J. King Saud Univ., Vol. 12, *Comp. & Info. Sci.*, pp. 1-43 (A.H. 1420/2000)

Transforming an Imperative Design into an Object-Oriented Design

Abad Shah and Hassan Mathkour
Computer Science Department, King Saud University
P.O. Box 51178, Riyadh 11543, Saudi Arabia

(Received 11 May 1997; accepted for publication 15 September 1998)

Abstract. Most of the traditional and legacy systems were designed using traditional methodologies such as Structured Analysis/Structured Design (SA/SD) methodology. Design of such a system is called an *imperative design*. After the introduction of the object-oriented technology, there are compelling reasons to redevelop those systems using this new technology to benefit from its merits. To redevelop them, there are two possible choices: either develop them from scratch using some object-oriented methodology, or use the available design documents (i.e., imperative design) of those systems and transform their designs into object-oriented designs. The second choice clearly results in saving both the development cost and time.

This paper reports on an effort to build support for the second choice mentioned above. We started our effort in 1992 and proposed a framework of a redesign methodology. Our proposed redesign methodology, i.e., imperative design to object-oriented design (ID-OOD), transforms a given imperative design of an already implemented system into an object-oriented design using the design documents of the system. The methodology works in four phases and they are presented formally. We also illustrate the methodology with a case study.

Keywords: Object-oriented design, Imperative design, ID-OOD methodology, Entity relationship, Data flow diagram, Redesign methodology.

1. Introduction

Numerous information systems comprising very many lines of code were developed during, and before, the 1970s using traditional software development methodologies such as Structured Analysis/ Structured Design (SA/SD), Structured Analysis and Design Technique (SADT; was developed by SofTech), Jackson Structured Development (JSD) etc. [1, 2, 3]. Most of these systems were developed using SA/SD methodology due to its rising popularity and maturity. Therefore, many companies used it for software development. Main emphasis of SA/SD methodology is on the system functionality, hence the approach of this methodology is referred to

as a *functional approach.* Due to its popularity and extensive use, it is considered as a representative of the imperative design methodologies. Design document of a system that is developed using SA/SD methodology is called an *imperative design* (ID) of the system.

At some point in the life of those legacy systems there will be a need for redesigning and redeveloping them into an object-oriented design. For example, the US Department of Defense has many large information systems that were developed using COBOL and a traditional design methodology with over a million lines of code. Now, with the mandate from the US Department of Defense to use Ada, and with the arrival of Ada 9X with it's object-oriented approach, there will be an increasing demand for the use of this design methodology.

The object-oriented approach has influenced all disciplines of computer technology due to its features such as a unifying and modular design to integrate the different phases of software development process, capability to cope with evolutionary changes more easily, and encouragement of code reusability [4].

The principle of *aggregation* in a functional approach is to group together functions that are constituents of a higher level function implementation. In the object-oriented approach, the principle of aggregation groups together functions that operate on the same set of data. Functions share data in the functional approach, whereas data is changed by the functions through message-passing feature of the object-oriented approach. In short, the object-oriented approach is considered superior to the functional approach in many respects [5].

Several object-oriented software development methodologies are proposed such as Object Modeling Technique (OMT), Booch methodology, Yourdon methodology and many more [6,7,8,9,10,11,12]. These methodologies suggest different methods for object-oriented analysis and design. A comprehensive comparison and study of six most popular object-oriented software development methodologies, is available in [13]. Designing of a system using an object-oriented methodology is referred to as an *object-oriented design* (OOD).

Among researchers, there is a difference of opinion on the compatibility of the imperative (functional) paradigm and the object-oriented paradigm, and transformation of an ID into an OOD [1,7,8,14,15]. We advocate the compatibility of these two paradigms and the transformation of the design from one paradigm to another for the following similarities between them:

(i) An entire system can be represented by a meta-class in an OOD and by a meta-data-flow diagram in an ID. A data flow diagram and a class are both refined using the top-down approach and specialization abstraction.

(ii) During the process of refinement, a system hierarchy emerges in the object-oriented approach, which is referred to as a *class-hierarchy* or a *class-lattice*. In the imperative (or traditional) approach, the refinement process is referred to as *functional decomposition*, and results into a *data flow diagram* (DFD).

If it becomes desirable to redevelop a system designed using SA/SD methodology into a system using object-oriented technology, then there are two possible choices to handle this situation. They are:

(i) use the available ID of the system as input to a redesigning process to transform the ID into an OOD of the system.
(ii) discard the available ID of the system and develop a new OOD from the scratch using some object-oriented software development methodology.

The second alternative needs a new analysis and design effort as it does not derive any help from the available ID of the system. This choice puts an additional burden on the development cost and time of the new system. Whereas, by adopting the first choice we can save a considerable amount of development cost and time. In this paper, we advocate the first choice and propose a methodology that transforms a given ID of a system into an OOD of the system. We call our proposed methodology as Imperative Design-Object-Oriented Design (ID-OOD) methodology.

In 1992, we initiated our effort in this direction and we proposed a framework of the ID-OOD methodology [16]. In our preliminary work we proposed the methodology with three phases. In 1994, we improved the framework of the methodology and added the fourth phase. This improvement is reported in [17]. Now, we have further improved the methodology and its four phases, and present the redesign methodology in a formal way. We also illustrate the methodology with a case study.

The remainder of this paper is organized as follows. In Section 2, we give the background and related work. In Section 3 we give the architecture and formal working of the ID-OOD methodology and explain its four phases. To illustrate the working of the methodology, we present a case study of a Hotel Management System in Section 4. Finally, in Section 5, we give our concluding remarks and future directions of this work.

2. Background and Related Work

The classical life-cycle of the software development process, called the *Water-Fall Model*, consists of five main steps (or phases): *requirement analysis*, *design*, *coding*, *testing*, and *maintenance*. These steps are sequential and iterative [18]. The analysis step is the first step towards a computerized solution of a system. During this step, we understand a system and prepare a document of our understanding about the system by

capturing its characteristics. In the design step, we take the analysis document and translate it to prepare a software architecture, data structures, details of the processes, input and output design of the system. We use different models and techniques in these two steps to visualize and specify the system and its characteristics, and make them understandable to programmers.

Most of the earlier software development methodologies used the classical life-cycle and the imperative approach. Among those methodologies, SA/SD methodology is a mature and widely used methodology [1,3]. Most of the legacy systems were developed using SA/SD methodology. Several software tools are commercially available to support this methodology. In this paper, we call the analysis and design document of a system as an *imperative design* (ID) if the system is designed by using SA/SD methodology. During the design of an ID various notations and models are used to formally visualize and specify the system and its characteristics. They are *data flow diagram* (DFD), *data dictionary*, and *entity-relationship diagram* (ERD). In this paper, we assume that in an ID document these notations are used. In the next two sections we describe these notations.

2.1 Data flow diagram (DFD)

A DFD is defined as an abstract data type graph whose nodes represent functions (or processes) and its edges represent data flows [10]. A DFD specifies data sources, data sinks, data stores, data transformations, and the flow of data between sources and stores. A data store is a conceptual data structure in the sense that physical implementation details are suppressed; only the logical characteristics of data are emphasized in the DFD. In a DFD, major emphasis is on the overall functional decomposition; emphasis on data stores is very weak. To show the details of what information is being transferred and how it is being transferred, two other tools - called the data dictionary and process specification, are used.

In a DFD, a bubble (or rectangle) represents a function (or process), an arrow towards the bubble represents input to the function and an arrow out of the bubble represents an output of the function. A function is a set of code which takes a set of input data values, and operates on them, and produces a set of data values as output. Several variations of the DFD are proposed and available in the literature.

2.2 Entity-relationship model

The Entity-Relationship (ER) model is a semantic data model which partitions an application domain into a set of *entities*, and the entities are related to each other by a set of *relationships* [19]. An entity is a representation of an object of a real-world. The entity type and the relationship type are defined by a set of attributes which define their structures.

The ER model has graphical power to model a logical and conceptual view of a system and its objects, and is referred to as an *entity-relationship diagram* (ERD) of the system. An ERD highlights relationships among data stores of a system, and each element of the ERD corresponds to one data store of a DFD. This diagram provides a powerful graphical design tool which helps a designer during the analysis and the design phases of a system development.

2.3 Object-oriented paradigm

The term "object" is not a new term to the computer community. It was first introduced in the programming language Simula in 1966 [20]. This term might have been used even before 1966, and according to Berard it was used in late 1940s or early 1950s in some programming language [21]. In the language Simula other object-oriented concepts like encapsulation and inheritance were also introduced.

In the object-oriented approach, an object is defined by the two parameters: *structure* and *state*. The *structure* of an object provides the structural and behavioral capabilities to the object, which is defined by a set of instance variables and methods. The *state* of an object assigns data values to the instance variables of the objects and the methods operate on them. A set of objects sharing the same structure is referred to as a *class*. An object-oriented design (OOD) is a collection of classes which are organized as a Directed Acyclic Graph (DAG) or a simple graph which is called a *class-hierarchy* or *class lattice*. Classes in a class-hierarchy are related into *parent-child* relationship where a child class is referred to as a *subclass,* and a parent class is referred to as a *super-class*. A class-lattice of a system defines a message-passing pattern among the objects of the system. A subclass inherits the structure of its super-class(es). The inheritance increases reusability in a system [22, 23].

As said earlier, several object-oriented software development methodologies are proposed by different researchers. The result of object-oriented analysis and design phases of these methodologies is a design document which is called an *object-oriented design* (OOD). An OOD document consists of 1) a system of class objects that specify all functional requirements, 2) details of structure and behavior of the class objects of the system, and 3) a message-passing pattern *via* a class-lattice [24].

Main focus of the traditional methodologies such as SA/SD is on decomposition of system functionality. According to Rumbough et al the approach of these traditional methodologies is the most direct and implementation-oriented. If functional requirements of a system change, then the design of the system needs a heavy duty restructuring. Therefore, the ID is considered as an unstable design [10].

On the other hand, object-oriented software development methodologies focus on identifying the class objects of a system, and then the methods (or functions) are attached to the identified class objects. This class of methodologies is flexible in

adopting new functional requirements of a system. In other words, new functional requirements of a system can be incorporated in the OOD without making any major restructuring of the design.

2.4 Efforts toward redesigning process

By the term *redesign* we mean that if imperative design (ID) document of a system is already available, then a transformation methodology that transforms the ID into an OOD is referred to as a *redesign methodology*. We will use the terms redesign and transformation interchangeably.

As we have mentioned earlier, there is a disagreement among the researchers on the compatibility of the traditional paradigm and the object-oriented paradigm, and they are divided in two groups. The first group believes that these two paradigms are not compatible, therefore the transformation of an ID into an OOD is not possible accurately [7,8,14,25]. The second group argues in favor of the compatibility of these two paradigms, and they propose frameworks for transformations [15,16,26,27,28]. The researchers, Shumate and Alabiso, in their separate redesign proposals made DFDs and data stores the basis of their transformation processes [15,26]. They did not consider the ERD a component of ID in their proposals. Also, their proposals are not presented in a formal way (for details see [1, 28]). Whereas, many object-oriented software development methodologies such as OMT consider ERD an important system diagram that provides relationships among the objects [10, 29].

We started our effort to develop a redesign methodology in 1992 and reported the preliminary work of the methodology in [16,17]. In our redesign methodology, we consider ERD of a system as an important component of the system's ID. Also, we identify four different phases of our redesign methodology, i.e., the ID-OOD methodology, and formally present its four phases.

3. ID-OOD Methodology

As mentioned earlier, we consider ERD of a system as an essential part of an ID, because it gives a logical view of the system. We identify different types of relationships among entities in an ERD. They are described as follows:

- Weak-relationship: If entities of a relationship have no attribute in common, then the relationship is referred to as a *weak-relationship*.
- Semi-relationship: If a relationship between two entities is of type *'part-of'* [23], then the relationship is referred to as a *semi-relationship*.
- Strong-relationship: If a relationship between two entities has many attributes in common, then the relationship is referred to as a *strong-relationship*.

The above relationships are used in the ID-OOD methodology while defining different links (or relationships) among the classes of a class-hierarchy.

