an object's implementation without affecting any of its clients. Implementation decisions of an object may now change without consequences to the outside world. One can freely explore different implementation options with no impact on the outside world as the client of an object is not exposed to this level of detail. By hiding the implementation, a client can make no assumptions about an object. It is not necessary to determine the other parts of the code that need to be changed when a change is made in encapsulated code. The cost of maintenance, a large portion of the total cost of a project, is reduced.

As a result of encapsulation, a barrier is placed between the internal object and external objects. The object's class selectively and judiciously puts forth only those properties of an object that make it useful externally. Exposing only those properties that make it useful to the outside world (or an object's clients) is the principle of least commitment. The principle of least commitment elicits a safer system as less "can go wrong." By hiding the implementation details of an object from the outside, an object has a safe and consistent interface for the modification of its state (member data) by the outside. An object can only be modified through known methods that the programmer gives conditional availability to the outside world (a class has a public, private and protected section).

Object Oriented Design can be described in terms of the client/server model. A client object uses the services of a server object. The contract, or protocol, between the client object and the server object consists of all the member functions of the server object that the client object may use. The contract defines the operations for which the object can be held accountable. For maximum safety, whenever an operation is available to a client, the object must be able to perform its duties as defined whether asked to do so or not. In order to provide a clear definition of an object's duties, certain conditions must hold true, or be invariant, before and after the operation. These are called pre-conditions and post-conditions, respectively. If a client violates pre-conditions, it has not performed its duty as defined in the contract and can not expect valid results from the operation. If an object violates post-conditions, it violates its duty under the contract and can no longer be trusted. The engineer has the choice of placing different aspects of operational responsibility either with the server object or with the client object. These design decisions are largely problem dependent.

4.3 Modularity

The classes and objects which compose a program are divided and placed into modules which are their containers. Logically related classes and objects are placed within the same module. The difference between modules and class categories is that modules have a physical representation, and class categories are a logical concept. A module consists of an implementation file and an interface or header file. A system which has been decomposed into modules has the property of modularity. Similar to an object, a module has an interface and an implementation and uses the idea of abstraction. A module's interface exposes only those elements of a module other modules must see.

One may change the implementation of a module (or objects within a module) without affecting the behavior of other modules. A module includes the interface of the modules it uses. (A module always includes its lower level units.) If an object or class is altered. only its enclosing module need be recompiled.

4.4 Hierarchy

The inheritance relationships between classes form a hierarchy. Inheritance represents an "is a" relationship and produces the "is a" hierarchy. A guppy is a kind of fish; a house is a kind of building. Another kind of hierarchy is the "part of" hierarchy. Aggregation represents a "part of" relationship and produces a "part of" hierarchy. As previously noted. the record structure is a structure that supports aggregation. Objects that exist independently and have lifetimes independent of their container objects are contained by reference with a pointer. Objects that can not exist without their container objects are contained by value and have the same lifetime as their enclosing objects. Another hierarchy is the stratification hierarchy of higher level and lower level abstractions mentioned in 1.2 (The Structure of Complex Systems - Human Biological Systems).

5 THE BOOCH METHOD

5.1 Diagrams

The Booch method prescribes that the engineer develop models or views of the system. These models consist of diagrams which embody analysis and design decisions with regard to two dimensions: a physical/logical view and a static/dynamic view. Each dimension consists of several diagrams. Classes or objects living in one diagram can also live within another diagram. The results of the relations produced by these diagrams are cumulative. It is by logically partitioning the classes or objects (depending on the diagram) that a diagram becomes particularly useful. Each diagram represents some aspect of the system model under consideration. Each diagram has a theme or name and by allowing classes and objects to be included in more than one diagram. the true interdependencies of the system are exposed.

The model of the system is built in stages incrementally and appropriate diagrams are generated depending on the current area of focus. Encapsulation assists the process as diagrams can be considered in relative isolation from each other. The area in which two diagrams overlap can be presented in another diagram. Using an Object-Oriented Design Tool such as Cloud9 or commercial products such as Rational Rose. the decisions made in these diagrams may be translated directly into the code. The three general categories of diagrams are Class Diagrams, Object Diagrams and Module Diagrams. There are also State Transition Diagrams. but it is beyond the scope of this paper to include this in the discussion.

5.2 Class Diagrams

Class diagrams represent the inheritance structure of classes with regard to the particular nouns or objects within the problem domain. In general any particular high level class (that from which subclasses are derived) has associated with it a class diagram showing its parent/inheritance relationship to its subclasses. This does not mean that each superclass has its own Class Diagram. but it is associated with at least one. Class diagrams present a logical view of the system under consideration and provide only static information (See Figure 2. page 64). In the diagram. the different links between the class clouds represent different relationship adornments. A straight line. such as the one labeled "Defines climate" represents an Association between the classes GardeningPlan and EnvironmentalController. The class EnvironmentalController contains a Heater. a Cooler and any number of Lights by Aggregation denoted by the line with the dark circle at the container class end. Heater and Cooler are subclasses of the Abstract Class Actuator as denoted by arrows pointing to the superclass. The Actuator Class chooses to display two of its methods. startUp and shutDown. The Actuator Class uses the Temperature Class as denoted by the line with an unfilled circle at the using class end.

5.3 Object Diagrams

Object Diagrams. also called Scenario Diagrams. are interaction diagrams which expose the different scenarios in which objects participate with each other to perform some higher level function. An object diagram represents a physical view of the system -- a dynamic snapshot of the interaction between physical objects. An object diagram contains an ordering of methods which objects invoke upon each other to perform the higher level function. Each object diagram represents a unique scenario. The name or description of the object diagram is typically the name or description of the higher level function portrayed in the diagram. (A good example of a scenario related to processing a Packing Order for Items at a Warehouse can be found in Figure 3, page 65.)

5.4 Module Diagrams

Module diagrams are used to represent the physical layering and partitioning of levels of abstraction within the system. A module corresponds to two files in C++: a .cpp (body) and .h (specification). Inter-module dependencies generated by `#include` are presented in the module diagrams by use of the arrow (Figure 4, page 66).

5.5 The Steps of the Booch Method

The Booch Method of Object-Oriented Design consists of three steps [12]:

1) Requirements Analysis

2) Domain Analysis

3) System Design

briefly summarized below.

In Requirements Analysis. the customer is asked to provide the key elements of functionality in the system. From this information. key domain specific vocabulary is determined. The vocabulary is then clarified and understood by the engineer using books. other works in the problem domain. and the customer or users of the system (people intimately familiar with the problem domain). Experts in the problem domain that have a body of knowledge from earlier similar projects are valuable in guiding the project. Examining publicly available information about similar systems can lead to a faster understanding of the problem domain. Providing a customer with a rapid prototype that is executable can promote early modifications based on feedback critical to expediting the system. Use cases are developed to summarize the desired functionality of the system. A use case is a "particular form or pattern or exemplar of usage. a scenario that begins with some user of the system initiating some transaction or sequence of interrelated events."[13]

In Domain Analysis. key classes are identified. Any noun in the problem domain is a candidate for a class. Nouns are examined only as abstractions without regard to their implementation which takes place later. These class candidate nouns are then filtered to determine the actual key classes. This filtering process determines the objects of the system and their roles. Class collaborations are determined and responsibilities are assigned. Relationships between classes are established and refined. Object-scenario diagrams of use cases are determined. Focus should be implementation independent.

The goal of System Design is to move the Domain Analysis into implementation. The architecture should be divided into layers based on class categories. As described in Section 2, the design should follow an incremental and iterative approach. The implementation of operations and their algorithms is specified. Any relationships between classes that are not completely "fleshed out" are determined. Any new classes needed to support the implementation are created.

6 CLOUD9 APPLICATION

6.1 Cloud9 Overview

Cloud9 is a C++ Object-Oriented Design Tool. Its primary purpose is to facilitate the programmer's expression and prototyping of a C++ Object-Oriented Design and associated code. Work started on Cloud9 in January 1995 at San Jose State University in CS240, Software Project under the direction of Dr. Cay Horstmann. Cloud9 enabled graduate students to work together on a large scale software. Although Cloud9 is somewhat distant from commercial quality, its scope is realistic as a commercial product. Teamwork is fundamental to commercial software projects and is a valuable experience for student software engineers too often focused on solo academic projects. The idea behind Cloud9 was also to provide a venue for students to work on a thesis project related to a "real-world" Windows application. Thesis contributions to Cloud9 would encompass some selective useful enhancement to Cloud9. While Cloud9 presents the user with a structure for creating an Object-Oriented Design, it is itself a product of Object-Oriented

Design. A similar micro-macro analogy would be an engineer developing a compiler for a high-level programming language. An engineer developing the compiler could not help but benefit in his understanding of the high-level language as well as the target machine instructions; a complete understanding of the language is produced by this exercise, which must account for all language and structural possibilities of the high-level language. Similarly, the student working on Cloud9 becomes intimately familiar with issues of object-oriented design because the student is confronted with a generic framework for any Object-Oriented Design.

Cloud9 provides the user with two primary output commands which may be applied to the C++ Object-Oriented Design under development: generate code and generate documentation. Generating documentation is the contribution of this thesis.

6.2 Class Frameworks

Cloud9 is implemented using the Borland Object Windows Library or OWL. OWL is a application class framework for developing a Windows application. Class Frameworks perform much of the work involved in creating a Windows application by providing a generic framework for the basic structural units of any Windows application. There is a generic Window class with built-in useful operations to perform on a Window, and there is a generic Dialog Box class with built-in useful operations to perform on it, and so on. An Application Class Framework is actually composed of high-level C++ wrappers around standard Windows function calls. The user of the Class Framework is largely protected from much of the implementation of Windows function calls: consequently the users can focus on the problem at hand of the unique contribution of their Windows application. Class frameworks provide the building blocks for any generic Windows application. The programmer needs to add derived classes and overrides generic member functions to produce the desired functionality. Default behavior for standard Windows messages is largely already in place, and the application framework is complete except that the programmer must specify how the application should handle application specific Windows messages. Most applications also have programmer defined classes specific to the purpose of the application itself which are not part of the Class Framework.

6.3 The Document/View Application Model

The single document and multiple view model of an application was created because it is often useful to view a single document with respect to different criteria. Many application frameworks (Borland Object Windows Library (OWL), Symantec Think C++ Application Framework, Microsoft C++ Foundation Class Library) provide support for this model using built-in classes. They all possess a built-in base class devoted to the Document (in Borland's OWL it is called TDocument). Some of the common functions that appear in the TDocument base class that can be applied to TDocument objects are open, close, save, save as, and view in a particular view. The application framework programmer has his custom document class inherit from the

TDocument base class and by doing so takes advantage of common built in behavior. The users then provide additional functionality to their document class by over-riding member functions of the TDocument base class and creating their own member functions. An instance of the user-defined document class is associated with a text file which will be referred to as the *underlying document*. The underlying document is anything that you can display and manipulate inside a View. discussed below.

