DIVISION OF COMPUTER SCIENCE

The Responsibility Driven Object-oriented Design Method advocated by Wirfs-Brock, Wilkerson and Weiner

Technical Report No. 149

A. Mayes

December 1992

The Responsibility Driven Object-oriented Design Method advocated by Wirfs-Brock, Wilkerson and Weiner.

Audrey Mayes

December 92

1 Introduction

This document presents the results of an investigation into a responsibility driven object-oriented design method [1]. This type of method views a system as a group of objects which represent the roles required to provide the desired functionality. This view of objects contrasts with the more usual view that objects represent both abstractions of entities in the problem space and the computer artifacts required to produce a system[2]. The responsibility driven method can lead to the identification of classes which provide one service only and act on data provided by or stored in other objects of another class.

Section 2 of this document contains a brief summary of the responsibility driven object-oriented design method. This is followed by a section giving the author's views on the method and program style produced. Section 4 contains a review of the author's experience of using the responsibility driven design method.

2 The method

Object-oriented design is defined as *'the process by which software requirements are turned into a detailed specification of objects'* [1]. The object specification defines the roles and responsibilities of the object, leading to the method being described as responsibility driven. This is different from the more conventional definition of an object which is concerned with the state and behaviour required [2].

The starting point for the method is a requirements specification document. This contains a description of what the software can and cannot do, the relative importance of the different features and real world constraints. Details of the user interface and the data storage facilities to be used are included in the requirements specification. The method deals with all aspects of the required system.

The method is divided into two phases, the exploratory phase and the analysis phase. Each of these phases is divided into three stages as summarised below.

2.1 Exploratory phase

1. Identify classes. This identification starts when the requirements specification has been read and understood. The classes are found by examining the requirements specification document and making a list of all the nouns and noun phrases. It is pointed out that the phrasing of the document can lead to nouns or verbs being disguised so care should be taken to compensate for this. The list is reduced by removing duplicates and alternative names for the same things and obvious nonsense. The remaining nouns and noun phrases are then examined to make sure that they fall into one of the following categories.

   - physical objects such as a display screen,
   - conceptual entities such as a PIN on a card,
   - external interfaces such as the user interface,
   - values of attributes such as float or real for the value of an attribute length.

   The candidate classes are then examined for groups with common attributes. These are used to define preliminary inheritance hierarchies. Fig.1 shows the notation used. The final list is then transferred on to cards or other storage medium. Each card contains the name of one class and a sentence describing the purpose of that class.

2. Allocate responsibilities. Responsibilities are the knowledge maintained by an object and the functions it can perform. The requirements specification is the starting point for the identification of responsibilities. This time verbs and verb phrases are extracted. The purpose recorded for each class is also useful. The relationships between classes are also used to help identify responsibilities. These responsibilities are then allocated to classes. The following guidelines are given.

   - State the responsibilities as generally as possible.
   - Distribute the intelligence evenly.
   - Put all the information about one thing in one place.

- Keep the behavior with the related information.
- Share responsibilities among related classes.

This information is also recorded on the cards as shown in Fig 2. More information is added to the cards later in the development process.

The attributes of a class are not modelled. It is the type of the attribute that is important, ie if it is an integer, string, etc. The attributes required by a class can be added as a responsibility for the class to know something, for example

Class: Account

Know the account balance.

