A Design Method For Object-Oriented Programming

WINNIE W. Y. PUN and RUSSEL L. WINDER

Department of Computer Science
University College London
Gower Street, London WC1E 6BT, England

ABSTRACT
This paper describes the primary framework of a design method targeted at object-oriented programming. The design method is not a revolutionary one but adopts appropriate ideas and graphical constructs from existing design methods. It is specially tailored to the characteristic features of object-oriented programming languages. This paper mainly discusses the structure and some of the preliminary ideas of the design method.

KEYWORDS: Design Methods, Object-Oriented Programming

1 INTRODUCTION TO THIS PAPER

Object-oriented programming has become more and more popular in the 1980's. This new programming paradigm requires people to think and design their systems differently from traditional design approaches [1]. In traditional procedural programming, modules are generated around operations; data structures are distributed between resulting routines. However, in object-oriented programming, the reverse occurs and the emphasis is on data structures; modularisation and operations are generated around data structures. This approach has reversed the viewpoint and changed the design behaviour of system designers. Thus, existing design methods, which are targeted at conventional programming languages, are not suitable for object-oriented programming. They are inadequate to help people identify objects or help in the process of object construction. They do not support the distinguishing features of object-oriented programming such as inheritance. Therefore, a new design method has to be developed for object-oriented programming.

This paper describes our first pass in developing a design method which is targeted at object-oriented programming languages. Section 2 of this paper discusses the characteristics of the design method. It also gives an introduction to the design method prior to the detailed description of each design level. Section 3, 4 and 5 examine each level of the design method in detail. They discuss the primary objective and the essential procedures in each individual design level. The paper concludes with a summary of the design method and outlines further work which will be carried out.

2 INTRODUCTION TO THE DESIGN METHOD

In this section, we first talk about the characteristics of the design method, we then give an overview of the design method itself.

2.1 Characteristics of the Object-Oriented Design Method

The design method mentioned in this paper is specially tailored for object-oriented programming and encompasses the following characteristics:

- The design method aims at providing a set of guidelines to help designers organise the system to be implemented in an object-oriented programming language. It does not impose rules which hinder the creativeness of the designer but provides guidance for designers to organise the design process.
- The design method serves as a teaching aid for novice object-oriented designers. It helps them to understand the philosophy and characteristics of the object-oriented programming paradigm. For advanced object-oriented designers, the design method also serves as a documentation tool. It assists them to document and realise each stage in the design process.
- The design method is not a completely revolutionary one which may alienate some users. In fact, it is our desire to have the method evolved from existing design methods. Thus, readers may find that some of the constructs of this design method are familiar. For example, the object interaction diagrams mentioned in the next section remind us of the data flow diagram in Yourdon's design method [2]. It is hoped that this approach helps to reduce the learning curve of new users.
- The design method tries to bring in user-interface issues at an early stage of the system design. Most existing design methods do not regard user-interface issues as part of the system design. Often designers either treat it as a separate issue or leave it till the end of the system development. The object-oriented design method mentioned here, however, emphasizes that user-interface design has to be looked at in the earliest stage.
- The design method uses graphical constructs. It is widely agreed that the human mind acquires information at a significantly higher rate by discovering graphical relationships in complex pictures than by reading texts [3]. The graphical constructs in the design method are designed to be easy to use. Users should not find that they have to spend too much time on drawing the graphical constructs but can spend most of their time on the main system design.
- The design method is targeted at an object-oriented programming language in the imperative programming style. Object-oriented features have been introduced in declarative programming languages such as in Concurrent Prolog [4] and Loops [5]. It may be that the same design method is not suitable to use in these domains.

2.2 An Overview of the Design Method

In software engineering, the classic life cycle demands a systematic, sequential approach to software development that spans system analysis, design, coding, testing and maintenance. The design phase normally takes requirement specifications, functional specifications as input and produces a set of system design specifications as output.

This object-oriented design method covers the system analysis and design phase in the conventional software life cycle. It expects system designers to have carried out the task analysis and feasibility study. A set of requirement specifications must be ready prior to using the design method. With the requirement specifications, system designers go through the stages suggested in the method and produce a set of design specifications. This set of design specifications will then be taken by the programmers and implemented in an object-oriented programming language.