All the application frameworks mentioned also provide a built-in base class devoted to the View (in Borland's OWL it is called TView). An instance of the TView class is a Window through which the underlying document can be modified. However. a View is more than a Window; it is also accompanied by a unique set of pull-down menus and/or push buttons appropriate for modification of the underlying document in that view. Similar to the built-in TDocument class, the application framework programmer has his custom view class inherit from the TView base class and by doing so takes advantage of common built in behavior. The data of the underlying document is organized by the programmer so that it can be inspected and modified by multiple programmer-defined custom views. Each custom view is designed to present the data of the underlying document according to distinct criteria, so the user of the application can modify aspects of the underlying document without having irrelevant information presented in the view. Cloud9 supports the single document multiple view model of an application.

6.4 Cloud9's Use of the Document/View Application Model

The Booch method describes two categories of models important to object-oriented development: the physical/dynamic model, and the logical/static model. The physical/dynamic model is captured in Object (also called Scenario) Diagrams, and the logical/static model is captured in Class Diagrams and Module Diagrams. Cloud9 provides two different ways of viewing the same design document, the "code view" and the "cloud view." The logical/static model of the class structure is represented by a "cloud view" of the design document, and the physical/dynamic model is represented by a "code view" of the design document. The design document viewed and modified using Cloud9 is an in-process product of object-oriented analysis and design. Decisions can be made quickly regarding the suitability of any particular design by determining a logical structure using the "cloud view," refining it in the "code view," generating the code, and "trying it out" which enables rapid prototyping of different designs.

In the "cloud view," new classes are entered by selecting the cloud icon and placing the cloud into the view. The class can then be named, and relationships can be established with other classes. Relationships of inheritance, aggregation and using relationships can be established visually using the "cloud view" input tool by drawing a line between the two class clouds and modifying a dialog box. The cardinality and data type of aggregation relationships can be established. While in the "cloud view," the user is not presented with largely irrelevant information such as the code of a particular method. Only logical/static information is presented, such as inheritance and aggregation relationships. When the user wants to specify the "guts" of the code, the user opens the same design document in the "code view." Mechanisms for each view updating the other independently are in progress.

The "code view" represents a tiered division of the code into separate blocks normally found in C++ code such as the public, private and protected blocks within a class. Blocks are further subdivided into sections for consistent placement of various types, such as enumerated types or record types Consistent placement of blocks and types within blocks produces code that is readable and facilitates finding specifics within the code as it is logically grouped. The physical/dynamic model can be gleaned by examining the "code view." Similarly, the "code view" does not present cloud diagrams, and the user will switch back to the "cloud view" of the same design document when such modeling is desired. Although the dynamic model is not explicitly present in Cloud9, as it does not offer an Object Diagram (or Scenario) View with an ordering of methods (see Figure 3, Packing Order Scenario Diagram), these mechanisms are likely to be relatively simple, and a drawing outside of Cloud9 can usually provide other engineers with the basic information. In Figure 3, the anAgent Object invokes the schedule() method of the aPackingOrder Object. This causes the aPackingOrder Object to invoke the assign() method of the aStockPerson Object. The aStockPerson Object invokes the query() and update() methods of the inventoryDatabase Object, the update() method of ..ie anOrder Object and the schedule() method of the shipping Object. The aStockPerson Object finally invokes the close() method of the aPackingOrder Object. The higher level function the scenario portrays is the processing of a Packing Order. As shown, the Object Diagram contains ordering of methods which objects invoke upon each other to perform some higher level function. The first method invoked appears at the top and the last method invoked appears at the bottom. The engineering work of this thesis, Cloud9's support for Literate Programming, has been done entirely in Cloud9's "code view."

6.5 Structure of Code View Hierarchy

Credit for the basic structure of the Hierarchical View (or "code view" above) can be given to another graduate student at San Jose State, Gene Yao. Although the Hierarchical View structure lacked documentation, it was sound and allowed the development of the Literate Programming enhancement on top of it with a few alterations.

All the information inside the Hierarchical View window is contained within an instance of the HViewView class. The HViewView class is derived from the Borland's OWL TWindowView class, which is in turn derived from the TView and TWindow classes. HViewView derives from the TView and TWindow classes to provide the Hierarchical View with default features common to many Windows applications, for example custom menu settings for a View and any Window accepting default Windows messages.

As mentioned before, an instance of the TView class is an interface to a document. It allows a document to access its view(s) (a document can be displayed in more than one view simultaneously). Views in turn can call document functions to request input and output streams. Public functions common to a TView include SetViewMenu which sets the menu for a view, GetWindow, to get the View's enclosing window, GetDocument to get the document displayed in the View, and GetViewId to get the identification number of a view. An instance of the TWindow class is a generic Window that can be resized and moved. It provides default behavior specific to any Window and additional adornments such as scroll bars.

The HViewView class contains a pointer to a single HViewNode class. The HViewNode class contains a hierarchical subtree structure for any of the C++ basic structural units, functions, classes, etc. For example, the section in the Hierarchical View where the programmer would enter a class is an HViewNode and consists of separate subtrees for each of the functions of the class, and a table (single subtree) which provides the area to modify the private fields of the class, and other appropriate subtrees. The HViewNode class displays its own subtree in the Hierarchical View window when its Paint function is invoked. Each of the subtrees of the HViewNode class is an instance of the HViewChild class. The HViewNode class contains an array of pointers to HViewChild.

The HViewChild class (each subtree) provides hierarchical text edit windows in which the programmer specifies code: for example, the arguments of a function, or the functions of a class. The programmer clicks on the appropriate section of the

HViewChild to invoke a text edit window. It was part of the work of this thesis to return a Rich Text Edit window when appropriate.

HViewNode class inherits from the HViewChild class and so the HViewNode is also an HViewChild. This class design allows a recursive nesting of hierarchical subtrees displayed in the View. In order to display the entire hierarchical structure in the Hierarchical View, the Paint function of the _root HViewNode in HViewView is called. This recursively calls the Paint function of all the HViewNodes in the Hierarchical View.

6.6 Unfinished Features

Cloud9 is at a stage at which it is beginning to become a useful tool to a software engineer. Currently, the "cloud" view and the "code" view are not strongly connected and work needs to be done linking the two. Until these two views are fully integrated, an essential aspect of Cloud9 is still lacking. This will become the work of future graduate students. Bugs have been ironed out in producing this thesis that make Cloud9 useable in the "code view." Cloud9 is now much less likely to cause General Protection Faults (none were experienced during testing).

PART II: LITERATE PROGRAMMING

7 LITERATE PROGRAMMING AND KNUTH'S WEB SYSTEM

7.1 Computer Programming as an Art

Computer Science has struggled to earn its name as a science since its beginning. A major goal of Formal Computability Theory was to make working with computers a legitimate scientific academic discipline. Most dictionaries, such as the New World Dictionary, 1974, define a science as "systematized knowledge derived from observation, study and experimentation carried on in order to determine the nature or principles of what is being studied." A science has laws and principles, and laws and principles are exactly what Formal Computability Theory provides computer science. If computer science were not called a science, then it would not have been a legitimate emerging academic discipline when viewed by a staunch academic community in the late 1950's. The purpose of all Association for Computing Machinery's (ACM) periodicals was described in 1959: "If computer programming is to become an important part of computer research and development, a transition of programming from an art to a disciplined science must be effected."[14] It was as if something was inherently bad about art, and a clear separation of science and art must be established. Inevitably computer science was to become an academic discipline with or without Formal Computability Theory, and the new technology has changed all of our lives.

Donald Knuth of Stanford University proposed the resurrection of the idea of computer programming as an art. In his 1984 treatise on Literate Programming, he explores this idea. Knuth states, "When I speak about computer programming as an art, I am thinking primarily of it as an art form, in an aesthetic sense. The chief goal of my work as educator and author is to help people learn to write beautiful programs. . . . My feeling is that when we prepare a program, the experience can be just like composing poetry or music; as Andrei Ershov has said programming can give us both intellectual and emotional satisfaction, because it is a real achievement to master complexity and to establish a system of consistent rules."[15] In conclusion, Knuth states "We have seen that computer programming is an art, because it applies accumulated knowledge to the work, because it requires skill and ingenuity, and especially because it produces objects of beauty. Programmers who subconsciously view themselves as artists will enjoy what they do and will do it better."[14]

7.2 Literate Programming and Knuth's Web System

Literate programming presented the concept of a program as a work of literature. The program itself was an expository text explaining to the reader the purpose of a program and how the program accomplished its task. The programmer focused on explaining the program to the reader rather than writing the actual code. In an effort to explain the program to the reader, the programmer ends up explaining the program to the computer, ideally with much less struggle. There is less struggle as the programmer presents the parts of a program in ". . . an order that makes sense on expository grounds" [13] which naturally parallels human psychology. The programmer designs the program as a narrative story told to another, rather than the computer driving the programmer's method. Although the computer is, of course, considered when writing a literate program, the idea behind true literate programming is that the computer is considered second.

Knuth's belief that programmers "who subconsciously view themselves as artists will enjoy what they do and will do it better" prompted him to write a new system for program documentation. In 1984, he proposed a system of program documentation called WEB. WEB files were a combination of two different languages, a programming language (Pascal) and the TEX documentation language (developed by Knuth). TEX is a typesetting command language and Knuth wrote a compiler that when input a TEX file, output a formatted descriptive document. A WEB file functioned as both a program and a descriptive document. The WEB file could be compiled with an ordinary Pascal compiler into an executable program, or with the TEX compiler into a readable descriptive document. The descriptive document produced using WEB could use any of the commands of the TEX formatting language, including tables, formulas, boxes, diagrams, producing a previously unknown level of documentation within a program. (See Figures 5 and 6 on pages 67-68.)

A style of literate programming commensurate with a procedural language, Pascal, was first introduced. The program was divided into logical units called sections.

Sections were grouped, and each section group was named by topic such as "The Output Phase" and "The Program Plan." The individual sections would further the goals of that particular section group topic. Each section consisted of "code", a single macro associated with the "code," and literate text. These macros were typically several word long descriptions of the task the "code" performed, such as "<Initialize the data structures>" or "<Increase j until next prime number>," although they could also be entities such as "<Other constants of the program>." "Code" consisted of Pascal code and other macros embedded within the Pascal code. The literate text described the theory behind the "code" presented in the section, or if the section related to a larger topic (such as a section under "The Program Plan") described the program or other larger abstraction. This allowed the programmer to focus on explaining the basics of the section under consideration. A top-most section was allotted for the purpose of the program. The high level procedures of the program were each allotted a section ("code" and associated macro, and text) and procedures were broken down into sub-sections ("code" and associated macro, and text) representing other logical scraps of code. The decomposition of the program could take place in a top-down order or a bottom up order. For example, the specifics of a function's implementation or the general purpose of a module could be explained first. It really did not matter as the program could be viewed as a "web" or tree. Whatever order suited one's stream of consciousness could be used.