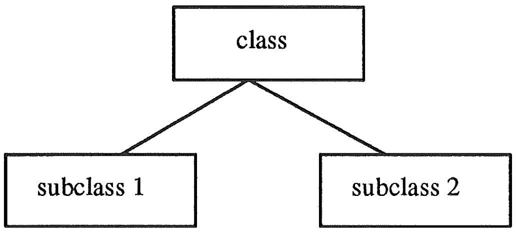


Figure 1: notation for preliminary heirarchies

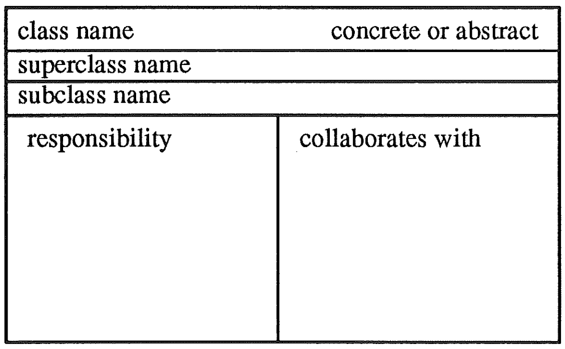


Figure 2: class cards

3. List the collaborations required. Some of the responsibilities recorded will require information or services from other classes. These are called

collaborations. Classes providing the services are called servers and classes using the services are called clients. Server classes are noted with the responsibilities on the cards (Fig 2). The relationships between classes are used to help identify collaborations. Any classes with no collaborations with other classes and which are not used by other classes are discarded.

This exploratory phase leads to a preliminary design which is studied in greater detail during the analysis phase to improve the hierarchies and identify subsystems.

2.2 Analysis phase

1. Develop inheritance hierarchies. This involves re-examining the classes and preliminary hierarchies. The common attributes are placed as high in the hierarchy as possible. Venn diagrams are used to ensure that all the responsibilities of the superclass are needed by the subclass. These diagrams are used to ensure that there is a type-subtype relationship between classes as well as a class-subclass relationship, but Venn diagrams seem to be an irrelevant complication. Abstract classes, which will not exist during the execution of the system are identified.

   The responsibilities of each class are divided into two groups. Those which can be requested by another class are called contracts. Those which represent behaviour a class must have but which cannot be used by other classes are called private responsibilities. The contracts supported by classes are simplified by grouping together those which are used by the same clients. An example of this is that the responsibility for an array to return the first element meeting a criterion can be grouped with the responsibility to return all the elements meeting a criterion. This simplifies the design. The notation used to show the developed hierarchies is shown in fig 3.

   The class cards have the extra information added to them. Each contract is numbered and named. The numbers of the contracts used by the class to fulfill its private responsibilities are noted.

2. Identify subsystems. A subsystem is a group of classes which work together to fulfill a set of closely related responsibilities eg a printing subsystem. Subsystems do not exist as the system executes. They are conceptual entities introduced to make the system easier to understand.

   A complete collaborations graph of the system is drawn. The notation used is again simple, see fig 4, but the graph gets complicated when many classes are involved.

   The subsystems are then identified and named. Again the collaborations between subsystem and classes within subsystems are examined and simplified. The introduction of subsystems permits layering of the design information. Information on the subsystems is noted in a similar way to class information.

