Modelling Techniques for Object-Oriented Design

Technical Report No.133

Carol Britton and Mary Buchanan

May 1991

Modelling Techniques for Object-Oriented Design

Carol Britton and Mary Buchanan.
Division of Computer Science, Hatfield Polytechnic,
College Lane, Hatfield, Herts. AL10 9AB.

May 1991.

Abstract

It is increasingly recognised (Booch 1991, Meyer 1989) that an object-oriented approach offers the most promising way forward for the design and production of software systems. In this paper we adopt this point of view, and discuss a problem which arises from it, namely that there is a basic incompatibility between the underlying concepts of object oriented design and the techniques available for modelling software systems in the early stages of development.

From the most popular system modelling techniques we select Entity Life Histories as the one which can be made to map most happily onto an object oriented design. We show how this can be achieved, and how, by a simple enhancement, Entity Life Histories can be adapted to reflect inheritance and reusability in object-oriented design.

The example we use is that of a simple card which is issued for identification purposes, and more sophisticated versions which incorporate a PIN number for the user and which may be used to allow access to restricted areas of a building.

Introduction : The Mis-match problem.

In the early 1970s a play about the French Revolution was put on at London's Roundhouse. The production caused quite a stir because it did not use traditional proscinium arch or 'in the round' staging. Instead, the performance took place on rostra placed, seemingly at random, in an otherwise bare auditorium. The audience had to follow the action of the play from rostrum to rostrum. In the programme, the play's director described how he had to find completely new techniques to stage the play in this unusual way. Although his training and experience in traditional theatre helped him, to a certain extent, to understand the content of the play, they were of little use when it came to transposing it into the medium of the stage. A new approach had to be found which would map successfully onto this novel way of staging the play.

The director's problem is similar to that of the system developer who wishes to design software using an object-oriented approach. In the process of developing a computer system it is difficult, if not impossible, to leap straight into fully fledged program design. Prior to this a system developer needs to model the problem area as a means of understanding it and of specifying a solution.There is a wide range of techniques available to help the developer at this stage of the work, from mathematically-based formal specification languages to structured techniques such as entity relationship and data flow diagrams. However, at present there are no widely used techniques which accurately mirror the object-oriented approach to system design. This means that, when object-oriented design techniques are used, there is always an underlying incompatibility between the modelling and the design of the computer system.

The underlying concept of object-oriented design is that of the classification of objects. Objects contain data and the operations which are required to manipulate the data. Classes of objects are used to capture common characteristics which objects of the class share. A class can be viewed as abstract in the sense that the data it describes is not instantiated and it can be defined by its operations alone. Since an abstract data type is a set of values of a type characterised by the operations which can be performed on the type, an abstract data type can be used to define a class of objects. A study of design using abstract data types can be found in (Britton and Doake 1988).

In this paper we use the example of a system which issues identification cards. Figure 1 shows a diagrammatic representation of the abstract data type CARD with some of the operations that can be performed on it.

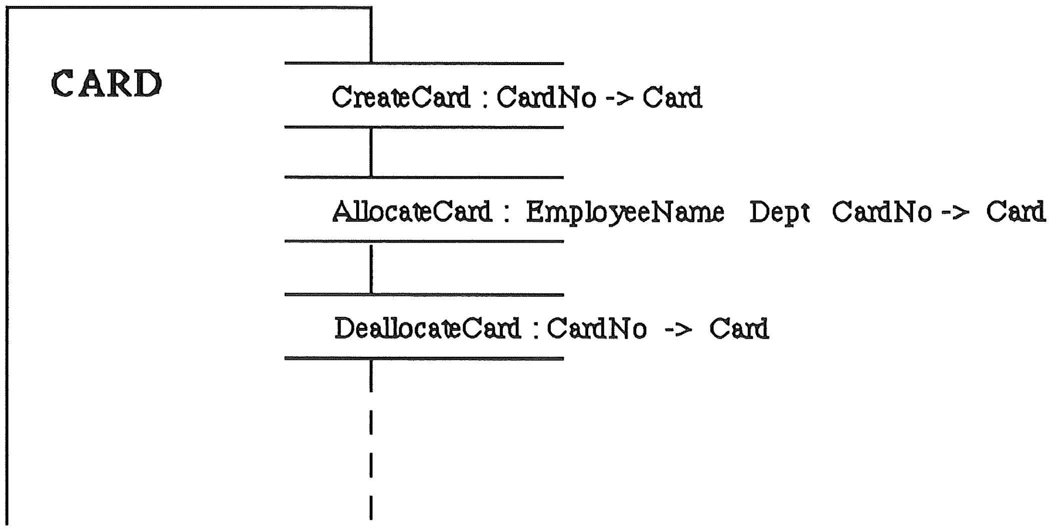

CARD
CreateCard : CardNo -> Card
AllocateCard : EmployeeName Dept CardNo -> Card
DeallocateCard : CardNo -> Card

Figure 1 : Part of the abstract data type CARD.

