Extra 1240 —>
entry

653

JA' ''VERSITY

OBJECT ORIENTED GRAPHICAL USER INTERFACE FOR HOTEL AUTOMATION SYSTEM

Dissertation submitted to the Jawaharlal Nehru University
in partial fulfilment of the requirements
for the award of the Degree of

MASTER OF TECHNOLOGY
(COMPUTER SCIENCE & TECHNOLOGY)

by
R. VIJAYASARATHI

SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110067
INDIA

CERTIFICATE

This is to certify that the thesis titled **Object-Oriented Graphical User Interface for Hotel Automation System**, being submitted by **Mr. R. Vijayasarathi** to the **Jawaharlal Nehru University**, New Delhi in partial fulfilment of the requirements for the award of the degree **Master of Technology (Computer Science & Technology)**, is a record of the original work done by him under the supervision of **Prof. K. K. Nambiar**, Professor, School of Computer & Systems Sciences, Jawaharlal Nehru University, New Delhi, during the Monsoon Semester, 1994.

The results reported in this thesis have not been submitted in part or full to any other University or Institution for the award of any degree etc.

[illegible] 16-1-95

Prof. K. K. Bharadwaj,
Dean, SC&SS,
J. N. U.,
New Delhi-110067

[illegible]

Prof. K. K. Nambiar,
Professor, SC&SS,
J. N. U.,
New Delhi-110067

ACKNOWLEDGEMENTS

It is a great pleasure for me to sincerely express my deep sense of respect and gratitude to **Prof. K. K. Nambiar** for his invaluable guidance and inspiration. His knowledge, scholastic experience and remarkable patience led me throughout the project, leading to the successful completion of the same. It would have been impossible for me to come out successfully without his constant guidance and constructive criticism.

I extend my thanks to **Prof. K. K. Bharadwaj**, Dean, for providing me the opportunity to undertake the project.

A friend in need is a friend indeed. I thank all my friends, especially Diana and John for the moral support provided throughout. Thanks are also due for all those who helped me directly or indirectly in completing the project.

There are others whose positions preclude them from being officially acknowledged for their help. Their unstinting assistance has been priceless.

R. VIJAYASARATHI

ABSTRACT

An attractive "look and feel" in a software product's user interface is as important to its success as its functionality. Graphical User Interfaces started a new era in man-machine communication. Microsoft Windows epitomized the concept of Graphical User Interface with its intuitive and user-friendly approach.

With the advant of object-oriented technology, Object-Oriented Programming(OOP) became the natural choice for the implementation of user interface systems. Object-Oriented Programming allows the mapping of visual objects on the display, directly to conceptual objects in the software, providing consistency and simplification.

Borland's Object Windows Library(OWL) came to the rescue of the Windows programmer, encapsulating the complexities of Windows programming. Object Windows thus provides a flexible and simplified application frame work.

Design and implementation of an Object Oriented Graphical User Interface for Hotel Automation System is discussed in this thesis. The Hotel Automation System is meant to automate the functioning of a Star Hotel for fast and efficient functioning. It provides various services such as making an enquiry about the availability of rooms for reservation etc. It also provides a flexible billing and menu system.

CONTENTS

Chapter 1

INTRODUCTION

Computer technology enables people to interact with enormous amounts of data using large numbers of functions. The user's controlling commands and computer's responses constitute a user interface, an exchange of information between the user and the computer. As the state of computing matured, users wanted the interface to be more intuitive, visually appealing and simple to use, and still meet the entire range of problem solving requirements. The significant improvement of computer hardware and software technology, high resolution bitmap graphics and pointing devices such as mouse, made it possible the evolution of modern Graphical User Interfaces (GUIs).

Microsoft Windows provides an excellent Graphical User Interface for DOS, with its capabilities like device independence, multitasking, and Graphics Device Interface(GDI). Microsoft Windows supports an Application Programming Interface(API) that allows programmers to create GUI applications by providing a powerful library of more than 600 function to:

* Create and draw screen objects like windows, bitmaps, and dialogboxes.
* Monitor and process mouse and other keyboard activations.

With the advent of Object-Oriented technology, object-oriented programming became the natural choice for the implementation of user interface systems. Rather than separate declarations of data structures and functions that operate on them, we have, integrated objects which maintain their own state and provide a set of applicable operations. Object-oriented programming allows the mapping of visual objects on the display (like windows, bitmaps) directly to conceptual objects in the software, providing consistency and simplification.

Borland's Object Windows Library (OWL) provides a programming interface to Windows, taking away the burden of window management and message processing from the programmer. Object Windows simplifies the development of Windows applications by encapsulating the behaviors that Windows applications commonly perform. Object Windows uses the object-oriented features of C++ to hide parts of Windows API, insulating the programmer from the internals of Windows programming. As a result, Windows applications can be developed with much less time and effort.

In this thesis, the development of a prototype of an object-oriented Graphical User Interface (GUI) for Hotel Automation System is discussed in detail. The Hotel Automation System is meant to automate the activities of a Star Hotel for fast and efficient functioning. This software system allows for making
enquiries about availability of rooms, as well as making reservation of rooms. It provides a flexible billing and menu system. It allows for general enquiry about a guest. It also provides other miscellaneous functions that are generally performed, such as check-in/check-out of guests. The various operations performed by the Hotel Automation System are listed below:

1. Room Reservation Enquiry
2. Room Reservation
3. Cancellation of a Reservation
4. Food Menu
5. Bar Menu
6. Prepare Bill

7. Update Date
8. Guest Enquiry
9. Check-In
10. Check-Out
11. Change Availability of Food/Drink

Chapter 2 introduces the various concepts involved in the development of this system. In this, the evolution and importance of Graphical User Interfaces (GUIs) and Object-Orientation concepts are discussed. This chapter also emphasizes the Microsoft Windows programming and introduces Borland's Object
Windows Library (OWL), providing the details of Object Windows class hierarchy.

Chapter 3 discusses the design and implementation of the prototype of the object-oriented GUI for Hotel Automation System. It gives the design and implementation issues of the GUI and hotel database separately, discussing various design considerations.

Chapter 4 gives the "look and feel" of the product developed by providing some snapshots from the execution of the program in a sample session.

Time constraints forced the project to confine to a prototype Hotel Automation System instead of a real world system. Chapter 5 discusses the possible enhancements and further extensions that can be made to this software system.

Chapter 6 gives a partial listing of the program.

References used in the project are provided at the end of the report.

Chapter 2

PROGRAMMING CONCEPTS

2. 1 Graphical User Interfaces

As the state of the computing matured, users wanted programs that were visually more appealing and simple to use. The user's inputs, commands and the computer's responses constitute a user interface. In the case of earlier interactive computer, the primary means of interaction with computer has been through command-based interface. This interface was rather awkward and the user has to memorize a large set of commands to use the system and run an application.

The purpose of the user interface is to facilitate user-computer communication by enveloping hardware and software in a dialogue. Users often prefer simpler sets of functions with good user interfaces to large sets of functions with awkward user interfaces. An attractive look and feel in a product's user interface can be as important to its success as its functionality. User interfaces were originally described by relatively small amounts of code embedded in the application. As people recognized the importance of the user interface, the code became specialized and separate. User interfaces are now become so important that for some applications, the amount of code devoted to the user interface may exceed that for the application.

The properties of a good user interface include the following:

* **User control.** Users can easily use the program and should feel that they directly control the program.

* **Predictability.** User interfaces should be as consistent as possible, so that the user would feel less unfamiliarity while using the application, for example, a pop-up menu should always appear at the same place relative to the cursor.

* **Economy of expression.** User interfaces should facilitate concise expression of user's wants, needs, and actions and should respond appropriately.

The user's ability to learn an interface is crucial to its acceptance. It must let the user accomplish tasks in the most straight forward way possible and still meet the entire range of problem solving requirements.

Object-oriented design is natural for user interface systems. Visual objects on the display (like windows, and bitmaps) map to conceptual objects in software. Implementing a user interface in a language which supports object-oriented programing provides consistency and simplification. Rather than separate declarations of data structures and functions that operate on them, we have integrated objects which maintain their own state and provide a set of applicable operations.

Another advantage of the object-oriented approach for user interface design is the clear decomposition of structure and
functionality, which makes it easier for a developer to design, maintain, and reuse components. Since object-oriented systems support inheritance, new components can be specialized or extended from existing ones. This makes object-oriented interfaces well suited for rapid prototyping.

Significant hardware improvements like powerful dedicated processors, high resolution bitmap graphics, and pointing devices such as mouse, facilitated the evolution of modern Graphical User Interfaces (GUIs). GUIs are systems that allow creation and manipulation of user interfaces employing graphical objects such as windows, icons, bitmaps, buttons and menus. GUIs not only do away with tedious task of data preparation, but also hide the user from the usually character based operating system of the computer. Another useful feature of GUIs is they are so easy to use that even novices can learn to drive the systems and their individual applications without any manual training.

Smalltalk MVC, Apple Macintosh Toolbox, Microsoft Windows, X-Windows, Sun Microsystem's NEWS, DEC's DECWindows are some of the examples of GUIs.

One of the most important aspects of GUIs is that they all have programmer's interfaces, called Software Development Kits (SDK) which allow independent software developers to create applications which 'look and feel' like the main GUI. The Software Development Kits provide an Application Programming Interface (API) which enables users to:

* Create and draw screen objects like windows, bitmaps and dialogboxes.

* Monitor and process mouse and other keyboard activations.

2. 2 Object-Orientation Concepts

The distinguished characteristic of industrial-strength software is that it is intensely difficult, if not impossible, for the individual developer to comprehend all the subtleties of its design. The complexity of such system often exceeds the human intellectual capability. We observe that this inherent complexity derives from four elements: the complexity of the problem domain, the difficulty of managing the developmental process, the flexibility possible through software, and the problems of characterizing the behavior of discrete systems.

Our failure to master the complexity of software results in projects that are late, over budget, and deficient in their stated requirements. We often call this condition 'software crisis'.

As Dijkstra suggests, "The technique of mastering complexity has been known since ancient times: divide et impera (divide and rule)"[4]. When designing a complex software system, it is essential to decompose it into smaller and smaller parts, each of which may then be refined independently.

Algorithmic Decomposition: Most of us have been formally trained in the dogma of top-down structured design, and so we approach decomposition as a simple matter of algorithmic decomposition, wherein each module in the system denotes a major step in some overall process.

Object-Oriented Decomposition: Another possible decomposition strategy decomposes the system according to the key abstractions in the problem domain. Rather than decomposing the problem into substeps, objects have been identified, which derive directly from the problem domain. From this perspective, an object is simply a tangible entity which exhibits some well defined behavior. Since the decomposition is based upon objects and not algorithms, it is called an 'object-oriented decomposition'.

Object-oriented decomposition has a number of highly significant advantages over algorithmic decomposition. Object-oriented decomposition yields smaller systems through the reuse of common mechanisms, thus providing an important economy of expression. Object-oriented systems are also more resilient to change and thus better able to evolve over time, because their design is based upon stable intermediate forms. Indeed, object-oriented decomposition greatly reduces the risk of building complex software systems, because they are designed to evolve incrementally from smaller systems in which we already have confidence.

As structured design methods evolved to guide developers who were trying to build complex systems using algorithms as their fundamental building blocks, Object-oriented design methods have evolved to help developers exploit the expressive power of object-oriented programming languages, using the class and object as basic building blocks.

Object-oriented programming employs object-oriented decomposition. Object-oriented programming is a "method of implementation in which programs are organized as co-operative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships". This definition implies that object-oriented programming uses 'objects'; each object is an instance of some 'class' ; classes are related to one another via 'inheritance' relationships. Under this definition, smalltalk, Object Pascal, C++ and CLOS are all object-oriented programming languages.

2. 2. 1 Object

The fundamental idea behind object-oriented languages is to combine both 'data' and the 'functions that operate on that data' into a single unit, called 'object'. An object's functions called 'member functions' or 'methods' provide the only way to access its private data.

In other words, an object is a tangible entity that exhibits some well defined behavior. An object has state, behavior and identity.

The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties. A property is an inherent or distinctive characteristic, quality, or feature that contributes to making an object uniquely that object.

The behavior of an object is how an object acts and reacts in terms of its state changes and message passing. An operation is some action that one object performs upon another in order to elicit a reaction. In object-oriented programming languages, operations performed on an object are typically declared as 'methods' or 'member functions', which are part of the declaration of the class of the object.

The identity of an object is that property which distinguishes it from all other objects. Since an object is distinguished from all others, the identity of each object is preserved even when the state of that object is completely changed.

2. 2. 2 Class

The concepts of a class and an object are so interrelated that we cannot talk about an object without regard for its class. However, there are important differences between these two terms. Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the "essense" of an object, as it were. In other words, "A class is a set of objects that share a common structure and a common behavior". A single object is simply an instance of a class.

Consider the following C++ class declaration for a 'Stack' class.

```
class Stack
 {
  private:
    int st[MAX];     // stack: array of integers
  protected:
```

```
  int top;          // top of the stack
public:
  Stack();          // constructor
  void push(int x)
   {
    if(top < MAX)
     st[top++] = x;
    else
     {
      cout<<"\nError: Stack is full";
      exit(1);
     }
   }
  int pop()
   {
    if(top > 0)
     return st[top--];
    else
     cout<<"\nError: Stack is empty";
   }
};
```

The above declaration of the 'Stack' class consists of three parts.

* Public : A declaration that is visible to all clients that are visible to it

* Protected : A declaration which is not visible to any other client except its sub-classes.

* Private : A declaration which is not visible to any other class; accessible only to its member functions.

Relationships Between Classes and Objects:

Classes and objects are separate yet intimately related concepts. Specifically, every object is an instance of some class, and every class has zero or more instances (objects). Practically for all applications, classes are static; therefore, their existence, semantics, and relationships are fixed prior to the execution of a program. Similarly, the class of most objects is static, meaning that once an object is created, its class is fixed. In sharp contrast, however, objects are typically created and destroyed at a furious rate during the lifetime of an application.

2. 2. 3 Inheritance

Inheritance is one of the powerful features of object-oriented programming. Inheritance is a relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. In other words, it is the process by which one class can acquire the properties of another class. It allows the creation of new classes, called 'derived classes' from the existing or 'base classes'. The derived class not only inherits all the capabilities of the base class, but can add additional features and refinements of its own, leaving the base class unchanged. Inheritance defines a kind of hierarchy among classes in which a derived class typically augments or redefines the existing structure and behavior of its base classes.

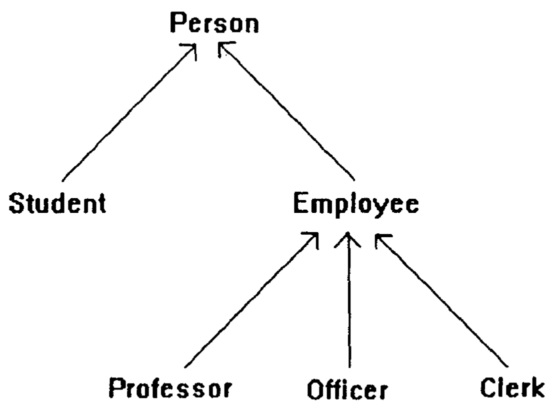


Fig. 2. 1 Class hierarchy for a University example.

The class hierarchy of a University example is shown in Fig. 2. 1. Both **Student** and **Employee** are persons. They inherit the characteristics of **Person**. Both **Student** and **Employee** *specialize* the properties of persons, and conversely **Person** *generalizes* the properties of both **Student** and **Employee**. Similarly, **Professor**, **Officer** and **Clerk** inherit the common properties from **Employee** and would specialize the employee characteristics. The arrows in the class hierarchy or inheritance diagram are directed from derived classes to base classes.