WEB provided excellent support for document readability. It automatically generated numbered sections and embedded the section number of the macro into the macro itself. It automatically created an index by section number of all identifiers and macros in the program and placed it at the end of the document. Actual mathematical symbols could be placed in the document code listing, instead of the symbols the computer interprets as mathematical symbols.

The primary ideas behind WEB were:

1) Code and documentation must come from one source, or they will diverge. Literate Programming copes with this problem by embedding the document inside the program itself.

2) Documentation order may differ from code order. In order to clearly describe a program to another person, it is better to traverse the program in a natural psychological progression rather than explanation by way of a linear traversal of each module.

3) Plain ASCII cannot provide decent documentation. Other visual cues are necessary to convey the organization and information hierarchy of the module. Visual cues provide a necessary means of explaining difficult material. It is easy for the reader to interpret a document differently than the author intended without visual cues. Visual cues include the use of whitespace and different typefaces to indicate the hierarchy of information. These cues tell the reader about the organization of the document. The reader should be able to look at a document and from the format determine the organization of the document. The use of visual cues

should be consistent throughout the document. Figures or images also help to convey information which is difficult to describe verbally.

8 CLOUD9 SUPPORT FOR LITERATE PROGRAMMING

8.1 On-line Documentation

Much has changed with regard to the presentation of documentation since Knuth introduced WEB in 1984. On-line documentation has become popular in the 1990's. The internet has promoted the idea of hyper-links (links from documentation to related documentation) and added the capability of user choice determining the logical flow of documentation. The user of an Internet Browser chooses what he or she would like to see next in accordance with their own style of learning. Although the implementor of a set of web pages devoted to a particular topic may present the information in a default format that is believed to be logical, the user often has the choice of whether to accept the default order of presentation or create his/her own.

The idea of a document as a growing, evolving and living entity has been promoted by on-line documentation. As such, on-line documentation does not necessarily become out-of-date but can be updated. Pages can be inserted into the document with minimum difficulty. The document is updated in real-time. From a conservation standpoint, on-line documentation also makes sense as it saves paper and can be printed when and if necessary.

The purpose of this thesis is to provide the engineer the additional capability of embedding literate text in a Cloud9 design document. Furthermore, in keeping with the advent of on-line documentation, the implementation of embedded literate text in Cloud9 is as Microsoft Rich Text. Microsoft Rich Text was selected for three reasons:

1) It is an industry standard and most Word Processing applications read it. There is no need to "re-invent the wheel" by having to worry about formatting issues for printing and displaying that have already been engineered.

2) With the proper formatting adornments, a Microsoft Rich Text File can be compiled into a hyper-linked Microsoft Help File.

3) It was possible to create a Rich Text control from an existing simple Text Edit control. Currently, there is no Hyper-Text-Meta-Language (HTML) control offered by any class framework. This is why HTML was not selected as the output documentation of Cloud9.

This thesis moves the ideas behind WEB into current on-line standards of documentation.

8.2 Cloud9 Support for Literate Programming

Each C++ module for a project is a separate Cloud9 document. As mentioned previously, the underlying document is simply a set of data. In the "code view," the module or document is broken down into two major sections, the public section and the private section. The public section and the private module sections represent a separation of concerns. The public section contains classes and code that are to be used by other programmers (the target audience of programmer's class, encapsulated from the private section). The private section contains classes and code that support the user classes defined in the public section. These are classes which the user of classes of the public section does not need to know about. They contain implementation dependent information.

Cloud9 can generate two distinct types of descriptive documents: a user document or an implementor document. The user document contains only the information in the public module section, and the implementor document contains information in both the public and private module sections. In this way, a true separation of concerns is effected. The user of a class will not be exposed to implementation details, yet the user of a class will have basic information on the use of any public classes, functions, data types, etc.

The example Cloud9 document presented during the thesis defense and in thesis defense pre-meetings illuminates this separation of concerns. The document called Camera consists of three classes: Camera, Box3, and Polyhedron3. This module encapsulates a common graphics primitive known as a Camera which is used to view shapes in 3-space. The Camera views the shapes in mathematical 3-space and projects the shapes onto a real 2-dimensional surface, the pixels of the screen. As mentioned, the document is a C++ module, and as such Cloud9 can be used to perform a code generation and produce camera.cpp and camera.h. The Camera class lives in the public module section, while Box3 and Polyhedron3 live in the private module section. Box3 and Polyhedron3 support the Camera class. An instance of Box3 is the "world" which the

Camera views, and Polyhedron3 are the shapes or objects which the Camera views. Of course, it is somewhat arbitrary in this case to assume that Polyhedron3 lives in the private module section and Camera lives in the public module section. The Camera could be seen as a supporting class to a Polyhedron3 which lives in the public module section, but, the author has chosen the Camera class to live in the public module section. This choice would affect the focus of the literate presentation of the module.

8.3 Use of Literate Programming in Cloud9

Similar to the Booch incremental and iterative method of Object-Oriented Design, the engineer can logically pick a level of abstraction and begin to document ideas at that level. Names may be chosen for classes, variables, and any other structures, and their purposes can be described at the literate level first. The iterative philosophy of system development and incremental integration into a functional whole in stages are facilitated by tools which give the programmer the ability to provide explanation anywhere within the code. The programmer has freedom to express in literate text in whatever level of detail is necessary at the current stage of development. Notes can be revisited and fleshed out in more detail during the process. As notes are modified and refined, the code is refined.

For example, suppose that the programmer wanted to create a virtual tape drive. The virtual tape drive would respond to exactly the same Escon Channel messages that a normal IBM 3490 tape drive would respond. Its status would be updated based on these commands, but instead of it being a physical tape drive, it is just software mimicking the behavior of an IBM 3490 tape drive. Instead of the storage media being magnetic tape. suppose the storage media were a physical hard drive which greatly outperforms the old tape drive. Some key abstractions the programmer might want to code would be some mechanism for the Escon Channel messages to be converted into software messages and virtual tape drive itself. Another would be a software simulation of physical paths which might be available to the tape drive and a unit to manage these paths to prevent conflict with other virtual tape drives using the same paths. First the names of classes appropriate to these abstractions would be selected, then space would be allotted in Cloud9 for each of these classes. The programmer could immediately begin filling in the literate text associated with these abstractions speaking to their purpose, use, necessity of their presence or whatever. Based on the information in some of this literate text, the programmer could begin to think about the data structures necessary for the problem. What attributes would a virtual tape drive have? What needs to be present to mimic a physical tape drive? Coding could begin at this abstract level, with literate text being revisited and updated as new discoveries are revealed by the coding process. The classes are iterated upon and refined, first with "broad strokes of the paint brush," then more details are fleshed out.

This enhancement is not meant to modify the philosophy behind all coding in Cloud9 to a literate philosophy, but to enable the programmer to use literate techniques. A convenient interface and mechanism to show and hide literate text is essential to making this task easy for the engineer. If it is not easy to use, it will not be used. This was a primary goal of the enhancement. A reminder that Cloud9 has a literate programming tool is the comment box provided for the module. all classes. all functions (class methods and external) and operations which invoke a Rich Edit control. The programmer will merely have to click on the comment field to add literate text. The hope is that because the tool interface is visible at all times. literate prose and comments will be added more frequently and that updating comments after modifications is more likely.

Associated with each area where it is possible to specify Rich literate text in the Cloud9 document are additional fields titled "See Also" and "Next Comment." "See Also" places a hyperlink to the page specified by the "See Also" field on the page associated with the Rich literate text. "Next Comment" ensures that the next page viewed when using the Browse Buttons of the Help file is the page specified here.

8.4 Cloud9 Addresses the Ideas of WEB

It is not the goal of this thesis to recreate Knuth's WEB System within Cloud9 but to add the ideas behind WEB to Cloud9.

1) Code and documentation must come from one source. or they will diverge. Because the code, object oriented design and descriptive document all come from the same Cloud9 document, the descriptive document and code are not allowed to diverge. The descriptive document automatically contain sections of literate text for all classes and class operations present in the document, as well as global functions and module level comments. The programmer can also specify logical links to related Classes using the "See Also" feature of Cloud9 which creates physical hyper-links.

2) Documentation order may differ from code order. Through use of the "Next Comment" feature. documentation traversal order can differ from the code order. Cloud9 allows the programmer to present the documentation associated with their module in an order which facilitates the reader's understanding. The programmer can have one topic lead into another related topic which is not necessarily physically adjacent in the code. For example, an operation of a class which calls another operation of a different class could be made adjacent in the documentation.

3) Plain ASCII cannot provide decent documentation. Documentation created using Cloud9 provides all the symbols available to a Rich Text Editor, including mathematical symbols. The programmer has the complete set of fonts and sizes available in Microsoft Rich Text. Code can be inserted into the Rich text in the Courier font, a standard for most typeset code documentation which makes code immediately recognizable as code. Font sizes and underlining can be used to give visual cues to the hierarchy, or relative importance, of the information being presented. Although cutting and pasting of figures into the Rich Edit control would have been a nice feature, the support for embedding objects through use of Microsoft OLE technology could not be accommodated in the time frame of this thesis.

In addition to addressing the main ideas of WEB, Cloud9 provides additional functionality. Similar to WEB, Cloud9 supports a table of contents of all the classes in the module (the second page: the first page is the literate description of the module). The table of contents in conjunction with automatic indexing using the built-in Help file search for topic capability provide instant alphabetized reference to any topic in the documentation. Similar to WEB. Cloud9 also supports associating specific code with literate text in the document by simple cutting and pasting. Although it would have been desirable to have the code change in the documentation when changed in the code view. this feature could not be accommodated in the time frame of this thesis. It also presented the issue of the documentation becoming automatically out of synch with the code when this occurred. Beyond the scope of WEB. on-line documentation with hyper-links brings the ideas of WEB into modern day standards of documentation.

8.5 Traversal Order of Documentation

The default traversal order of documentation is first a help file page of top-level module comments. then a help file page of all the class names (names are links) in the module. then a help file page of the first class with names of its functions (names are links). then a help file page for each function/operation of the first class. then a help file page of the second class with names of its functions (names are links). and so on following a breadth first traversal. This traversal order is known as breadth first. (See Figure 7. page 69.) The user is provided with a reasonable default if he/she does not wish to provide any information in the "Next Comment" field. If the user specified a title in the "Next Comment" field. the program follows "Next Comment" links as far as possible. then returns to first unvisited node given by the default traversal.