The objective of the ID-OOD methodology is to utilize the available design documents of a given ID to transform it into an OOD. This transformation can save both the development time and cost. A high level view of the methodology is shown in Fig.1.

The ID of a system has two main components: Data Flow Diagrams (DFDs), and a conceptual diagram (control flows or context diagram) of the system [1]. In the ID-OOD methodology, we use ERD to get the first realization of classes and their logical links. But, the ERD of a system lacks functional information of entities and their relationships. This transformation comes from DFDs of the system.

The ID-OOD methodology works in four phases in order to transform a given ID into an OOD. The OOD contains the components mentioned in Section 2.3, and the OOD is functionally equivalent to the ID. Fig. 2 depicts the ID-OOD methodology and its four phases. The four phases of the methodology are described in the following sections. Note that we use the terms attribute and instance-variable in the traditional paradigm and object-oriented paradigm, respectively, otherwise these terms are interchangeable.

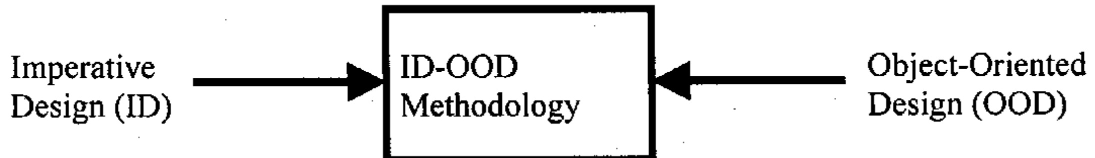

Fig. 1. A high-level view of the ID-OOD methodology.

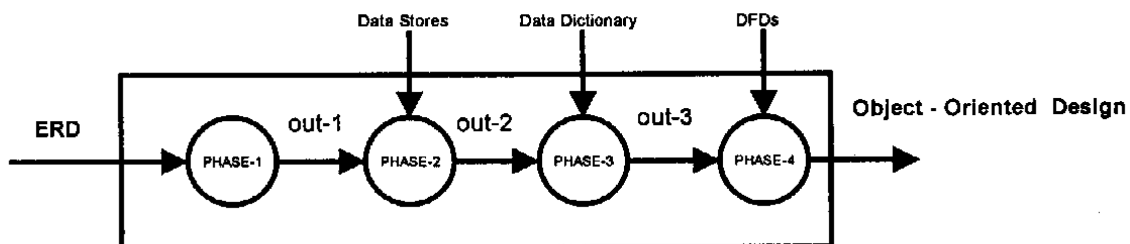

Fig. 2. A detailed view of the ID-OOD methodology.

3.1 PHASE-1: Using ERD to build preliminary-class-hierarchy

This phase takes the ERD of a given ID and transforms them into a *preliminary-class-hierarchy*. A preliminary-class-hierarchy (PCH) is the first skeleton of classes and their relationship links. In many object-oriented software development methodologies, during analysis phase, *nouns* are picked up from the problem statements as potential classes and the *verbs* are considered as potential methods of the classes [24,29,30]. We also take entity names of the ERD as class names of a PCH. We outline the following procedure to transform the ERD of the ID into a PCH.

Procedure PHASE-1 (ERD): PCH

Step 1: Make each entity as a class and attributes of the entity as instance-variables of the class.

Step 2: If a relationship and the entities involved in the relationship have some attributes in common, then create a new class as a subclass of the classes which are involved in the relationship, and put the common attributes in the new class.

Step 3: Link the classes that are classified as classes in Step 1 as follows:

- **a.** create a system class as the *root class* of the PCH, and
- **b.** connect all classes as subclasses to the root class.

Step 4: Convert each relationship between two entities into a link between the two classes of the PCH by using the *rules* that are defined as follows:

- **R1-** If a relationship between two entities is a weak-relationship, then make the relationship as an *associations- relationship* between the two classes.
- **R2-** If a relationship between two entities is a semi-relationship, then make the relationship as an *aggregation-relationship* between the two classes.
- **R3-** If a relationship between two entities is a strong-relationship, then make the relationship as *a generalization-relationship* between the two classes.

Step 5: Convert the cardinality of each relationship between two entities, generally represented as $1 : 1$, $1 : M$, $M : 1$, and $M : N$, into the multiplicity of object-oriented design.

Step 6: If the root class is connected directly to only one class, then make the class that is connected to the root class as the root class of the PCH; otherwise do nothing.

Step 7: If two or more classes of the PCH have some or all attributes in common, then create a new class as a super-class of all these classes and put those common attributes in the super-class.

End Procedure PHASE-1

The full cardinality constraints which are maintained in Step 5 of the above procedure, are also maintained in most of object-oriented design methodologies [6, 13]. The output of PHASE-1 is denoted by *out-1* in Fig. 2.

3.2 PHASE-2: Using data stores structures to add classes to the PCH

PHASE-2 adds more classes (if possible) to the PCH which is the output of PHASE-1, by using data stores of DFDs of the ID. Note that PHASE-2 deals only with

attributes of data store, not the data itself. The input of PHASE-2 is the data stores of the DFDs and the PCH (see Fig. 2). The output of PHASE-2 is the PCH with some additional classes. The following procedure describes PHASE-2.

Procedure PHASE-2 (attributes of data stores & PCH): PCH with additional classes

Step 1: Create a new class for each data store of a DFD and the attributes of data store as the instance-variables of the new class.

Repeat

Step 2: Compare each newly created class with every class of the PCH and do as follows:

IF all the instance-variables of the new class are the same as the instance-variables of an existing class in the PCH
THEN ignore the new class;
ELSE IF the instance-variables of the new class partially match instance-variables of an existing class of the PCH
THEN add the new class as a super-class of the class in the PCH;
ELSE IF the set of instance-variables of the new class is subset of the set of instance-variables of a class of the PCH
THEN do the following two steps in a sequence:

1. create a new class, which includes the common instance-variables between them, and
2. connect the new class and the existing class of the PCH as subclasses of the newly created class in the PCH.

ELSE IF all the instance-variables of a class of the PCH and the new class are same
THEN add the new class as a subclass of the existing class;
ELSE IF the set of instance-variables of the new class and an existing class of the PCH are mutually disjoint
THEN add the new class as a subclass of the root class.

Step 3: Establish the multiplicity between the new class and the classes connected with the newly created classes in the PCH.

Until all new classes are exhausted.

End PHASE-2

3.3 PHASE-3: Using data dictionary to place instance-variables in the classes

The output of PHASE-2 is the PCH with classes containing incomplete sets of instance-variables and without methods. In PHASE-2 we have used the attributes of data stores to add new classes to the PCH, but we did not add those attributes as instance-variables to the classes. Methods will be added later to the classes in PHASE-4. PHASE-3 adds instance-variables to the classes using the data dictionary of the ID. This phase also provides name, data type and the location of the instance-variables of each class. The following procedure describes PHASE-3. Its output which is denoted by out 3 in Fig.2, is the PCH. After this phase the PCH classes contain complete sets of instance-variables.

Procedure PHASE-3 (data Dictionary): PCH

Repeat

Step 1: Take each attribute name and data type from the data dictionary. If an attribute name matches with instance-variable name of any PCH, then make as the same type of the instance-variable in the class.

Step 2: Pick up the location of every instance-variable of each class from the data dictionary.

Until all attributes in data dictionary are exhausted

End PHASE-3

3.4 PHASE-4: Using Processes from DFDs to place methods in the classes

This phase adds methods from the DFDs to the classes of the PCH which is the output of the previous phase. We assume that the DFDs of the ID are decomposed to their most primitive form (i.e., each function in each DFD performs only one single function). The following procedure describes PHASE-4.

Procedure PHASE-4 (DFDs): final class-lattice (OOD)

Repeat

Step 1: Consider each function of a DFD as a method of some class, input to the function as arguments of the method (in OOD we call a function as a method), and the output (or outputs) of the function as outputs of the method.

Step 2: Add the method to an appropriate class, using one of the following rules:

R1. Compare the arguments and output of the method with the instance-variables of each class of the PCH, put the method in that class which has a maximum match.

R2. Add the method to the class that was created from the corresponding data store.

R3. Examine the *functionality* of the method with the class functionality and add the method to an appropriate class (e.g., if we have 'Update_Age' method, then it will be added to 'Person' class, because the 'Age' attribute is a basic characteristic of the 'Person' class).

Until all DFDs are exhausted

Repeat

Step 3: Relocate each method in an appropriate class. There are two possible relocation cases, which are described below.

Case 1: If a method M_1 needs input from j different methods and gives the output to a single method (see Fig. 3), then create a new method in the class between the input and the method M_1. The new method is referred to as a *coordinator method*. When the method M_1 is invoked, the *coordinator method* is also activated to collect all inputs for the method M_1.

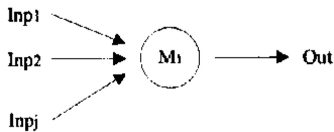

Fig. 3. Method M_1 receives inputs from *j* different methods and gives one output.

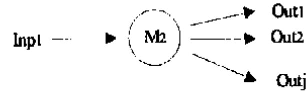

Fig. 4. Method M_2 receives one input and gives outputs to *j* different methods.

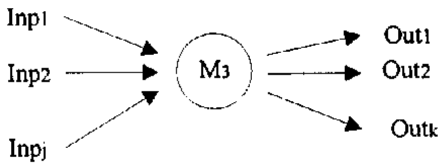

Fig. 5. Method M_3 receives inputs from *j* different methods and gives outputs to *k* different methods.

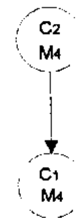

Fig. 6. Super-class and it's subclass have the same method.

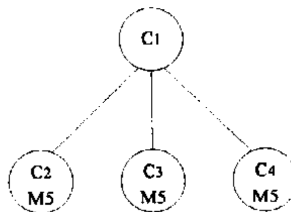

Fig. 7. Subclasses that have the same super-class.

The main function of the coordinator method is to fetch all necessary and correct arguments from different methods and feed them to the method M_1 before it starts its execution. It is also the responsibility of the coordinator method to give output of the method M_1 to the appropriate method (methods) after each successful execution of the method M_1. Concurrency control is also the responsibility of the coordinator method when more than one object is active at the same time in a class. The nature of the coordinator method is different in the subsequent cases in terms of input and output.

Case 2: If a method M_2 of a class receives a single instance-variable as input and gives the output to *j* methods (see Fig. 4), then create a coordinator method between the method M_2 and the output, and add the coordinator method to the same class to which the method M_2 belongs. When the method M_2 sends its outputs to *j* different methods, then the coordinator method is activated and passes the outputs to the appropriate methods which may belong to different classes.

Case 3: If a method M_3 needs input from *j different* methods and gives its outputs to *k* different methods (see Fig. 5), then create two coordinator methods MC_3 and MC_4 between the method M_3 and its inputs and the methods M_3 and its outputs, respectively. The coordinator methods will be added to the same class to which the method M_3 belongs. When a message is received in the class for invocation of the method M_3, the coordinator method MC_3 is activated to collect all necessary inputs for the method M_3. After completing each execution of the method M_3, the second coordinator method MC_4 is activated to pass on the outputs to *k* methods. The *k* methods may belong to different classes.

Case 4: If a method M_4 is already present in both the super-class and its subclass (see Fig. 6), then place the method M_4 only in the super-class, and the subclass will inherit the method.

Case 5: If a method M_5 lies in all t subclasses of a super-class (see Fig. 7), then place the method M_5 only in the super-class and all the subclasses will inherit the method.

Step 4: Construct the *Scenario* of interactions between methods from a DFD. A scenario is defined as a sequence of methods that occurs when a part of a system operates, or we can say that, a scenario is a sequence of methods for an operation (or a function). Note that an operation is a group of methods.

Until no more relocation is needed

End PHASE-4

The methodology gives a class-lattice with complete details of its classes as an OOD of a given ID (see Fig. 2).

4. Case Study

In this section, we present a case study to illustrate the ID-OOD methodology. For this purpose we take the ID of a hotel management system. This system was designed and implemented in the College of Computer and Information Sciences, King Saud University [31].