Inheritance lets the programmers reuse code and redefine its application within the current environment. Reusing existing code saves time and money and increases the reliability of software. It is the key to building maintainable, reusable systems, and it provides a form of configuration management.

2. 2. 4 Abstraction

An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer. An abstraction focuses on the outside view of an object, and so allows to separate an object's essential behavior from its implementation. The different kinds of abstractions include:

* **Entity abstraction** : An object that represents a useful model of a problem-domain entity.

* **Action abstraction** : An object that provides a generalized set of operations, all of which perform the same kind of fuction.

* **Virtual machine abstraction:** An object that groups together operations that are all used by some superior level of control, or operations that all use some junior-level set of operations.

* **Coincidental abstraction** : An object that packages a set of operations that have no relation to each other.

The different kinds of abstractions above are listed in the decreasing order of usefulness. We strive to build entity abstractions because they directly parallel the vocabulary of a given problem domain.

2. 2. 5 Encapsulation

Abstraction and encapsulation are complementary concepts: abstraction focuses upon the outside view of an object and encapsulation, also known as 'information hiding', prevents users from looking its inside view, where the behaviour of the abstraction is implemented. In this manner, encapsulation provides explicit barriers among different abstractions. For example, in designing a database application, it is standard practice to write programs so that they don't depend on the physical representation of data, but depend only upon a scheme that denotes the data's logical view. Hence, objects at higher levels of abstraction are shielded from lower level implementation details.

Whereas abstraction "helps people to think about what they are doing", encapsulation "allows program changes to be reliably made with limited effort".

In other words, Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics. For abstraction to work, implementations much be encapsulated. In practice, this means that each class must have two parts: an interface and an implementation. The interface of a class captures only its outside view, encompassing our abstraction of the behavior common to all instances of the class. The implementation of a class comprises the representation of the abstraction as well as the mechanisms that achieve the desired behavior. This explicit division of interface/implementation represents a clear separation of concerns: the interface of a class is the only place where we assert all of the assumptions that a user may make about any instance of the class; the implementation encapsulates details about which no user may make assumptions.

2. 3 The Windows Concepts

2. 3. 1 The Windows Paradigm

In a traditional data-processing or task-oriented application, the application itself performs a series of tasks, under its own control. This model works well for non-interactive applications, but it is not very well suited to interactive applications such as user-interfaces. For an interactive application, a programming environment that reacts in a natural way to the user's actions, is needed. If the user clicks the mouse to push a button in a dialog, the necessary action should be performed. Instead if the user closes a window, some other necessary task should be performed. The main advantage of event-driven programming is that it addresses this need.

An event-driven program has an orientation that is different from a traditional program. In an event-driven program, the program waits for a command. If the user gives any message (by clicking the mouse, for example) the program sends a message that describes that event, to the application. The application wakes up, receives the message, processes the message, and then waits for another message.

Another advantage of using the event-driven programming model is that it helps the program manage several independent applications. After an application finishes processing a message, the program can send the next message to a different application, making the user feel as if more than one application is running simultaneously.

Microsoft Windows is a Graphical User Interface (GUI) built on DOS, employing the event-driven programming. Microsoft Windows provides an intuitive and easy to use GUI for DOS, using windows, icons, bitmaps, buttons, dialog boxes etc.

Microsoft Windows offers many benefits to the user. They include:

* *Provides an intuitive interface.* If you know how to use one Windows application, you know how to use them all.
* *No need to set up devices and drivers for each application.* Windows provides drivers to support various vendors' peripherals.
* *Multitasking.* You can use many application simultaneously by opening any number of overlapped windows.
* *Data interchange between different windows applications.* You can transfer data between your application and Clipboard etc.
* *Access to more memory:* Windows runs in 386 Enhanced mode on 80386 processor to provide the virtual memory capabilities. With this, applications have access to more memory than is physically available on the system.
* *Device-independent graphics.* So graphical applications run on all standard display adapters.

Microsoft Windows supports an Application Programming Interface (API) that allows programmers to create Windows applications by providing a powerful library of more than 600 functions to:

* Create and draw screen objects like windows, bitmaps, and dialog boxes.

* Monitor and process mouse and other keyboard activations.

2. 3. 1 The OWL Paradigm

The Object Windows Library of Borland C++ doesn't fundamentally alter Windows' event driven paradigm, but it clearly provides a much more convenient and trouble free way to use the event-driven paradigm. Object Windows simplifies the process, allowing the programmer to focus on the application's function, rather than its form. Object Windows uses the object-oriented features of Borland C++ to encapsulate parts of the Windows API, insulating us from the internals of Windows programming. As a result, we can develop Windows programs with much less time and effort.

Object Windows let us use objects to represent the fairly complex elements of a Windows program. Its window objects encapsulate data that all windows require, perform common window operations, and respond to common Windows messages and events. Object Windows' window and application classes completely manage the processing of messages, which normally comprises the bulk of a Windows application.

Specifically, Object Windows provides the following advantages:

Advantages of object-oriented programming.

Object-oriented programming is natural for user interface systems. Visual objects on the display map to conceptual objects in the software. C++ provides much stronger type and usage-checking than C, which is a tremendous boon in Windows programming.

* *Encapsulating Window information.*

Object Windows supplies classes that define behavior and data storage for the windows, dialog boxes, and controls of Windows applications. There is a TWindow class for working with windows, a TDialog class for creating dialog boxes, various other classes for the controls, and so on.

These classes often take care of many details automatically, and they can all be used as base classes, which makes it easy to create specialized behavior.

* *Abstraction of many Windows API functions.*

Object Windows simplifies the calling of Windows API functions by offering object member functions that abstract many of the function calls. As many of the parameter for Windows API functions are already stored in the data members of interface objects, (such as windows, dialog boxes, and controls), the member functions can use this data to supply Windows functions with parameters. In addition, Object Windows groups related function calls into single member functions that perform higher level tasks. This results in a streamlined, easier-to-use interface to Windows.

While this approach greatly reduces the dependence on the hundreds of Windows API functions, it does not restrict us from calling the API functions directly.

* *Automatic message response.*

Object Windows lets us associate class member functions with individual messages. Such member functions are called 'message response functions'. This eliminates the need for huge, unwieldy switch statements and it provides a far cleaner way to manage messages.

2. 3. 3 Object Windows class hierarchy

Object Windows is a comprehensive set of classes that simplifies the development of Windows programs with Borland C++. It is a library consisting of a hierarchy of classes that can be used, modified, or added to, using inheritance. The class hierarchy of Object Windows is shown in Fig. 2.2.

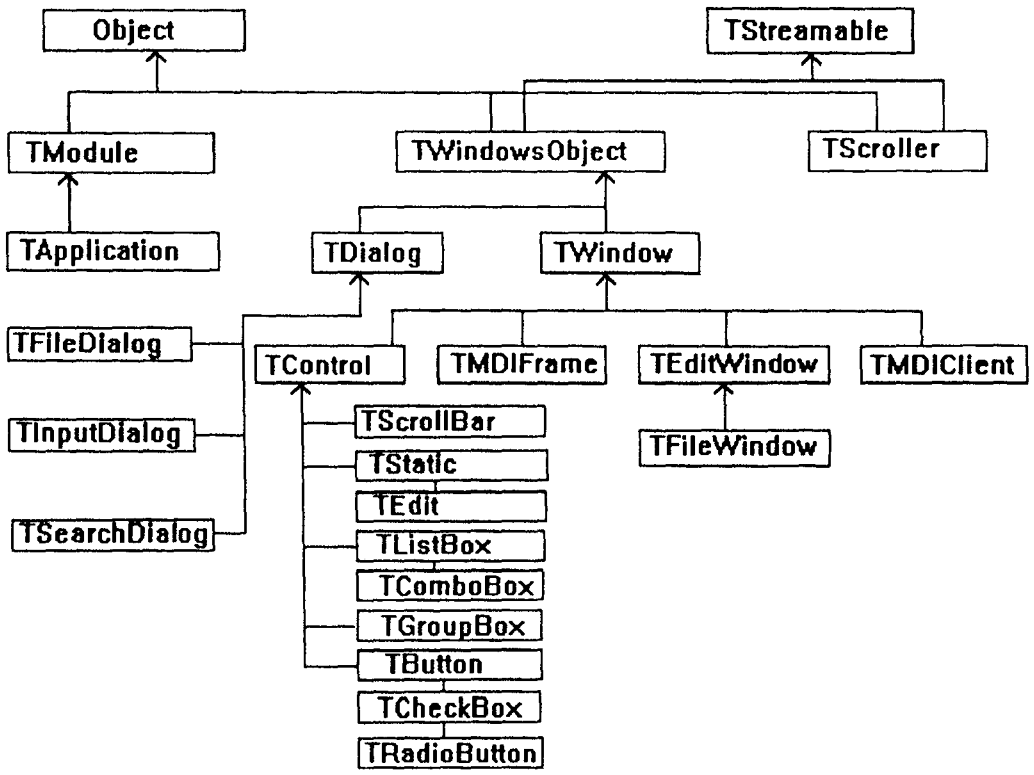


Fig. 2. 2 Object Windows Class Hierarchy

Object is the base class for all Object Windows derived class and is defined in the Borland C++ container class library. **TApplication** defines the behavior required of all Object Windows applications, and is derived from **TModule**. **TModule** defines behavior shared by both dynamic-link libraries (DLLs) and application modules.

The remaining objects in the Object Windows hierarchy are generally termed as interface objects. They are interface objects in the sense that they represent elements in the Windows user interface, and because they serve as a kind of interface between the application code and the Windows environment.

TWindowsObject is a base class that unifies the three main types of Object Windows interface objects: Windows, dialog boxes, and controls. It provides member functions to handle the creation, message processing and destruction of window objects.

Window Objects:

Window objects represent not only the familiar windows of the windows environment, but also most of the visual elements within that environment, such as controls. **TWindow** is a general-purpose window class whose instances can represent main, pop-up or child windows of an application. **TEditWindow** is derived from **TWindow** and defines a class that allows text editing in a window. **TFileWindow** which is also derived from **TWindow**, defines a class that allows text editing in a window, but can also load and save text files.

Dialog Objects:

Dialog objects represent interactive dialog boxes containing controls such as buttons, list boxes, and scroll bars. **TDialog** class serves as a base for derived classes that manage Windows dialog boxes. **TFileDialog** is derived from **TDialog** and defines a dialog that allows user to choose a file for any purpose, such as opening or editing. **TInputDialog** is also derived from **TDialog** and defines a dialog box for user input of a single text item.

Control Objects:

Within windows and dialogs, controls allow users to enter data and select options. Control objects provide a consistent and simple means of dealing with all the different kinds of controls defined by Windows.

TControl is an abstract class that serves as a common base class for all control objects. **TButton** defines Windows push buttons. **TCheckBox**, derived from **TButton**, defines Windows check boxes and provide member functions to manage their state. **TRadioButton** derived from **TCheckBox**, defines the creation and state management for Windows radio buttons. **TListBox** class handles creation of and selection from Windows list boxes, and defines member functions to manipulate items in a list. **TComboBox**, derived from **TListBox** defines behavior for Windows combo boxes. **TGroupBox** defines Windows group boxes. **TStatic** provides member functions that set, query, and clear the text of a static control. **TEdit,** derived from **TStatic** provides the extensive text processing capabilities for a windows edit control. **TScrollBar** defines member functions that manage the range and thumb position of a standalone scroll bar control.

MDI Objects:

Windows implements a standard for handling multiple windows within the framework of a single window. This standard is called the 'Multiple Document Interface (MDI)'. Object Windows provides a means of setting up and manipulating MDI windows.

TMDIFrame provides the windowing behavior appropriate for the main window of an application that follows the Windows MDI specification. **TMDIClient** provides additional support for MDI windows. The MDIClient object is the object that actually manages the MDI window's client area.

Scroller Objects:

TScroller is the object that gives life to Windows scroll bars, providing an automated way to scroll the text and graphics in windows. **TScroller** also scrolls its owner when the user drags the mouse from the inside to the outside of a window's client area; therefore **TScroller** works for windows that don't even have scroll bars.

Chapter 3

DESIGN AND IMPLEMENTATION

The design and implementation of the Object-Oriented GUI for Hotel Automation System can be divided into two parts: design and implementation of hotel database and user interface. The user interface allows the user to access the database through various predefined operations.

3. 1 Hotel Database Design

The hotel database has been designed as a relational database. The database consists of five relations **Room**, **Guest**, **Restaurant**, **Reservation** and **Bill**. The scheme and details of the five relations are given below:

* <u>Room</u>

The relation **Room** is defined on the following scheme.

Room = (RoomNumber, Type, Status, Rent)

The attribute 'RoomNumber' is the primary key of the **Room** relation. The domain of 'RoomNumber' is {1, ..., 200}. The attribute 'Type' denotes the type of the room. Its domain is {SINGLE NON_AC, SINGLE AC, DOUBLE NON_AC, DOUBLE AC}. The 'Status' attribute denotes the status of the room and its domain is {RESERVED, UNRESERVED}. 'Rent' attribute indicates the unit rent of the room per day, and its domain is the set of real numbers.

* <u>Guest</u>

The relation **Guest** is defined on the following scheme.

Guest = (Name, Sex, Age, Address)

The attributes 'Name', 'Sex', 'Age' and 'Address' denote the guest's name, sex, age and address respectively. The domain of 'Name' and 'Address' is any character string of length NAMELEN and ADDRESSLEN respectively, defined in the header file. 'Sex' can take either 'M' or 'F' and the domain of age is {1, ..., 100}.

* __Restaurant__

The **Restaurant** relation is defined on the following scheme.

Restaurant = (Item, Rate, Availability)

The attributes 'Item', 'Rate' and 'Availability' denote the name, unit rate, and availability of items, respectively in the restaurant. The domain of 'Item' is the set of character strings of length ITEMLEN, defined in the header file. The domain of 'Rate' is the set of real number and the domain of 'Availability is {YES, NO}.

* __Reservation__

The relation **Reservation** is defined on the following scheme.

Reservation = (RoomNumber, Name, From, To)

To attribute 'RoomNumber' can take any value from the domain {1, ..., 200}. The attribute 'Name' denote the name of the customer on whose name the reservation is made for and its domain is the set of character strings of length NAMELEN, defined in the header file. The attributes 'From' and 'To' denote the dates from which date to which date the room is reserved for (inclusive of both the dates). The domain of 'From' and 'To' is the set of character strings of the form 'DD/MM/YEAR', whose length is DATELEN, as defined in the header file.

* __Bill__

The **Bill** relation is defined on the following scheme.

Bill = (Name, RoomNumber, Bill)

The attribute 'Name' denotes the name of the customer and its domain is the set of character strings of length NAMELEN, defined in the header file. The domain of 'RoomNumber' is {1, ..., 200}. The attribute 'Bill' indicates the total bill added to the customer's account. The domain of 'Bill' is the set of real numbers.

3. 2 Hotel Database Implementation

The relations **Room**, **Guest**, **Restaurant**, **Reservation** and **Bill** have been stored in the files room.$$$, guest.$$$, restaurant.$$$ and bill.$$$ respectively.

Each of the five relations have been implemented using the data structure 'B-tree'. Instead of developing routines for B-tree implementation, the Borland C++ Container Class Library's **Btree** class has been used. In the implementation of the database, the Container Class Library's **Date** and **String** classes have been used to represent dates and character strings respectively. The class hierarchies in the Borland C++ **Object**-based Container Class Library is shown in Fig. 3. 1.