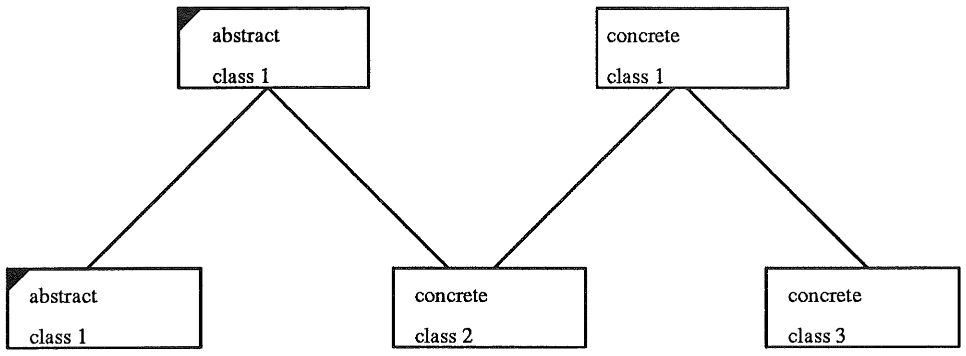


Figure 3: notation for final inheritance hierarchy

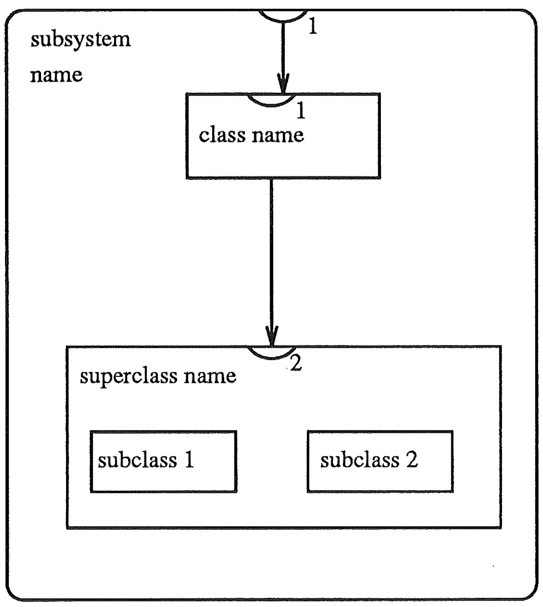


Figure 4: notation for collaboration graphs

3. Define protocols. A protocol is the list of signatures, procedure calls, to which a class will respond. These are defined for each class. Some classes may have to fulfill the same responsibility, such as display, even though they are not in the same hierarchy. A common abstract superclass is introduced from which the classes inherit the responsibility. The protocols are made as general as possible. This can be done by providing many versions of the same method. The most general version will require several parameters to be provided by the client of the method. Other versions require fewer parameters because the other values are implemented as default values. This gives a user of the class a choice of implementations.

   The hierarchy and collaborations graphs are then amended to include any changes made. The classes are then fully specified giving details of the

   - superclasses,
   - subclasses,
   - purpose,
   - contracts supported,
   - signatures of the methods they contain to support any contracts they do not inherit,
   - private responsibilities and the signatures for their methods if defined,
   - any behavioural constraints,
   - error conditions.

The completed design documentation consists of

1. hierarchy graphs - showing the class inheritance hierarchy,
2. collaborations graphs - showing the classes and subsystems within a system and how they collaborate,
3. class, subsystem and contract specifications.

As stated above the method considers the whole of the required system together, although it is recommended that larger systems are split up into major subsystems. There is no real advice about how to do this. The large system used as an example, in Wirfs-Brock's book, is an online documentation system, a simple word processor. This is divided into subsystems. Three of these are interfaces to the external world, a windowing system already provided, a printing subsystem and a file subsystem. The other is the document subsystem. Each subsystem is designed individually following the above stages. The stages are iterative. At each stage it is possible that some more details will be found which require the repetition of a previous stage.

3 Comments on the method

Most of the examples, given by Wirfs-Brock [1] to explain the method, refer to an automatic bank-teller machine called the ATM system. An online documentation system is also used. The following comments on the method use the same example case studies.

3.1 The notation used

The notation and documentation system is mostly clear and simple. There are some mistakes in the documentation for the ATM system which indicates that some form of consistency checking tool would be required for the method. The contract specifications show which of the services provided by a server is required by each client class. The clients are thought of as clients of a service provided by a class rather than of the class itself. This contrasts with Coad and Yourdon [3] where only the server class is noted. The collaborations graphs are simplified by removing subclasses which do not define new contracts. This is a sensible way to reduce complexity but a reminder that there are more classes involved would be useful. This can be done quite simply by adding (+) after the relevant class name.

The sample documentation shows classes and subsystems arranged alphabetically in one list. This can be confusing, separate subsystem specifications and class specifications would be preferable.

3.2 The design style

1. Classes

   The approach concentrates on defining the responsibilities of each class. A class must have a purpose which can be easily stated and either provide services to other classes or use services provided by other classes. The concentration on responsibilities results in some of the classes appearing to be little more than functions.