Hotel management is one of the fastest growing business industry in the modern era. The rapid development of the technology makes it more challenging and demanding. The activities of the hotel management system not only involve the processing of daily transactions of *occupancy* or *vacancy* of rooms, ordering and receiving room services, but it also involves the performance evaluation of the business in general and the evaluation of the services in particular. This can be made possible by generating reports

by the hotel information system. This case study was done for Salahudin Hotel in Riyadh to automate its manual hotel management system. The system concentrates on the major activities, such as reception, room reservations, occupancies, restaurant, laundry, etc. The services and facilities that are provided to guests are explained in the ID. The ID of the hotel management system is given in Appendix.

We transform the ID of the hotel management system into an OOD using ID-OOD methodology in the next four sections.

4.1 PHASE-1: Using ERD to build preliminary-class-hierarchy

Step 1: From the ERD of the hotel system (see Fig. A-1 in Appendix), the entities *Hotel, Guest, Account, Restaurant, Room, Villa, Comm-Ser* (Communication Services), *Laundry, FAX, Telex, Large-Villa, Small-Villa, Single-Room, Double-Room, Suite-Room*, and *Royal-Room* are identified as isolated classes (i.e., the classes without links).

Step 2: In the ERD of the system, the *eat-in* relationship between the classes ***Guest*** and ***Restaurant*** have three instance-variables: Meal-Date, Meal-Desc and Meal-Cost in common. Create a new class, ***eat-in*** as subclass of these two classes.

Step 3: Create a system class and connect the classes that are identified in Step 1 to the system class as shown in Fig. 8 and call it *Preliminary-Class-Hierarchy* (PCH).

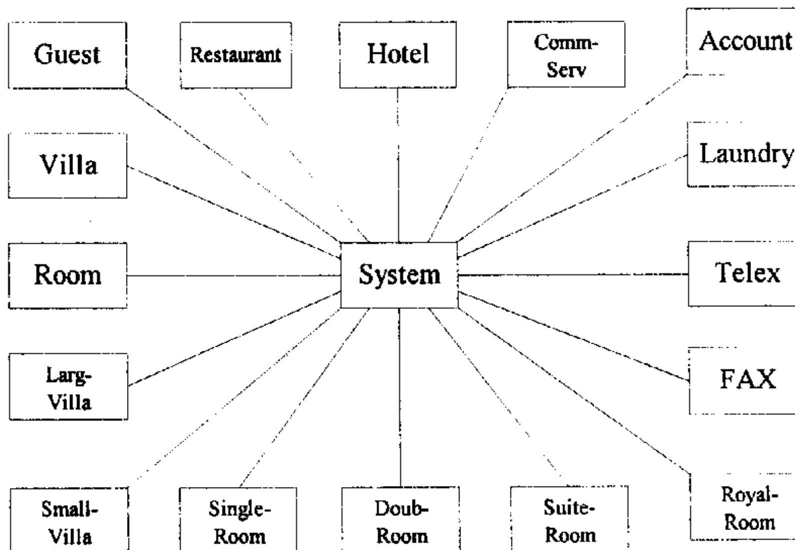

Fig. 8. The first diagram of the PCH of the system (step 2).

Step 4: Convert each relationship between two entities into an equivalent relationship in the PCH.

1. The *live-in* relationship which is between the entities *Hotel* and *Guest* (see Fig. A-1) is converted into association-relationship because this is a weak relationship, since the guest lives temporarily in the hotel.
2. The *part-of* relationships between the entity *Hotel,* and the entities *Account, Restaurant, Villa, Room, Comm-Ser,* and *Laundry* (see Fig. A-1)are converted into the aggregation-relationships, because these classes ***Account, Restaurant, Villa, Room, Comm-Ser,*** and ***Laundry*** take part in the construction of the ***Hotel*** class. Note that we denote class names in bold and italic.
3. The *is-a* relationships between the entity *Room,* and the entities *Single-Room, Double-Room, Suite-Room* and *Royal-Room* (see Fig. A-1) are converted into generalization relationships because they are inheritance relationships.
4. The *is-a* relationships between the entity *Villa,* and the entities *Small-Villa* and *Large-Villa* are converted into generalization relationships because they are inheritance relationships.
5. The *is-a* relationships between the entity *Comm-Ser,* and the entities *FAX* and *Telex* are converted into generalization relationships because they are inheritance relationships.
6. The *eat-in* relationship between Guest and Restaurant is converted into the association relationship because this is a weak-relationship.

Fig. 9 depicts the PCH after incorporating the above relationships.

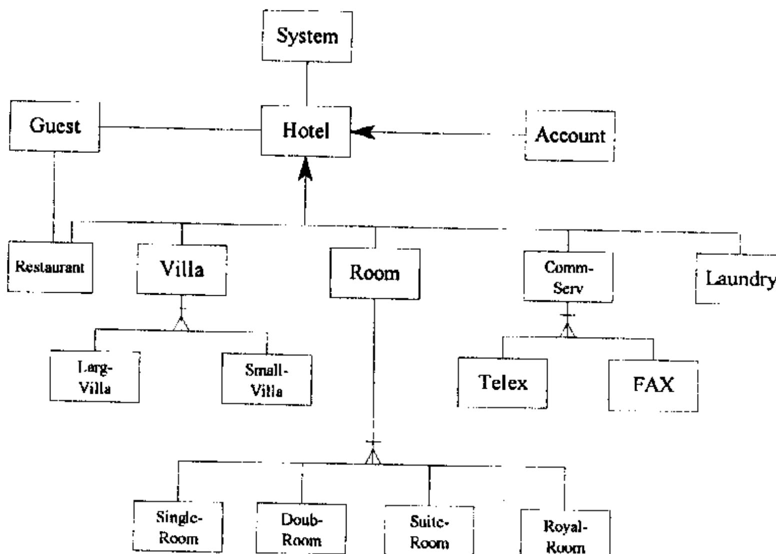

Fig.9. PCH after converting the ERD relationships into the relationships of OOD (step 3).

Step 5: **Convert** the cardinality of each relationship between the entities of the ERD into multiplicity between the classes (see Section 3.1).

Step 6: In Fig. 10 the ***system*** class is connected only to the ***Hotel*** class, therefore, the ***Hotel*** class will become the ***root*** or ***system*** class of the PCH.

Step 7: This step is not applicable for the given design ID because the PCH classes that share the same instance-variables, such as ***Large-Villa*** and ***Small-Villa***, have their own super-classes. Fig. 11 shows the resultant PCH at the end of this phase.

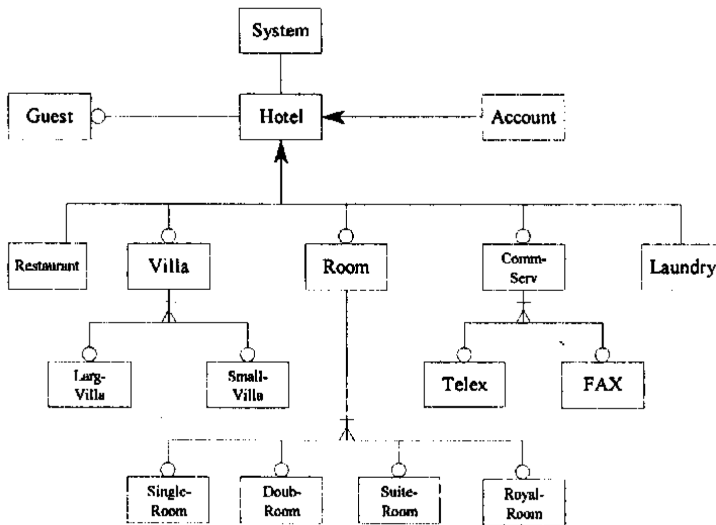

Fig. 10. PCH after adding multiplicity to the classes.

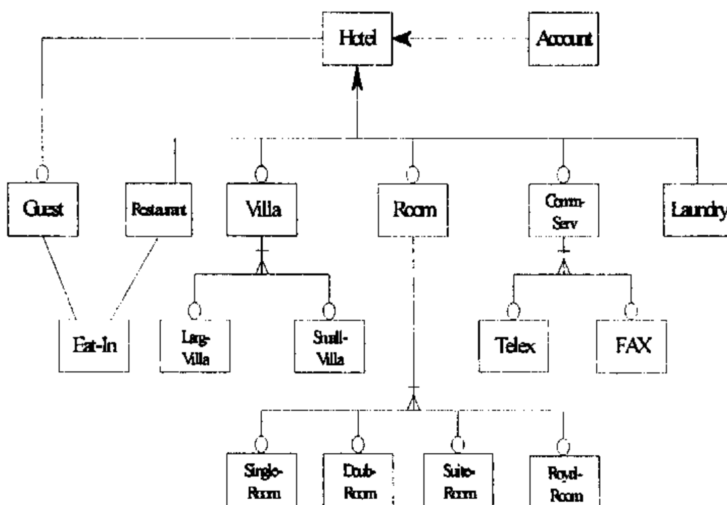

Fig. 11. PCH after completion of PHASE-1.

4.2 PHASE-2: Using data-stores to add classes to the PCH

PHASE-2 adds additional classes (if possible) to the PCH using data stores from DFDs of the ID. Fig. A-2 through Fig. A-21 are the DFDs of the ID. The data stores and their attributes are listed below:

1. Room-info: Room-No, Room-Type, Room-Status, and Room-Price
2. Villa-info: Villa-No, Villa-Type, Villa-Status, and Villa-Price
3. Gust-info: Guest-No, Guest-Name, Guest-Addr, Arrival-Date, and Arrival-Time
4. Account: Mode-Pay, Invoice-No, Invoice-Desc, and Invoice-Type
5. FAX/Telex: Rate-Per-Minute
6. Laundry: Clean-Cost

Now we use the procedure given in PHASE-2 (see Section 3.2 for details) to add new classes to the PCH shown in Fig. 11.

Step1: Consider each data store in the DFDs as a class. Therefore, all six data-stores that are listed above can be considered as new classes of the PCH.

Table 1. Instance variables of the PCH

Instance-Variable Name	Instance-Variable Type	Instance-Variable Location
Room-No	Digits	Room
Room-Status	One Char	Room
Room-Type	One Char	Room
Room-Price	Digits	Room
Villa-No	Digits	Villa
Villa-Status	One Char	Villa
Villa-Type	One Char	Villa
Villa-Price	Digits	Villa
Guest-No	String	Guest
Guest-Name	Char	Guest
Guest-Addr	Char	Guest
Arrival-Date	Date	Guest
Arrival-Time	Time	Guest
Call-No	Digits	FAX
Destination	Char	FAX
Duration	Digits	FAX
FAX-Cost	Float	FAX
Rate-Per-Minute	Float	FAX
Meal-Date	Date	Eat-In
Meal-Desc	Char	Eat-In
Meal-Cost	Float	Eat-In
Call-Date	Date	FAX
Call-Time	Time	FAX
Telex-No	Char	Telex
Telex-Cost	Float	Telex
Telex-Date	Date	Telex
Telex-Time	Time	Telex
Clean-Cost	Float	Laundry
Clean-Desc	Char	Laundry
Hotel-Name	Char	Hotel
Hotel-Addr	Char	Hotel
Clean-Date	Date	Laundry
Mode-Pay	Two Char	Account
Invoice-No	Digits	Account
Invoice-Desc	Char	Account
Invoice-Type	One Char	Account

Step 2: Compare each new class with the classes of the PCH (in Fig. 11) as follows:

1. ***Room-info*** class: by comparing the instance-variables of the class ***Room-info*** with the classes of PCH, it is found that they match with the instance-variables of the ***Room*** class of the PCH. Therefore, the new class ***Room-info*** is ignored.
2. ***Villa-info*** class: by comparing the instance-variables of the class ***Villa-info*** with the classes of PCH, it is found that they match the instance-variables of the ***Villa*** class of the PCH. Therefore, the new class ***Villa-info*** is ignored.
3. Similarly, on the same ground the new classes ***Guest-Info***, ***Account***, ***FAX/Telex*** and ***Laundry*** are also ignored. Therefore, PHASE-2 does not add any new class to the PCH of Fig. 11.