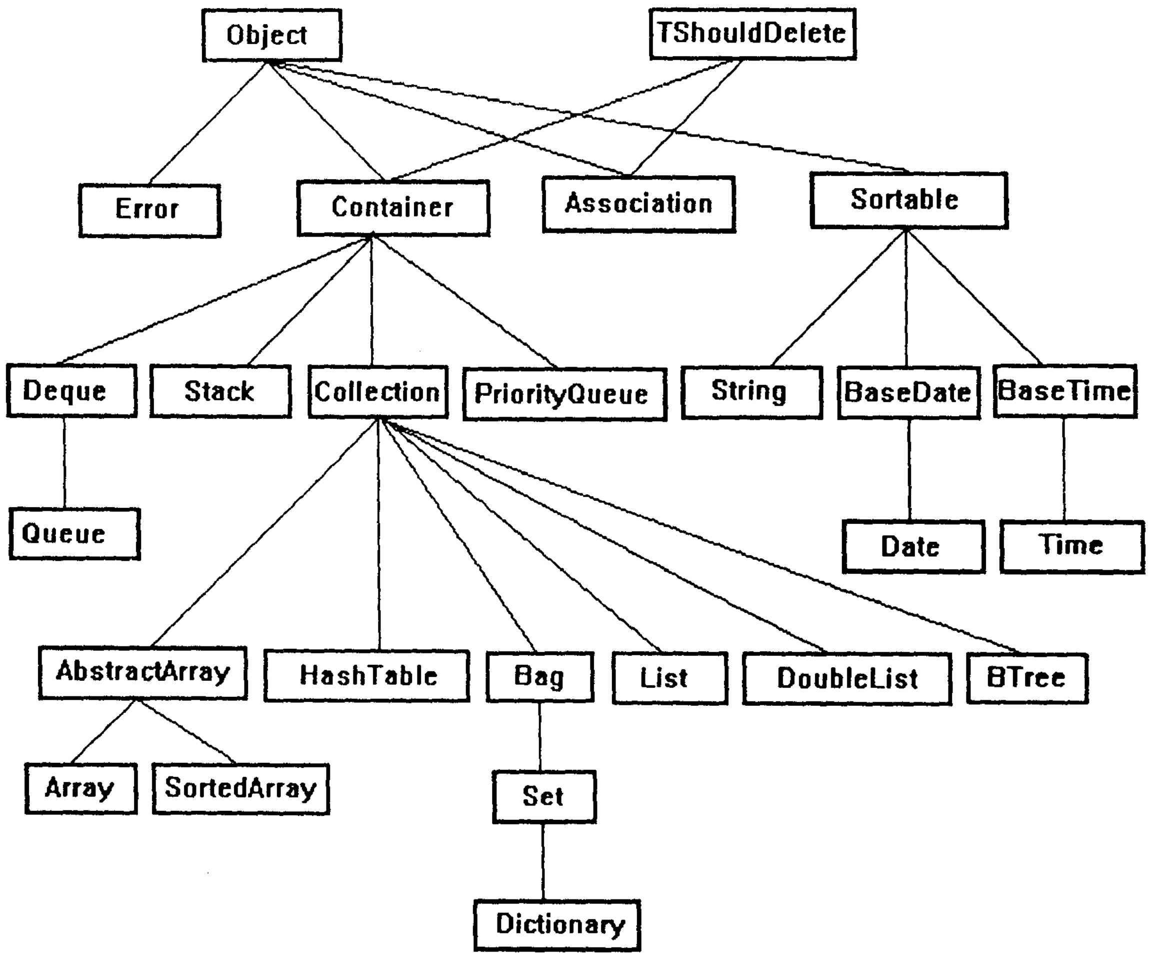


Fig. 3.1 Class hierarchies in Borland C++ Object-based Container Class Library.
(Source: Borland C++ 3.1 Programmer's Guide)

BRoom, **BGuest**, **BRestaurant**, **BReservation** and **BBill** objects are declared as **Btree** objects. **BRoom** B-tree stores the ***Room*** relation (RoomNumber, Type, Status, Rent). **BGuest** B-tree stores the ***Guest*** relation (Name, Sex, Age, Address). **BRestaurant** B-tree stores the ***Restaurant*** relation (Item, Rate, Availability). **BReservation** B-tree stores the ***Reservation*** relation (RoomNumber, Name, From, To). **BBill** B-tree stores the ***Bill*** relation (Name, RoomNumber, Bill). 'RoomEntry', 'GuestEntry', 'RestaurantEntry', 'ReservationEntry', and 'BillEntry' classes which are all derived from the **Sortable** class, are created to hold the tuples of the relations ***Room***, ***Guest***, ***Restaurant***, ***Reservation***, and ***Bill*** respectively. The inheritance diagram of the relations is shown in Fig 3.2.

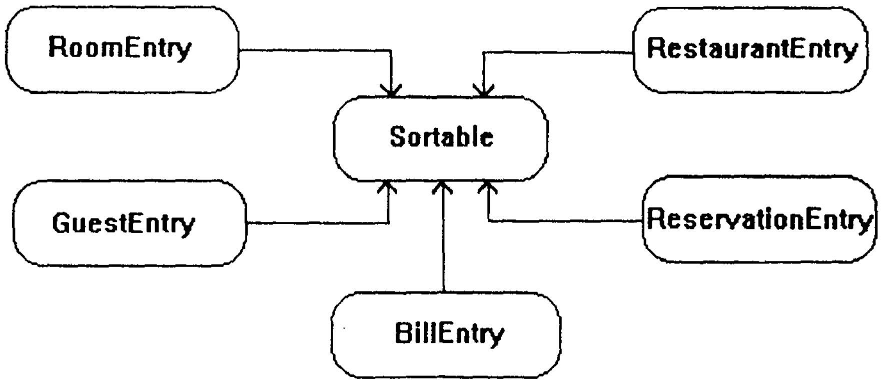


Fig. 3.2 Class Inheritance Diagram for the classes RoomEntry, GuestEntry, RestaurantEntry, ReservationEntry, BillEntry

Initially, when the application is started, the relations stored in the files room.$$$, guest.$$$, restaurant.$$$, and bill.$$$ are used to construct the corresponding B-trees. All the processing of the database is done on the B-trees. At the end of the session, when the application is about to be closed, all the data in the B-trees is stored back into the corresponding files.

3. 3 User Interface Design:

The user interface system interacts with both the database and the user as shown in the following Fig. 3. 3.

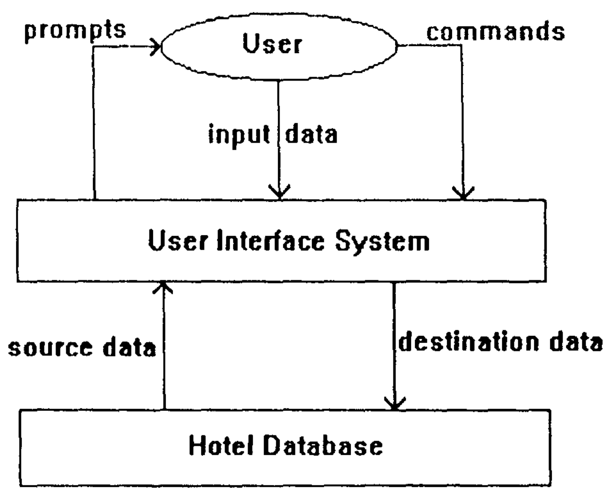


Fig. 3. 3 Interaction of user interface with user and database.

The user accesses the database through the user interface. When the user gives commands to the interface system, it prompts the user for the input and the user responds by giving the input data. The user interface system receives the user input and retrieves the source data from the database and writes the updated data back to the database, after processing. The user interface has been designed keeping in mind, the facilities offered by Microsoft Windows and Borland's Object Windows Library (OWL), such as windows, menus and dialog boxes. It is designed such that the information flow across the user interface is minimized.

The user interface is designed as a menu driven system. The menu structure is described in Chapter 4 SAMPLE SESSION.

3. 4 User Interface Implementation

The user interface has been implemented in Borland C++ 3.1 using Object Windows Library (OWL) in MS-Windows 3.1 environment.

Object Windows greately simplified the implementation of the user interface. The capabilities of MS-Windows and Object Windows Library have been widely used in the implementation in order to reduce the programming effort.

The application object 'HotelApp' has been derived from the **TApplication** class. The application's main window 'HotelMainWindow' has been derived from the **TWindow** class. The dialog box objects for each of the menu items 'TReserveEnqDialog', 'TReserveDialog', TCancelDialog', 'TFoodMenuDialog', 'TBarMenuDialog', 'TCheckInDialog', 'TCheckOutDialog', 'TPrepareBillDialog', 'TChangeAvailDialog' have all been derived from **TDialog** class. For the menu items *GuestEnquiry* and *UpdateDate*, instead of defining separate classes, the **TInputDialog** class has been used. The inheritance diagrams for each of the above classes have been showed in Figures 3. 4 through y.

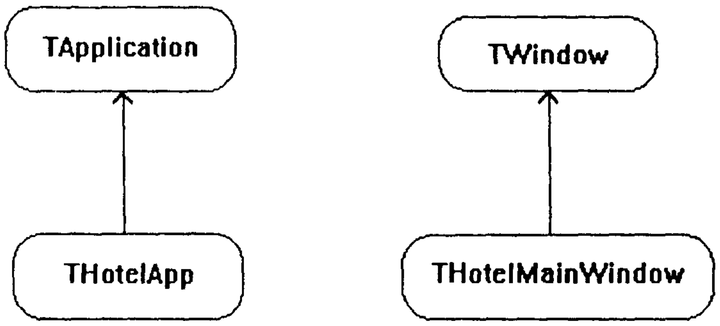


Fig. 3. 4 Inheritance Diagrams for the Application and Main Window objects

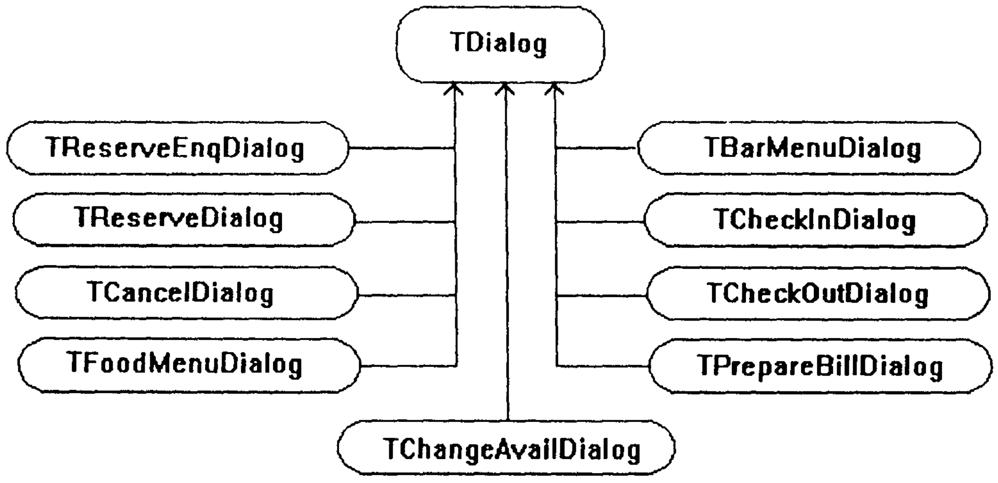


Fig. 3. 5 Inheritance Diagram for the Windows Dialog Boxes

The user interface application's main program consists of just three statements. The first statement of the WinMain (Windows main program) constructs the application object by calling its constructor. The constructor initializes the data members of the application object. The second statement calls the application's **Run** member function. The third statement returns the final status of the application that Object Windows stores in the *Status* data member. **Run** calls **InitApplication** and **InitInstance** to perform the first-instance and each instance initialization, respectively. **InitMainWindow** is then called to create the main window. **Run** then sets the application in motion by calling **MessageLoop** to begin processing incoming Windows messages, which directly affect the application's flow. **MessageLoop** calls member functions that process particular incoming messages.

The application flow of the user interface application is shown below. Chapter 6 gives a partial listing of the source code.

3. 4. 1 Application flow

```
Begin
    Construct a TApplication object
    Construct aTMainWindow object
    While( message = WM_QUIT )
        Process messages for the application
    Destruct TMainWindow object
    Destruct TApplication object
    Return status
End
```

3. 4. 1. 1 Construct a TApplication object

```
Begin
    Initialize TApplication object data members
End
```

3. 4. 1. 2 Construct a TMainWindow object

```
Begin
    Initialize TWindow object data members
    Assign menu to TMainWindow ogject
    Build BRoom B-tree from room.$$$ file
    Build BGuest B-tree from guest.$$$ file
    Build BRestaurant B-tree from restrnt.$$$ file
    Build BReservation B-tree from reserve.$$$ file
    Build BBill B-tree from bill.$$$ file
End
```

3. 4. 1. 3 Process messages for the application

```
Begin
    Case Active object:
      THotelMainWindow    =>  Process main window messages
      TReserveEnqDialog   =>  Process reservation enquiry dialog messages
      TReserveDialog      =>  Process reservation dialog messages
      TCancelDialog       =>  Process cancel reservation dialog messages
      TFoodMenuDialog     =>  Process food menu dialog messages
      TBarMenuDialog      =>  Process bar menu dialog messages
      TInputDialog        =>  Process guest enquiry dialog messages
       (Guest Enquiry)
      TCheckInDialog          =>  Process check-in dialog messages
      TCheckOutDialog     =>  Process check-out dialog messages
      TPrepareBillDialog  =>  Process prepare bill dialog messages
      TInputDialog        =>  Process update date messages
      (Update Date)
      TChangeAvailDialog  =>  Process change availability dialog messages
      THelpWindow         =>  Process help window messages
  End Case
End
```

3. 4. 1. 3. 1 Process main window messages

```
Begin
    Case User Selection:
      menu item RoomReservationEnquiry   =>  Execute TReserveEnqDialog
      menu item RoomReservation          =>  Execute TReserveDialog
      menu item Cancellation             =>  Execute TCancelDialog
```

```
        menu item FoodMenu                  =>  Execute TFoodMenuDialog

        menu item BarMenu                   =>  Execute TBarMenuDialog

        menu item GuestEnquiry              =>  Execute TInputDialog
                                                    ( Guest Enquiry )

        menu item CheckIn                   =>  Execute TCheckInDialog

        menu item CheckOut                  =>  Execute TCheckOutDialog

        menu item PrepareBill               =>  Execute TPrepareBillDialog

        menuitem  UpdateDate                =>  Execute TInputDialog
                                                    ( Update Date )

         menuitem ChangeAvailabilityOfFood/Drink

                                            => Execute TChangeAvailDialog

         menu item Help                     =>  Execute Help Window

    End Case

End
```

Chapter 4

Sample Session

The Hotel Automation System is a window-based menu-driven system. The application has the same "look and feel" as any standard MS-Windows application and all the common operations such as selecting an item, closing, minimizing and maximizing etc. will work in a similar fashion as any standard MS-Windows application.

The application's main window(Fig. 4. 1) shows the main menu of the application. The **Lodging** menu provides the services related to the lodging such as reservation of rooms etc. The **Boarding** menu provides the services related to the restaurant. The **General** menu provides the general services such as enquiry of a guest etc. The Internal menu provides the services which are of the internal use of the hotel management, such as change availability of food/drink items. The **Help** menu provides the information about the application and its services.