The object-oriented design method is divided into three levels:

- conceptual level,
- system level and
- specification level.

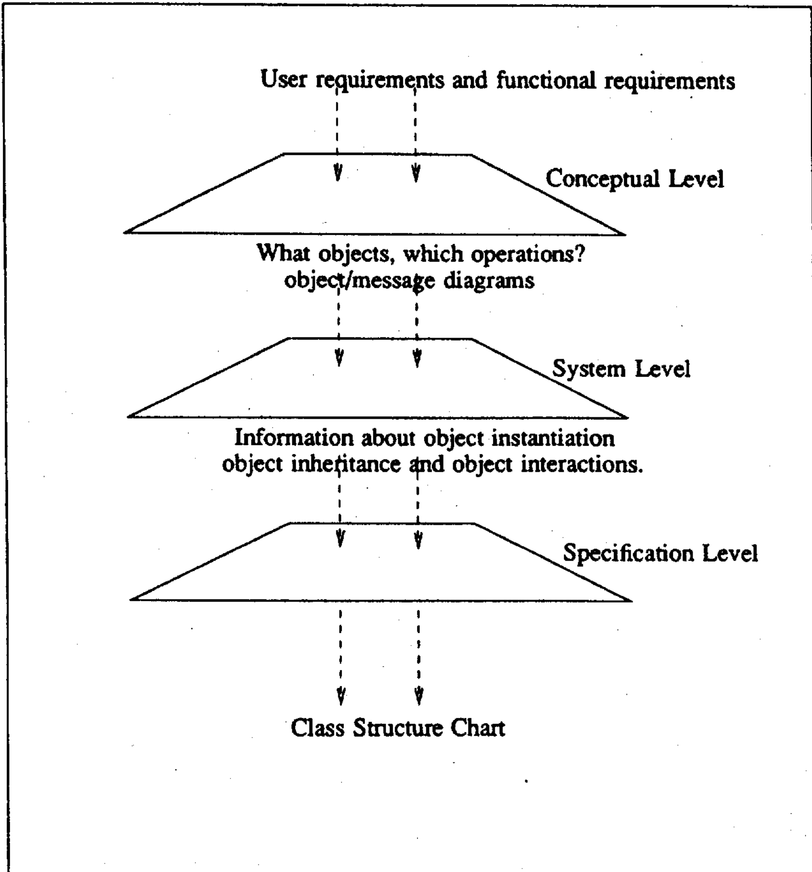

Figure 2. An Overview of the Design Method

The main objective of the conceptual level is to identify objects and operations occuring in a particular application. The designer takes the requirement specification as the input, analyses it and generates a set of object interaction diagrams. The object interaction diagrams are the documentation of the objects and interactions found in the system. Thus, at the end of the conceptual level design, a list of objects and actions have been identified.

After identifying the objects and operations which are involved in the system, designers have to construct these objects. The system level is where the construction methods for

objects are specified. Objects in object-oriented programming are created as instances of classes. There are three issues which are of interest in this level:

- object instantiation,
- class inheritance and
- object interaction.

In the process of object construction, the application object has to interact with the existing system classes at some point. For example, the application object may be created as an instance of an existing system class. If the system lacks anything like the required class, then the system designers have to create the class. The construction of new classes may involve working on the inheritance hierarchy. This process requires system designers to know exactly what inheritance is and how to arrange the inheritance hierarchy. Furthermore, system designers have to search around and get to know what system classes are available in the system. In order to assist them in this search, an intelligent browser is recommended for use in this level. The browser would provide information about the system classes to help system designers to make decisions.

With the information obtained from the conceptual and the system level, system designers should be able to construct the application objects and the interaction between objects. This kind of information is presented as a 'class structure chart' in the specification level. In fact, these class structure charts will be taken as design specifications by the programmers to implement the system. It is probably true that the system level and the specification level intermingle a little. In order to produce the 'class structure chart' at the specification level, system designers may find themselves having to use the browser at the system level, especially when working on inheritance issues. The detail of each level will be discussed in the next section. Throughout the discussion, we have used examples drawn from the development of a 'GP Surgery Notes System', a simple database system which we have developed with this object-oriented design method to test its feasibility.