4.3 PHASE-3: Using data dictionary to place instance-variables in the classes

As mentioned before, PHASE-3 uses the data dictionary (given in Fig. A-22 of Appendix) of the ID and adds instance-variables to the classes of PCH.

- **Step 1** The Room-No attribute is considered as instance-variable.
- **Step 2** The data type of Room-No is an integer type in the data dictionary.
- **Step 3** The Room class is the location for Room-No instance-variable.

The above three steps are repeated for all attributes of the data dictionary of the ID, and Table 1 shows the output of the phase.

4.4 PHASE-4: Using processes from DFDs to place methods in the classes

Figures A-2 through A-21 in Appendix show the DFDs of the ID. Each Fig. shows a single function of the hotel information system. For example, Fig. A-2 and Fig. A-3 show *Occupy-room* and *Occupy-Villa* functions, respectively. PHASE-4 places the processes of the DFDs into the classes of the PCH as their methods. Note that each DFD represents a function and each function may have more than one process.

4.4.1 Fig. A-2. *Occupy-room* function

Step 1: Each process of a function is considered as a method, the input to the process as argument to the method, and the output of the process as the output of the method. Table 2 shows the methods of the *Occupy-room* function with their arguments and outputs. Table 2 is derived from the DFD of the function (Fig. A-2). Similarly, for the following functions, tables are derived from their DFDs.

Table 2 . Methods with their arguments and outputs of *occupy-room* function

Method Name	Arguments	Output
Receive guest request	Room-Type	Room-Type
Verify guest request	Room-Type	Room-No
Change room status	Room-No	
Get guest info	Guest-No, Guest-Name, Guest-Addr, Guest-Credit	

Step 2: Distribute the methods given in Table 2 to the appropriate classes of the PCH as follows:

By comparing the arguments and the outputs of the *Receive-guest-request* method with the instance-variables of the classes of the PCH, it is found that Room-Type argument and its output match with the Room-Type instance-variable of the ***Room*** class. Therefore, the *Receive-guest-request* method is added to the ***Room*** class.

The *Verify-guest-request* and the *Change-room-status* methods are added to the ***Room*** class, because these methods operate on the same data store Room-info (see Fig. A-2).

The *Get-guest-info* method is added to the ***Guest*** class, because the method operates on the Guest-info data store.

Step 3: This step reorganizes the methods into the PCH using one of the five cases of this step (see Section 3.4). In Fig. A-2, there is no method that receives arguments from more than one method, or gives its output to more than one method. Also, there is no subclass or superclass that shares the same method. Therefore, none of the five cases is applicable here (for details see Section 3.4).

Step 4: This step constructs a sequence of the methods of the *Occupy-room* function to clarify the interaction among the classes of the PCH. This sequence gives the excavation order of the methods of the *Occupy-room* function. Table 3 shows the sequence of the methods.

Table 3. Sequence of the methods of *occupy-room* function

SequenceNo.	Method Name
1	Receive guest request
2	Verify guest request
3	Change room status
4	Get guest info

4.4.2 Fig. A-3. *Occupy-villa* function

Step 1: Table 4 shows the methods of *Occupy-villa* function with their arguments and outputs.

Step 2: Distribute the methods given in Table 4 to the appropriate classes of the PCH as follows:

By comparing the argument and the output of *Receive-guest-request* method with instance-variables of the classes of the PCH, it is found that the Villa-Type argument and

output match with Villa-Type instance-variable of the ***Villa*** class. Therefore, the *Receive-guest-request* method is added to the ***Villa*** class.

Table 4. Methods with their arguments and outputs of *occupy-villa* function

Method Name	Arguments	Output
Receive guest request	Villa-Type	Villa-Type
Verify guest request	Villa-Type	Villa-No
Change villa status	Villa-No	
Get guest info	Guest-No, Guest-Name, Guest-Addr, Guest-Credit	

The *Verify-guest-request* and *Change-villa-status* methods are also added to the ***Villa*** class, because they operate on the same data store Villa-info (see Fig. A-3). The *Get-guest-info* method is added to the ***Guest*** class, because the method operates on the Guest-info data store.

Step 3: In Fig. A-3, there is no method that receives arguments from more than one method, or gives its output to more than one method. Also no subclass or superclass of the PCH shares the same method.

Step 4: Table 5 gives the sequence of the methods.

Table 5. Sequence of the methods of *occupy-villa* function

SequenceNo.	Method Name
1	Receive guest request
2	Verify guest request
3	Change villa status
4	Get guest info

4.4.3 Fig. A-4. *Reserve-room* function

Step 1: Table 6 shows the methods of *Reserve-room* function with their arguments and outputs.

Table 6. Methods with their arguments and outputs of *reserve-room* function

Method Name	Arguments	Output
Receive guest request	Room-Type	Room-Type
Verify guest request	Room-Type	Room-No
Reserve room	Room-No	
Get guest info	Guest-No, Guest-Name, Guest-Addr, Guest-Credit	

Step 2: Distribute the methods given in Table 6 to the appropriate classes of the PCH as follows:

By comparing the argument and the output of the method *Receive-guest-request* with instance-variables of the classes of the PCH, it is found that, the Room-Type argument and its output match with the Room-Type instance-variable of the ***Room*** class. The method should therefore be added to the ***Room*** class. However, since this method is already present in the class, it is ignored.

The method *Reserve-room* is added to the **Room** class, because it operates on the data store Room-info of the function. The methods *Verify-guest-request* and *Get-guest-info* are ignored because they are already present in the ***Room*** and ***Guest*** classes, respectively.

Step 3: There is no method of the function that receives arguments from more than one method, or gives its output to more than one method. Also, there is no subclass and superclass in the PCH that shares the same method.

Step 4: The sequence of the methods of the DFD of *Reserve-room* function is given Table 7.

Table 7. Sequence of the methods of *reserve-room* function

SequenceNo.	Method Name
1	Receive guest request
2	Verify guest request
3	Reserve room
4	Get guest info

4.4.4 Fig. A-5. *Reserve-villa* function

Step 1: Table 8 shows the methods of *Reserve-villa* function with their arguments and outputs.

Table 8. Methods with their arguments and outputs of *reserve-villa* function

Method Name	Arguments	Output
Receive guest request	Villa-Type	Villa-Type
Verify guest request	Villa-Type	Villa-No
Reserve Villa	Villa-No	
Get guest info	Guest-No, Guest-Name, Guest-Addr, Guest-Credit	

Step 2: Distribute the methods given in Table 8 to the appropriate classes of the PCH as follows:

The method *Reserve-villa* is added to the ***Villa*** class, because it operates on the data store villa-info of the *Reserve-villa* function. The methods *Receive-guest-request, Verify-guest-request* and *Get-guest-info* are ignored because they are already present in the ***Villa*** and ***Guest*** classes, respectively.

Step 3: There is no method of the function that receives arguments from more than one method or gives its output to more than one method. Also, there is no subclass and superclass in the PCH that shares the same method.

Step 4: Table 9 shows the sequence of the methods.

Table 9. Sequence of the methods of *reserve-villa* function

Sequence No.	Method Name
1	Receive guest request
2	Verify guest request
3	Reserve villa
4	Get guest info

4.4.5 Fig. A-6. *Room-cancel* function

Step 1: Table 10 shows the methods of the *Room-cancel* function with their arguments and outputs.

Step 2: Distribute the methods given in Table 10 to the appropriate classes of the PCH as follows:

Table 10. Methods with their arguments and outputs of *room-cancel* function

Method Name	Arguments	Output
Get guest name	Guest-Name	Guest-Name
Retrieve guest info	Guest-Name	Room-No, Guest-Name
Change room status	Room-No	
Delete guest	Guest-Name	

By comparing the argument and output of the *Get-guest-name* method with instance-variables of the classes of the PCH, it is found that the Guest-Name argument and its output match with the instance-variable of the ***Guest*** class. Therefore the method is added to the ***Guest*** class.

The methods *Retrieve-guest-info* and *Delete-guest* are added to the ***Guest*** class, because they operate on the data store Guest-info of the *Room-cancel* function (see Fig. A-6). The method *Change-room-status* is added to the ***Room*** class, because it operates on the data store Room-info of the function.

Step 3: In Fig. A-6, the *Retrieve-guest-info* method gives outputs to the methods *Change-room-status* and *Delete-guest*, therefore a new method *Distribute-outputs-for-room-cancel* is defined in the ***Guest*** class as a coordinator method to distribute the output to the methods *Change-room-status* and *Delete-guest*.

Step 4: Table 11 gives the sequence of the methods.

Table 11. Sequence of the methods of *room-cancel* function

Sequence No.	Method Name
1	Get guest name
2	Retrieve guest info
3	Distribute results for room cancel
4	Change room status
5	Delete guest

4.4.6 Fig. A-7: *Villa-cancel* function

Step 1: Table 12 gives the methods of the *Villa-cancel* function with their arguments and outputs.

Table 12. Methods with their arguments and outputs of *villa-cancel* function

Method Name	Arguments	Output
Get guest name	Guest-Name	Guest-Name
Retrieve guest info	Guest-Name	Villa-No, Guest-Name
Change villa status	Villa-No	
Delete guest	Guest-Name	

Step 2: Distribute the methods given in Table 12 to the appropriate classes of the PCH as follows:
The methods *Get-guest-name*, *Retrieve-guest-info* and *Delete-guest* are ignored, because they already present in the ***Guest*** class. The method *Change-villa-status* is added to the ***Villa*** class, because it operates on the data store Villa -info of the function (see Fig. A-7).

Step 3: The method *Retrieve-guest-info* gives outputs to *Change-villa-status* and *Delete-guest*. Therefore, a new method *Distribute-outputs-for-villa-cancel* is defined as a coordinator method in the ***Guest*** class to distribute the output to the methods *Change-villa-status* and *Delete-guest*.

Step 4: Table 13 gives the sequence of the methods.

Table 13. Sequence of the Methods of *Villa-cancel* function

Sequence No.	Method Name
1	Get guest name
2	Retrieve guest info
3	Distribute results for villa cancel
4	Change villa status
5	Delete guest

4.4.7 Fig. A-8: *Updating-guest-info* function

Step 1: Table 14 gives the methods of the function *Updating-guest-info* with their arguments and output.

Step 2: Distribute the methods given in Table 14 to the appropriate classes of the PCH as follows:

Table 14. Methods with their arguments and outputs of *updating-guest-info* function

Method Name	Arguments	Output
Retrieve guest info	Guest-Name	Guest-Name, Guest-No, Guest-Addr, and Guest-Credit
Display guest info	Guest-Name, Guest-No, Guest, Addr, and Guest-Credit	Guest-Name, Guest-No, Guest, Addr, and Guest-Credit
Get new guest info	Guest-Name, Guest-No, Guest, Addr, and Guest-Credit	Guest-Name, Guest-No, Guest, Addr, and Guest-Credit
Updating guest info	Guest-Name, Guest-No, Guest, Addr, and Guest-Credit	

By comparing the arguments and output of the methods *Display-guest-info*, *Get-new-guest-info*, and *Updating-guest-info* with the instance-variables of classes of the PCH, it is found that they match with the instance-variables of the ***Guest*** class. Therefore, the methods *Display-guest-info*, *Get new-guest-info*, and *Updating-guest-info* are added to the class ***Guest***. The method *Retrieve-guest-info* is ignored, since it is already present in the ***Guest*** class.

Step 3: There is no method of the function that receives arguments from more than one method, or gives its output to more than one method. Also, in the PCH there is no subclass or superclass that shares the same method.

Step 4: Table 15 gives the sequence of the methods.

Table 15. Sequence of the methods of *updating-guest-info* function

Sequence No.	Method Name
1	Retrieve guest info
2	Display guest info
3	Get new guest info
4	Updating guest info

4.4.8 Fig. A-9: *Updating-room-info* function

Step 1: Table 16 depicts the methods of the function *Updating-room-info* with their arguments and outputs.

Step 2: Distribute the methods given in Table 5-16 to the appropriate classes of the PCH as follows:

By comparing the arguments and output of the methods *Retrieve-room-info*, *Display-room-info*, *Get new-room-info*, and *Updating-room-info* with the instance-variables of classes of the PCH, it is found that they match with the instance-variables of ***Room*** class, therefore, they are added to the ***Room*** class.