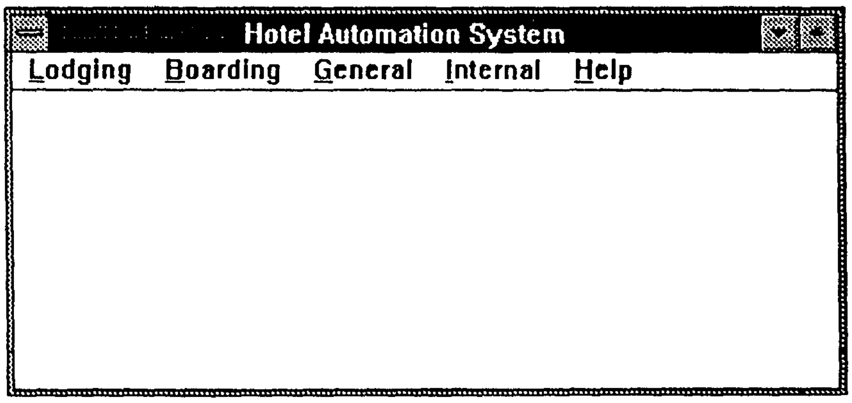


Fig. 4. 1 The main window of the application

The selection of the **Lodging** menu item leads to a sub-menu(Fig. 4. 2), showing the individual services that are available in the **Lodging** menu.

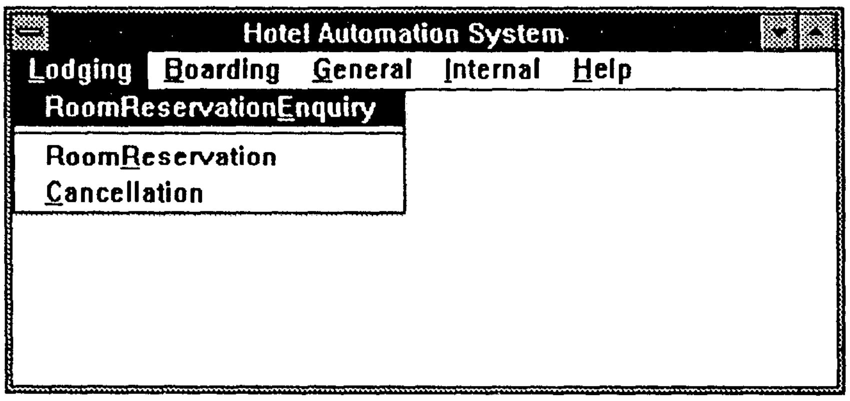


Fig. 4.2 The Lodging menu

The **Boarding** pulldown menu is shown in Fig. 4. 3. It lists the services related to the restaurant and the bar.

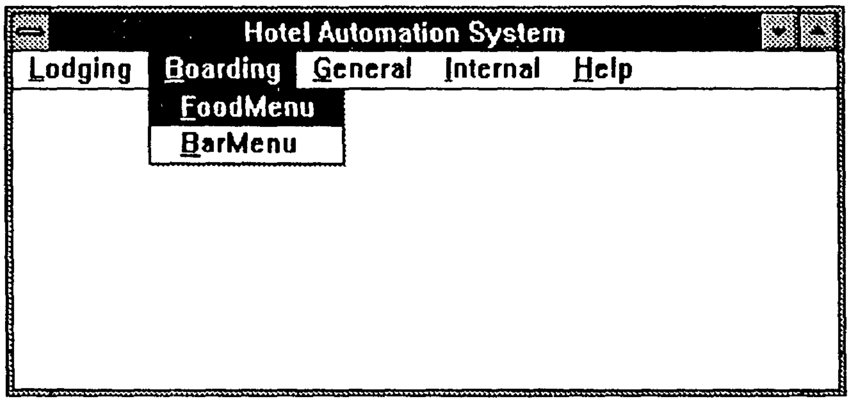


Fig. 4. 3 The Boarding menu

Selection of the **General** pulldown menu shows the general services as shown in Fig. 4.4.

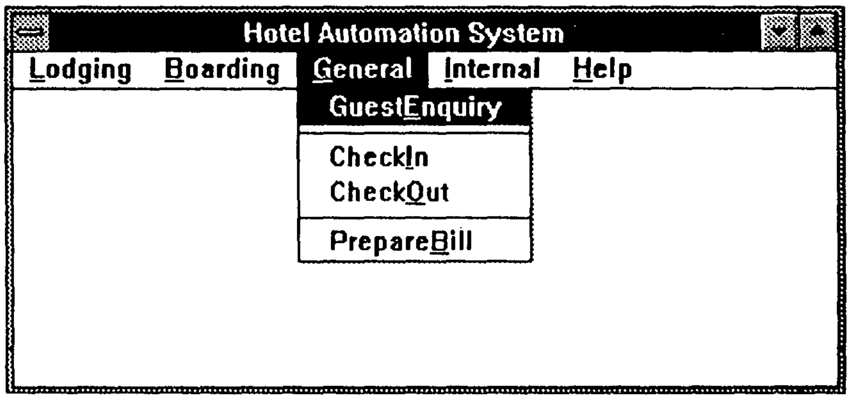


Fig. 4. 4 The General menu

Selection of the **Internal** menu item in the main menu gives the pull down menu as shown in Fig. 4. 5.

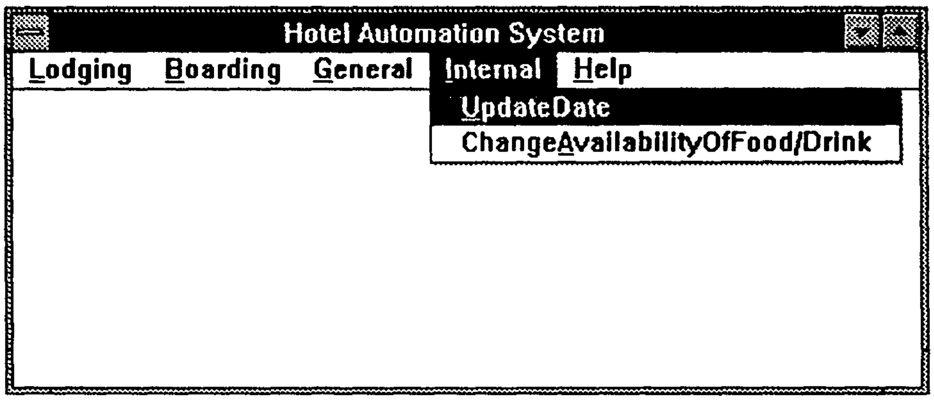


Fig. 4. 5 The Internal menu

Selecting the Help menu item in the main window gives the following pulldown menu(Fig.4.6), which provides the general help about the application and its services.

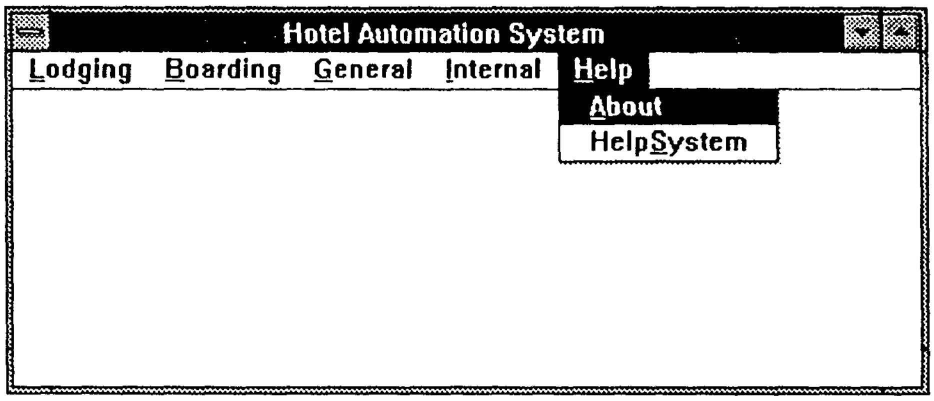


Fig. 4. 6 The Help menu

When the user selects the **RoomReservationEnquiry** menu item in the **Lodging** menu, the Room Reservation Enquiry dialog, shown in Fig. 4. 7 appears. It takes the room type and dates of booking as input and returns a message whether the reservation exists or not.

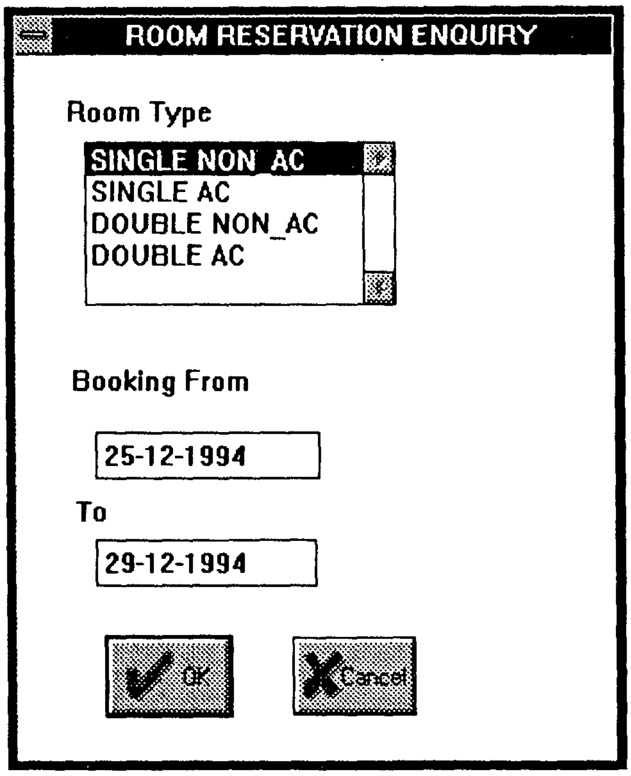
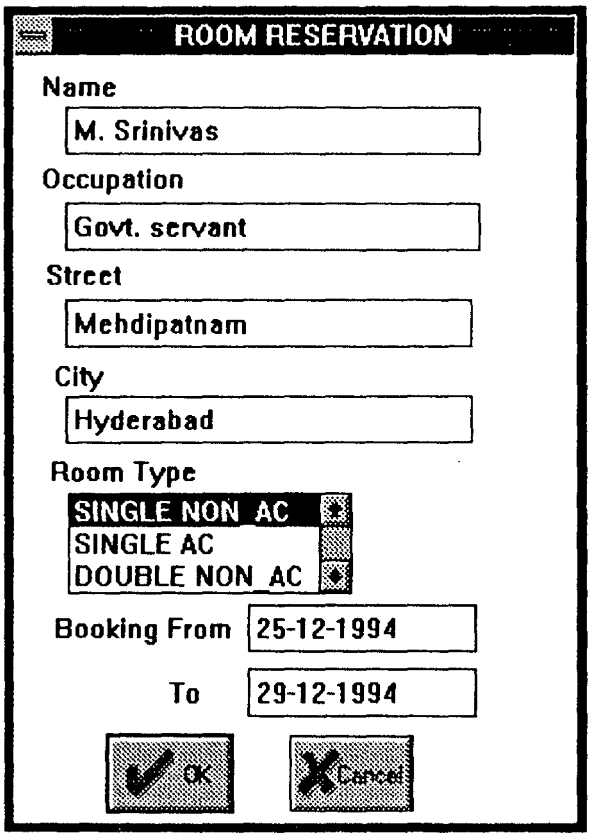


Fig. 4. 7 Room Reservation Enquiry

Selection of **RoomReservation** menu item from the **Lodging** menu gives the Room Reservation dialog (Fig. 4. 8). This dialog box takes the guest's details such as name, address etc., the room type and booking dates as input and reserves the room for the specified period, returning the reservation number. If the reservation does not exist, it gives a message to that effect.


ROOM RESERVATION

Name: M. Srinivas

Occupation: Govt. servant

Street: Mehdipatnam

City: Hyderabad

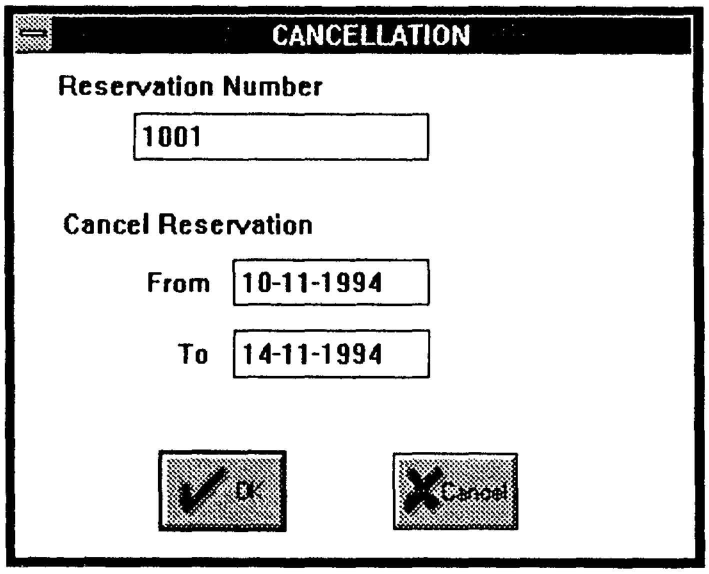
Room Type: SINGLE NON AC / SINGLE AC / DOUBLE NON AC

Booking From: 25-12-1994

To: 29-12-1994

Fig. 4. 8 Room Reservation dialog

If the user selects the **Cancellation** menu item from the **Lodging** menu, the Cancellation dialog appears on the screen. It takes the reservation number and cancellation dates and cancels the reservation.


CANCELLATION

Reservation Number

1001

Cancel Reservation

From 10-11-1994

To 14-11-1994

Fig. 4. 9 Cancellation dialog

When the user selects the **FoodMenu** menu item from the **Boarding** menu, the Food Menu dialog appears(Fig. 4. 10). It lists the available food items in a list box and allows the user to select any number of items from the menu.

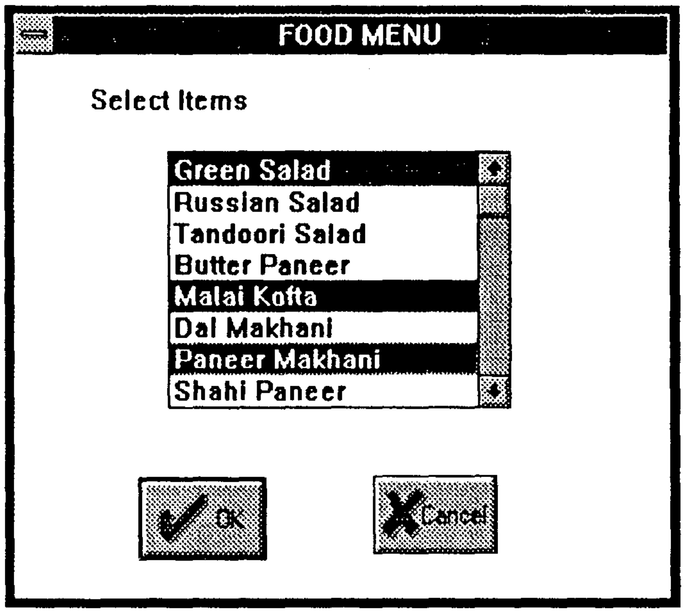


Fig. 4. 10 Food Menu dialog

If the user selects the **BarMenu** menu item from the **Boarding** menu, the Bar Menu dialog appears(Fig. 4. 11). This lists the available drink items in a list box and allows the user to select any number of items from the list.