   All the classes in the transaction hierarchy have one public responsibility, that is they perform a financial transaction.In order to fulfill that responsibility an instance of a transaction class, for example a withdrawal transaction,interacts with:

   (a) the user via instances of user interaction classes to prompt the user for the amount to be withdrawn.

   (b) an instance of class account to tell the account to reduce its balance by the required amount,

   (c) the user via the cash dispenser to dispense the required amount of money.

There is no responsibility to remember the transaction so the objects of class transaction have no state. They are functional abstractions.

A more conventional object-oriented approach might result in an abstract class 'account' being developed and used to produce a 'transaction account' by inheritance. The derived class would add the ability to perform all the required transactions to the basic 'account' class. This would reduce the number of classes and help to simplify the design. At a later stage in the case study, the 'account' and 'transaction' classes are grouped into the financial subsystem. This brings about the same apparent simplification.

The transaction class hierarchy has arisen because the design method concentrates on responsibilities. The use of these functional classes may have the following advantages :

(a) any additional transaction types could be added more easily by inheriting from transaction instead of from account. More changes might be required in the ATM class if a new type of 'transaction account' was introduced rather than a new transaction. If this is true it suggests that object-oriented programs should have operational procedures separate from the data types on which they act,

(b) the user input and output is in a separate class which allows for easy change of natural language or style of output,

(c) the account to be accessed has to be specified in an object, so the same object can find out the amount as well. This also has the benefit of distributing the system's intelligence more evenly.

2. System Intelligence

   It is advised that the system's intelligence should be distributed evenly. The reason is that putting as much intelligence as possible into one object results in an object which is very much like a main program in a structured system and a group of other objects which are more like data structures. An advantage of this uneven distribution is that the flow of control is easy to understand. However the system's behaviour is "hard-wired" and the resulting system may be less flexible. This method is also said to require the writing of more unintelligent classes and therefore take longer. The recommended approach of evenly distributing the intelligence is said to require relatively fewer classes, be easier to modify and be more flexible. The disadvantage is that the overall system may take longer to understand as the flow of control is not readily seen.

   The following discussion looks at the ATM case study developed by the authors of the method with reference to the formation of a main program, the flow of control and the number of classes required.

(a) Main program

The ATM system has a class 'ATM'. The stated purpose of this class is to represent a teller machine through which bank customers can perform financial services. This has private responsibilities to

- create and initialise transactions
- display the greeting message
- display the main menu
- eject the receipt
- eject a bank card.

This is very similar, but not identical to the documentation for a main driver module in a Modula-2 program and thus appears to contradict the advice to distribute the intelligence evenly throughout the system. The difference between the ATM specification and a Modula-2 main program, is that the main driver module has the procedures listed in the order they are required to be carried out, instead of in alphabetical order. The order of execution must be determined somewhere. This can only be from within the ATM itself because all its responsibilities are private and so cannot be accessed by other classes. One method by which this can be implemented is by a create procedure within the ATM which would list the procedure (methods) in the correct order. Alternatively it may be possible to impose the order by using pre- and post- conditions. This would ensure that eject bankcard could not be called until the receipt had been printed and that the receipt could not be printed until a transaction had been created and initialised etc. The ATM has a large amount of control irrespective of the mode of implementation.

(b) Flow of control

It is stated that when the machine is idle the greetings message is displayed. This indicates a repeating structure is needed to allow the system to return to the greeting message. There is no indication of this in the design. The idea of when an action must be performed does not appear to form part of the documentation of the design. This may be reasonable for the design of 'what' the individual classes must do but it does not seem sensible for the system design to ignore the 'when' aspect altogether. This information is known to the designer and so should be available to the people responsible for implementation. It could be added in the documentation of the private responsibilities for each class. It should be noted that 'when' is not always important. For abstractions representing stored data there are no private responsibilities and no concept of 'when' is needed.