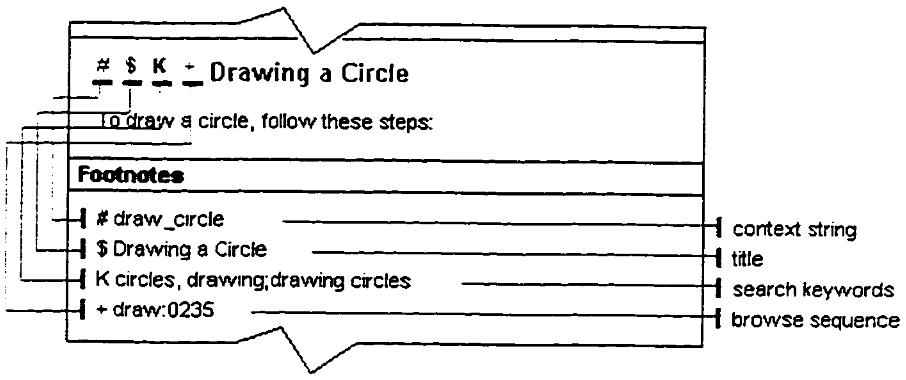
8.6 Microsoft Help Compiler Source File Format

There are two source files that the Microsoft Help Compiler needs in order to produce a Help file: a Rich Text Format (.RTF) file, and a Help Project (.HPJ) file. The Rich Text file contains the topics of the compiled Help file, separated by page breaks: each page is a Help file topic. The Rich Text from the comment field of the Cloud9 topic appears exactly the same in the Help file, and all formatting is maintained. Characteristics of the Help file topic such as browse page ordering, page identification for hyper-links, title of the topic, and related key words for searches are determined by hidden embedded footnotes in each page (see Diagram 1). The footnotes precede the title of each topic and are as follows:

Footnote character	Used to identify	Purpose
#	Context String	Uniquely identifies the topic
$	Title	Appears as the topic title in the Search dialog box and the history list
K	Keywords (and phrases)	These words and phrases appear in the Search dialog box
+	Browse sequence	Determines the order of the topics when the user browses through them.

Diagram 1

Footnote Insertion

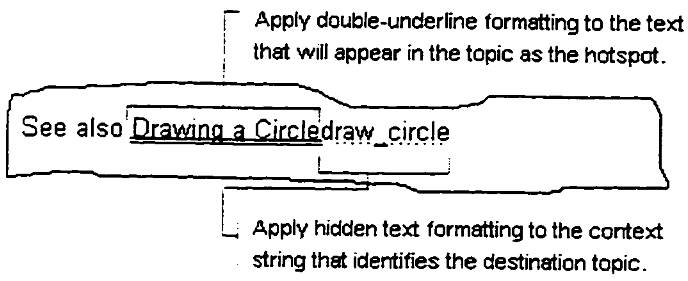


(diagram from Creating Windows Help. Microsoft Corporation. 1993)

Code to generate these footnotes with appropriate title/identifiers as specified in the Cloud9 document has been added. Hyper-links are also added if the user specifies a topic in the "See Also" section. The Microsoft Help compiler recognizes hyperlinks by a special formatting. The hyperlink highlighted text is double underlined and is immediately followed by special formatted hidden text of the Context String (# footnote) of the page of the link (see Diagram 2 below).

Diagram 2

Hotspot Insertion


Apply double-underline formatting to the text that will appear in the topic as the hotspot.
See also Drawing a Circledraw_circle
Apply hidden text formatting to the context string that identifies the destination topic.

(diagram from Creating Windows Help. Microsoft Corporation. 1993)

The "Next Comment" field determines the Browse Sequence (+ footnote). The "+" footnote determines the order of traversal of documentation when using the Help file browse buttons. Each topic is given a distinct integer browse number. and there is no possibility of a cycle. In this implementation. search Keywords (K footnote) are the same as the Title ($ footnote) of the topic and when invoking the search for topic function of a Help file. the titles of the topic pages are presented.

The Help Project (.HPJ) file contains information that the Help compiler uses to construct the Help file. It consists of instructions which control characteristics of the Help file. Only a few simple instructions are needed to generate the Help file using Cloud9. The Help project file is the same for all Cloud9 generated help files, except the name of the Help project file and Rich Text file reflects the current name of the document. Cloud9 uses only three instructions in the Help project file:

```
[OPTIONS]
CONTENTS=context_string     //specifies the context_string
                            // or ID of the first page
[CONFIG]
BrowseButtons()             //include browse buttons
[FILES]
RTF_filename                //Rich Text file name
```

`BrowseButtons()` enable left and right arrow keys which appear at the top of the Help file and allow a programmer specified traversal order. Cloud9 automatically calls the DOS program HCP.EXE, the Microsoft Help file compiler when performing document generation.

8.7 Implementation

8.7.1 Creation of the Rich Text Source File for the Help Compiler

The data structure that contains an individual Help file page or Help file topic is called a `page_node`. It is as follows:

```
typedef struct page_node {
                struct page_node    *left, *right;      // ptrs to subtrees
                Chi_String          title;              // title string
                int                 id;          // # context integer id
                Chi_String      browse;          // K search keywords
                Chi_String      rich;            // rich text associated
                                                 // with page
                Chi_String      next;       // name of "Next Comment",
                                            // if any
                Chi_String      seeAlso;    // name of "See Also",
                                            // if any
                int     marked;  // TRUE if the page has been traversed
                                 // when determined page order output
                                 // (related to BrowseButtons() feature;
                                 //   although traversal order may
                                 //   change, each page is visited once)
} PAGE_NODE, *PAGE_NODE_PTR;
```

Each page of the Help file is placed into a binary tree based on the alphabetical title of the page. A binary tree was used to preserve the uniqueness of topics. If a page already exists in the binary tree it can not be inserted. (Although two pages may have links to the same page, there should only be one copy of the page in the tree.) Another reason for selecting a binary tree is that it is an easy structure to traverse recursively. The fields of the data structure are standard `left` and `right` pointers to the nodes' sub-trees, the `title` of the page (printed at the top of the page), the `id` of the node (also called the context string (ID) and is associated with the "#" footnote), the `browse` field that identifies the search keywords which appear when using the search for topic function of Microsoft Help files (set the same as the `title`), the `rich` text associated with the page, the title of the page of the `next` comment (if the field is used), the `title` of the page of the `seeAlso` field (if the field is used), and a Boolean representing if the page has been traversed or `marked` (used for page ordering).

The rich text associated with the page of the Help file in the `rich` field grows in stages. First it only contains the rich text of the page or whatever appears in the Cloud9 Rich Text comment field. Then the footnotes are inserted at the front of the rich text based on information found in the other fields of the `page_node` structure. Next, links to functions and operations for the page of each class and see also links are added to the `rich` field (if any). As a last step, only the `rich` field of each page need be written to the .RTF file. The `rich` field eventually contains all the necessary information.

The `page_node` binary tree is constructed by first inserting the `page_node` associated with the module, then the `page_node` listing the names of all the classes in the module (links to all the classes), then starting with the public module section, the first public class page, then pages for each of its member functions and operations, the second public class page, then pages for each of its member functions and operations. Then the private module section classes, functions and operations are iterated as above if implementor documentation is being generated.

The last page to be inserted is a page containing only a closing paren "}" which has the `title` "zzzzzz". Microsoft Rich Text is a completely parenthesized language, and by traversing the binary tree in alphabetical order by title, the last closing paren is placed at the end of the Microsoft Rich Text file. This ensures that the Microsoft Rich Text file is always valid because an unclosed parenthesis in a Microsoft Rich Text file will cause the Help compile to fail.

The function to enter a Help file page, called `enter_page`, follows a standard binary tree insertion algorithm and is listed in Figure 8, page 70. Prior to calling the `enter_page` function, a function called `try_enter_page` ensures that the page has not already been entered.

It is helpful to dissect the Rich Text of a page of a Help file and expose the Help file attributes that have been entered. Figure 9 on page 71 lists a standard Help File page source code in Rich Text (note the C++ comments to determine the meaning of symbols; actually, C++ style comments are illegal in Rich Text but have been added to aid the reader). Similar to the "C" programming language, Microsoft Rich Text is fully parenthesized and the "\" character represents special formatting. Fonts are listed in the font table and are referred to by "\f#" associated with them in the font table. The font table closest to the formatted rich text is the one that applies.

Following the insertion of the all basic page data, a recursive function named `iterator` is called given the binary tree root node as an argument. This function inserts all four footnotes at the beginning of each page (in the `rich` field) with the corresponding footnote information based on the fields of the `page_node` structure (except the "+" browse traversal order footnote, which is determined last by following the "Next Comment" links).

After iterator is called, a function named update_functions is called. This function searches the page_node binary tree for the page associated with each class. Each class page is modified (the rich field) by adding the names of its member functions as links. Recall that links are created by applying hidden text to the context string associated with the member function page immediately following the double underlined member function name. The context string associated with the member function page is provided by searching the binary tree for the title of the member function, then reading the associated id field of the page_node.

After update_functions is called, a recursive function named update_seealso is called with the binary tree root node as an argument. The code for update_seealso is in Figure 10, on page 72. It compares the seeAlso field of each page with the empty string. If the seeAlso string is not empty, the title of the page of the seeAlso string is searched in the binary tree. Similar to update_functions, the context string id of the seeAlso entry is added as hidden text following the double underlined title of the seeAlso string creating a link to the seeAlso page. As shown in the code, it was convenient to define some commonly used Rich Text strings such as short_title_open and newline. These contain Rich Text instructions to format for the insertion of a title name and addition of a new line, respectively.

Next, the sequence function updates the "+" footnote (traversal sequence) with the appropriate page number. It determines the page number of the next page by following the next comment field of the current page (if it has any) and searching for a match in the binary tree with that title. If a match is found, the value of page number is incremented and placed in the "+" footnote of the binary tree node associated with this next page. marked is set to TRUE for this next page, and the "Next Comment" of this next page is followed (if any). If the page has no next comment field, the next-next page is determined from the default traversal sequence (see Section 9.7).