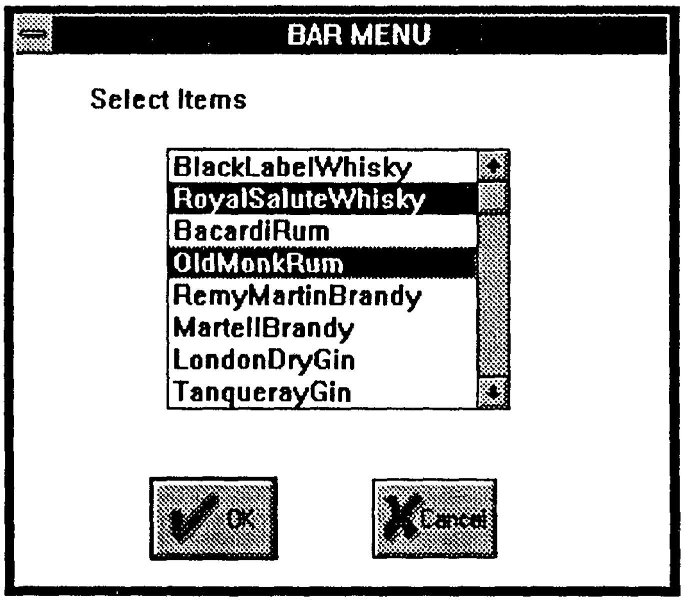
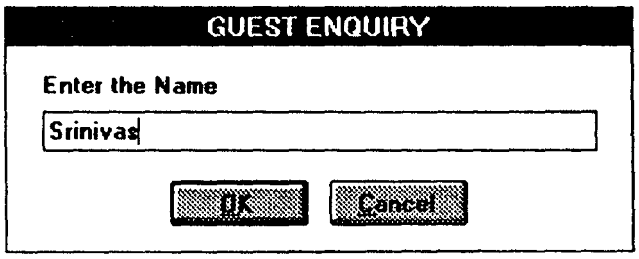


Fig. 4. 11 The Bar Menu dialog

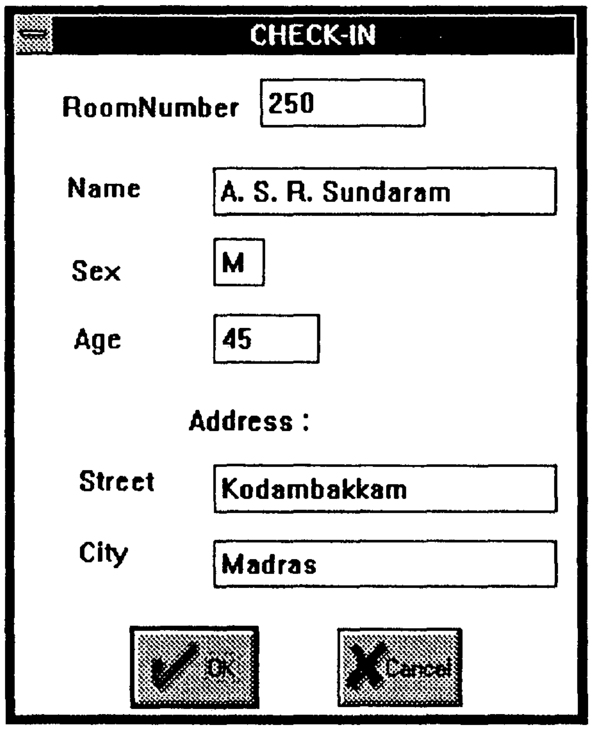
Selecting the **GuestEnquiry** menu item from the **General** menu prompts the Guest Enquiry dialog as shown in Fig. 4. 12. This dialog takes the guest's name and returns the guest's details.


Enter the Name

Srinivas

Fig. 4. 12 The Guest Enquiry dialog

Selecting the **Check-In** menu item from the **General** menu gives the Check In dialog (Fig. 4. 13). It takes the guest's details and performs the check-in operation. This operation is performed when a guest checks in the hotel against a reservation. It changes the status of the room from 'VACANT' to 'OCCUPIED'.



CHECK-IN

RoomNumber 250

Name A. S. R. Sundaram

Sex M

Age 45

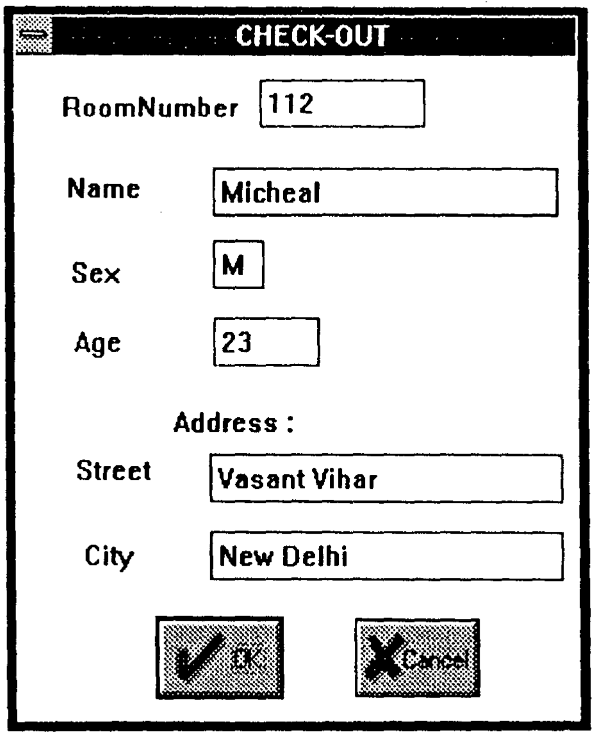
Address :

Street Kodambakkam

City Madras

Fig. 4. 13 The Check In dialog

Selection of the **Check-Out** menu item from the **General** menu gives the Check Out dialog(Fig. 4. 14). It takes the guest's details and performs the check-out operation. This operation is performed when a guest vacates the room on expiry of the reservation or in advance. It changes the statut of the room from 'OCCUPIED' to 'VACANT'.



RoomNumber 112

Name Micheal

Sex M

Age 23

Address :

Street Vasant Vihar

City New Delhi

Fig. 4. 14 The Check Out dialog

Selection of the **PrepareBill** menu item from the **General** menu gives the Prepare Bill dialog(Fig. 4. 15). It takes the customer's name and room number(if resident) and prepares the bill.

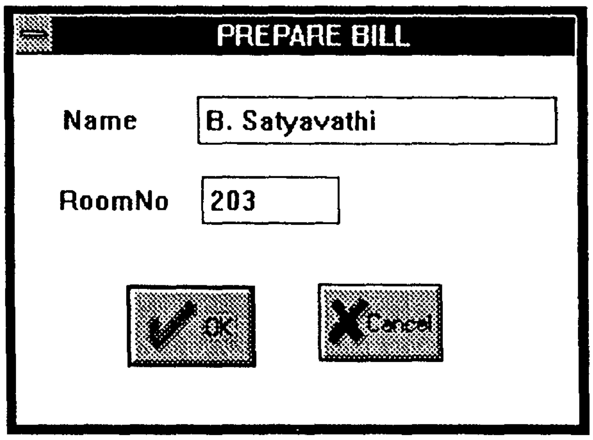


Fig. 4. 15 Prepare Bill dialog

Selecting the **UpdateDate** menu item from the **Internal** menu gives the Update Date input dialog box(Fig. 4. 16). This operation allows to update the 'current date' by taking the new date as input. The 'current date' is used in various calculations. This operation is usually performed at the beginning of a day.

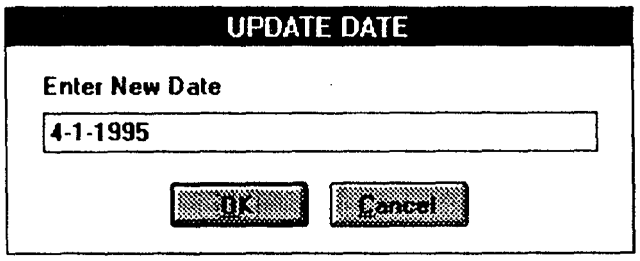
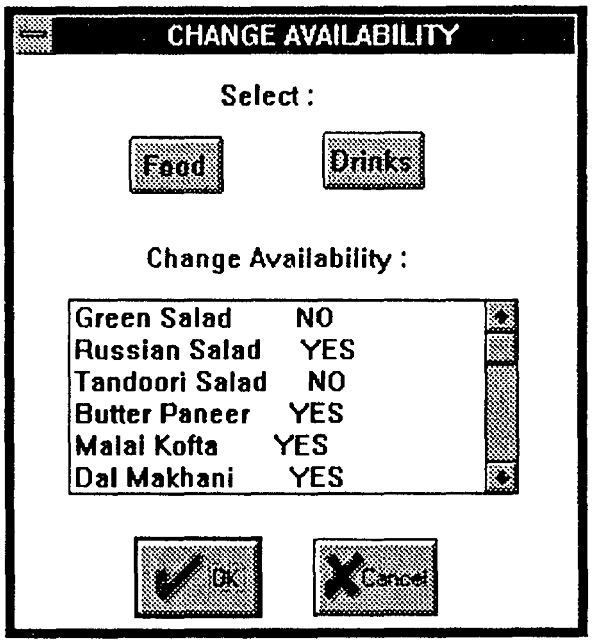


Fig. 4. 16 The Update Date dialog

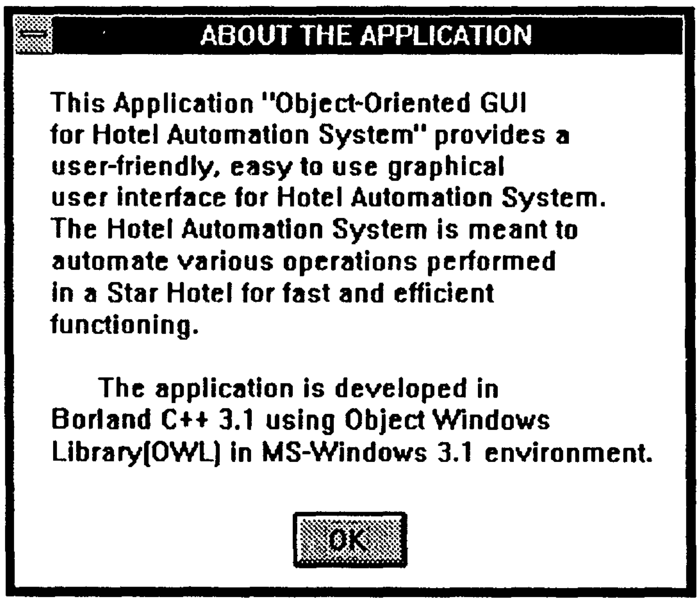
Selecting the **ChangeAvailabilityOfFood/Drink** menu item from the **Internal** menu gives the Change Availability dialog. Selecting the 'Food' or 'Drinks' button fills the list box with food or drink items respectively. Fig. 4. 17 shows the Change availability dialog for food items. The user can change the availability by clicking on the particular item.



```
Green Salad      NO
Russian Salad    YES
Tandoori Salad   NO
Butter Paneer    YES
Malai Kofta      YES
Dal Makhani      YES
```

Fig. 4. 17 Change Availability dialog

If the user selects the **About** menu item from the **Help** menu, a message box giving information about the application appears on the screen.

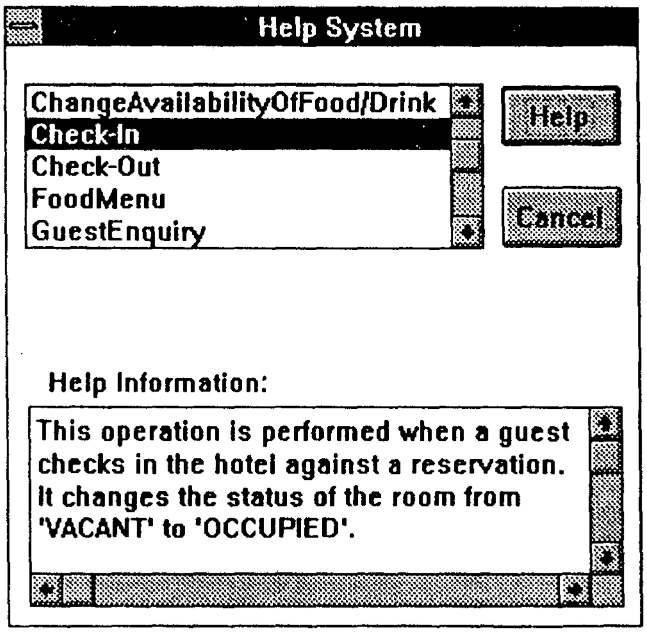


This Application "Object-Oriented GUI for Hotel Automation System" provides a user-friendly, easy to use graphical user interface for Hotel Automation System. The Hotel Automation System is meant to automate various operations performed in a Star Hotel for fast and efficient functioning.

The application is developed in Borland C++ 3.1 using Object Windows Library(OWL) in MS-Windows 3.1 environment.

Fig. 4. 18 About message box

If the user selects the **HelpSystem** menu item from the **Help** menu, the Help System window appears on the screen(Fig. 4. 19). The Help System provides information about various services the Hotel Automation System provides by listing them in a list box. Selection of any particular service displays the information regarding that service.



This operation is performed when a guest checks in the hotel against a reservation. It changes the status of the room from 'VACANT' to 'OCCUPIED'.

Fig. 4. 19 The Help System window

Chapter 5

Conclusion And Enhancements

Graphical User Interfaces play very important role in the success of any software product. The concept of GUI has taken a revolutionary change with the introduction of systems like MS-Windows. Object-Oriented Programming is well suited for GUI development and made the development of GUI easy. Borland's Object Windows Library (OWL) further simplified the development of Windows applications.

The Hotel Automation System automates the activities of a Star Hotel for fast and efficient operations.

Time constraints forced us to limit the project to a prototype Hotel Automation System instead of a real world system. Some of the possible enhancements that can be made to this system are discussed below.

The present Hotel Automation System performs 11 selected operations. More operations can be added to represent other activities like laundry, discotheque etc. Extra operations can be easily added by defining classes to represent them and adding aditional member response functions to the appropriate window objects.

The present system supports only a single restaurant. Provision can be made to support number of different restaurants.

Not every person is allowed to access every operation. Some previleged operations should be restricted to specific persons only for security reasons. For example, 'ChangeAvailabilityOfFood/Drink' operation should be allowed to be performed only by Manager(Catering).

Accounting and auditing of the hotel's financial accounts can be done. Employee information and salary calculation and maintanance can be included in the Hotel Automation System.

As the present system provides only a limited help facility, a full-fledged on-line help system can be added to improve the user interface.

User interface can be made more effective and user-friendly by including more controls, like check boxes, combo boxes, group boxes, radio buttons etc. Database design can be normalized so that data redundancy and other pitfalls can be minimized.

Chapter 6

PROGRAM LISTING

The project file of the application contains the files ***hotel.cpp***, ***helpwind.cpp***, ***hotrl.rc*** and ***standard.def***. The header files of the application are ***hotel.h***, ***resource.h***, ***helpwind.h*** and ***project.h***. In this Chapter, a listing of the project's main file ***hotel.cpp*** and the project's header file ***project.h*** is provided to give a brief outline of the application.

The ***hotel.cpp*** file contains WinMain and member functions for HotelApp, THotelMainWindow and all the dialog box classes. The project.h file contains class declarations for all window classes and classes to represent the members of different B-trees. The hotel.rc file contains the resource data. The ***standard.def*** file contains the memory stack and memory heap requirements of the application.