Thus the 'ATM' class appears to have a large amount of control over the system. There is no flow of control defined so it cannot be readily seen.

(c) Number of classes

The ATM system appears to have a large number of classes. The account class has responsibility for knowing the account balance, accepting deposits and withdrawals and committing the results of the transactions to the database. This is an opaque abstract data type which knows how to store itself, an alternative name for this type of class is a passive class. The transaction classes all represent requests by the user to do something to an account. This results in a hierarchical structure to the system with the higher layers interacting with the user and the lower layers interacting with the hardware responsible for the storage of the data (Fig 5).

3. Requirements Capture

   The ATM class has a responsibility to display the main menu. The menu class acts as a template for the production of instances of class 'menu' and provides facilities to get a user choice from a list of options.The main menu is an instance of menu and as such does not appear in the class specification. The designers of the system know at least part of what is required in the menu, they have defined the different sorts of transactions. This information should be available to the programmer. This lack of detailed information suggests that the method may be designing the classes from which a system can be built rather than the system itself.

4. System Structure

   In the ATM case study, the ATM 'root' class, or main program, at the top of the hierarchy is responsible for starting the system. This interacts with the user to find out which transaction is required, before starting the required transaction which interacts with the user and with the account abstract data type at the bottom of the hierarchy. The user interaction is performed by user interface classes.

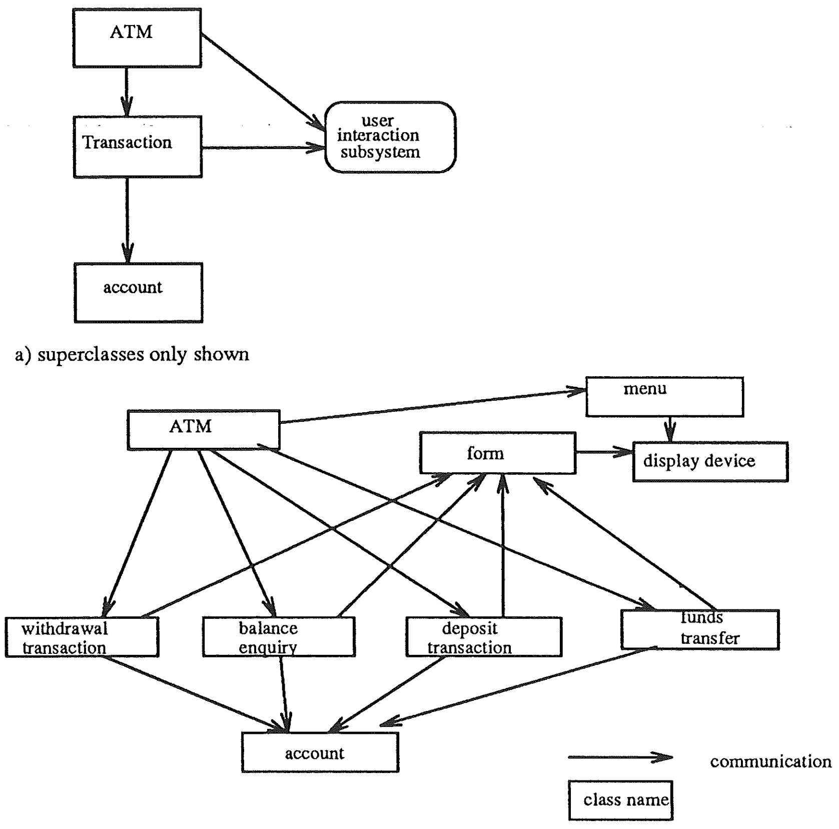


Figure 5: ATM system structure

The separation of the user interaction from the ADT has the benefit of allowing the user interface to be changed more easily and allows the generic classes to be used in other systems.