As we are using a very small example, it is possible to draw the abstract data type for the CARD object class without any previous data modelling. However it is most unlikely that this would be the case in a real system. In this paper we are using our simple abstract data type to emphasize the nature of the mapping from models produced by different structured techniques to object-oriented design. In the rest of the paper we look at examples of the most popular structured modelling techniques : data dictionary, entity relationship diagrams, data flow diagrams and entity life histories. Our aim is to see what common ground, if any, exists between the underlying concepts of these techniques and the concepts of object-oriented design.

Data Dictionary and Entity Relationship Diagrams.

Data dictionary and entity relationship diagrams both model the data objects in a system, the former in terms of their internal structure and the latter in terms of the relationships that exist between objects.

As an example, we consider the modelling of a card which is used for identification purposes. In its simplest form, the data dictionary definition of such a card might be :

Card = CardNumber + EmployeeName + Department

We might also want to model the fact that an employee may have more than one card during the course of his employment, and a card may be returned and reissued to someone else. The fragment of an Entity-Relationship diagram which relates cards to employees might look as shown below.

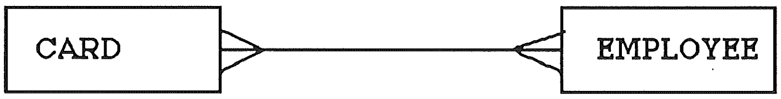


Figure 2 : Part of the initial entity relationship diagram for the card issuing system.

This does not give us a very accurate view of the situation, since we could infer from it that one card is issued to several employees at once. We can, however, refine the model to show that the issuing of a card only involves one employee and one card.

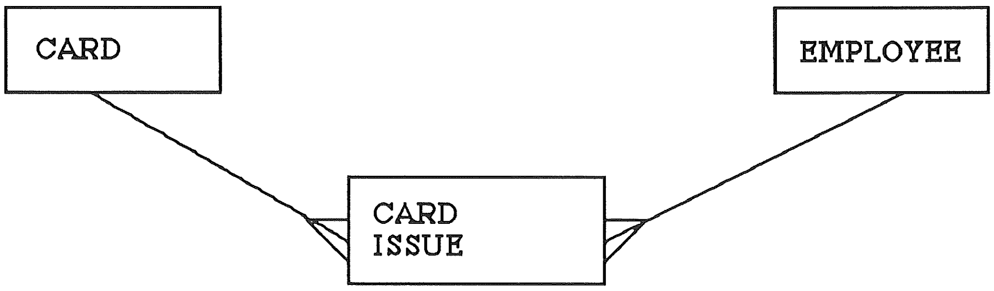


Figure 3 : Refinement of the entity relationship diagram for the card issuing system.

The combination of the data dictionary definition and the entity relationship model gives us a clear picture of the data object CARD and how cards and employees are related in the system. Since the model produced is based on the data in the problem area, it could be argued that these techniques are similar to the object-oriented approach. However, neither of these techniques addresses the problem of modelling the functionality of the data. We cannot , for example, tell from these models anything about the operations which might be performed on a card object in the system. In terms of mapping these models on to an object-oriented design we still have only half the picture.

Data Flow Diagrams.

A data flow diagram identifies the data stores, data flows, external entities and processes in a problem area. It could be argued that these diagrams model both data and functionality. However, in looking for a correspondence to object-oriented design, we find problems with the data flow approach.

The driving force of decomposition in data flow diagrams is based on processes. The starting point for each subsidiary level of a data flow diagram is a process from the level above. The data stores which the process accesses and the data flows which constitute its input and output remain the same. Consequently the data represented in these diagrams remains global. There is no concept of data hiding as is found in object-oriented design.

As an example we consider a process Allocate Card, in a level 1 data flow diagram, which issues a new identification card to an employee. This process accesses a data store, Allocations. It takes as input the data flow Employee Name and outputs the flow Allocated Card.

Figure 4 shows this partial data flow diagram.