Table 16. Methods with their arguments and outputs of the *updating-room-info* function

Method Name	Arguments	Output
Retrieve room info	Room-No	Room-No, Room-Type, Room-Status, and Room-Price
Display room info	Room-No, Room-Type, Room-Status, and Room-Price	Room-No, Room-Type, Room-Status, and Room-Price
Get new room info	Room-No, Room-Type, Room-Status, and Room-Price	Room-No, Room-Type, Room-Status, and Room-Price
Updating room info	Room-No, Room-Type, Room-Status, and Room-Price	

Step 3: There is no method of the function that receives arguments from more than one method or gives its outputs to more than one method. Also, in the PCH there is no subclass or super class that shares the same method.

Step 4: Table 17 gives the sequence of the methods.

Table 17. Sequence of methods of the *updating-room-info* function

Sequence No.	Method Name
1	Retrieve room info
2	Display room info
3	Get new room info
4	Updating room info

4.4.9 Fig. A-10: *Updating-villa-info* function

Step 1: Table 18 gives the methods of the function *Updating-villa-info* and their arguments and outputs.

Table 18. Methods with their arguments and outputs of *updating-villa-info* function

Method Name	Arguments	Output
Retrieve villa info	Villa-No	Villa-No, Villa-Type, Villa-Status, and Villa-Price
Display villa info	Villa-No, Villa-Type, Villa-Status, and Villa-Price	Villa-No, Villa-Type, Villa-Status, and Villa-Price
Get new villa info	Villa-No, Villa-Type, Villa-Status, and Villa-Price	Villa-No, Villa-Type, Villa-Status, and Villa-Price
Updating villa info	Villa-No, Villa-Type, Villa-Status, and Villa-Price	

Step 2: Distribute the methods given in Table 18 to the appropriate classes of the PCH as follows:

By comparing the arguments and the outputs of the methods *Retrieve-villa-info, Display-villa-info*, *Get-new-villa-info*, and the *Updating-villa-info* with the instance-variables of classes of the PCH, it is found that they match with the instance-variables of the ***Villa*** class. Therefore, the methods are added to the V***illa*** class.

Step 3: There is no method of the function that receives arguments from more than one method or gives its output to more than one method. Also, in the PCH there is no subclass or superclass that shares the same method.

Step 4: Table 19 gives the sequence of the methods.

Table 19. Sequence of the methods of *updating -villa-info* function

Sequence No.	Method Name
1	Retrieve villa info
2	Display villa info
3	Get new villa info
4	Updating villa info

4.4.10 Fig. A-11: *Checkout* function

Step 1: Table 20 gives the methods of the function *Checkout* and their arguments and outputs.

Table 20. Methods with their arguments and outputs *of checkout* function

Method Name	Arguments	Output
Retrieve arrival date	Guest-Name	Arrival-Date, Room-No, Villa-No
Calculate total room rate	Arrival-Date, Room-No	Total-Room-Rate
Calculate total villa rate	Arrival-Date, Villa-No	Total-Villa-Rate
Calculate total invoice	Total-Room-Rate, Room-No, Total-Villa-Rate, Villa-No	Total-Cost, Room-No, Villa-No
Change room status	Room-No	
Change villa status	Villa-No	

Step 2: Distribute the methods given in Table 20 to the appropriate classes of the PCH as follows:

The method *Retrieve-arrival-date* is added to ***Guest*** class, the methods *Calculate-total-room rate* and *Change-room-status* are added to the ***Room*** class, the methods *Calculate-total-villa-rate* and *Change-villa-status* are added to ***Villa*** class, and the *Calculate-total-invoice* method is added to ***Account*** class, because they operate on the data stores Guest, Account, Room-info, and Villa-info data stores of *Checkout* function, respectively (see Fig. A-11).

Step 3: The method *Calculate-total-invoice* gives the output either to the method *Change-room-status* or to the method *Change-villa-status*. Both the methods belong to the ***Guest*** class. Case 2 of Step 3 (see Section 3.4) will be applied here and a new method *Distribute-outputs-for-invoice* is defined as a coordinator method in the ***Guest*** class to distribute the output of one of the methods *Change-room-status* or *Change-villa-status*.

Step 4: Table 21 gives the sequence of the methods.

Table 21. Sequence of methods of *checkout* function

Sequence No.	Method Name
1	Retrieve arrival date
2	Calculate total room rate, or Calculate total villa rate
3	Calculate total invoice
4	Distribute outputs for invoice
5	Change room status, or Change villa status

4.4.11 Fig. A-12: *Guest-dealing-with-restaurant* function

Step 1: Table 22 gives the methods of *Guest-dealing-with-restaurant* function and their arguments and outputs.

Step 2: Distribute the methods given in Table 22 to the appropriate classes of the PCH as follows:

By comparing the arguments and outputs of the methods *Receive-meal-request,* and *Prepare-meal-cost* with the instance-variables of the PCH classes, it is found that they match with the instance-variables of the ***Eat-In***, ***Room***, and ***Villa*** classes, since this function deals with the meal and with different restaurant activities. Therefore, these two methods are added to the ***Eat-In*** class. The method *Register-restaurant-invoice* is added to the ***Account*** class, since the method operates on Account data store of the function.

Table 22. Methods with their arguments and outputs of *guest-dealing-with-restaurant* function

Method Name	Arguments	Output
Receive meal request	Meal-Des, (Room-No or Villa-No)	Meal-Desc, (Room-No, or Villa-No)
prepare meal cost	Meal-Des, (Room-No or Villa-No)	Meal-Cost, (Room-No, or Villa-No)
Register restaurant invoice	Meal-Cost, (Room-No, or Villa-No)	

Step 3: There is no method of the function that receives arguments from more than one method or gives its output to more than one method. Also, in the PCH there is no subclass or super class that shares the same method.

Step 4: Table 23 gives the sequence of the methods.

Table 23. Sequence of the methods of *guest-dealing-with- restaurant* function

Sequence No.	Method Name
1	Receive meal request
2	Prepare meal cost
3	Register restaurant invoice

4.4.12 Fig. A-13: *Guest-dealing-with-FAX-and-Telex* function

Step 1: Table 24 gives the methods of the function *Guest-dealing-with-FAX-and-Telex* and their arguments and outputs.

Step 2: Distribute the methods given in Table 24 to the appropriate classes of the PCH as follows:

Table 24. Methods with their arguments and outputs of *guest-dealing-with-fax-and-telex* function

Method Name	Arguments	Output
Receive call request	Call-No, (Room-No or Villa-No)	Call-No, (Room-No, or Villa-No)
Serve guest	Call-No, (Room-No or Villa-No)	Duration, Call-No, (Room-No or Villa-No)
Calculate cost of service	Duration, Call-No, (Room-No or Villa-No)	Call-Cost, (Room-No, Villa-No)
Register call invoice	Call-Cost, (Room-No or Villa-No)	

By comparing the arguments and outputs of the methods *Receive-call-request*, *Serve-guest*, and *Calculate-cost-of-service* with the instance-variables of the PCH classes, it is found that they match with the instance-variables of the ***FAX***, ***Telex***, ***Room***, and ***Villa*** classes. Since this function handles the communication services which are part of the FAX and Telex functions and the Duration, FAX-No, and Telex-No are part of instance-variables of the classes ***FAX*** and ***Telex***, therefore, the three methods are added to the ***FAX*** and ***Telex*** classes. The method *Register-call-invoice* is added to the ***Account*** class, since this method operates on Account data store of the function (see Fig. A-13).

Step 3: There is no method of the function that receives arguments from more than one method or gives its output to more than one method. In the PCH, it is found that the classes ***FAX*** and ***Telex*** share the same superclass. Also, these two classes have the same methods (i.e., *Receive-call-request, Serve-guest*, and *Calculate-cost-of-service* methods). Therefore, by applying Case 5 of Step 3 (see Section 3.4), the three methods are removed from the classes ***FAX*** and ***Telex*** and are placed into the ***Comm-Ser*** class; the superclass of the ***FAX*** and ***Telex*** classes.

Step 4: Table 25 gives the sequence of the methods.

Table 25. Sequence of the methods of *guest-dealing-with-fax-and-telex* function

Sequence No.	Method Name
1	Serve guest
2	Receive call request
3	Calculate cost of service
4	Register call invoice

4.4.13 Fig. A-14: *Guest-dealing-with-laundry* function

Step 1: Table 26 gives the methods of the *Guest-dealing-with-laundry* function and their arguments and outputs.

Step 2: Distribute the methods given in Table 26 to the appropriate classes of the PCH as follows:

Table 26. Methods with their arguments and outputs of *guest-dealing-with-laundry* function

Method Name	Arguments	Output
Receive laundry request	Cloth-Parts, (Room-No or Villa-No)	Cloth-Parts, (Room-No, or Villa-No)
Specify cloth parts	Cloth-Parts, (Room-No or Villa-No)	Cloth-Price, (Room-No or Villa-No)
Calculate cost of laundry	Cloth-Price, (Room-No or Villa-No)	Laundry-Cost, (Room-No, Villa-No)
Register laundry invoice	Laundry-Cost, (Room-No or Villa-No)	

By comparing the arguments and outputs of the methods *Receive-laundry-request*, *Specify-cloth-parts*, and *Calculate-cost-of-laundry* with the instance-variables of the PCH classes, it is found that they match with the instance-variables of the ***Laundry***, ***Room***, and ***Villa*** classes. Since this function handles the laundry service and the instance-variables Cloth-Parts, Cloth-Price, and Laundry-Cost are part of the instance-variables of the ***Laundry*** class, therefore, the first three methods of the function are added to the ***Laundry*** class. The last method *Register-laundry-invoice* is added to the ***Account*** class, because this method operates on the Account data store of the function.

Step 3: There is no method of the function that receives arguments from more than one method or gives its output to more than one method. Also, there is no subclass or super-class that shares the same method.

Step 4: Table 27 gives the sequence of the methods.

Table 27. Sequence of the methods of *guest-dealing-with-laundry* function

Sequence No.	Method Name
1	Receive laundry request
2	Specify cloth parts
3	Calculate cost of laundry
4	Register laundry invoice

4.4.14 Fig. A-15: *Guest-inquiry* function

Step 1: Table 28 gives the methods of the *Guest-inquiry* function and their arguments and outputs.

Step 2: Distribute the methods given in Table 28 to the appropriate classes of the PCH as follows:

Table 28. Methods with their arguments and outputs of *guest-inquiry* function

Method Name	Arguments	Output
Retrieve guest info	Guest-Name	Guest-No, Guest-Name, Guest-Addr, Guest-Credit
Display guest info	Guest-No, Guest-Name, Guest-Addr, Guest-Credit	Guest-No, Guest-Name, Guest-Addr, Guest-Credit

The methods *Retrieve-guest-info*, and *Display-guest-info* are ignored because they are already present in the ***Guest*** class.

Step 3: There is no method of the function that receives arguments from more than one method or gives its output to more than one method. Also, there is no subclass or super-class that shares the same method.

Step 4: Table 29 gives the sequence of methods.

Table 29. Sequence of the methods of *guest-inquiry* function

Sequence No.	Method Name
1	Retrieve guest info
2	Display guest info

4.4.15 Fig. A-16: *Room-inquiry* function

Step 1: Table 30 gives the methods of the *Room-inquiry* function and their arguments and outputs.

Table 30. Methods with their arguments and outputs of *room-inquiry* function

Method Name	Arguments	Output
Retrieve room info	Room-No	Room-No, Room-Type, Room-Status, Room-Price
Display room info	Room-No, Room-Type, Room-Status, Room-Price	Room-No, Room-Type, Room-Status, Room-Price

Step 2: The methods *Retrieve-room-info* and *Display-room-info* are ignored because they are already present in ***Room*** class.

Step 3: There is no method of the function that receives arguments from more than one method or gives its output to more than one method. Also, there is no subclass or super-class in the PCH that shares the same method.

Step 4: Table 31 gives the sequence of the methods.

Table 31. Sequence of the methods of *room-inquiry* function

Sequence No.	Method Name
1	Retrieve room info
2	Display room info

4.4.16 Fig. A-17: *Villa-inquiry* function

Step 1: Table 32 gives the methods of the *Villa-inquiry* function and their arguments and outputs.