The online documentation system is divided into subsystems. Three of these are interfaces to the external world, a windowing system already provided, a printing subsystem and a file subsystem. This allows the designers to concentrate on the documentation subsystem. This is further subdivided into subsystems. These are the document subsystem and the editor subsystem.

The editor subsystem is used to interpret user inputs - I think this is a functional breakdown although probably different to the breakdown arrived at by structured analysis and design. A main difference between traditional development and this method seems to be the responsibility of the data stores to control their contents. For example, a sorted array would be responsible for finding the right place and putting the element in there. Conventionally an array would be traversed until the correct place was found and the user would be responsible for adding the element.

The overall architecture of the system appears to give flexibility for reusing the components. The grouping of classes into functional subsystems makes it possible

(a) to reuse each subsystem independently in other systems,

(b) to allow the possibility of changing the user interface and the storage medium.

4 Experience using the method

The garden planning system [4] was designed following the responsibility driven approach. The final documentation is not included, but is available if required. The complete requirements specification was used to supply the nouns and verbs needed to identify classes.

4.1 Requirements Capture

The approach examines the required system in great detail and ensures that most if not all of the requirements are understood.

The consideration of the required output and user interaction throughout the design helped to clarify the requirements.

The recommended documentation did not allow all the knowledge gained to be passed on to the implementation stage.

The lack of layering led to a rather large number of classes being considered at the same time. The garden planning system with around forty classes is not large but the number was rather overwhelming.

The authors of the method suggest that large problems are split into smaller pieces ie subsystems. The experience of following the method suggests that anything other than small systems should be divided. The following subsystems would be the starting point for the development of an information system similar to the garden planner:

- user interaction,
- stored data,
- external devices
- the control system.

The first to be considered would be the user interaction defining the user interface and the output required. This would make sure all the requirements were fully understood before any of the rest of the system was designed. Some of the more general classes needed for the user interaction in the first system might be reusable.

4.2 Abstract classes

There are two main types of responsibility:

- to maintain information
- to perform an action.

The design method makes no distinction between the types of responsibility until the implementation is being considered. The responsibilities to maintain information then become part of the state of the object or part of its structure. For example, a car class might maintain information about its colour as part of its state and information about its engine as part of its structure. The actions a class can perform will become the sevices or methods provided by the class.

During the analysis phase of the design, inheritance hierarchies are developed. This involves placing common responsibilities as high as possible in a hierarchy. Abstract superclasses are added to the system as necessary to provide the common responsibilities. These classes can be added to encapsulate knowledge storing and/ or processing responsibilities.

The type of responsibility being encapsulated in the abstract superclass becomes important when the implementation is considered. For example, in the garden case study, a class 'display form' was introduced to encapsulate the responsibility 'to display stored information in a text form'. The fact that the responsibility being encapsulated is a processing requirement means that the class does not contain sufficient information to allow instances to be created. The class contains an abstraction of many possible implementations. In the programming language Eiffel [5], this type of abstraction is coded as a deferred class which prevents any direct instances of the class being created. A sub-class must be formed to add the required information before the class can be used. A abstract super class added to encapsulate the responsibility to maintain information will contain all the information required to allow an instance to be created and is a potential concrete class. The type of responsibility being encapsulated, therefore affects the way in which the classes are implemented. The author has decided to include the information that a superclass encapsulates an action that can be performed in the documentation, see Fig 6.

4.3 Control

A class to to encapsulate user choice has been included in the design. This class was identified when several classes were found to have similar responsibilities

- namely to display a menu and initiate a process. The new class is called 'dispatch'. This is to be used whenever a choice of function can be made by the user. It is an example of an action abstraction. The ability to execute is exported or, alternatively stated, it has a responsibility to allow other classes to execute it! If a series of choices can be made, it must be able to execute another instance of itself. This is a reusable main program or driver module.

There is a problem with modelling this type of class in this responsibility driven method. The responsibility to execute a process should specify which class(es) it will collaborate with. This requires a slight change in the notation, the class specification will have to allow for a 'uses' statement to contain a deferred class and contract number.