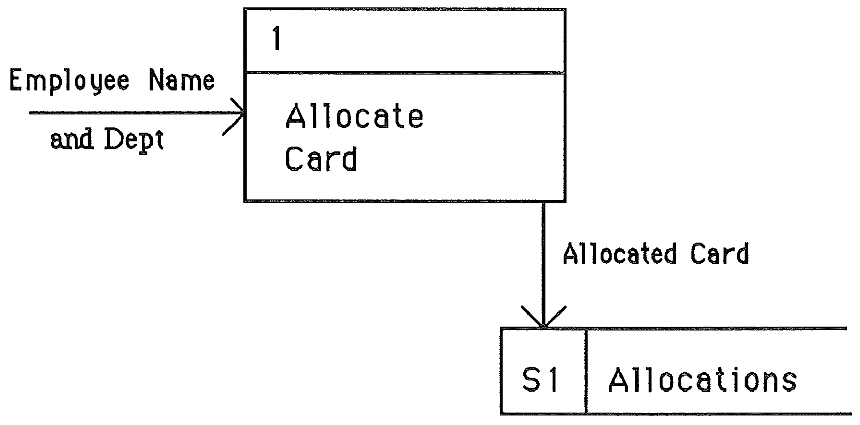


Figure 4 : Part of level 1 data flow diagram for the card issuing system.

The level 2 data flow diagram decomposes the Allocate Card process into two separate sub-processes: Check Employee and Issue Card. Check Employee ascertains that the employee whose name has been input is entitled to a card. Issue Card takes a card number from the set of available cards and issues it to the employee.

Figure 5, below, shows the level 2 diagram.

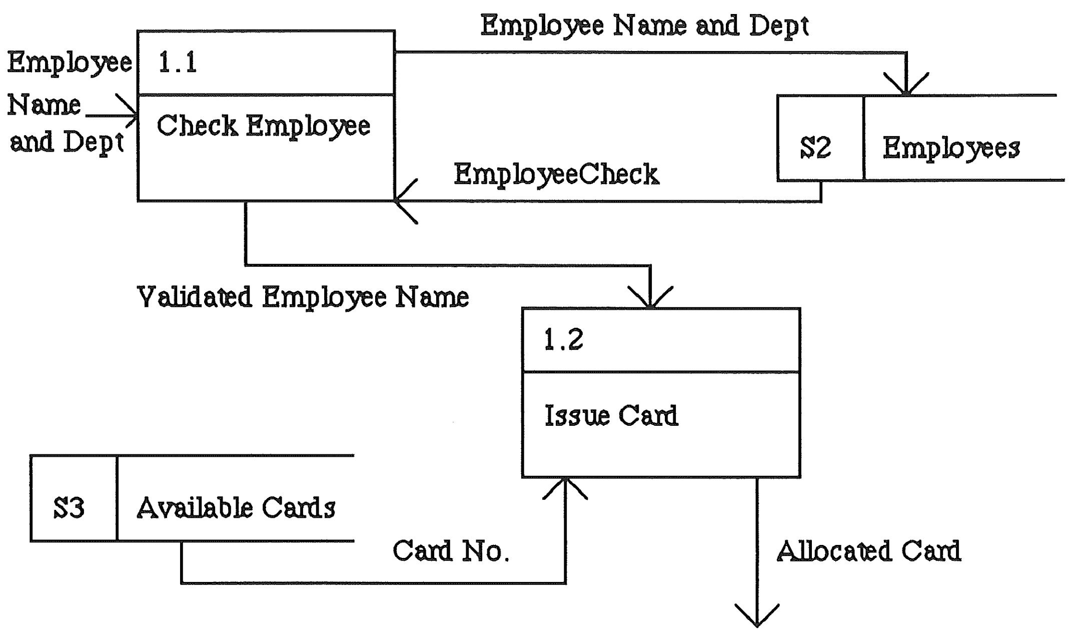


Figure 5 : Level 2 decomposition of process 1, Allocate Card.

From the level 2 diagram we discover a lot more about the level 1 Allocate Card process. We can now see that the process is made up of the two sub-proceses, Check Employee and Issue Card, together with two further data stores, Employees and Available Cards, and some internal data flows. We do not, however, discover anything more about the data from the level 1 diagram, such as the structure of the data store Allocations or the data flow Allocated Card. It is true that both data and functionality are present on all levels of a data flow diagram, but it is the functionality, as represented by the processes, which this technique illustrates.

A further problem with the data flow approach specifically concerns the mapping between the model and the design. Assuming that we wish to structure a design according to object-oriented principles, it will be helpful to produce a model which reflects the underlying object-oriented structure. In a data flow diagram the data is transferred round the system from process to process. In our example the data flow 'EmployeeName' flows into the process 'CheckEmployee' and then into the data store 'Employees'. This is completely opposite to the object-oriented approach, where the data is encapsulated in an object and it is the messages between these objects which are passed round the system.

Figure 6 shows the original CARD abstract data type (Figure 1) and the levelled data flow diagrams (Figures 4 and 5). A comparison of the diagrams shows the large gap that exists between the object-oriented and data flow views of the system. The data flow approach can give the system developer an understanding of the problem domain, but it cannot provide a structure for the specification which will map cleanly onto an object-oriented design.