3 DETAILED DESCRIPTION OF THE CONCEPTUAL LEVEL

The main objective of the conceptual level is to set up the conceptual model of the application. In this level, system designers take the user requirements and functional specifications and analyse the system, aiming to identify the objects and interactions involved in the application. The result of the analysis is presented diagrammatically as object interaction diagrams.

3.1 Identifying Objects

As mentioned in most of the literature [6], the initial step in object-oriented programming is to identify the objects involved in the application. At the conceptual level, system designers are interested in identifying objects which are application oriented. Here, an object represents an entity in the user's mental conception of the real world. An object in object-oriented programming is effectively an instance of an abstract data type which consists of:

- a set of variable names that describes its state, and
- a set of operations which can act upon the object to alter its state.

The selection of suitable objects will be based on this definition. To determine whether an entity in the application should be regarded as an object in the system, one has to make sure that the entity has a set of variables that undergoes some actions in the application. For example, in a GP system, the 'Patient-Card Database' is an object. It embeds operations such as 'add', 'delete' etc, which can be invoked by other objects in the application. Sometimes, one may encounter entities which are not obviously treated as objects. For example, a 'buttonstate' is not an obvious object because it is not tangible or readily associated with everyday objects. It is a state which occurs in the implementation model. However, one can construct it as an object in the implementation phase because it contains attributes such as 'stateValue' and operations such as 'setState', 'resetState' etc.

3.2 Two-layered Conceptual Level

Almost all systems developed involve interactions with end-users at some stage, thus the user-interface design is part of the design phase. In order to highlight this issue, the conceptual level is divided into two layers:

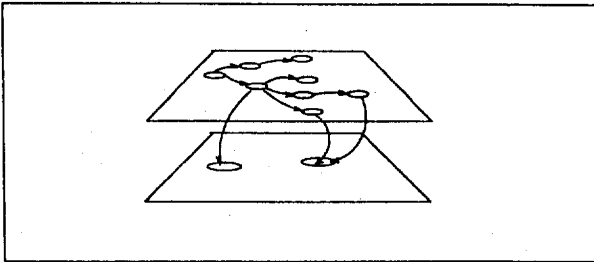

Figure 3. A Two-Layered Conceptual Level

- The User-Interface Layer.
  - This layer contains objects which communicate and interact directly with end-users. It provides a visual presentation of what the system is to end-users. Objects such as forms and menus are typical interfacing objects.
- The User-Transparent Layer
  - This layer contains application objects which are transparent to end-users. As far as end-users concern, they are not aware of the existence of these objects. For example in the GP system, the 'Patient-Card Database' is transparent to end-users.

Since the conceptual level is divided into two layers, system designers are obliged to arrange the identified objects into these two layers. The connection between these two layers is via message communication amongst the objects between the two layers. This is illustrated in Figure 3.

The separation of the conceptual level into two layers has not only highlighted the importance of user-interface design in system design, but also gives more flexibility to user-interface designers. As user-interface design becomes more important, methodologies have emerged to assist system designers to design better interface. The object-oriented paradigm has been introduced in this particular area. For example the Model-View-Controller (MVC) [6] and the Presentation, Abstraction and Control (PAC) [7] are two different frameworks

which apply the object-oriented paradigm to user-interface management. By separating the conceptual level into two layers, the user-interface layer can be extracted and handled by interface experts. As long as the user-interface layer provides a proper interface, i.e., message communication to interact with objects in the user-transparent layer, the conceptual model of the application is still correct.

3.3 Object Interaction Diagrams

The object interaction diagram is a diagrammatic technique used in the conceptual level. It allows designers to express the system analysis graphically. The graphical notation of the diagram is very simple. There are basically two different constructs:

- circles represent objects which are involved in the application,
- arcs and arrows denote the interactions between objects in the system.

Figure 4 is a simple object interaction diagram which denotes the GP System.