Table 32. Methods with their arguments and outputs of *villa-inquiry* function

Method Name	Arguments	Output
Retrieve villa info	Villa-No	Villa-No, Villa-Type, Villa-Status, Villa-Price
Display villa info	Villa-No, Villa-Type, Villa-Status, Villa-Price	Villa-No, Villa-Type, Villa-Status, Villa-Price

tep 2: The methods *Retrieve-villa-info*, and *Display-villa-info* are ignored because they are already present in the ***Villa*** class.

Step 3: There is no method of the function that receives arguments from more than one method or gives its output to more than one method. Also, there is no subclass or superclass in the PCH that shares the same method.

Step 4: Table 33 gives the sequence of the methods.

Table 33. Sequence of the methods of *villa-inquiry* function

Sequence No.	Method Name
1	Retrieve villa info
2	Display villa info

4.4.17 Fig. A-18: *List-all-empty-rooms* function

Step 1: In Fig. A-18, there is only one method *List-all-empty-rooms* of the function with two outputs: Room-No and Room-Status.

Step 2: The method is added to the ***Room*** class, because it logically belongs to this class.

Step 3: No case of Step 3 is applied here.

Step 4: There is only one method, so there is no need of any sequence.

4.4.18 Fig. A-19: *List-all-empty-villa* function

Step 1: There is only one method of the function with outputs: Villa-No and Villa-Status.

Step 2: The method is added to the ***Villa*** class because it logically belongs to this class.

Step 3: No case of Step 3 is applied here.

Step 4: There is only one method, so there is no need of any sequence.

4.4.19 Fig. A-20: *List-all-guests* function

Step 1: There is only one method *List-all-guests* with two outputs: Guest-No and Guest-Name.

Step 2: This method is added to the ***Guest*** class, because it logically belongs to this class.

Step 3: No case of Step 3 is applied here.

Step 4: There is only one method, so there is no need of any sequence.

4.4.20 Fig. A-21: *Display-daily-revenue* Function

Step 1: Table 34 shows the methods of the *Display-daily-revenue* function and their arguments and outputs.

Table 34. Methods with their arguments and outputs of *display-daily-revenue* function

Method Name	Arguments	Output
Receive invoice date	Date	Date
Calculate total cost of laundry	Date	Total-Laundry-Cost
Display total laundry cost	Total-Laundry-Cost	Total-Laundry-Cost
Calculate total cost of restaurant	Date	Total-Restaurant-Cost
Display total restaurant cost	Total-Restaurant-Cost	Total-Restaurant-Cost
Calculate total FAX/Telex cost	Date	Total-FAX-Cost, Total-Telex-Cost
Display total FAX/Telex cost	Total-FAX-Cost, Total-Telex-Cost	Total-FAX-Cost, Total-Telex-Cost
Calculate total room cost	Date	Total-Room-Cost
Display total room cost	Total-Room-Cost	Total-Room-Cost
Calculate total villa cost	Date	Total-Villa-Cost
Display total villa cost	Total-Villa-Cost	Total-Villa-Cost

Step 2: Distribute the methods given in Table 34 to the appropriate classes of the PCH as follows:

By comparing the arguments and outputs of the methods *Receive-invoice-date, Calculate-total-cost-of-laundry,Display-total-laundry-cost, Calculate-total-cost-of-restaurant, Display-total-restaurant-cost, Calculate-total-cost-of-FAX/Telex, Display-total-FAX/Telex-cost, Calculate total cost of room, Display-total-room-cost, Calculate-total-cost-of-villa,* and *Display-total-villa-cost* with the instance-variables of the PCH classes, it is found that they match with the instance-variables of the ***Account*** class. Therefore, all these methods are added to the ***Account*** class.

Step 3: There is no method of the function that receives arguments from more than one method or gives output to more than one method. Also, there is no subclass or super-class of the PCH that shares the same method.

Step 4: Table 35 shows the sequence of the methods.

Table 35. Sequence of the Methods of *display-daily-revenue* function

Sequence No.	Method Name
1	Receive invoice date
2	Calculate total cost of laundry
3	Display total laundry cost
4	Calculate total cost of restaurant
5	Display total restaurant cost
6	Calculate total FAX/Telex cost
7	Display total FAX/Telex cost
8	Calculate total room cost
9	Display total room cost
10	Calculate total villa cost
11	Display total villa cost

Figure 12 shows the class-lattice of the system. The details of the classes of the class-lattice are given bellow.

- **Class Name: *Hotel***
  - **Instances-Variables:** Hotel-Name, Hotel-addr
  - **Methods:** None

- **Class Name: *Account***
  - **Instances-Variables:**Invoice-No, Invoice-Desc, Invoice-Type, Mode-Pay
  - **Methods:** Calculate-total-invoice, Register-restaurant-invoice, Register-call-invoice, Register-laundry-invoice, Receive-invoice-date, Calculate-total-cost-of-laundry, Display-total-laundry-cost, Calculate-total-cost-of-restaurant, Display-total-cost-of-restaurant, Calculate-total-cost-of-FAX/Telex-cost, Display-total-FAX/Telex-cost, Calculate-total-room-cost, Display-total-room-cost, Calculate-total-villa-cost, Display-total-villa-cost

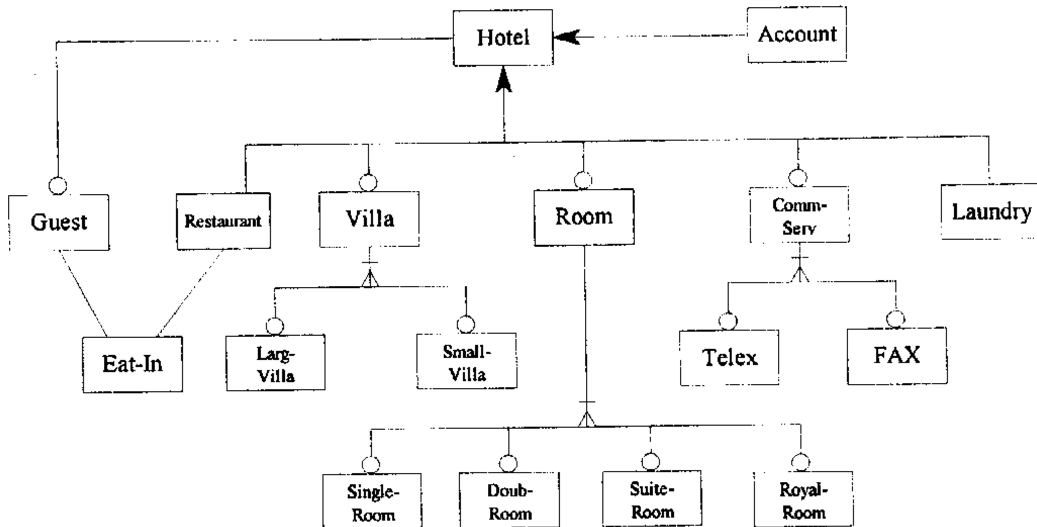

Fig. 12. The final class-lattice of the hotel information system.

- Class Name: ***Guest***
  - **Instances-Variables:** Guest-No, Guest-Name, Guest-Addr, Arrival-Date, Arrival-Time
  - **Methods:** Get-guest-info, Get-guest-name, Retrieve-guest-info, Delete-guest, Display-guest-info, Get-new-guest-info, Updating-guest-info, Retrieve-arrival-date, List-all-guest

- Class Name: ***Restaurant***
  - **Instances-Variables**: None
  - **Methods:** None

- Class Name: ***Eat-In***
  - **Instances-Variables:** Meal-Date, Meal-Desc, Meal-Cost
  - **Methods:** Receive-meal-request, Prepare-meal-cost

- Class Name: ***Villa***
  - **Instances-Variables:** Villa-No, Villa-Type, Villa-Price, Villa-Status
  - **Methods:** Receive-guest-request, Verify-guest-request, Change-Villa-status, Reserve-Villa, Retrieve-villa-info, Display-villa-info, Get-new-villa-info, Updating-villa-info, Calculate-total-villa-rate , List-all-empty-villa

- Class Name: ***Large-Villa***
  - **Instances-Variables:** Inheritance from "Villa" class

- **Methods:** Inheritance from ***Villa*** class

- Class Name: ***Small-Villa***
  - **Instances-Variables:** Inheritance from "Villa" class
  - **Methods:** Inheritance from ***Villa*** class

- Class Name: ***Room***
  - **Instances-Variables:** Room-No, Room-Status, Room-Type, Room-Price
  - **Methods:**Receive-guest-request,Verify-guest-request, Change-room-status, Reserve-room, Retrieve-room-info, Display-room-info, Get-new-room-info, Updating room-info, Calculate-total-room-rate, List all-empty-room

- Class Name: ***Single-Room***
  - **Instances-Variables:** Inheritance from ***Room*** class
  - **Methods:** Inheritance from ***Room*** class

- Class Name: ***Doub-Room***
  - **Instances-Variables:** Inheritance from the class ***Room***
  - **Methods:** Inheritance from the class ***Room***

- Class Name: ***Suite-Room***
  - **Instances-Variables:** Inheritance from the class ***Room***
  - **Methods:** Inheritance from the class ***Room***

- Class Name: ***Royal-Room***
  - **Instances-Variables:** Inheritance from the class ***Room***
  - **Methods:** Inheritance from the class ***Room***

- Class Name: *Comm-Serv*
  - **Instances-Variables:** None
  - **Methods:** None

- Class Name: *Telex*
  - **Instances-Variables:** Telex-No, Telex-Cost, Telex-Date, Telex-Time
  - **Methods:** Receive-call-request, Serve-guest, Calculate-cost-of-service

- Class Name: *FAX*
  - **Instances-Variables:** Call-No, Destination, Duration, Fax-Cost, Rate-Per-Minute, Call-Date, Call-Time
  - **Methods:** Receive-call-request, Serve-guest, Calculate-cost of service

- **Class Name:** ***Laundry***
  - **Instances-Variables:** Clean-Cost, Clean-Desc, Clean-Date
  - **Methods:** Receive-laundry-request, Specify-cloth-parts, Calculate-cost-of-laundry

  The final output (OOD) of PHASE-4 is summarized as follows:
  - Class-Lattice of the system
  - Instance-variables and methods in each class or class schema
  - Arguments and outputs of the methods
  - Relationships among the classes of the class-lattice
  - Sequence of methods for each function

5. Conclusions and Future Remarks

We have reported ID-OOD methodology that transforms a given ID into an OOD, and this transformation takes place in four different phases. Every phase has its procedural form. Both development cost and time can be reduced considerably by using this methodology. Now we are actively working towards the automation of this methodology.

Acknowledgments. We like to thank the anonymous reviewers for their valuable comments. We also like to thank Mr. Ali Ahmed Sabir for drawing the figures, and Dr. Amhed Sharaf Eldin and Mr. S. H. Shah Khan for reviewing early drafts of this paper.

References

[1] DeMarco, T. "Structured Analysis and System Specification." Yourdon Press, New York (1978).

[2] Jackson, M. A. "System Development." Englewood Cliffs, New Jersey: Prentice Hall International (1983).

[3] Yourdon, E. and Constantine, L. "Structured Design." Prentice Hall, Englewood Cliffs, New Jersey, (1979).

[4] Booch, G. "Object-Oriented Development." *IEEE Transactions on Software Engineering, Volume SE-12, Number 2* (1986), 211-221.

[5] Maier, D. "Why Object-Oriented Databases Can Succeed Where Other Have Failed." *Proceedings of International Workshop on Object-Oriented Database Systems* (1986), 227.

[6] Booch, G. "Object-Oriented Design With Applications." Benjamin Cumming, Redwood City, California (1991).

[7] Coad, P. and Yourdon, E. "Object-Oriented Analysis." Yourdon Press, Englewood Cliffs, New Jersey (1991).

[8] Coad, P. and Yourdon, E. "Object-Oriented Design." Yourdon Press, Englewood Cliffs, New Jersey (1991).