6. 1 Application's MainProgram (hotel.cpp)

```
#include "project.h"

/************** Implementation of TReserveEnqDialog functions ***********/


// Construct the TReserveEnqDialog dialog
TReserveEnqDialog::TReserveEnqDialog(PTWindowsObject AParent,LPSTR AName,
                                    PTModule AModule = NULL)
                  : TDialog(AParent,AName,AModule)
 {
  new TListBox(this,ID_LISTBOX);
  new TEdit(this,ID_FROMEDIT,DATELEN,NULL);
  new TEdit(this,ID_TOEDIT,DATELEN,NULL);
  new TButton(this,IDOK);
  new TButton(this,IDCANCEL);
  TransferBuffer = &ReserveEnqData;
 }

TReserveEnqDialog::~TReserveEnqDialog()
 {
 }


/************ Implementation of TReserveDialog functions ***************/


// Construct the TReserveDialog dialog
TReserveDialog::TReserveDialog(PTWindowsObject AParent,LPSTR AName,
                                    PTModule AModule = NULL)
              : TDialog(AParent,AName,AModule)
 {
  new TEdit(this,ID_NAMEEDIT,NAMELEN,NULL);
  new TEdit(this,ID_OCCUPATIONEDIT,75,NULL);
  new TEdit(this,ID_STREETEDIT,75,NULL);
  new TEdit(this,ID_CITYEDIT,75,NULL);
  new TEdit(this,ID_RFROMEDIT,DATELEN,NULL);
  new TEdit(this,ID_RTOEDIT,DATELEN,NULL);
  new TListBox(this,ID_RLISTBOX);
  new TButton(this,IDOK);
  new TButton(this,IDCANCEL);
  TransferBuffer = &ReserveData;
 }

 TReserveDialog::~TReserveDialog()
 {
 }


/*************** Implementation of TCancelDialog functions **************/


// Construct the TCancelDialog dialog
```

```
TCancelDialog::TCancelDialog(PTWindowsObject AParent,
                       LPSTR AName,PTModule AModule = NULL)
                      : TDialog(AParent,AName,AModule)
{
 new TEdit(this,ID_RNUMBER,75,NULL);
 new TEdit(this,ID_CFROMEDIT,DATELEN,NULL);
 new TEdit(this,ID_CTOEDIT,DATELEN,NULL);
 new TButton(this,IDOK);
 new TButton(this,IDCANCEL);
 TransferBuffer = &CancelData;
}


/************** Implementation of TFoodMenuDialog functions *************/


// Construct the TFoodMenuDialog dialog
TFoodMenuDialog::TFoodMenuDialog(PTWindowsObject AParent,LPSTR AName,
                                  PTModule AModule = NULL)
                 :TDialog(AParent,AName,AModule)
 {
  ListBox = new TListBox(this,ID_FMLISTBOX);
  new TButton(this,IDOK);
  new TButton(this,IDCANCEL);
  ListBox->ClearList();
  ListBoxData = new TListBoxData();
  for(int i=0;i<MAXFOODITEMS;i++)
   {
    if(strcmp(YFood[i],Food[i]) == 0)
     ListBoxData->AddString(IFood[i]);
   }
  TransferBuffer = &ListBoxData;
 }

 TFoodMenuDialog::~TFoodMenuDialog()
  {
  }


 /***************** Implementation of TBarMenu functions *****************/


 // Construct the TBarMenuDialog dialog
 TBarMenuDialog::TBarMenuDialog(PTWindowsObject AParent,LPSTR AName,
                                PTModule AModule = NULL)
               : TDialog(AParent,AName,AModule)
  {
   ListBox = new TListBox(this,ID_BMLISTBOX);
   new TButton(this,IDOK);
   new TButton(this,IDCANCEL);
   ListBox->ClearList();
   ListBoxData = new TListBoxData();
   for(int i=0;i<MAXDRINKITEMS;i++)
    {
     if(strcmp(YDrink[i],Drink[i]) == 0)
      ListBoxData->AddString(IDrink[i]);
 }
 TransferBuffer = &ListBoxData;
}

TBarMenuDialog::~TBarMenuDialog()
{
}


/*************** Implementation of TCheckInDialog functions *************/


// Construct the TCheckInDialog dialog
TCheckInDialog::TCheckInDialog(PTWindowsObject AParent,LPSTR AName,
                  PTModule AModule = NULL)
              : TDialog(AParent,AName,AModule)
 {

  new TEdit(this,ID_ROOM,ROOMLEN,NULL);
  new TEdit(this,ID_NAME,NAMELEN,NULL);
  new TEdit(this,ID_SEX,SEXLEN,NULL);
  new TEdit(this,ID_AGE,AGELEN,NULL);
  new TEdit(this,ID_STREET,75,NULL);
  new TEdit(this,ID_CITY,75,NULL);
  new TButton(this,IDOK);
  new TButton(this,IDCANCEL);
  TransferBuffer = &CheckInData;
 }

TCheckInDialog::~TCheckInDialog()
 {
 }


/*************** Implementation of TCheckOutDialog functions *************/


//Construct the TCheckOutDialog dialog
TCheckOutDialog::TCheckOutDialog(PTWindowsObject AParent,LPSTR AName,
                  PTModule AModule = NULL)
              : TDialog(AParent,AName,AModule)
 {
  new TEdit(this,ID_CROOM,ROOMLEN,NULL);
  new TEdit(this,ID_CNAME,NAMELEN,NULL);
  new TEdit(this,ID_CSEX,SEXLEN,NULL);
  new TEdit(this,ID_CAGE,AGELEN,NULL);
  new TEdit(this,ID_CSTREET,75,NULL);
  new TEdit(this,ID_CCITY,75,NULL);
  new TButton(this,IDOK);
  new TButton(this,IDCANCEL);
  TransferBuffer = &CheckOutData;
 }

TCheckOutDialog::~TCheckOutDialog()
 {
 }
```

```
/************* Implementation of PrepareBillDialog functions ************/


// Construct the TPrepareBillDialog dialog
TPrepareBillDialog::TPrepareBillDialog(PTWindowsObject AParent,LPSTR AName,
                        PTModule AModule = NULL)
                       : TDialog(AParent,AName,AModule)
{
 new TEdit(this,ID_BNAME,NAMELEN,NULL);
 new TEdit(this,ID_BROOM,ROOMLEN,NULL);
 new TButton(this,IDOK);
 new TButton(this,IDCANCEL);
 TransferBuffer = &PrepareBillData;
}

TPrepareBillDialog::~TPrepareBillDialog()
{
}


/************** Implementation of ChangeAvailDialog functions ************/


//Construct the TChangeAvailDialog dialog
TChangeAvailDialog::TChangeAvailDialog(PTWindowsObject AParent,LPSTR AName,
                                    PTModule AModule = NULL)
                      : TDialog(AParent,AName,AModule)
{
 ListBox = new TListBox(this,ID_ALISTBOX);
 new TButton(this,ID_FOODBTN);
 new TButton(this,ID_DRINKBTN);
 new TButton(this,IDOK);
 new TButton(this,IDCANCEL);
}

TChangeAvailDialog::~TChangeAvailDialog()
{
}

// Process Food button message
void TChangeAvailDialog::HandleFoodButtonMsg(RTMessage)
{
 int i;
 ListBox->ClearList();
 FoodListData = new TListBoxData();
 for(i=0;i<MAXFOODITEMS;i++)
  FoodListData->AddString(Food[i]);
 TransferBuffer = &FoodListData;
 ListBox->Transfer(&FoodListData,TF_SETDATA);
}

// Process the Drink button message
void TChangeAvailDialog::HandleDrinkButtonMsg(RTMessage)
{
 ListBox->ClearList();
 DrinkListData = new TListBoxData();
 for(int i=0;i<MAXDRINKITEMS;i++)
  DrinkListData->AddString(Drink[i]);
 TransferBuffer = &DrinkListData;
 ListBox->Transfer(&DrinkListData,TF_SETDATA);
}

// Process the List box message
void TChangeAvailDialog::HandleListBoxMsg(RTMessage Msg)
{
 char Selection[25];
 BOOL flag;
 if(Msg.LP.Hi == LBN_SELCHANGE)
  {
   int Ind = ListBox->GetSelIndex();
   ListBox->GetSelString(Selection,25);
   if((Ind<MAXFOODITEMS)&&(Ind<MAXDRINKITEMS))
    {
     if(!strcmp(Selection,Food[Ind]))
      flag = TRUE;
     else if(!strcmp(Selection,Drink[Ind]))
      flag = FALSE;
    }
   else if(Ind<MAXFOODITEMS)
           flag = TRUE;
   else if(Ind<MAXDRINKITEMS)
           flag = FALSE;
   if(flag)
    {
     if(!strcmp(Selection,YFood[Ind]))
      strcpy(Food[Ind],NFood[Ind]);
     else if(!strcmp(Selection,NFood[Ind]))
      strcpy(Food[Ind],YFood[Ind]);
     ListBox->DeleteString(Ind);
     ListBox->InsertString(Food[Ind],Ind);
    }
   else
    {
     if(!strcmp(Selection,YDrink[Ind]))
      strcpy(Drink[Ind],NDrink[Ind]);
     else if(!strcmp(Selection,NDrink[Ind]))
      strcpy(Drink[Ind],YDrink[Ind]);
     ListBox->DeleteString(Ind);
     ListBox->InsertString(Drink[Ind],Ind);
    }
  }
 }


/************** Implementation of THotelMainWindow functions ***************/

// Construct the Application's MainWindow object
THotelMainWindow::THotelMainWindow(PTWindowsObject AParent,LPSTR ATitle)
                    : TWindow(AParent,ATitle)
 {
```

```
OFSTRUCT of;
int RmHandle,GHandle,RtHandle,RvHandle,BHandle;
char RoomNo[ROOMLEN],Type[TYPELEN],Status[STATUSLEN],Bill[BILLLEN];
char Sex[SEXLEN],Age[AGELEN],Address[ADDRESSLEN],From[DATELEN],To[DATELEN];
char Rate[RATELEN],Name[NAMELEN],Item[ITEMLEN],Availability[AVAILLEN];
char Rent[RATELEN];
AssignMenu("MAINMENU");
strcpy(DateBuffer,"DD/MM/YY");
strcpy(GuestBuffer,"Name");

/* Constructing the B-tree objects from the files at the beginning of the
 session */
RmHandle = OpenFile("room.$$$",&of,OF_READ);
while(!eof(RmHandle))
{
 _lread(RmHandle,RoomNo,ROOMLEN);
 _lread(RmHandle,Type,TYPELEN);
 _lread(RmHandle,Status,STATUSLEN);
 _lread(RmHandle,Rent,RATELEN);
 BRoom.add(*new RoomEntry(RoomNo,Type,Status,Rent));
}
_lclose(RmHandle);

GHandle = OpenFile("guest.$$$",&of,OF_READ);
while(!eof(GHandle))
{
 _lread(GHandle,Name,NAMELEN);
 _lread(GHandle,Sex,SEXLEN);
 _lread(GHandle,Age,AGELEN);
 _lread(GHandle,Address,ADDRESSLEN);
 BGuest.add(*new GuestEntry(Name,Sex,Age,Address));
}
_lclose(GHandle);

RtHandle = OpenFile("restrnt.$$$",&of,OF_READ);
while(!eof(RtHandle))
{
 _lread(RtHandle,Item,ITEMLEN);
 _lread(RtHandle,Rate,RATELEN);
 _lread(RtHandle,Availability,AVAILLEN);
 BRestaurant.add((Object&)*new RestaurantEntry(Item,Rate,Availability));
}
_lclose(RtHandle);

RvHandle = OpenFile("reserve.$$$",&of,OF_READ);
while(!eof(RvHandle))
{
 _lread(RvHandle,RoomNo,ROOMLEN);
 _lread(RvHandle,Name,NAMELEN);
 _lread(RvHandle,From,DATELEN);
 _lread(RvHandle,To,DATELEN);
 BReservation.add(*new ReservationEntry(RoomNo,Name,From,To));
}
_lclose(RvHandle);

BHandle = OpenFile("bill.$$$",&of,OF_READ);
```

```
 while(!eof(BHandle))
 {
  _lread(BHandle,Name,NAMELEN);
  _lread(BHandle,RoomNo,ROOMLEN);
  _lread(BHandle,Bill,BILLLEN);
  BBill.add((Object&)*new BillEntry(Name,RoomNo,Bill));
 }
 _lclose(BHandle);

}

void THotelMainWindow::HandleEnquiryMsg(RTMessage)
{
 if(GetApplication()->ExecDialog(new TReserveEnqDialog(this,
                          "ENQUIRYDIALOG"))==IDOK)
  {
   GlobalRoomNo = 1;
   ReservationEntry& m = (ReservationEntry&)BReservation.firstThat(
                                              testMember,0);
   if(GlobalRoomNo < MAXROOMS)
    MessageBox(HWindow,"Yes, Reservation Exists","MESSAGE",MB_OK);
   else
    MessageBox(HWindow,"Sorry! Reservation Does not Exist","MESSAGE",MB_OK);
  }
}

void THotelMainWindow::HandleResvMsg(RTMessage)
 {
  if(GetApplication()->ExecDialog(new TReserveDialog(this,
                            "RESERVATIONDIALOG")) == IDOK)
   {
    char temp[ROOMLEN];
    char Buf[100];
    GlobalRoomNo = 1;
    ReservationEntry& m = (ReservationEntry&)BReservation.firstThat(
                                           testMember,0);
    if(GlobalRoomNo < 3)
     {
      BReservation.add(*new ReservationEntry(itoa(GlobalRoomNo,temp,10),
               ReserveData.Name,ReserveData.FromEdit,ReserveData.ToEdit));
      wsprintf(Buf,"RoomNumber %d Is Reserved From %s To %s",GlobalRoomNo,
                              ReserveData.FromEdit,ReserveData.ToEdit);
      MessageBox(HWindow,Buf,"MESSAGE",MB_OK);
     }
    else
      MessageBox(HWindow,"Sorry! The Reservation Does Not Exist","MESSAGE",
                                          MB_OK);
   }
 }

 void THotelMainWindow::HandleCancelMsg(RTMessage)
 {
  GetApplication()->ExecDialog(new TCancelDialog(this,"CANCELDIALOG"));
 }

 void THotelMainWindow::HandleFoodMsg(RTMessage)
```

```
{
 GetApplication()->ExecDialog(new TFoodMenuDialog(this,"FOODDIALOG"));
}

void THotelMainWindow::HandleBarMsg(RTMessage)
{
 GetApplication()->ExecDialog(new TBarMenuDialog(this,"BARDIALOG"));
}

void THotelMainWindow::HandleGuestEnqMsg(RTMessage)
{
 GetApplication()->ExecDialog(new TInputDialog(this,"GUEST ENQUIRY",
                  "Enter the Name",GuestBuffer,sizeof GuestBuffer));
}

void THotelMainWindow::HandleCheckInMsg(RTMessage)
{
 GetApplication()->ExecDialog(new TCheckInDialog(this,"CHECKINDIALOG"));
}

void THotelMainWindow::HandleCheckOutMsg(RTMessage)
{
 GetApplication()->ExecDialog(new TCheckOutDialog(this,
                                 "CHECKOUTDIALOG"));
}

void THotelMainWindow::HandlePrepareBillMsg(RTMessage)
{
 GetApplication()->ExecDialog(new TPrepareBillDialog(this,
                               "PREPAREBILLDIALOG"));
}

void THotelMainWindow::HandleDateMsg(RTMessage)
{
 GetApplication()->ExecDialog(new TInputDialog(this,"UPDATE DATE",
                  "Enter New Date",DateBuffer,sizeof DateBuffer));
}

void THotelMainWindow::HandleChangeAvailMsg(RTMessage)
{
 GetApplication()->ExecDialog(new TChangeAvailDialog(this,
                                 "CHANGEAVAILDIALOG"));
}

void THotelMainWindow::HandleAboutMsg(RTMessage)
{
 char *Buff;
 Buff =
  "This Application \"Object-Oriented GUI\r\n"
  "for Hotel Automation System\" provides a\r\n"
  "user-friendly, easy to use graphical\r\n"
  "user interface for Hotel Automation System.\r\n"
  "The Hotel Automation System is meant to\r\n"
  "automate various operations performed\r\n"
  "in a Star Hotel for fast and efficient\r\n"
  "functioning.\r\n\n"
 "     The application is developed in\r\n"
 "Borland C++ 3.1 using Object Windows\r\n"
 "Library(OWL) in MS-Windows 3.1 environment.";
MessageBox(HWindow,Buff,"ABOUT THE APPLICATION",MB_OK);
}

void THotelMainWindow::HandleHelpMsg(RTMessage)
{
 HlpWindow = new HelpWindow(this);
 GetApplication()->MakeWindow(HlpWindow);
}

BOOL THotelMainWindow::CanClose()
{
 OFSTRUCT of;
 int RmHandle,GHandle,RtHandle,RvHandle,BHandle;
 int no_of_items,i;
 String& RoomNo(*new String("")),Type(*new String(""));
 String& Status(*new String("")),Age(*new String("")),Bill(*new String(""));
 String& Availability(*new String("")),Sex(*new String(""));
 String& Address(*new String("")),From(*new String("")),To(*new String(""));
 String& Rate(*new String("")),Name(*new String("")),Item(*new String(""));

// Storing the data from the B-trees to the files at the end of the session
 no_of_items = BRoom.getItemsInContainer();
 RmHandle = OpenFile("room.$$$",&of,OF_READWRITE|OF_CREATE);
 for(i=0;i<no_of_items;i++)
  {
   RoomNo = ((RoomEntry&)(BRoom[i])).RoomNo();
   Type = ((RoomEntry&)(BRoom[i])).Type();
   Status = ((RoomEntry&)(BRoom[i])).Status();
   Rate = ((RoomEntry&)(BRoom[i])).Rent();
   _lwrite(RmHandle,RoomNo,ROOMLEN);
   _lwrite(RmHandle,Type,TYPELEN);
   _lwrite(RmHandle,Status,STATUSLEN);
   _lwrite(RmHandle,Rate,RATELEN);
  }
 _lclose(RmHandle);

 no_of_items = BGuest.getItemsInContainer();
 GHandle = OpenFile("guest.$$$",&of,OF_READWRITE|OF_CREATE);
 for(i=0;i<no_of_items;i++)
  {
   Name = ((GuestEntry&)(BGuest[i])).Name();
   Sex = ((GuestEntry&)(BGuest[i])).Sex();
   Age = ((GuestEntry&)(BGuest[i])).Age();
   Address = ((GuestEntry&)(BGuest[i])).Address();
   _lwrite(GHandle,Name,NAMELEN);
   _lwrite(GHandle,Sex,SEXLEN);
   _lwrite(GHandle,Age,AGELEN);
   _lwrite(GHandle,Address,ADDRESSLEN);
  }
 _lclose(GHandle);

 no_of_items = BRestaurant.getItemsInContainer();
 RtHandle = OpenFile("restmt.$$$",&of,OF_READWRITE|OF_CREATE);
```

```
for(i=0;i<no_of_items;i++)
 {
  Item = ((RestaurantEntry&)(BRestaurant[i])).Item();
  Rate = ((RestaurantEntry&)(BRestaurant[i])).Rate();
  Availability = ((RestaurantEntry&)(BRestaurant[i])).Availability();
  _lwrite(RtHandle,Item,ITEMLEN);
  _lwrite(RtHandle,Rate,RATELEN);
  _lwrite(RtHandle,Availability,AVAILLEN);
 }
_lclose(RtHandle);

no_of_items = BReservation.getItemsInContainer();
RvHandle = OpenFile("reserve.$$$",&of,OF_READWRITE|OF_CREATE);
for(i=0;i<no_of_items;i++)
 {
  RoomNo = ((ReservationEntry&)(BReservation[i])).RoomNo();
  Name = ((ReservationEntry&)(BReservation[i])).Name();
  From = ((ReservationEntry&)(BReservation[i])).From();
  To = ((ReservationEntry&)(BReservation[i])).To();
  RoomNo.printOn(cout);
  Name.printOn(cout);
  From.printOn(cout);
  To.printOn(cout);
  _lwrite(RvHandle,RoomNo,ROOMLEN);
  _lwrite(RvHandle,Name,NAMELEN);
  _lwrite(RvHandle,From,DATELEN);
  _lwrite(RvHandle,To,DATELEN);
 }
_lclose(RvHandle);

no_of_items = BBill.getItemsInContainer();
BHandle = OpenFile("bill.$$$",&of,OF_READWRITE|OF_CREATE);
for(i=0;i<no_of_items;i++)
{
 Name = ((BillEntry&)(BBill[i])).Name();
 RoomNo = ((BillEntry&)(BBill[i])).RoomNo();
 Bill = ((BillEntry&)(BBill[i])).Bill();
 _lwrite(BHandle,Name,NAMELEN);
 _lwrite(BHandle,RoomNo,ROOMLEN);
 _lwrite(BHandle,Bill,BILLLEN);
}
_lclose(BHandle);

delete &RoomNo;
delete &Type;
delete &Status;
delete &Availability;
delete &Sex;
delete &Age;
delete &Address;
delete &From;
delete &To;
delete &Rate;
delete &Name;
delete &Item;
delete &Bill;
```

```
 return TRUE;
}


/*************** Hotel Application Object HotelApp functions ************/

// Construct the Application object
HotelApp::HotelApp(LPSTR AName,HINSTANCE hInstance,HINSTANCE hPrevInstance,
                   LPSTR lpszCmdLine,int nCmdShow)
         : TApplication(AName,hInstance,hPrevInstance,lpszCmdLine,nCmdShow)
         {}

void HotelApp::InitMainWindow()
 {
  MainWindow = new THotelMainWindow(NULL,Name);
 }


/*********************** Main function WinMain **************************/

//Main function for the Application
int PASCAL WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
                   LPSTR lpszCmdLine,int nCmdShow)
 {
  HotelApp HotelObj("Hotel Automation System",hInstance,hPrevInstance,
                    lpszCmdLine,nCmdShow);
  HotelObj.Run();
  return HotelObj.Status;
 }
```