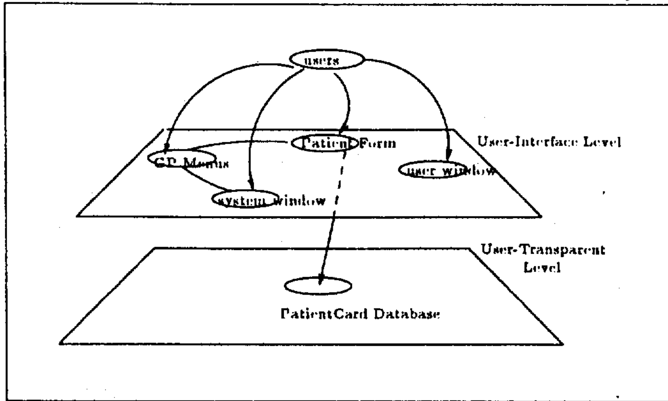

Figure 4. An example of an Object Interaction Diagram

Very often, an object is actually an abstraction of several objects. To reveal the structure of the abstraction, the object can be expanded into another set of object/message diagram as shown in Figure 5.

The object interaction diagram itself is not useful without proper description. Thus, designers should enter detailed information and descriptions of individual object into an object description form. A possible format of the form is shown in Figure 6.

This information together with the object interaction diagrams will be passed to the system level where individual objects are constructed.

4 DETAILED DESCRIPTION OF THE SYSTEM LEVEL

The first step of the object-oriented design method helps to construct an object-oriented conceptual model of the application. It identifies a set of objects and interactions within the system. This information will be passed down to the next level, the system level.

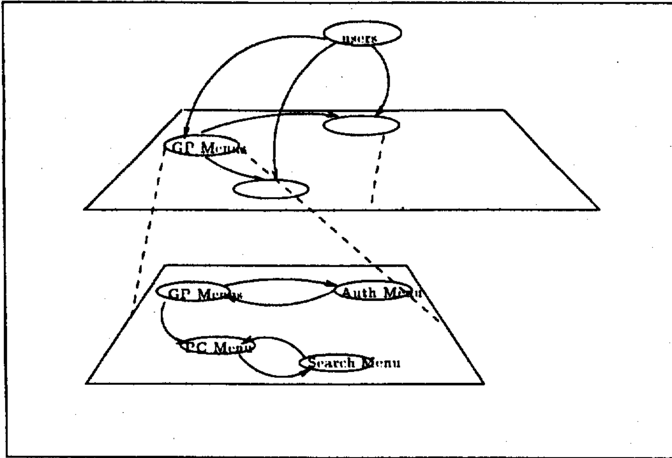

Figure 5. Expansion of an Object

Project name:
Object no:
Object name:
Description:

Possible Interactions:

Figure 6. Object Description Form

The system level is mainly concerned with how to transform a conceptual model into an implementation model. This involves:

- Further analysis of the object interaction diagrams.
- Decision of which control flow approach to be applied to the implementation model.
- Construction of individual objects. Since an object is created as an instance of a class, system designers have to decide:
    - — the set of attributes of a class, i.e., the instance variables and operations.
    - — whether a particular application object can be created as an instance of an existing class.
    - — if the object cannot be created as an existing class, then a new class has to be introduced. This new class may be a complete new class which has its own class hierarchy. Alternatively, this class may be built on existing classes using inheritance.

4.1 Centralised and Decentralised Control Flow

The model obtained in the conceptual level has highlighted objects which are application-oriented. However, these objects are only a subset of all the objects required to build the system. The rest of the relevant objects will be identified at this system level.

To pursue this, the object interaction diagrams drawn in the conceptual level have to be analysed further. Three points have to be noticed in doing this:

- Objects that are identified in the conceptual model do not necessarily become useful in the implementation model. For example, 'user' is an object which is identified in the conceptual model but becomes transparent in the implementation model.
- At the same time, additional objects may be needed in order to construct the implementation model for the application. For example, one may need an object called 'errorState' to determine which type of error has occured and its corresponding actions.
- By having a further analysis of the object interaction diagrams, one can confirm the set of objects that are needed for the implementation model.

The conceptual model has not only identified a set of objects which may be useful for the implementation model, it has also shown the interaction routes of the objects involved. However, the interaction routes that are denoted in the conceptual model may be altered when transformed into an implementation model. In fact, the interaction routes in the implementation model depends very much on which control flow approach one applies.

There are basically two different approaches to express the control flow of a system, the centralised and the decentralised control flow.