[9] Jacksen, I. "Object-Oriented Development in a Industrial Environment." *Proceedings of the ACM International Conference on Object-Oriented Programming Languages, Systems and Applications (OOPSLA'87),* (1987), 183-191.

[10] Rumbaugh *et al*, "Object-Oriented Modeling and Design." Prentice Hall, Englewood Cliffs, New Jersey (1991).

[11] Seidewitz, E. V. and Stark, M. "Towards a General Object-Oriented Software Development Methodology." *the ACM Ada Letters, 7(4),* (1987), 54-67.

[12] Shlaer, S., Mellor, S., Ohlsen, D. and Hywari, W. "The Object-Oriented Method for Analysis." *Proceedings of the tenth Structured Development Forum* (1988).

[13] Embley, D., Jackson, R. and Woodfield, S. "Object-Oriented Systems Analysis: Is It or Isn't It?." *IEEE Software*, July (1995), 19-33.

[14] Firesmith, D. "Structured Analysis and Object-Oriented Development are not Compatible." *the ACM Ada Letters, XI(9)* (1991), 56-65.

[15] Shumate, K. "Structured Analysis and Object-Oriented Design are Compatible." *the ACM Ada Letters, XI(4),* (1991), 78-90.

[16] Shah, A. and Kaushal, R. "A Methodology for Converting an Imperative Design to an Object-Oriented Design." *Proceedings of the 23rd Pittsburgh Conference on Modeling and Simulation*, USA (1992), 1505-1512.

[17] Al-Harbi, S., Shah, A., Mathkour, M. and Agrawal, J. "Methodology to Convert a Traditional Design to Object-Oriented Design." *Proceedings of the Third Golden West International Conference on Intelligent Systems* USA (1994), 833-842.

[18] Pressman, R. "Software Engineering - A Practitioner's Approach." McGraw-Hill, Inc. (1992).

[19] Chen, P. "The Entity-Relationship Model - Toward A Uniform View of Data." *ACM Transactions on Database Systems*, 1, No.1 (1976), 9-36.

[20] Dahl, O.J. and Nygaard, K. "SIMULA - An ALGOL-Based Simulation Language." *Combinations of the ACM, 9(9),* (September 1966), 671-678.

[21] Berard, E. "Essays on Object-Oriented Software Engineering." Vol. 1, Prentice Hall, Englewood Cliffs, New (Jersey 1991).

[22] Banerjee, J. et al, " Data Model Issues for Object-Oriented Applications." *ACM Transactions on Office Information Systems,* 5, No. 1 (January 1987), 3-26.

[23] Kim, W. "Introduction to Object-Oriented Databases." The MIT Press, Cambridge, Massachusetts (1990).

[24] Wirfs-Brock, R., Wilkerson, B. and Wiener, L. "Designing Object-Oriented Software." Prentice Hall, Englewood Cliffs, New Jersey (1990).

[25] Bailin, S. "An Object-Oriented Requirements Specification Method." *Communications of the ACM,* 32, No. 5 (May 1989), 608-632.

[26] Alabiso, B. "Transformation of Data Flow Analysis Models to Object-Oriented Design." *Proceedings of the ACM International Conference on Object-Oriented Programming Languages, Systems and Applications* (OOPSLA'88), (1988), 227-242.

[27] Gray, L. "Transitioning from Structured Analysis to Object-Oriented Design." *Proceedings of the Fifth Washington Ada Symposium*, USA (1988), 151-162.

[28] Ward, P. "How to Integrate Object-Orientation With Structured Analysis and Design." *The IEEE Software* (March 1989), 74-82.

[29] Pomberger, G. and Blaschek. G. "Object-Orientation and Prototyping in Software Engineering." The Object-Oriented Series, Prentice Hall (1996).

[30] Capertz, L. and Lee, S. "Object-Oriented Design: Guidelines and Techniques." *Information and Software Technology,* 35, No.4 (1993).

[31] Al-Dweesh, T. and Al-Qhtany, K. "Hotel Information System." Graduate Project, College of Computer and Information Sciences, King Saud University (1994).

Appendix

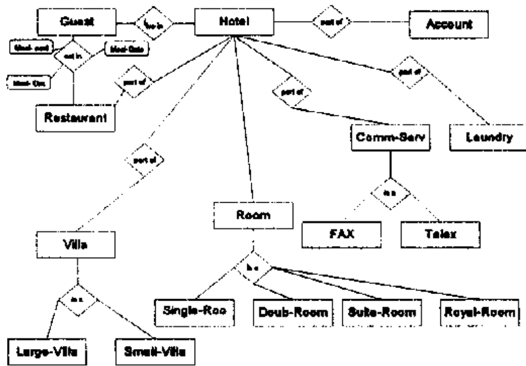

Fig. A-1. The ERD of hotel information system.

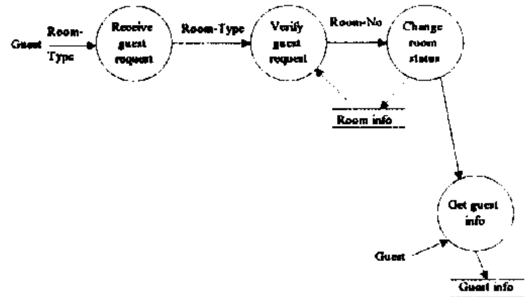

Fig. A-2. An explosion of occupy-room function.

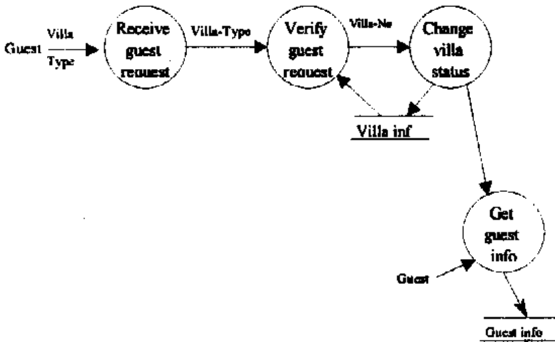

Fig. A-3. An explosion of occupy-villa function.

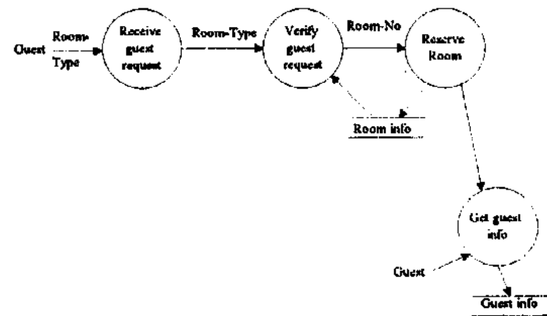

Fig. A-4. An explosion of reserve-room function.

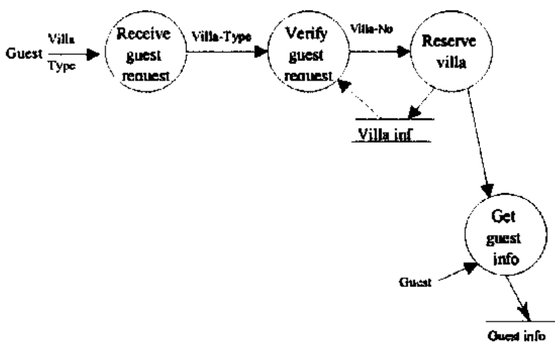

Fig. A-5. An explosion of the reserve-villa function.

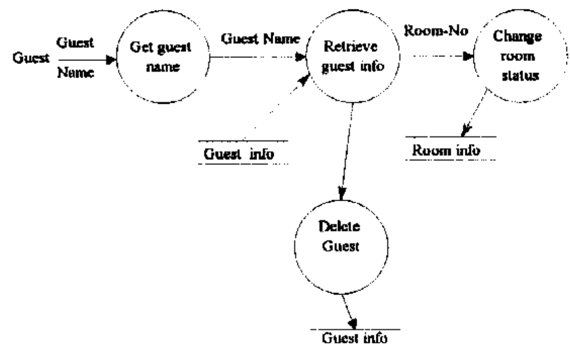

Fig. A-6. An explosion of room-cancel function.

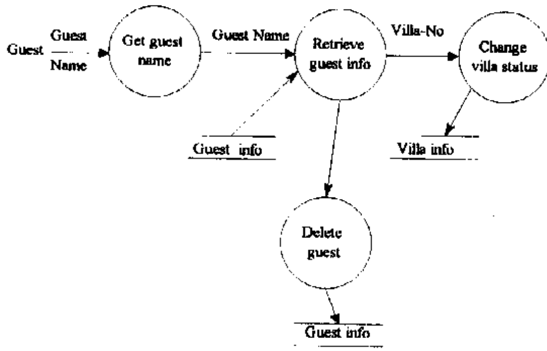

Fig. A-7. An explosion of villa-cancel function.

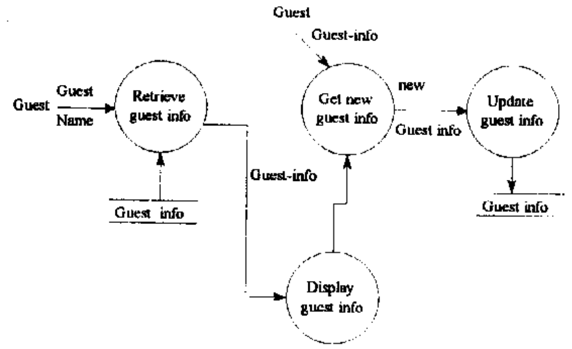

Fig. A-8. An explosion of updating-guest-info function.

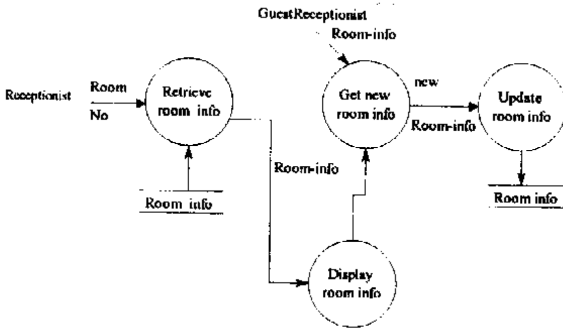

Fig. A-9. An explosion of updating-room- info function.

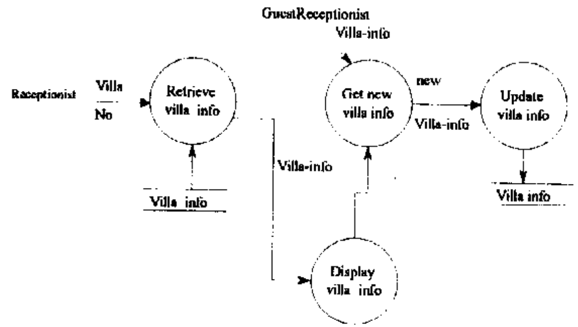

Fig. A-10. An explosion of updating-villa-info function.

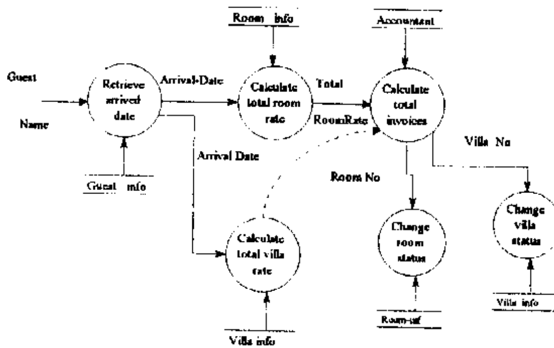

Fig. A-11. An explosion of checkout function.

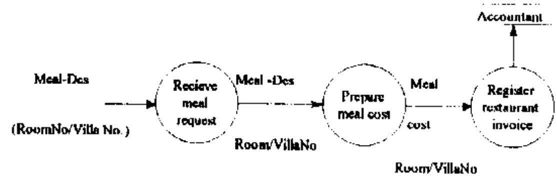

Fig. A-12. An explosion of guest-dealing-with-restaurant function

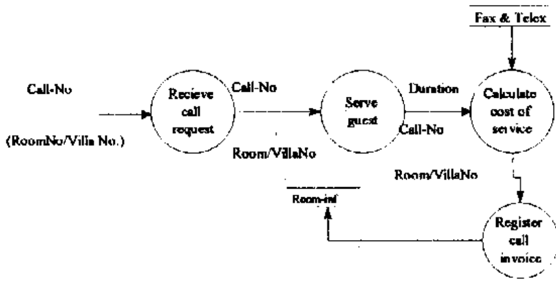

Fig. A-13. An explosion of guest-dealing-with-FAX-and-telex function.