All the pages are written to the .RTF file using the recursive iterator function shown below. Because the Help compiler takes care of browse sequence ordering based on the "+" footnote, the files can be written to the .RTF file in any order (used in-order traversal of the binary tree, although pre or post would have been fine). Lastly, the memory of the page_node binary tree is freed using the recursive delete_page_display function.

```
void iterator(PAGE_NODE_PTR page_display,DOC_OutputState& rtfos)
{
        if (page_display)
        {
                iterator(page_display->left,rtfos);
                 rtfos.print(page_display->rich).endl();
                iterator(page_display->right,rtfos);
        }
}
```

8.7.2 Development of Rich Text Editor Class

In order to provide Rich Text for the Help compiler, Cloud9 needed a Rich Text Edit control. Borland's OWL, used to create Cloud9, does not offer an encapsulated class to provide the functionality of a Rich Text Edit control. Borland does, however, offer a TEdit class which provides the basic functionality of a simple Text Edit control. One of the obstacles involved in this thesis was to implement a Rich Text Editor class similar to Borland's TEdit class but having Rich Text functionality. Specifically, what was missing from Borland's TEdit class was the ability to select fonts and sizes, perform bold, italic and underlining. Research into a Rich Text Editor application written in C which used calls to Windows 95 Rich Text Library suggested that it could be possible to simply add the functionality of a Rich Text Editor to Borland's TEdit class. Part of the design work of this thesis involved creating new methods for the TEdit class which encapsulate Rich Text Editor functionality. For example, if the new Rich Edit class received a "CmBold" command, it would have to perform the appropriate Rich Text modification. It would also have to by default perform all of the same functions of a TEdit control when appropriate. For this reason, the new Rich Edit class created for this thesis inherited from the TEdit class. Depending on the area of the Hierarchical View clicked on, an appropriate Rich Edit control or TEdit control would be returned. It is interesting to note that while this thesis was in development, Microsoft Foundation Class Library Version 4.0 implemented a class to encapsulate a Rich Text Edit control. It performs everything the version created for this thesis performs, but also supports cutting and pasting of pictures. The header file for the new Rich Text Edit control class appears in Figure 11. on page 73. The actual name in the Cloud9 application for the new Rich control class is TExampleRich. Although it appears to inherit from a class called TRich. TRich is an exact duplicate of TEdit, except for the name change TRich for TEdit. This was done because a direct inheritance from the TEdit class produced compile errors related to the internal workings of OWL and this was a simple work-around. As shown in Figure 11. the new member functions were added to express the new desired functionality.

8.7.3 Rich Text Control Usage

Once the Rich Text control had been created, it was necessary to modify Cloud9 so that the new Rich Text control was returned if a Rich comment box was selected in the "code view." This was easily accommodated by looking at the name of the node in the "code view." If it was an edit control for comments associated with a function, for example, a Rich Text control would be returned. (See Figures 12, 13, and 14 for common uses of the Rich Control, pages 74-76.) In order to discern whether a Rich Edit control or a TEdit control had been invoked when updating the contents of a edit control (when the user clicked outside the active edit control) a `dynamic_cast` was convenient. As shown in Figure 15 on page 77 in the C++ code to update a "code view" string (`HViewString::update`), if the edit control updated was indeed a TExampleRich, some special behavior needed to take place.

First. a special operation specific to saving Rich Text had to be invoked. called `SaveText()`. `SaveText()` calls a Windows callback function. A callback function is a function which once invoked automatically invokes itself until it is finished. Each time it is invoked. a known amount of data is processed. As a result the callback function pulls consecutive units of data out of the Rich Text control repeatedly and saves them as text in a specified location. This repeats until the entire length of the data of the Rich Text control has been processed. Notice that with the deletion of the Rich control at the end of the operation. the position and length had to be reset to zero for the callback function as the callback function had to remember where it left off in-between its recursive calls to itself.

8.7.4 Other Enhancements to Cloud9

Cloud9 had a number of problems at the outset of this thesis. While the framework existed for public and private module sections in the "code view." data entered into the public module section would appear in the private module section. Apparently. the flag for determining the module section that an edit was to be inserted or replaced was neglected for "code view" editing updates. The solution was simple. An additional field for public vs. private module sections was added to all modification and creation function for the basic types supported by Cloud9 to ensure the appropriate updates. For example. a function to change the name of a function. called update_function_name. would have the additional parameter of module section (public or private).

Several small bugs were fixed during the course of this thesis. including General Protection Faults caused by dereferencing NULL pointers. poor memory deallocation. and incorrect array indexing for the names of files produced using Cloud9. Some of these problems made Cloud9 unusable due to frequent crashes. and they had to be dealt with immediately before progress could be made on the topic of this thesis. Some cosmetic changes to Cloud9 were also performed. including hiding the of inactive Rich Text Edit controls.

In order to ensure unique Help File page names for functions and operations. arguments entered into the Function Argument Table had to automatically update the function name. Uniqueness is a necessary attribute due to the common overloading of functions in C++ producing functions with identical names but with different argument lists. Function Argument Table changes and insertions are now instantly reflected in the function name.

9. CONCLUSION

Object-Oriented Design is most useful when applied to real world modeling because it is natural for us to think of the world as composed of objects interacting. It allows the programmer to break down a complex system into different levels of abstraction and to focus on encapsulating behavior with respect to each level. Similarly.

explaining a program to another person more naturally parallels human psychology than explaining the program to a computer. Using Literate Programming techniques, in an effort to explain the program to another person, the programmer ends up explaining the program to the computer, ideally with much less struggle. Although Cloud9 is still in its development, it already possesses some attributes which make it unique and allow programmers to express their ideas in ways which facilitate Object-Oriented Design and promote understanding.

FIGURE 1

HYDROPONICS GARDENING SYSTEM TOP-LEVEL CLASS DIAGRAM

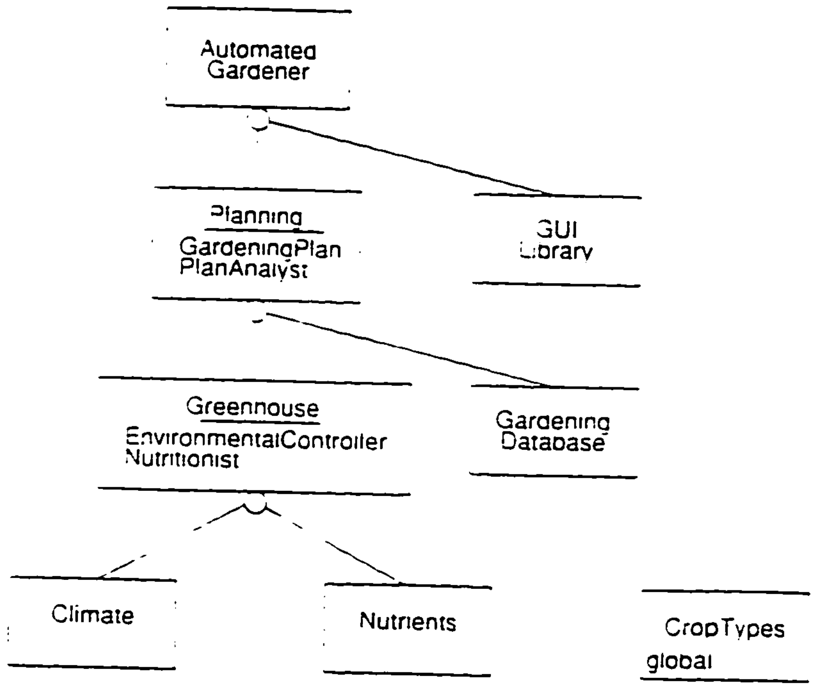


FIGURE 2

HYDROPONIC GARDENING SYSTEM CLASS DIAGRAM

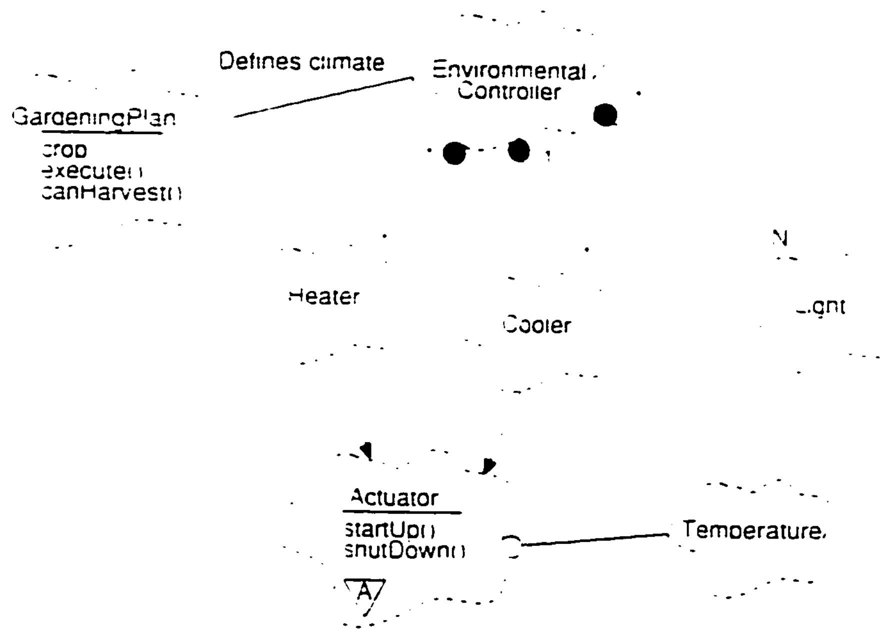
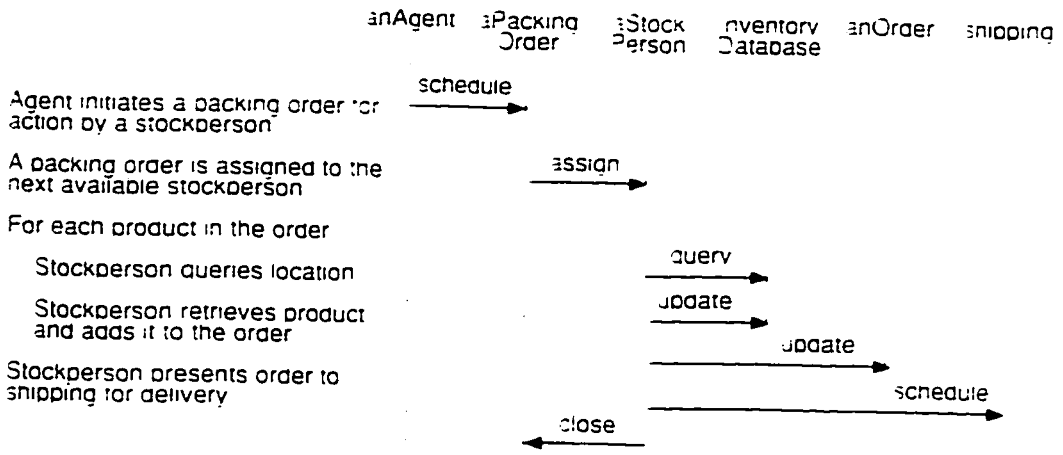