- For a centralised control flow approach, there is always an object which acts as a scheduler. When a receiver object finishes executing the method of a message, control returns to the scheduler object before the next message is sent.
- For a decentralised control flow approach, there is no unique object which can give an overall view of the activity of a system. To invoke the system, a message is sent to an object. This object then sends messages to other objects and the system control flows from one object to the next.

In conventional procedural languages, there is always a main program which encourages a centralised control flow. In object-oriented programming, modularisation and operations are generated around data structures. If one thinks of control flow as operations on objects then control flow is also generated around data structures. Thus, object-oriented programming tends to demonstrate a decentralised control flow. By embedding the appropriate part of the control flow in the object, one can enhance the autonomy of the object involved. The more autonomous the object in a software architecture, the more likelihood that a simple change will affect just one object, or a smaller number of objects, rather than trigger off a chain reaction of objects over the whole system. Thus, a decentralized control flow seems to be more appropriate in object-oriented programming. However, both centralised and decentralised control flow are valid and can be implemented in object-oriented programming. Although deciding which approach to apply is a design issue, the decision has to be made

in this level. The decision of which approach to use significantly affects what objects will be needed for the implementation model. This is shown in the following example.

In the GP system, the conceptual model has indicated that 'system menu' has to interact with 'system window' and the 'Patient Card Form'. If the centralised control flow approach is applied, one needs a 'GP Scheduler' to implement such idea. The 'GP Scheduler' governs the control flow of the system. When a user has selected an option from the menu, the system control will return to the scheduler. The scheduler may then send a message to the 'system window' to display some messages. Thus, the object 'GP Scheduler' is an important object found in the implementation model. On the other hand, if decentralised control flow approach is chosen, there will be no need for a scheduler object. In this case, 'system menu' directly sends messages to the 'system window' and the 'Patient Card Form' without going through another object. In order to implement this, the object 'system menu' has to know the appropriate reference of the other objects i.e. the references of the 'system window' and the 'Patient Card Form'. In this case, the control structure must be embedded in all the objects.

4.2 Inheritance in Object-Oriented Programming

Inheritance is an important feature that distinguishes object-oriented from object-based and class-based programming and it plays an important role in the construction of classes. Inheritance is always associated with the specification/generalisation relationship between classes. For example, if classA and classB have an 'is-a' relationship and classA is more general than classB i.e., classB needs a larger set of attributes to describe its behaviour, then attributes defined in classA can be used by classB through inheritance. There are basically two kinds of usage of inheritance:

- **Non-strict Inheritance** implies code sharing. Its main aim is to reuse as much of the existing implementation as possible. Here, to determine whether an appropriate superclass exists, system developers need only look at the attributes offered by a class. If there exist some attributes in classA which classB can use, classB can be created as a subclass of classA. One does not worry about the possibility that some meaningless, redundant attributes may be introduced into the subclass. This way of using inheritance does not require one to satisfy the 'is-a' relationship which was mentioned earlier. It allows complete freedom in building a class hierarchy and obtains maximal software reuse. However, in this case, inheritance may not be able to be used as a classification technique. In the extreme, one can put two conceptually unrelated classes in the same hierarchy provided they share some common attributes. This may lead to the formation of a class hierarchy which is conceptually confusing. Furthermore, if multiple inheritance is applied, one can get a very unstructured network which is impossible to comprehend. For example, in the Smalltalk system classes, the class 'semaphor' is created as a subclass of 'linked-list'. Conceptually there is no relation between 'semaphore' and 'linked-list' which may lead people to think that they should be put together. One suspects that the reason for doing this is because class 'semaphore' utilises some of the operations defined in the class 'linked-list'.
- **Strict Inheritance** is not only concerned with inheriting the code but also the specification of the superclass. By specification, we are refering to the abstract data type specification which involves the signature and the semantic meaning (axioms) of the operations in a class.

With strict inheritance, a subclass can only be derived from a superclass if and only if:

- All operations defined in the superclass mean something to the subclass i.e. no redundant operations must be inherited.
- Operations which share the same name, types of input and output arguments also perform the same functon i.e. operations which share the same name must share the same semantics. For example, a 'stack' and a 'queue' may have the same operations such as 'add' to add an element and 'delete' to remove an element. However, a stack applies a 'last in first out' policy whereas a queue applies a 'first in first out' policy. The operation 'delete' employs different methods in the implementation. Thus, according to the definition of strict inheritance, 'stack' and 'queue' do not share the same class hierarchy.