4.4 Architecture

The introduction of the 'dispatch' class gives a system architecture which can be used for many different systems. The top level collaborations graph gives no indication of the system under development because the dispatch class is an abstract class. It is necessary to go down a level to find out about the current system. The system is not fully specified. The menus are not defined because they are instances of a class menu.

4.5 Notation

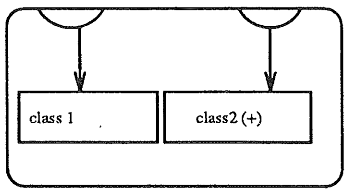
In the garden case study, the following changes were made to the notation to give a more complete representation of the known information. See Fig 6.

1. The collaborations graphs show, by adding (+), where classes have been left out to simplify the diagrams. The written specifications of the classes and subsystems are not arranged alphabetically. All the subsystem specifications are given with the controlling subsystem first. This is followed by the classes grouped by subsystems and arranged alphabetically within each group.

2. The heirarchy graphs show, by filling in the bottom left hand corner of a class box , when the abstract class encapsulates an action to be performed.

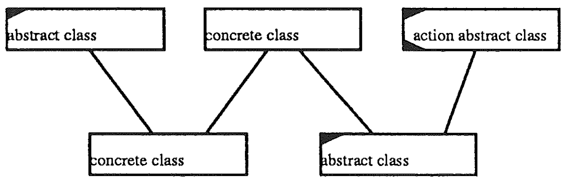
3. The class specification allows a uses- statement to contain a deferred class and contract number.

4. The requirement for some of the responsibilities to be carried out in a specified order has been added to the class specifications.

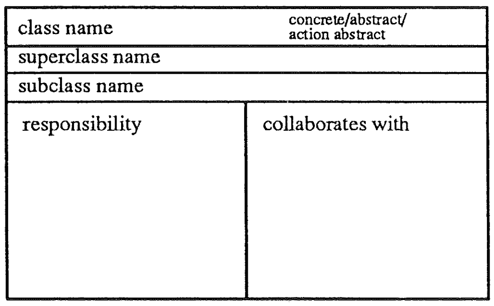


+ indicates the presence of subclasses which do not add behaviour

a)collaborations graph



b) class hierarchies



c) class cards

Figure 6: changed notation

5 Conclusion

The Wirfs-Brock method seems to lead to a thorough understanding of the required system. The system is divided into functional subsystems. The documentation appears simple and clear but does not capture all the information known to the designer. In particular,

- there is no concept of how the system is controlled,

- information about the type of abstraction is not recorded, that is entity abstraction or action abstraction,
- detail known to the designer about the entries required in the menus is not recorded.

Several questions are raised by this approach. These include:

- Is this design method producing classes for using in the design of a system or developing a system?
- Should classes provide more than one service?
- At what stage in the development should the requirement for a program to repeat be included in the design?
- At what stage should instances of template classes be specified? eg the contents of menus.
- Should there be special classes to interpret user input?
- Is it easier to modify a design with many simple classes than a design for the same system which has fewer complex classes? How can ease of modification be measured?
- Does the fact that a hierarchical system with one class resposible for overall control is easier to understand make this a more maintainable structure than one with distributed control?

The method begins with the requirements specification for a system and involves studying the whole of the system, including data storage and user interaction, but it suffers from the drawback that it appears to design the classes from which a system can be built rather than designing the required system.

References

[1] R. Wirfs-Brock, B. Wilkerson, and L. Weiner. *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[2] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings Publishing Company, Redwood City,California, 1991.

[3] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, Inc, Englewood Cliffs, New Jersey, second edition, 1991.

[4] J. A. Mayes. An investigation into the reusability of functional and object-oriented designs. MSc Report, Hatfield Polytechnic, 1991.

[5] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, Hemel Hempstead, 1988.