6. 2 Application's Header File (Project.h)

```
#include <owl.h>
#include <io.h>
#include <iostream.h>
#include <listbox.h>
#include <button.h>
#include <edit.h>
#include <inputdia.h>
#include <string.h>
#include <bwcc.h>
#include <assoc.h>
#include <btree.h>
#include <strng.h>
#include <ldate.h>
#include "vc5.h"
#include "resource.h"
#include "helpwind.h"

Btree BRoom(5),BGuest(5),BRestaurant(5),BReservation(5),BBill(5);
int GlobalRoomNo;
PTListBox FdListBox,BrListBox;
```

```
char *YFood[MAXFOODITEMS] = {"Green Salad      YES",  "Russian Salad     YES", "Tandoori
Salad    YES", "Butter Paneer     YES", "Malai Kofta        YES",  "Dal Makhani      YES", "Paneer
Makhani   YES", "Shahi Paneer      YES", "Palak Paneer   YES", "Gobi Masala       YES",
"Navrattan Korma   YES", "Butter Chicken     YES",   "Mutton Masala      YES", "Vegetable Pulao
YES", "Kashmiri Pulao    YES", "Tandoori Chicken  YES", "Kashmiri Kabab    YES", "Chicken
Kabab      YES", "Vegetable Raita    YES", "Tandoori Nan        YES"};

char *YDrink[MAXDRINKITEMS] = {"BlackLabelWhisky  YES", "RoyalSaluteWhisky YES",
"BacardiRum          YES", "OldMonkRum          YES", "RemyMartinBrandy  YES", "MartellBrandy
YES", "LondonDryGin       YES", "TanquerayGin        YES", "StolichnayaVodka  YES",
"AbsolutVodka        YES" };

char *NFood[MAXFOODITEMS] = {"Green Salad        NO", "Russian Salad       NO", "Tandoori
Salad      NO", "Butter Paneer       NO", "Malai Kofta          NO", "Dal Makhani          NO",
"Paneer Makhani      NO", "Shahi Paneer        NO", "Palak Paneer         NO", "Gobi Masala
NO",                 "Navrattan Korma    NO", "Butter Chicken       NO", "Mutton Masala        NO",
"Vegetable Pulao     NO", "Kashmiri Pulao       NO", "Tandoori Chicken    NO", "Kashmiri Kabab
NO", "Chicken Kabab        NO", "Vegetable Raita     NO", "Tandoori Nan         NO"};

char *NDrink[MAXDRINKITEMS] = {"BlackLabelWhisky   NO", "RoyalSaluteWhisky  NO",
"BacardiRum           NO", "OldMonkRum            NO", "RemyMartinBrandy    NO", "MartellBrandy
NO", "LondonDryGin         NO", "TanquerayGin          NO", "StolichnayaVodka    NO",
"AbsolutVodka          NO" };

char *Food[MAXFOODITEMS] = {"Green Salad        YES", "Russian Salad      YES","Tandoori
Salad    YES", "Butter Paneer     YES", "Malai Kofta", "Dal Makhani        YES", "Paneer Makhani
YES",  "Shahi Paneer       YES", "Palak Paneer        YES", "Gobi Masala         YES", "Navrattan
Korma   YES", "Butter Chicken     YES","Mutton Masala       YES",  "Vegetable Pulao   YES",
"Kashmiri Pulao     YES", "Tandoori Chicken  YES", "Kashmiri Kabab     YES", "Chicken Kabab
YES",  "Vegetable Raita    YES", "Tandoori Nan        YES"};

char *Drink[MAXDRINKITEMS] = {"BlackLabelWhisky  YES", "RoyalSaluteWhisky YES",
"BacardiRum          YES", "OldMonkRum          YES", "RemyMartinBrandy  YES", "MartellBrandy
YES", "LondonDryGin       YES", "TanquerayGin        YES", "StolichnayaVodka  YES",
"AbsolutVodka        YES"  };

char *IFood[MAXFOODITEMS] = {"Green Salad        ", "Russian Salad      ", "Tandoori Salad    ",
"Butter Paneer      ", "Malai Kofta        ", "Dal Makhani        ", "Paneer Makhani     ", "Shahi Paneer
", "Palak Paneer       ", "Gobi Masala        ", "Navrattan Korma    ", "Butter Chicken     ",
"Mutton Masala      ", "Vegetable Pulao    ", "Kashmiri Pulao     ", "Tandoori Chicken   ", "Kashmiri
Kabab      ", "Chicken Kabab      ", "Vegetable Raita    ",  "Tandoori Nan       "};

char *IDrink[MAXDRINKITEMS] = {"BlackLabelWhisky  ", "RoyalSaluteWhisky ", "BacardiRum
", "OldMonkRum         ", "RemyMartinBrandy  ", "MartellBrandy      ", "LondonDryGin       ",
"TanquerayGin       ", "StolichnayaVodka  ", "AbsolutVodka       "};

/******* Structures to store the dialog box's TransferBuffer data ********/

struct TReserveEnqData
 {
  PTListBoxData ListBoxData;
  char FromEdit[DATELEN];
  char ToEdit[DATELEN];
  TReserveEnqData();
 } ReserveEnqData;
```

```
struct TReserveData
 {
  char Name[NAMELEN];
  char Occupation[75];
  char Street[75];
  char City[75];
  char FromEdit[DATELEN];
  char ToEdit[DATELEN];
  PTListBoxData ListBoxData;
  TReserveData();
 } ReserveData;

struct TCancelData
 {
  char RvNo[75];
  char FromEdit[DATELEN];
  char ToEdit[DATELEN];
  TCancelData();
 } CancelData;

struct TCheckInData
 {
  char RoomNo[ROOMLEN];
  char Name[NAMELEN];
  char Sex[SEXLEN];
  char Age[AGELEN];
  char Street[75];
  char City[75];
  TCheckInData();
 } CheckInData;

struct TCheckOutData
 {
  char RoomNo[ROOMLEN];
  char Name[NAMELEN];
  char Sex[SEXLEN];
  char Age[AGELEN];
  char Street[75];
  char City[75];
  TCheckOutData();
 } CheckOutData;

struct TPrepareBillData
 {
  char Name[NAMELEN];
  char RoomNo[ROOMLEN];
  TPrepareBillData();
 } PrepareBillData;

/********* Constructors to initialize the TransferBuffer data ***********/

TReserveEnqData :: TReserveEnqData()
 {
  ListBoxData = new TListBoxData;
  memset(FromEdit,0,DATELEN);
  memset(ToEdit,0,DATELEN);
 ListBoxData->AddString("SINGLE NON_AC",TRUE);
 ListBoxData->AddString("SINGLE AC");
 ListBoxData->AddString("DOUBLE NON_AC");
 ListBoxData->AddString("DOUBLE AC");

}

TReserveData :: TReserveData()
{
 memset(Name,0,NAMELEN);
 memset(Occupation,0,75);
 memset(Street,0,75);
 memset(City,0,75);
 memset(FromEdit,0,DATELEN);
 memset(ToEdit,0,DATELEN);
 ListBoxData = new TListBoxData;
 ListBoxData->AddString("SINGLE NON_AC",TRUE);
 ListBoxData->AddString("SINGLE AC");
 ListBoxData->AddString("DOUBLE NON_AC");
 ListBoxData->AddString("DOUBLE AC");

}

TCancelData::TCancelData()
{
 memset(RvNo,0,75);
 memset(FromEdit,0,DATELEN);
 memset(ToEdit,0,DATELEN);
}

TCheckInData::TCheckInData()
{
 memset(RoomNo,0,ROOMLEN);
 memset(Name,0,NAMELEN);
 memset(Sex,0,SEXLEN);
 memset(Age,0,AGELEN);
 memset(Street,0,75);
 memset(City,0,75);
}

TCheckOutData::TCheckOutData()
{
 memset(RoomNo,0,ROOMLEN);
 memset(Name,0,NAMELEN);
 memset(Sex,0,SEXLEN);
 memset(Age,0,AGELEN);
 memset(Street,0,75);
 memset(City,0,75);
}

TPrepareBillData::TPrepareBillData()
{
 memset(Name,0,NAMELEN);
 memset(RoomNo,0,ROOMLEN);
}
```

```
// Function to convert a character string to a Date object

Date& StringToDate(char *s)
 {
  char from[DATELEN],temp1[3],temp2[3],temp3[3];
  strcpy(from,s);
  int i,k=0;
  for(i=0;i<2;i++,k++)
   temp1[i] = from[k];
  temp1[i+1] = '\0';
  k++;
  for(i=0;i<2;i++,k++)
   temp2[i] = from[k];
  temp2[i+1] = '\0';
  k++;
  for(i=0;i<4;i++,k++)
   temp3[i] = from[k];
  temp3[i+1] = '\0';
  int day = atoi(temp1);
  int month = atoi(temp2);
  int year = atoi(temp3);
  Date D1(month,day,year);
  return (Date&)D1;
 }

/* Class definition for RoomEntry class. Defines RoomEntry objects to be
   stored in the BRoom B-tree. */

class RoomEntry : public Sortable
 {
  String& aRoomNo;
  String& aType;
  String& aStatus;
  String& aRent;

  public:

  enum {roomEntryClass = __firstUserClass};
  RoomEntry(char *roomno,char *type,char *status,char *rent)
    : aRoomNo(*new String(roomno)),aType(*new String(type)),
      aStatus(*new String(status)),aRent(*new String(rent)){}

  RoomEntry() : aRoomNo(*new String("")),aType(*new String("")),
             aStatus(*new String("")),aRent(*new String("")){}

  classType isA() const
   {
    return roomEntryClass;
   }

  char _FAR *nameOf() const
   {
    return "RoomEntry";
   }
int isEqual(const Object& e) const
 {
  return RoomNo() == ((RoomEntry&)e).RoomNo()
         && Type() == ((RoomEntry&)e).Type()
         && Status() == ((RoomEntry&)e).Status()
         && Rent() == ((RoomEntry&)e).Rent();
 }

int isLessThan(const Object& e) const
 {
  return RoomNo() < ((RoomEntry&)e).RoomNo();
 }

hashValueType hashValue() const
 {
  return aRoomNo.hashValue()+aType.hashValue()+
         aStatus.hashValue()+aRent.hashValue();
 }

void printOn(ostream& os) const
 {
  os<<aRoomNo<<":"<<aType<<":"<<aStatus<<":"<<aRent<<endl;
 }

String& RoomNo() const
 {
  return aRoomNo;
 }

String& Type() const
 {
  return aType;
 }

String& Status() const
 {
  return aStatus;
 }

String& Rent() const
 {
  return aRent;
 }

~RoomEntry()
 {
  delete &aRoomNo;
  delete &aType;
  delete &aStatus;
  delete &aRent;
 }
};

/* Class definition for the GuestEntry class. Defines GuestEntry objects
  to be stored in the BGuest B-tree. */
```

```
class GuestEntry : public Sortable
 {
  String& aName;
  String& aSex;
  String& aAge;
  String& aAddress;

  public:

  enum {guestEntryClass = __firstUserClass+1};
  GuestEntry(char *name,char *sex,char *age,char *address)
    : aName(*new String(name)),aSex(*new String(sex)),aAge(*new String(age)),
      aAddress(*new String(address)){}

  GuestEntry() : aName(*new String("")),aSex(*new String("")),
            aAge(*new String("")),aAddress(*new String("")){}

  classType isA() const
   {
    return guestEntryClass;
   }

  char _FAR *nameOf() const
   {
    return "GuestEntry";
   }

  int isEqual(const Object& e) const
   {
    return Name() == ((GuestEntry&)e).Name()
           && Sex() == ((GuestEntry&)e).Sex()
           && Age() == ((GuestEntry&)e).Age()
           && Address() == ((GuestEntry&)e).Address();
   }

  int isLessThan(const Object& e) const
   {
    return Name() < ((GuestEntry&)e).Name();
   }

  hashValueType hashValue() const
   {
    return aName.hashValue()+aSex.hashValue()+
           aAge.hashValue()+aAddress.hashValue();
   }

  void printOn(ostream& os) const
   {
    os<<aName<<":"<<aSex<<":"<<aAge<<":"<<aAddress<<endl;
   }

  String& Name() const
   {
    return aName;
   }
  String& Sex() const
   {
    return aSex;
   }

  String& Age() const
   {
    return aAge;
   }

  String& Address() const
   {
    return aAddress;
   }

  ~GuestEntry()
   {
    delete &aName;
    delete &aSex;
    delete &aAge;
    delete &aAddress;
   }
 };

/* Class definition for the RestrauntEntry class. Defines RestaurantEntry
   objects to be stored in the BRestaurant B-tree. */

class RestaurantEntry : Sortable
 {
  String& altem;
  String& aRate;
  String& aAvailability;

  public:

  enum {restrauntEntryClass = __firstUserClass+2};
  RestaurantEntry(char *item,char *rate,char *availability)
    : altem(*new String(item)),aRate(*new String(rate)),
      aAvailability(*new String(availability)){}

  RestaurantEntry() : altem(*new String("")),aRate(*new String("")),
                aAvailability(*new String("")){}

  classType isA() const
   {
    return restrauntEntryClass;
   }

  char _FAR *nameOf() const
   {
    return "RestaurantEntry";
   }

  int isEqual(const Object& e) const
   {
    return Item() == ((RestaurantEntry&)e).Item();
  }

 int isLessThan(const Object& e) const
  {
   return Item() < ((RestaurantEntry&)e).Item();
  }

 hashValueType hashValue() const
  {
   return aItem.hashValue()+aRate.hashValue()+
           aAvailability.hashValue();
  }

 void printOn(ostream& os) const
  {
   os<<aItem<<":"<<aRate<<":"<<aAvailability<<endl;
  }

 String& Item() const
  {
   return aItem;
  }

 String& Rate() const
  {
   return aRate;
  }

 String& Availability() const
  {
   return aAvailability;
  }

 ~RestaurantEntry()
  {
   delete &aItem;
   delete &aRate;
   delete &aAvailability;
  }
};


/* Class definition for the ReservationEntry class. Defines ReservationEntry
   objects to be stored in the BReservation B-tree. */

class ReservationEntry : public Sortable
 {
  String& aRoomNo;
  String& aName;
  String& aFrom;
  String& aTo;

  public:

  enum {reservationEntryClass = __firstUserClass+3};
  ReservationEntry(char *roomno,char *name,char *from,char *to)
 : aRoomNo(*new String(roomno)),aName(*new String(name)),
   aFrom(*new String(from)),aTo(*new String(to)){}

ReservationEntry() : aRoomNo(*new String("")),aName(*new String("")),
                aFrom(*new String("")),aTo(*new String("")){}

classType isA() const
 {
 return reservationEntryClass;
 }

char _FAR *nameOf() const
 {
 return "ReservationEntry";
 }

int isEqual(const Object& e) const
 {
 return RoomNo() == ((ReservationEntry&)e).RoomNo()
         && Name() == ((ReservationEntry&)e).Name()
         && From() == ((ReservationEntry&)e).From()
         && To() == ((ReservationEntry&)e).To();
 }

int isLessThan(const Object& e) const
 {
 return (RoomNo() < ((ReservationEntry&)e).RoomNo())
         || ((RoomNo() == ((ReservationEntry&)e).RoomNo())
            && (StringToDate((char *) *(To()))
               < StringToDate((char *) *(((ReservationEntry&)e).From()))));
 }

hashValueType hashValue() const
 {
 return aRoomNo.hashValue()+aName.hashValue()+
         aFrom.hashValue()+aTo.hashValue();
 }

void printOn(ostream& os) const
 {
 os<<aRoomNo<<":"<<aName<<":"<<aFrom<<":"<<aTo<<endl;
 }

String& RoomNo() const
 {
 return aRoomNo;
 }
String& Name() const
 {
 return aName;
 }

String& From() const
 {
 return aFrom;
 }
 String& To() const
  {
   return aTo;
  }

 ~ReservationEntry()
  {
   delete &aRoomNo;
   delete &aName;
   delete &aFrom;
   delete &aTo;
  }
};


/* Class definition for the BillEntry class. Defines BillEntry objects to
   be stored in the BBill B-tree. */

class BillEntry : Sortable
 {
  String& aName;
  String& aRoomNo;
  String& aBill;

  public:

  enum {BillEntryClass = __firstUserClass+4};
  BillEntry(char *name,char *roomno,char *bill)
   : aName(*new String(name)),aRoomNo(*new String(roomno)),
     aBill(*new String(bill)){}

  BillEntry() : aName(*new String("")),aRoomNo(*new String("")),
             aBill(*new String("")){}

  classType isA() const
   {
    return BillEntryClass;
   }

  char _FAR *nameOf() const
   {
    return "BillEntry";
   }

  int isEqual(const Object& e) const
   {
    return Name() == ((BillEntry&)e).Name()
             && RoomNo() == ((BillEntry&)e).RoomNo()
             && Bill() == ((BillEntry&)e).Bill();
   }

  int isLessThan(const Object& e) const
   {
    return Name() < ((BillEntry&)e).Name();
   }
```

```
hashValueType hashValue() const
 {
  return aName.hashValue()+aRoomNo.hashValue()+aBill.hashValue();
 }

void printOn(ostream& os) const
 {
  os<<aName<<":"<<aRoomNo<<":"<<aBill<<endl;
 }

String& Name() const
 {
  return aName;
 }

String& RoomNo() const
 {
  return aRoomNo;
 }

String& Bill() const
 {
  return aBill;
 }

~BillEntry()
 {
  delete &aName;
  delete &aRoomNo;
  delete &aBill;
 }
};

/* Function to search for a reservation. This function is used as an
   argument to the firstThat() function of BReservation Btree objects */

BOOL testMember(const class Object& x,void *)
 {
  Date& D1 = StringToDate(ReserveEnqData.FromEdit);
  Date& D2 = StringToDate(ReserveEnqData.ToEdit);
  Date& D3 = StringToDate((char *) *((ReservationEntry&)x).From());
  Date& D4 = StringToDate((char *) *((ReservationEntry&)x).To());
  int temp = atoi((char *) *((ReservationEntry&)x).RoomNo());
  if(GlobalRoomNo < temp)
   return TRUE;
  else if(GlobalRoomNo == temp)
   {
    if(D2.isLessThan(D3))
     return TRUE;
    else if(D4.isLessThan(D1))
     return FALSE;
    else
     {
      GlobalRoomNo++;
      return FALSE;
    }
   }
  return FALSE;
 }

 // Class declaration for the Application object HotelApp
class HotelApp : public TApplication
{
 public:

  HotelApp(LPSTR AName,HINSTANCE hInstance,HINSTANCE hPrevInstance,
           LPSTR lpszCmdLine,int nCmdShow);
  virtual void InitMainWindow();
};


// Class declaration for the main window class THotelMainWindow
class THotelMainWindow : public TWindow
 {
  public:

  char DateBuffer[10];
  char GuestBuffer[10];
  HelpWindow *HlpWindow;

  THotelMainWindow(PTWindowsObject AParent,LPSTR ATitle);
  BOOL CanClose();
  void HandleEnquiryMsg(RTMessage Msg) = [CM_FIRST+CM_ENQUIRY];
  void HandleResvMsg(RTMessage Msg) = [CM_FIRST+CM_RESERVATION];
  void HandleCancelMsg(RTMessage Msg) = [CM_FIRST+CM_CANCELLATION];
  void HandleFoodMsg(RTMessage Msg) = [CM_FIRST+CM_FOODMENU];
  void HandleBarMsg(RTMessage Msg) = [CM_FIRST+CM_BARMENU];
  void HandleGuestEnqMsg(RTMessage Msg) = [CM_FIRST+CM_ROOMPERSONENQ];
  void HandleCheckInMsg(RTMessage Msg) = [CM_FIRST+CM_CHECKIN];
  void HandleCheckOutMsg(RTMessage Msg) = [CM_FIRST+CM_CHECKOUT];
  void HandlePrepareBillMsg(RTMessage Msg) = [CM_FIRST+CM_PREPAREBILL];
  void HandleDateMsg(RTMessage Msg) = [CM_FIRST+CM_DATE];
  void HandleChangeAvailMsg(RTMessage Msg) = [CM_FIRST+CM_AVAILABILITY];
  void HandleAboutMsg(RTMessage Msg) = [CM_FIRST+CM_ABOUT];
  void HandleHelpMsg(RTMessage Msg) = [CM_FIRST+CM_HELP];
 };

// Class declaration for the TReserveEnqDialog class
class TReserveEnqDialog : public TDialog
 {
  public:

  TReserveEnqDialog(PTWindowsObject AParent,LPSTR AName,PTModule AModule);
  ~TReserveEnqDialog();
 };

// Class declaration for the TReserveDialog class
class TReserveDialog : public TDialog
 {
  public:
```

```
 TReserveDialog(PTWindowsObject AParent,LPSTR AName,PTModule AModule);
 ~TReserveDialog();
};

// Class declaration for the TCancelDialog class
class TCancelDialog : public TDialog
{
 public:

 TCancelDialog(PTWindowsObject AParent,LPSTR AName,PTModule AModule);
};

// Class declaration for the TFoodDialog class
class TFoodMenuDialog : public TDialog
{
 public:

 PTListBox ListBox;
 PTListBoxData ListBoxData;
 TFoodMenuDialog(PTWindowsObject AParent,LPSTR AName,PTModule AModule);
 ~TFoodMenuDialog();
};

// Class declaration for the TBarMenuDialog class
class TBarMenuDialog : public TDialog
{
 public:

 PTListBox ListBox;
 PTListBoxData ListBoxData;
 TBarMenuDialog(PTWindowsObject AParent,LPSTR AName,PTModule AModule);
 ~TBarMenuDialog();
};

// Class declaration for the TCheckInDialog class
class TCheckInDialog : public TDialog
{
 public:

 TCheckInDialog(PTWindowsObject AParent,LPSTR AName,PTModule AModule);
 ~TCheckInDialog();
};

// Class declaration for the TCheckInDialog class
class TCheckOutDialog : public TDialog
{
 public:

 TCheckOutDialog(PTWindowsObject AParent,LPSTR AName,PTModule AModule);
 ~TCheckOutDialog();
};

// Class declaration for the TPrepareBillDialog class
class TPrepareBillDialog : public TDialog
{
 public:
```

```
 TPrepareBillDialog(PTWindowsObject AParent,LPSTR AName,PTModule AModule);
 ~TPrepareBillDialog();
};

// Class declaration for the TChangeAvailDialog class
class TChangeAvailDialog : public TDialog
{
 public:

 PTListBox ListBox;
 PTListBoxData FoodListData,DrinkListData;
 TChangeAvailDialog(PTWindowsObject AParent,LPSTR AName,PTModule AModule);
 ~TChangeAvailDialog();
 void HandleFoodButtonMsg(RTMessage Msg) = [ID_FIRST+ID_FOODBTN];
 void HandleDrinkButtonMsg(RTMessage Msg) = [ID_FIRST+ID_DRINKBTN];
 void HandleListBoxMsg(RTMessage Msg) = [ID_FIRST+ID_ALISTBOX];
};
```