As one can see, this kind of inheritance imposes a very strict discipline in building class hierarchies. A subclass must inherit both the syntactic and the semantic aspects of its superclass. However, the amount of software reuse achieved in strict inheritance is minimal. In addition, strict inheritance suggests that system developers need to have full knowledge about the method of an operation.

In the design phase, system designers are not interested in what sort of inheritance is supported in a particular object-oriented programming language. Even if the programming language does not support a certain type of inheritance, system designers can attain it by imposing discipline in the usage of inheritance. What is more important is which is the better way for system designers to apply.

A system designer's main job is to design the system and prepare for the implementation stage. Strictly speaking, they should not need to worry about the detail of the implementation. For example, in the GP system, system designers need to realise that the class 'PatientCardDbase' is a subclass of the class DataBase. However, how the class DataBase is implemented is not important. This suggests that strict inheritance which requires knowledge of implementation, is not suitable for application in the design phase.

In contrast to strict inheritance, non-strict inheritance is too flexible to use. It does not impose a good discipline to guide system designers to handle inheritance. A good discipline is especially essential for novice object-oriented system designers who have no prior knowledge of inheritance. It prevents the designers from putting two unrelated classes in the same hierarchy. Therefore, it seems that the optimal kind of inheritance usage should lie in between the non-strict and the strict inheritance. This kind of inheritance has the following characteristics:

- The 'is-a' relationship should exist between superclass and subclass.
- The subclass inherits the whole external interface of the superclass i.e. the instance variable names and operation names.
- Methods of the inherited operations are allowed to be overwritten by subclasses if necessary.

As inheritance is an important feature in object-oriented programming, it is crucial for

the design method to incorporate some kind of mechanisms to help system designers to handle inheritance. Currently, we are working on an idea based on algebraic factorisation to help system designers to construct class hierarchies. The idea is inspired from the designers/programmers' behaviour in constructing class hierarchies. When constructing class hierarchies, designers always start with extracting common properties from the related classes. The idea is based on this observation and we are attempting to construct a formal manipulation system to automate such a process. The result of this formal system might mean that system designers need only to specify the attributes of particular classes and an optimal class hierarchical structures would be generated automatically [8].

5 DETAILED DESCRIPTION OF THE SPECIFICATION LEVEL

The system level mainly helps designers to set up the implementation model of the application. The implementation model is presented as design specifications in the specification level. The specification level is the level just prior to the implementation stage. It utilises the knowledge obtained in the system level to generate design specifications which programmers can take away and implement. The design specification is to be in the form of class structure charts.

5.1 Class Structure Chart

The class structure chart is a graphical technique which allows designers to express an individual class structure. A first impression probably suggests that the 'class structure chart' is similar to the 'structure chart' in structured design methods. In structured design methods, the structure chart helps to modularise a system into manageable units. These manageable units can be directly implemented as individual procedures/functions in a procedural programming language such as C, Pascal or Cobol. The 'class structure chart' is designed to match directly with an object-oriented programming language.

5.2 Objectives of the Class Structure Chart

It is necessary to reiterate that in this object-oriented design method, an object in the conceptual level is an instance of a class in the system level. That is why it is called 'class structure chart' instead of 'object structure chart'. The construction of an object-oriented system relies heavily on the construction of these structure charts. A class is a description of a group of similar objects. The description includes:

- a set of variables and
- a set of operations/messages that can act upon it.

Further, the structure of a class is built around the three 'I' relationships.

- inheritance,
- interaction and
- instantiation

Thus, the objectives of the 'class structure chart' are:

- to explicitly express the class hierarchical structure, i.e. the inheritance relationship between related classes,
- to lay out the method of the operations/messages of that class. This leads to further understanding of action sequences.

5.3 Graphical Notations of the 'Class Structure Chart'

A class structure chart reveals the method of a particular operation/message of a class. Examples of the class structure charts can be found in Figure 7 and Figure 8.