FIGURE 3

PACKING ORDER SCENARIO DIAGRAM



Agent initiates a packing order for action by a stockperson

A packing order is assigned to the next available stockperson

For each product in the order

Stockperson queries location

Stockperson retrieves product and adds it to the order

Stockperson presents order to shipping for delivery

FIGURE 4

HYDROPONICS GARDENING SYSTEM MODULE DIAGRAM

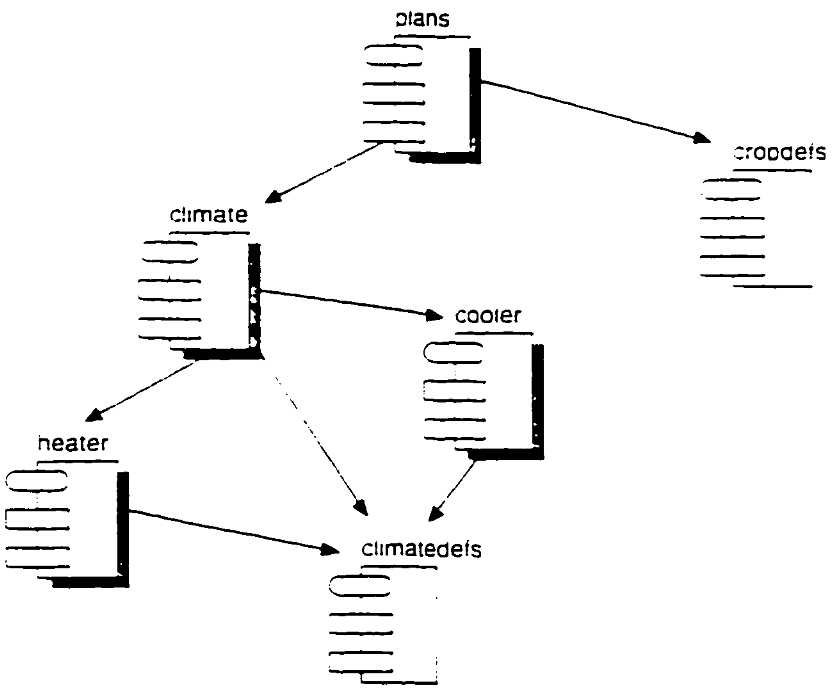


FIGURE 5

PASCAL PROGRAM GENERATED FROM WEB FILE

```
{1:}{2:}PROGRAM PRINTPRIMES(OUTPUT);CONST M=1000;{5:}
RR=50;CC=4;WW=10;{:5}{19:}ORDMAX=30;{:19}VAR{4:}
P:ARRAY[1..M]OF INTEGER;{:4}{7:}PAGENUMBER:INTEGER;
PAGEOFFSET:INTEGER;ROWOFFSET:INTEGER;C:0..CC;{:7}{12:}
J:INTEGER;K:0..M;{:12}{15:}JPRIME:BOOLEAN;{:15}{17:}
ORD:2..ORDMAX;SQUARE:INTEGER;{:17}{23:}N:2..ORDMAX;{:23}
{24:}MULT:ARRAY[2..ORDMAX]OF INTEGER;{:24}BEGIN{3:}{11:}
{16:}J:=1;K:=1;P[1]:=2;{:16}{18:}ORD:=2;SQUARE:=9;{:18};
WHILE K<M DO BEGIN{14:}REPEAT J:=J+2;{20:}
IF J=SQUARE THEN BEGIN ORD:=ORD+1;{21:}
SQUARE:=P[ORD]*P[ORD];{:21}{25:}MULT[ORD-1]:=J;{:25};
END{:20};{22:}N:=2;JPRIME:=TRUE;
WHILE(N<ORD)AND JPRIME DO BEGIN{26:}
WHILE MULT[N]<J DO MULT[N]:=MULT[N]+P[N]+P[N];
IF MULT[N]=J THEN JPRIME:=FALSE{:26};N:=N+1;END{:22};
UNTIL JPRIME{:14};K:=K+1;P[K]:=J;END{:11};{8:}
BEGIN PAGENUMBER:=1;PAGEOFFSET:=1;
WHILE PAGEOFFSET<=M DO BEGIN{9:}
BEGIN WRITE('The First ');WRITE(M:1);
WRITE(' Prime Numbers --- Page ');WRITE(PAGENUMBER:1);
WRITELN;WRITELN;
FOR ROWOFFSET:=PAGEOFFSET TO PAGEOFFSET+RR-1 DO{10:}
BEGIN FOR C:=0 TO CC-1 DO IF ROWOFFSET+C*RR<=M THEN WRITE
(P[ROWOFFSET+C*RR]:WW);WRITELN;END{:10};PAGE;END{:9};
PAGENUMBER:=PAGENUMBER+1;PAGEOFFSET:=PAGEOFFSET+RR*CC;
END;END{:8}{:3};END.{:2}{:1}
```

FIGURE 6

TEX PROGRAM GENERATED FROM WEB FILE

```
\input webmac
% Prime example
\font\ninerm=cmr9
     :
descriptions are replaced by their expanded meanings, a
syntactically correct \PASCAL\ program will be obtained.\]

\Y\P$\4\X2:Program to print the first thousand prime
numbers\X\S$\6\
4\&{program}\1\  \37$\\{print\_primes}(\\{output})$;\6
\4\&{const} \37$\|m=1000$;\5
\X5:Other constants of the program\X\6
\4\&{var} \37\X4:Variables of the program\X\6
\&{begin} \37\X3:Print the first \|m prime numbers\X;\6
\&{end}.\par
\U section~1.\fi
     :
The first three macro definitions here are parametric;
the other two are simple.\]

\Y\P\D \37$\\{print\_string}(\#)\S\\{write}(\#)$\C{put %
a given string into the \\{output} file}\par
\P\D \37$\\{print\_integer}(\#)\S\\{write}(\#:1)$\C{put %
a given integer into the \\{output} file, in decimal %
     :
\inx
\:{Bertrand, Joseph, postulate}, 21.
\:\\{boolean}, 15.
     :
\:\.{WEB}, 1.
\:\\{write}, 6.
\:\\{write\_ln}, 6.
\:\\{ww}, [5], 6.
\fin
     :
\:\X4, 7, 12, 15, 17, 23, 24:Variables of the program\X
\U section~2.
\con
```

FIGURE 7

DOCUMENTATION DEFAULT PRE-ORDER TRAVERSAL ORDER

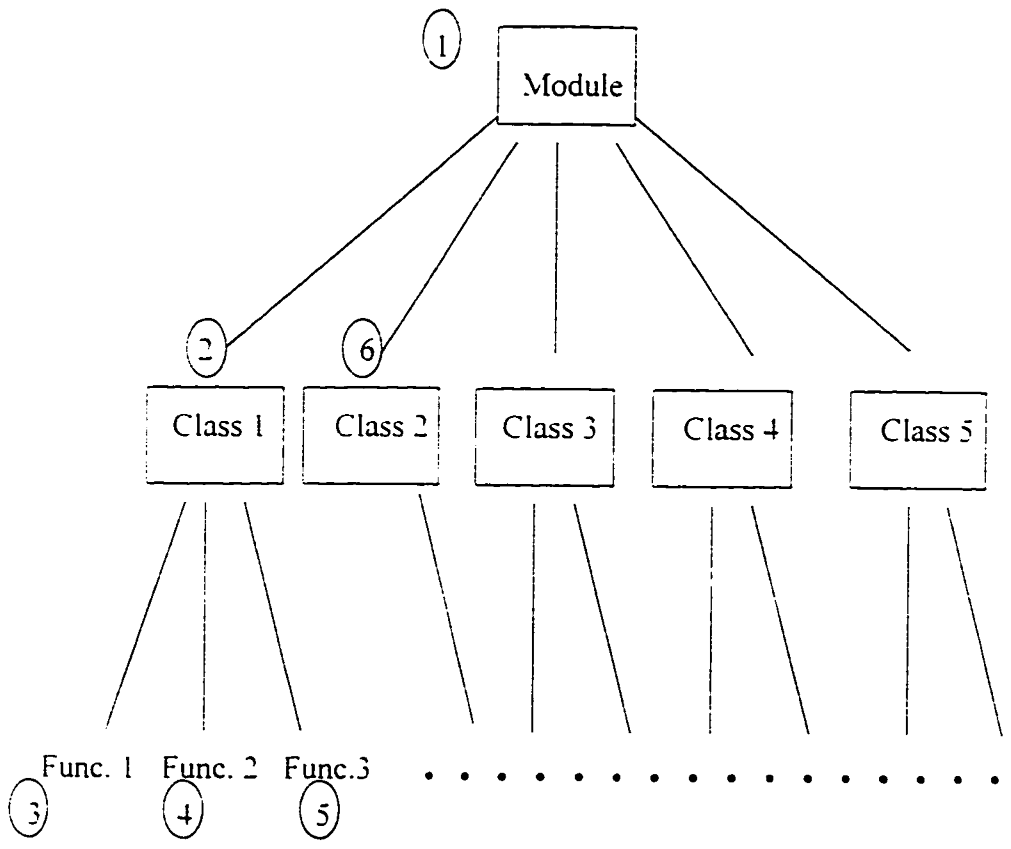


FIGURE 8

ENTER_PAGE FUNCTION

```
void enter_page(Chi_String &title,Chi_String &browse,Chi_String &rich,
      Chi_String &seeAlso,Chi_String &next, PAGE_NODE_PTR *npp)
                                           // npp = ptr to ptr to page root
{
      int            cmp;          //  result of string compare
      PAGE_NODE_PTR  new_nodep;    //  ptr to new entry
      PAGE_NODE_PTR  np;           //  ptr to node to test

      //  Create a new node for the name.

      new_nodep = new PAGE_NODE;

      new_nodep->title = title;
      new_nodep->browse = browse;
      new_nodep->rich = rich;
      new_nodep->seeAlso = seeAlso;
      new_nodep->next = next;
      new_nodep->left = new_nodep->right =  NULL;
      new_nodep->id = id;
      new_nodep->marked = 0;

      id++;

      //  Loop to search for the insertion point.
      while ((np = *npp)!= NULL)
      {
            if (title < np->title)
                  npp = &(np->left);
            else
                  npp =  &(np->right);
      }
      *npp = new_nodep;
}
```