BIBLIOGRAPHY

[1] Aaron Marcus and Andries Van Dam
"User-Interface Developments for the Nineties", IEEE Computer, September, 1991.

[2] Alan Synder
"The essence of Objects : Concepts and Terms", IEEE Software, January, 1993.

[3] Bjarne Stroustroup
"The C++ Programming Language", Addison Wesley Publishing Company, 1993.

[4] Dijkstra E
"Programming considered as a Human Activity", *Classics in Software Engineering*, Yourdon Press, 1979.

[5] Gary Syck
"Object Windows How - To", Galgotia Publications Private Limited, 1993.

[6] Grady Booch
"Object Oriented Design With Applications", The Benjamin/Cummings Publishing Company,Inc., 1991.

[7] Henry F. Korth, Abraham Silberschatz,
"Database System Concepts", Mc Graw Hill Publishing Company, 1993.

[8] James Conger
"Windows API Bible", Galgotia Publications private limited, 1994.

[9] Jim Conger
"Windows Programming Primer Plus", Galgotia Publications Private Limited, 1994.

[10] Preece P
"An Introduction to Graphical User Interfaces(GUIs)", *Object Oriented Programming Systems : Tools and Applications*, Chapman & Hall, 1991.

[11] Raimund K. Ege and Christian Stary
"Designing Maintainable, Resusable Interfaces", IEEE Software, November, 1992.

[12] Robert Lafore
"Object Oriented Programming In Turbo C++", Galgotia Publications Private Limited, 1994.

[13] Setrag Khoshafian and Razmik Abnous
"Object Orientation Concepts, Language, Database, User Interfaces", John Wiley & Sons Inc., 1990.

[14] Stanley Lippman
"C++ Primer", Addison Wesley Publishing Company, 1991.

[15] Vijay Mukhi
"C++ and Graphics", *C Odyssey*, *Vol 1*, BPB Publications, 1993.

[16] Vijay Mukhi
"Borland C++ 3.0 for Windows 3.1", BPB Publications, 1993.

[17] Borland C++ 3. 1 User's Guide, Borland Inc.

[18] Borland C++ 3.1 Programmer's Guide, Borland Inc.

[19] ObjectWindows for C++ User's Guide, Borland Inc.