The graphical notations used in the class structure charts are:

- **Rectangular boxes** represent an individual class. The box which is of current interest will be divided into two compartments. The first compartment denotes the class name. The second compartment contains public/external operations of that class.
- **Dashed Rectangular** boxes. Sometimes, the message receiver belongs to the same class as the message sender. In order to denote this, a dashed rectangular box is used for such message receiver.
- **Curved Rectangular** boxes denote the superclass. They are also used to denote the message receivers which are created in the superclass.
- **Connections** between boxes are denoted by arcs and arrows. There are two basic types of connection:

  — inheritance
  An inheritance connection is denoted by a dotted arrow line, - - - → - - - -, which points from a parent to a child class.

  — interaction
  An interaction connection is denoted by a single arrow, ⟶, which points from a sender to a receiver.

- **A Diamond** denotes an alternative path. It is used to represent an 'if' statement or a 'case' statement.
- **An Ellipse Loop** denotes a repetition of message sending.

These class structure charts are design specifications which will be taken by programmers for implementation.

6 SUMMARY AND FUTURE WORK

This paper has described the preliminary framework of an object-oriented design method. The design method is not meant to be a revolutionary one. In fact, it has employed some of the idea in existing design methods. The design method is specially tailored to reflect the computational model for object-oriented implementation.

The design method is divided into three levels:

- The first level assists system designers to analyse and examine the application system in an object-oriented fashion.
- The second level concentrates on some of the important issues in the design phase of object-oriented programming. For example, centralised and decentralised control flow and class inheritance are issues which have to be handled in this level.
- The third level concerns the production of design specifications which will be passed down to the implementation phase.