FIGURE 9

RICH TEXT SOURCE CODE FOR HELP COMPILER (SINGLE PAGE)

```
{\fonttbl {\f0\fnil MS Sans Serif;}{\f1\fnil\fcharset2 Symbol;}
      {\f2\fswiss\fprq2 System;}{\f3\fmodern Times New Roman;}
      {\f4\fswiss\fprq2 Arial;}
      {\f5\fmodern\fprq2 Modern;}}
                              // specifies the font table and
                              // associated f identifier

{\colortbl\red0\green0\blue0;}
                              // specifies use default coloring

{\cs16\super #{\footnote \pard\plain \s15 \f4\fs20 {\cs16\super #} 13}}
                       // id or context string (# footnote) of the page is 13

{\cs16\super ${\footnote \pard\plain \s15 \f4\fs20 {\cs16\super $}
      AddBounce(Vector3& position,Vector3& velocity)}}
                       // title of the page ($ footnote) is function signature

{\cs16\super K{\footnote \pard\plain \s15 \f4\fs20 {\cs16\super K}
      AddBounce(Vector3& position,Vector3& velocity)}}
                       // search keywords (K footnote) same as function name

{\cs16\super +{\footnote \pard\plain \s15 \f4\fs20 {\cs16\super +} 017 }}
                       // page is 17th page using BrowseButtons (+ footnote)

\deflang1033\pard\plain\f4\fs32 Box3::AddBounce(Vector3& position,Vector3&
velocity)
\par \par {\fonttbl{\f0\fnil MS Sans Serif;}{\f1\fnil\fcharset2
Symbol;}{\f2\fswiss\fprq2 System;}{\f3\fmodern Times New
Roman;}{\f4\fmodern\fprq1 Courier;}{\f5\froman\fprq2 Roman;}}
{\colortbl\red0\green0\blue0;}
\deflang1033\pard\plain\f3\fs26 For a \plain\f4\fs26 Box3\plain\f3\fs26 , add
the velocity to the position of the object.  If you pass the border
\par of the box, then reflect the velocity and keep the old position.
The position of \par the object is the position of the center (or centroid) of
the object.\par
\par Here is an example for the position exceeding the x-dimension of the box:
\par \plain\f2\fs20\b
\par   \plain\f4\fs26  if (position.x() < _lox || position.x() > _hix)
\par      \{
\par      \tab velocity.Set(-velocity.x(), velocity.y(), velocity.z());
\par           bounceflag = TRUE;
\par      \}
\par
\par \plain\f5\fs26 Note that either AddBounce or Wrap are used for the
application, \par but not both at the same time.
\par \plain\f4\fs26
\par
                              // rich text source of the page.
```

FIGURE 10

UPDATE_SEEALSO FUNCTION

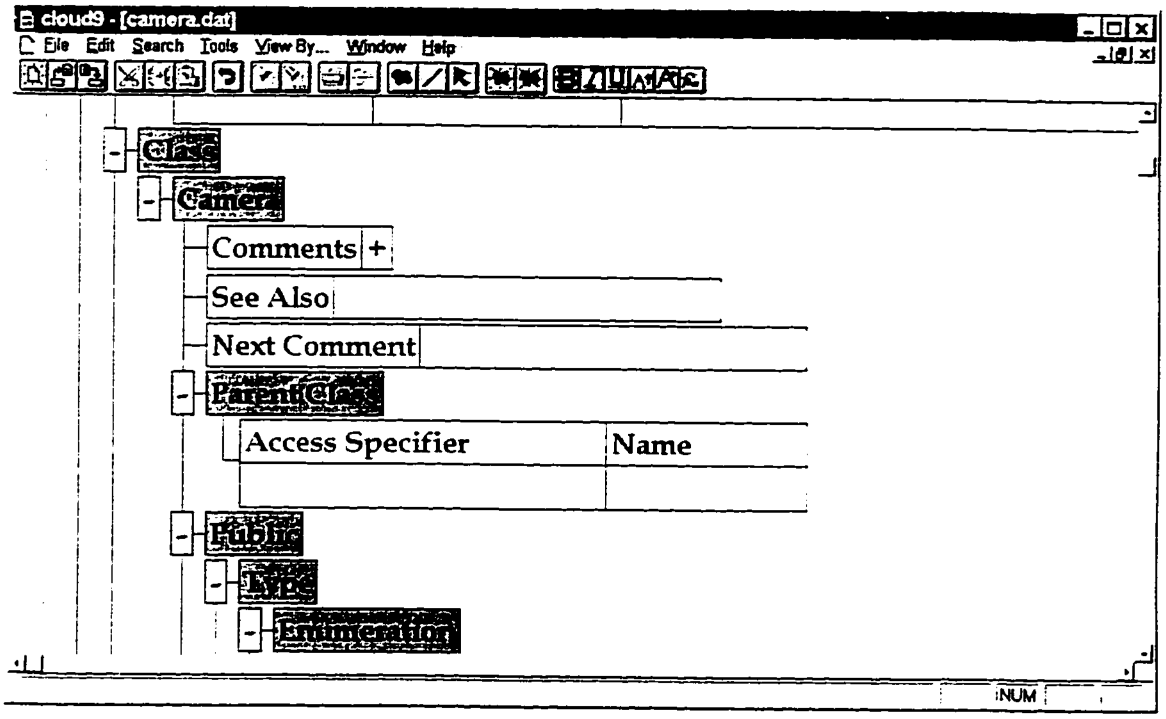
```
void update_seealso(const DOC_Module &module,PAGE_NODE_PTR page)
{
        int p,x,y,z;
        char number[5];
        Chi_String fName;
        PAGE_NODE_PTR ptr;

        if (page)
        {
                update_seealso(module,page->left);

                if (!((page->rich) == close))
                {
                        if (page->seeAlso.compare(""))
                        {
                                ptr = search_page(page->seeAlso,page_display);
                                if (ptr != NULL)
                                {
                                        fName = dbl_u;
                                        fName.append(page->seeAlso);
                                        fName.append(close);
                                        p = page->rich.find(the_page,0);
                                        Chi_String hidden = hide;
                                        sprintf(number,"%3d\0",ptr->id);
                                        hidden.append(number);
                                        hidden.append(close);
                                        Chi_String func = short_title_open + deflang +
                                                newline+ "See Also: " + fName + hidden +
                                                newline;
                                        page->rich.insert(p-1,func);
                                }
                        }
                }
                update_seealso(module,page->right);
        }
}
```

FIGURE 11
RICH TEXT CONTROL CLASS DEVELOPED FOR CLOUD9

```
class TExampleRich : public TRich
{
  public:
       TExampleRich(TWindow*        parent,
                                       int              id,
                                       const char far* text,
                                       int x, int y, int w, int h,
                                       UINT             textLen = 0,
                                       BOOL             multiline = TRUE,
                                       TModule*         module = 0 )
            : TRich(parent, id, text, x, y, w, h, textLen, multiline, module)
      {
            _view_string_container = NULL;
                  //New constructor calls parent TEdiT (recall TRich is TEdit)
                  // constructor and sets new container string attribute to NULL
      }
      void SetupWindow();
      void   CmEditCut();                          // Standard TEdit Cut
      void   CmEditCopy();                         // Standard TEdit Copy
      void   CmEditPaste();                        // Standard TEdit Paste
      void   CmEditDelete();                       // Standard TEdit Delete
      void   CmEditClear();                        // Standard TEdit Clear
      void   CmEditUndo();                         // Standard TEdit Undo
      void   CmdBold();                            // New Rich Bold
      void   CmdItalic();                          // New Rich Italic
      void   CmdUnderline();                       // New Rich Underline
      void   CmdFontDialog();                      // New Rich Font Dialog Box
      void   CmdIncreaseFont();                    // New Rich Increase Font
      void   CmdDecreaseFont();                    // New Rich Decrease Font
      void   setHViewStrContainer(HViewString *); // New Rich command
                  // "sets" the string the control contains
     HViewString *  hViewStrContainer() const;    // New Rich command
                  // "gets" the string the control contains
      void    EvKeyDown(UINT key, UINT repeatCount, UINT flags);
                                                   //  Standard TEdit event key down
      void    EvLButtonDown(UINT modKeys, TPoint& point);
                                                   //  Standard TEdit event button
                                                   //   down
      void     NotifyParent(int notification);
                                                   //  Standard TEdit notify parent
                                                   //   Window of update
      void SaveText();                             // Saves the Rich Text into
                                                   //  an ASCII string --
                                                   //  formatting instructions
      void RestoreText();                          // Restores the Rich Text
                                                   //  from an ASCII string--
                                                   //  formatting instructions
      void RTF_ChangeCharAttribute(DWORD dwMask, DWORD dwEffects);
                                                   // Changes the attribute (italic,
                                                   //  underline, or bold) of
                                                   //  selected Rich Text
      void RTF_ChangeFont();                       // Changes the font of
                                                   //  selected Rich Text
      void RTF_ChangeSizeAttribute(int iPointChange);
                                                   // Change the size of
                                                   //  selected Rich Text
      HWND *parent;          //  Standard TEdit handle to parent Window
  private:
      HViewString *_view_string_container;        //  New ASCII string --
                                                   //   formatting instructions
  DECLARE_RESPONSE_TABLE(TExampleRich);
};
```

FIGURE 12

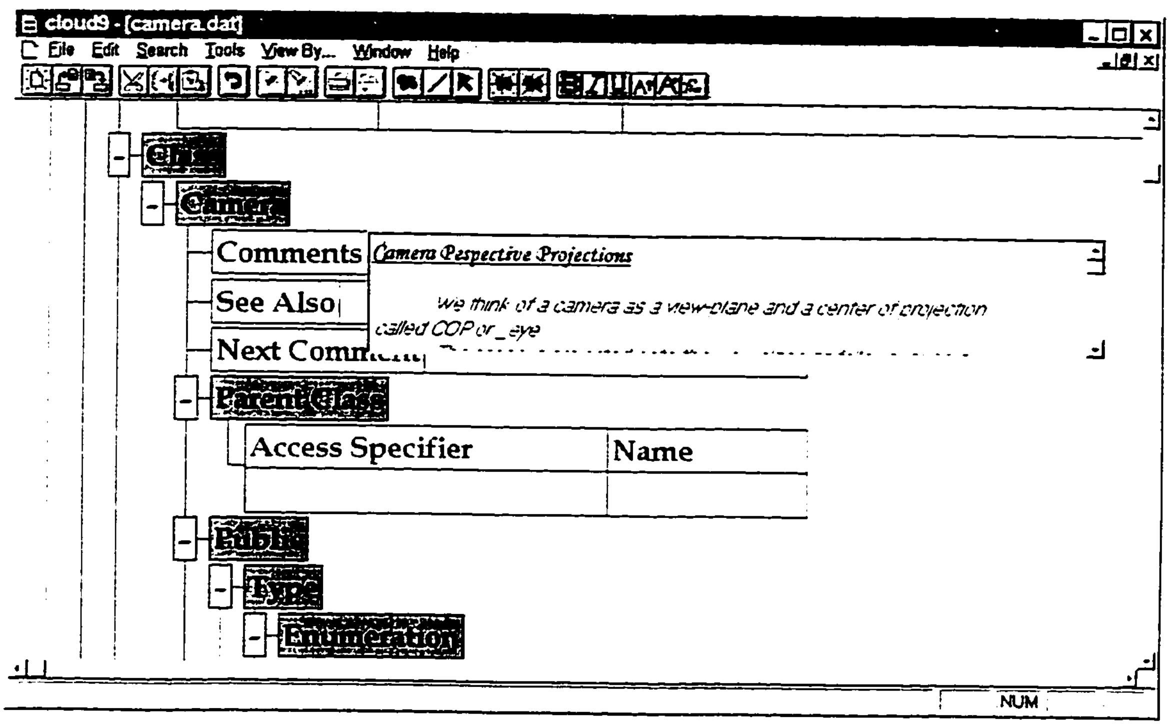
DOCUMENTING A CLASS
('+' is Rich Text Editor Activation Hot-Spot)