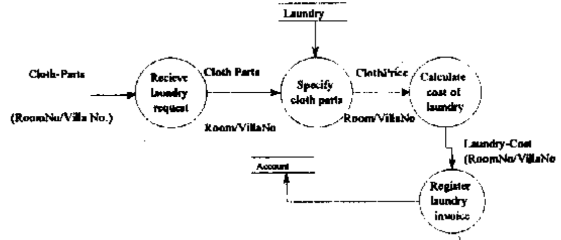

Fig. A-14. An explosion of guest-dealing-with-laundry function.

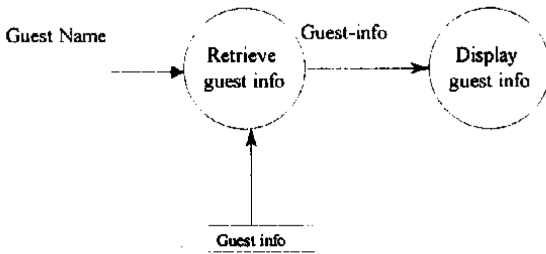

Fig. A-15. An explosion of guest-inquiry function.

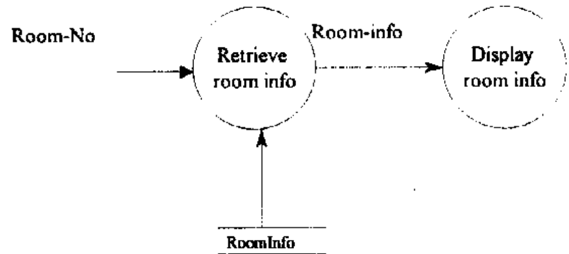

Fig. A-16. An explosion of room-inquiry function.

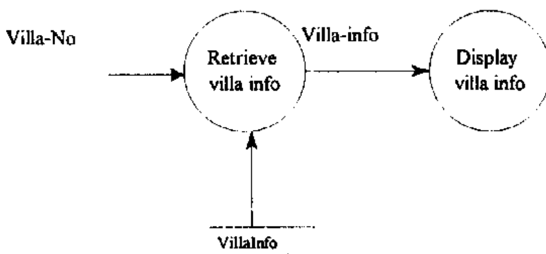

Fig. A-17. An explosion of villa inquiry function.

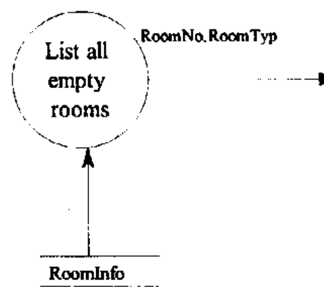

Fig. A-18. List-all-empty-rooms function.

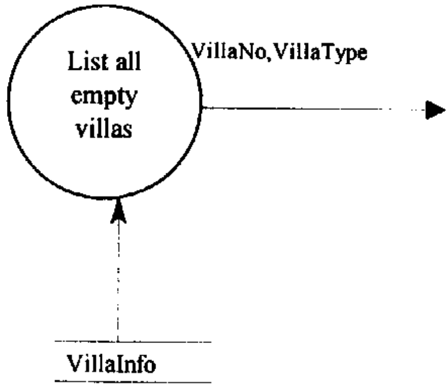

Fig. A-19. List-all-empty-villa function.

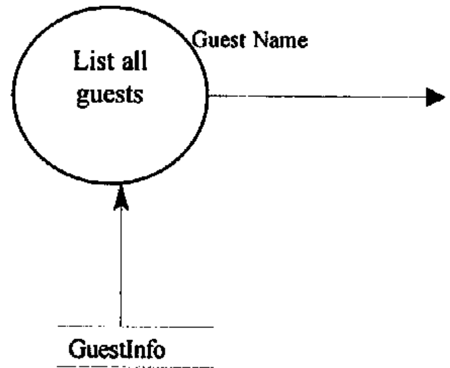

Fig. A-20. List-all-guests function.

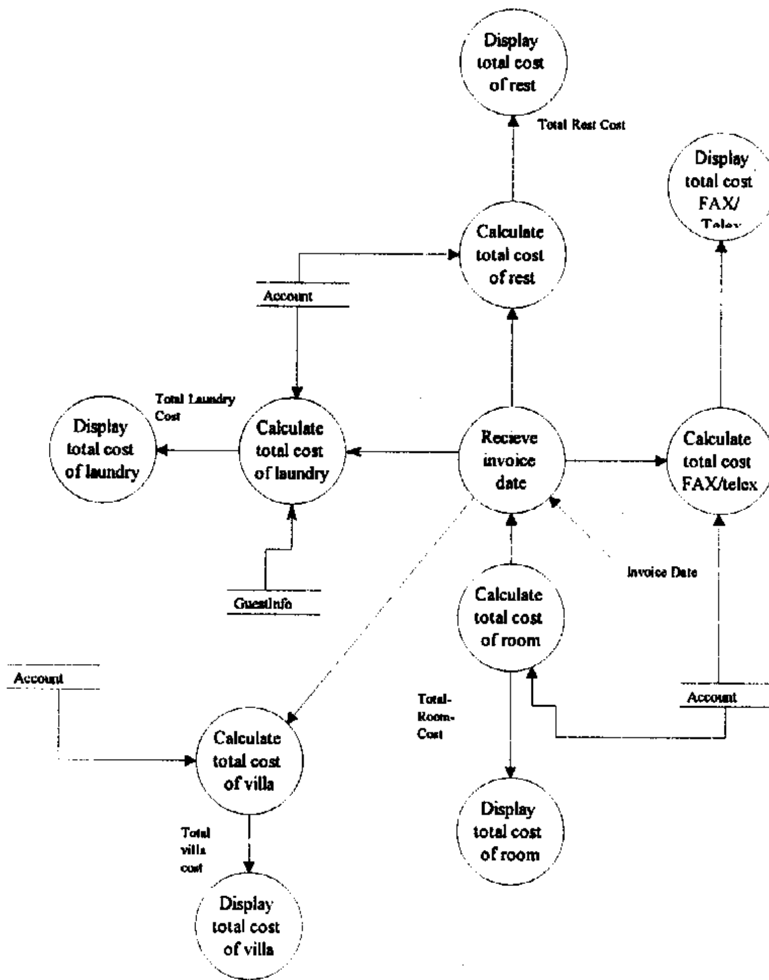

Fig. A-21. An explosion of display-daily-revenue.

Name: Room-No
Description: First digit represents floor no and that floorsecond two digits represent room no in
Format: Digits
Location: Room

Name: Room-Type
Description: Room type represents:
S = Single, D = Double, L = Suite, and R = Royal
Format: Char
Location: Room

Name: Room-Status
Description: Room status represents:
O = occupied, V = vacant, and R = reserved
Format: Char
Location: Room

Name: Room-Price
Description: Room price for each night
Format: Digits
Location: Room

Name: Villa-No
Description: The number of villa
Format: Digits
Location: Villa

Name: Villa-Type
Description: Villa type represents:
S = Small, and L = Large
Format: Char
Location: Villa

Name: Villa-Status
Description: Villa status represents:
O = occupied, V = vacant, and R = reserved
Format: Char
Location: Villa

Name: Villa-Price
Description: The villa price for each night
Format: Digits
Location: Villa

Name: Guest-No
Description: Identification number
Format: String
Location: Guest

Name: Guest-Name
Description: Guest name who wants to get room or villa
Format: String
Location: Guest

Name: Guest-Addr
Description: The address of guest
Format: Char
Location: Guest

Name: Arrival-Date
Description: Arriving date of guest
Format: Date
Location: Guest

Name: Arrival-Time
Description: The time when the guest came
Format: Time
Location: Guest

Name: Meal-Date
Description: The date when guest got his meal
Format: Date
Location: "eat in" Relationship

Name: Meal-Des
Description: Kind and description of the meal
Format: Char
Location: "eat in" Relationship

Name: Meal-Cost
Description: Cost of meal
Format: Float
Location: "eat in" Relationship

Name: Call-No
Description: The required number to send FAX
Format: Digits
Location: FAX

Name: Destination
Description: The country or city which send FAX to it
Format: Char
Location: FAX

Fig. A-22. Data dictionary of hotel information system.

Name: Duration **Description:** The number of minutes taken during FAX **Format:** Float **Location:** FAX	**Name:** FAX-Cost **Description:** The cost of using the FAX **Format:** Float **Location:** FAX
Name: Rate-Per-Minute **Description:** The price of one minute **Format:** Float **Location:** FAX	**Name:** Call-Date **Description:** The date of using FAX **Format:** Date **Location:** FAX
Name: Call-Time **Description:** The time of using FAX **Format:** Time **Location:** FAX	**Name:** Telex-No **Description:** The number of the telex which send telex to it **Format:** Char **Location:** Telex
Name: Telex-Cost **Description:** The cost of sending telex it **Format:** Float **Location:** Telex	**Name:** Telex-Date **Description:** The date of sending telex **Format:** Date **Location:** Telex
Name: Telex-Time **Description:** The time of sending telex **Format:** Time **Location:** Telex	**Name:** Clean-Cost **Description:** The cost of clean item **Format:** Float **Location:** Laundry
Name: Clean-Des **Description:** The items cleaned **Format:** Char **Location:** Laundry	**Name:** Clean-Date **Description:** The date of cleaning **Format:** Date **Location:** Laundry
Name: Mode-Pay **Description:** The payment methods where: CS = Cash, CC = Credit Card **Format:** Two Char **Location:** Account	**Name:** Invoice-No **Description:** The invoice number **Format:** Digits **Location:** Account
Name: Invoice-Des **Description:** The invoice description **Format:** Char **Location:** Account	**Name:** Invoice-Type **Description:** The invoice type represents: R = Room invoice, V = Villa invoice, S = Restaurant invoice, L = Laundry invoice, F = FAX invoice, and T = **Format:** Telex invoice **Location:** Char Account
Name: Hotel-Name **Description:** The name of hotel **Format:** String **Location:** Hotel	**Name:** Hotel-Name **Description:** The name of hotel **Format:** String **Location:** Hotel

Fig. A-22. [Continued] Data dictionary of hotel information system.

تحويل تصميم حتمي إلى تصميم شيئي

عباد شاه وحسن مذكور

قسم علوم الحاسب، كلية علوم الحاسب والمعلومات، جامعة الملك سعود،
ص. ب ٥١١٧٨، الرياض ١١٥٤٣، المملكة العربية السعودية

(قدم للنشر في ١١/ ٥/ ١٩٩٧م ؛ وقبل للنشر في ١٥/ ٩/ ١٩٩٨م)

ملخص البحث. لقد تم تصميم معظم النظم الحاسبوية، الحالية والقديمة باستخدام المنهجيات التقليدية مثل طرق التحليل والتصميم الهيكلية ويسمى مثل هذا التصميم بالتصميم الحتمي. ونتيجة لظهور التقنية الشيئية أصبحت هناك ضرورة لإعادة تطوير اتباع أحد الخيارين إما إعادة تطوير هذه النظم من البداية باستخدام طريقة شيئية وإما تحويل التصميم الحالي من خلال وثائق إلى تصميم شيئي وبالطبع فإن الحل الزخير يوفر في الوقت والتكلفة.

تقدم هذه الورقة تقريراً عن المجهودات التي قمنا بها في هذا الشأن. ولقد بدأنا هذا المجهود عام ١٩٩٢م حيث اقترحنا هيكلاً عاماً لطريقة إعادة التصميم وهذه الطريقة تقوم بتحويل التصميم الحتمي لنظام منفذ فعلاً إلى نظام شضيئي باستخدام وثائق النظام. وهذه الطريقة بأربع مراحل تم توصيفها، كما أننا دعمنا هذه الطريقة بحالة دراسية.