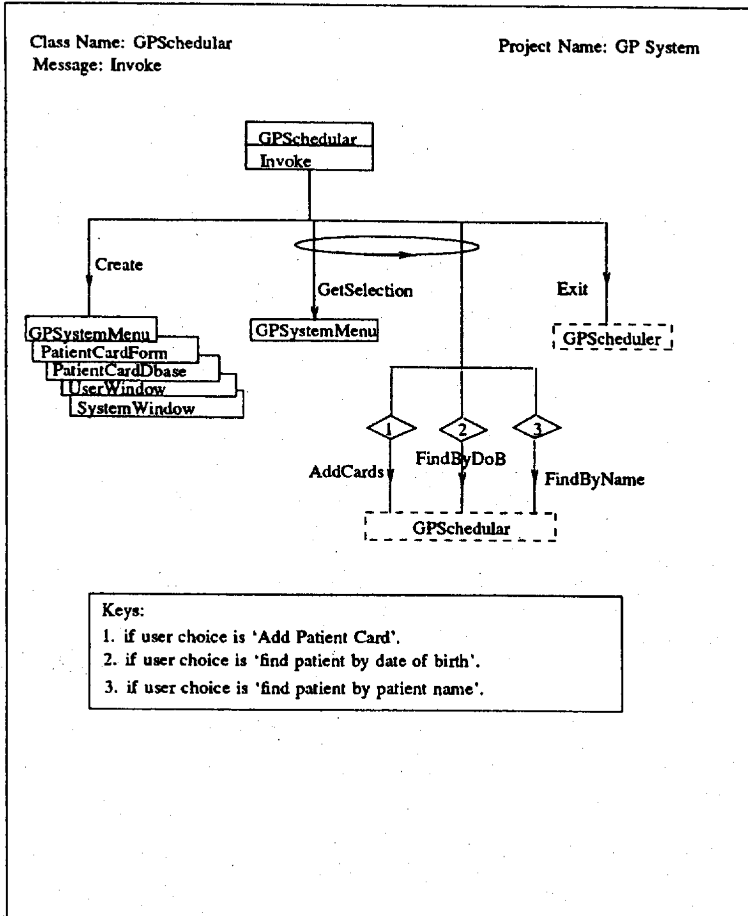

Figure 7 Class Structure Chart for the Invoke message in the GPScheduler

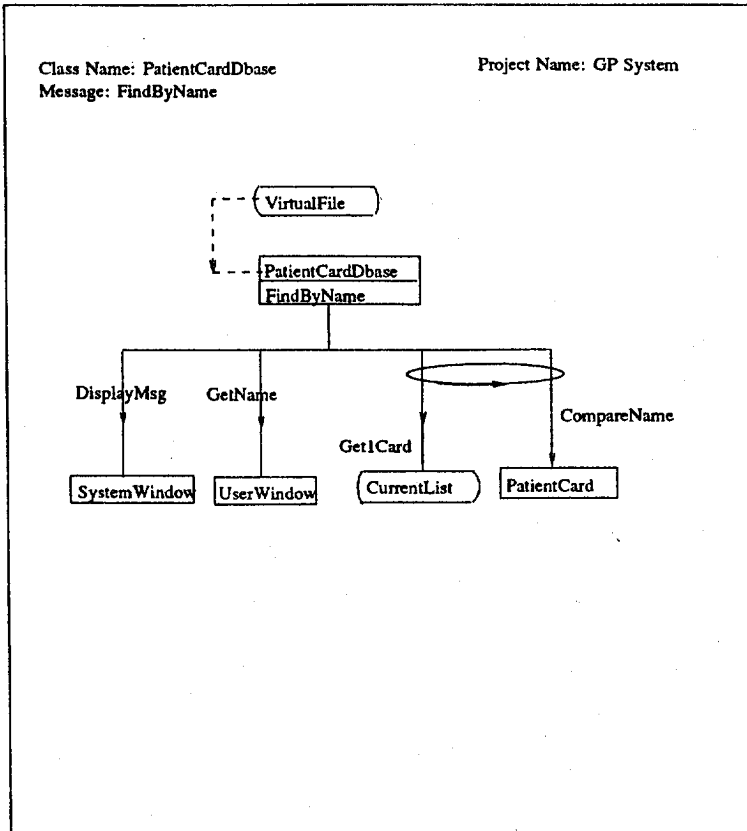

Figure 8 Class Structure Chart for the FindByName message in the PatientCardDbase

As it stands, the design method is still at its initial stage. There are a lot of areas which can be explored in the future.

- Currently, the design method has been applied in designing a small system, the 'GP Surgery Note system'. The next step is to apply the design method to a larger case study. This will improve the design method to cope with larger system development.
- We are developing a formal system which is based on algebraic factorisation to help system designers construct class hierarchies. We are also considering implementing the necessary tools to automate such a process.
- When the design method is fully developed, we would like to set up three experimental groups to test the feasibility of this design method. The first experimental group will involve subjects who are novice object-oriented designers. The second experimental group will involve subjects who are experienced object-oriented designers but are not familiar with any design methods. The last group will involve subjects who are familiar with existing design methods and know about object-oriented programming. We believe the result of such experiments can improve the usability and the practicability of the design method.

7 ACKNOWLEDGEMENT

We would like to thank the Croucher Foundation for the support given to Winnie W. Y. Pun during the period of this research. Also, we would like to thank the unknown referees whose comments have helped us to develop our ideas further. We hope that they will see the impact of their contributions in future publications.

8 REFERENCES

[1] Winnie W. Y. Pun, "Towards a Design Method for Object-Oriented Programming," *Research Note RN/88/1, University College London* (Jan 1988.).

[2] Tom De Marco, *Structured Analysis and System Specification*, Yourdon Press, Dec 1978.

[3] Georg Raeder, "A Survey of Current Graphical Programming Techniques," *IEEE Software Engineering* (Aug 1985).

[4] Ehud Shapiro & Akikazu Takeuchi, "Object-oriented Programming in Concurrent Prolog," *New Generation Computing* (1983).

[5] William Weil & Mary Ann Quayle, *The Friendly LOOPS Primer*, Learning Research and Development Center, University of Pittsburgh, 1984.

[6] Grady Booch, "Object-Oriented Development," *IEEE Transaction on Software Engineering* 12 (Feb 1986).

[7] M. J. Trebaul, "Smalltalk: The User Interface (a translated paper)," *Actes des Journees Afcet-Informatique, Langages Orientes Objet.* (1984).

[8] Joelle Coutaz, "The Construction of User Interfaces and the Object Paradigm," *Proceedings of the ECOOP'87 Conference* (1987).

[9] Winnie W. Y. Pun & Russel Winder, "Automating Class Hierarchy Graph Construction," *Research Note RN/89/23, University College London* (Mar 1989).