Access Specifier	Name

FIGURE 13

DOCUMENTING A CLASS - RICH TEXT EDITOR ACTIVE

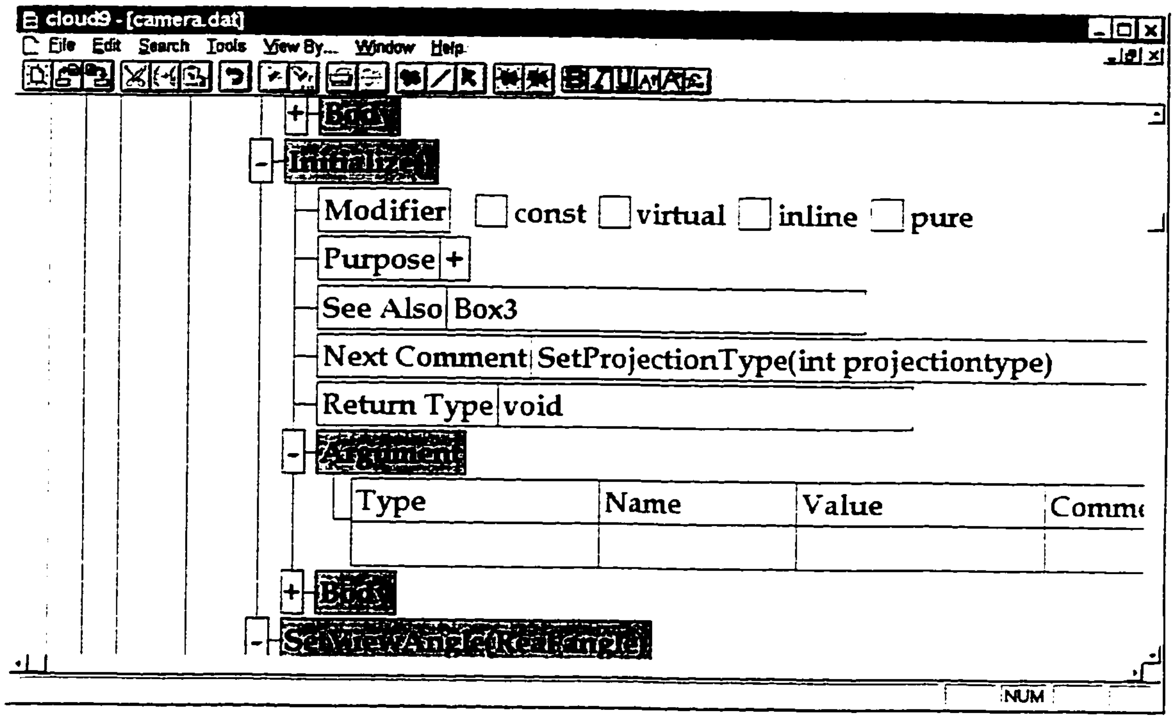


Camera Pespective Projections

we think of a camera as a view-plane and a center of projection called COP or _eye

FIGURE 14

DOCUMENTING A METHOD
('+' is Rich Text Editor Activation Hot-Spot)



Modifier [ ] const [ ] virtual [ ] inline [ ] pure

Purpose +

See Also Box3

Next Comment SetProjectionType(int projectiontype)

Return Type void

Type	Name	Value	Comm

FIGURE 15

RICH TEXT CONTROL USAGE IN C++ CODE

```
void HViewString::update(TEdit * edit)
{

        int length, num_lines, num_char;
        char *the_contents;
        Chi_String Label = getLabel();

        // we only do that for an existing edit control
        if (edit)
        {
                length = 0;
                num_lines = edit->GetNumLines();
                for (int i=0; i< num_lines; i++)
                        length += edit->GetLineLength(i); // length of the line

                TExampleRich *theCntrl = dynamic_cast<TExampleRich *>(edit);
                                             // convenient dynamic_cast

                 if (theCntrl == 0)  // Standard TEdit Contrl
                 {
                        the_contents = new char[length+1];
                        // fetch texts from edit control
                        edit->GetText(the_contents, length+1);
                        Chi_String contents(the_contents);
                        setContents(contents);
                  }
                  else  // New TExampleRich control
                  {
                                 theCntrl->SaveText();
                                             // calls the callback function and
                                             //  setContents() for the Rich
                                             //  control
                  }

                if (theCntrl == 0)
                        delete [] the_contents;

                Chi_String newContents = contents();    // retrieves set contents
                                                        //  for either control
```

FIGURE 15 (end of listing)

RICH TEXT CONTROL USAGE IN C++ CODE

```
        // Get the module, determine what type of node
        // has been modified, and modify that node
        HViewView* view = getView();
        Document& document = view->GetDocument();
        DOC_Module* module = document.getModule();
        HViewChild::class_type clss_type = getClassType();
        Chi_String clss = getClass();
        int parent_index = getParent()->getChildIndex();
        HViewChild::module_type mod_type = getModuleType();

        switch (_node_type)
        {
                case MODULE:
                        if (Label == "Path")
                                module->set_path(newContents);
                        else if (Label == "Comments")
                                module->set_comment(newContents);
                        else if (Label == "See Also")
                                module->set_seealso(newContents);
                        else if (Label == "Next Comment")
                                module->set_next(newContents);

                        break;
                case OPERATION:

                •

                •

                •

        }

        deActivate();
        edit->Destroy();

        if (theCntrl != 0)   // Special TExampleRich deletion resets
                             //  attributes necessary for Rich control
                             //  text saving
        {
                delete theCntrl;
                setRichLength(0);
                setRichPosition(0);
                setRichFile(NULL);
        }
    }
}
```

GLOSSARY

The following terms are taken directly from Cay Horstmann's book. Mastering Object-Oriented Design in C++[16]. except those with an *.

literate programming*

Changing the focus of programming from instructing the computer what we want it do to instructing a person(s) what we want the computer to do. Viewing programs as works of literature.

abstract class

A class without instance objects. An abstract class serves as a base class for other classes but is not specific enough to provide implementations for all operations.

accessor

A class operation that does not modify the object on which it is invoked but reports the value of the object (called selector herein*).

class

A collection of objects with the same operations and the same state range.

constructor

An operation that turns raw storage into an object by initializing the fields and bases.

design phase

The phase of a software project that concerns itself with the discovery of the structural components of the software system to be built. without concern for implementation details.

destructor

An operation that turns an object into raw storage, carrying out any actions that are necessary before the object is abandoned.

encapsulation

The act of hiding the implementation details of a class or module.

implementation phase

The phase of software development that concerns itself with realizing the design in a programming environment.

inheritance

The definition of a derived class as an extension of a base class. The derived class specifies how it differs from the base class and keeps all base class features that it does not redefine.

instance

An instance of class is an object of the class type (part omitted*).

instantiation

The process of making an instance.

member (data member*)

A feature of the class.

method (operation*, member function*)

A class operation.

GLOSSARY (continued)

mixin

Inheriting from one or more abstract classes to add a specific service or protocol to a class.

module

A collections of variables, constants, functions and types that have common functionality.

mutator

An operation that modifies the state of an object. A field mutator is a mutator modifying the value of a single field.

object

An entity in a programming system that has state, operations, and identity.

postcondition

A logical condition that an operation guarantees on completion.

polymorphism

Associating different features to a name, together with a mechanism for selecting the appropriate one.

precondition

A logical condition that the caller of an operation guarantees before making the call.

state

The current value of an object, which is determined by the cumulative action of all operations on it and influence the reaction to future operations.

GLOSSARY (continued)

subclass

A class that modifies another class (its base class) by adding fields and adding or redefining operations.

virtual operation

A family of operations that is specified in a base class and redefined in derived classes and can be dynamically bound in a call.

WEB*

A system of program documentation in which files are a combination of two different languages, a programming language (Pascal) and a documentation language.

REFERENCES

[1] G. Jones, *Software Engineering*. 1990. New York. New York: John Wiley & Sons., p. 287.

[2] P. Courtois. *On Time and Space Decomposition of Complex Structures*. June 1985. Communications of the ACM vol. 28(6), p. 596.

[3] G. Booch. *Object-Oriented Analysis and Design*. 1994. Redwood City. California: The Benjamin/Cummings Publishing Company. Inc., p. 12.

[4] H. Simon. *The Sciences of the Artificial*. 1982. Cambridge. MA: The MIT Press. p. 217.

[5] Ramamoorthy. C. and Sheu. P. Fall 1988. *Object Oriented Systems*. IEEE Expert vol. 3(3), p.14.

[6] J. Gall. *Systemantics: How Systems Really Work and How They Fail*. 1986. Second Edition. Ann Arbor. MI: The General Systemantics Press. p. 65.

[7] I. White. M. Goldberg, *Using the Booch Method: A Rational Approach*. 1994. Redwood City, California: The Benjamin/Cummings Publishing Company. Inc.

[8] G. Booch, p. 61.

[9] G. Booch, p. 65.

[10] G. Booch, p. 183.

[11] M. Shaw, *ALPHARD: Form and Content*. 1981. New York. NY: Springer-Verlag, p. 6.

[12] I. White, M. Goldberg, p. 6.

[13] I. Jacobson. *Object-Oriented Software Engineering, A Case Driven Approach*. 1992. Reading. MA: Addison-Wesley.

REFERENCES (Continued)

[14] W. Bauer, M. Juncosa, A. Perlis, "ACM Publication Policies and Plans". J. ACM 6 (Apr. 1959)., p.121-122.

[15] D. Knuth. *Literate Programming*, 1992. Stanford, California: Center for the Study of Language and Information, p. 7-8, 126.

[16] C. Horstmann. *Mastering Object-Oriented Design in C++*,1995. Brisbane, California: John Wiley and Sons, Inc., p. 441-445.

[17] C. Pokorny, C. Gerald, *Computer Graphics: The Principles Behind the Art and Science*, 1989. Franklin, Beedle & Associates, Irvine, CA.