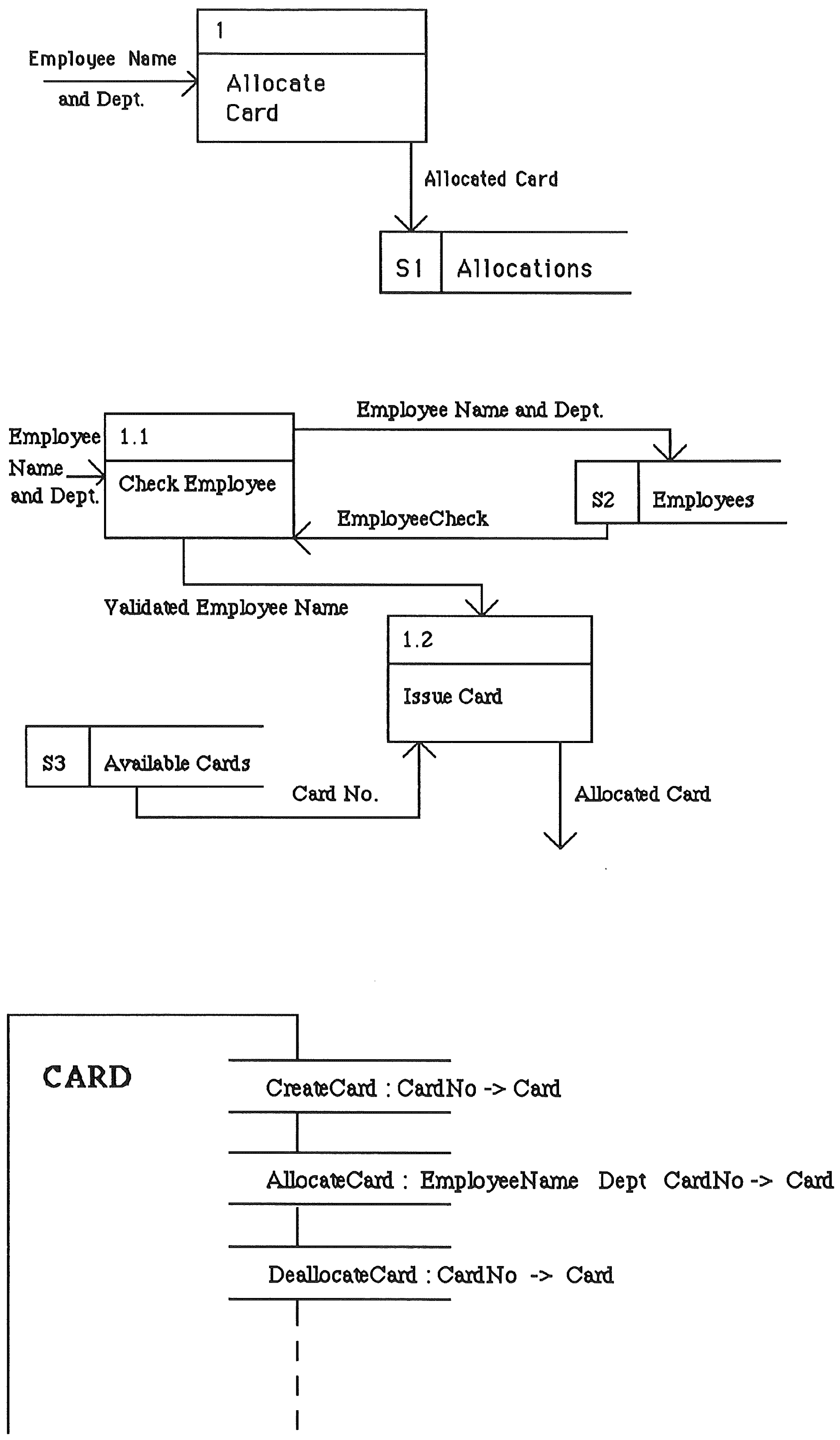


Figure 6 : A comparison of the data flow diagrams and the CARD abstract data type.

Entity Life Histories.

The technique of entity life histories is used in currently popular methodologies such as JSD and SSADM. In SSADM entity life histories are used during the anaylsis phase to give a perspective of sequence and time to the overall model of the system. In JSD a very similar technique, which Jackson calls the Entity Structure Diagram, is used at the start of the development process to describe objects of interest in the problem area in terms of the time-ordered actions which can be performed on them. A study of this can be found in (Buchanan 1990).

Figure 7 shows a simple entity life history for a card in the card issuing system using the JSD notation (Jackson 1983).

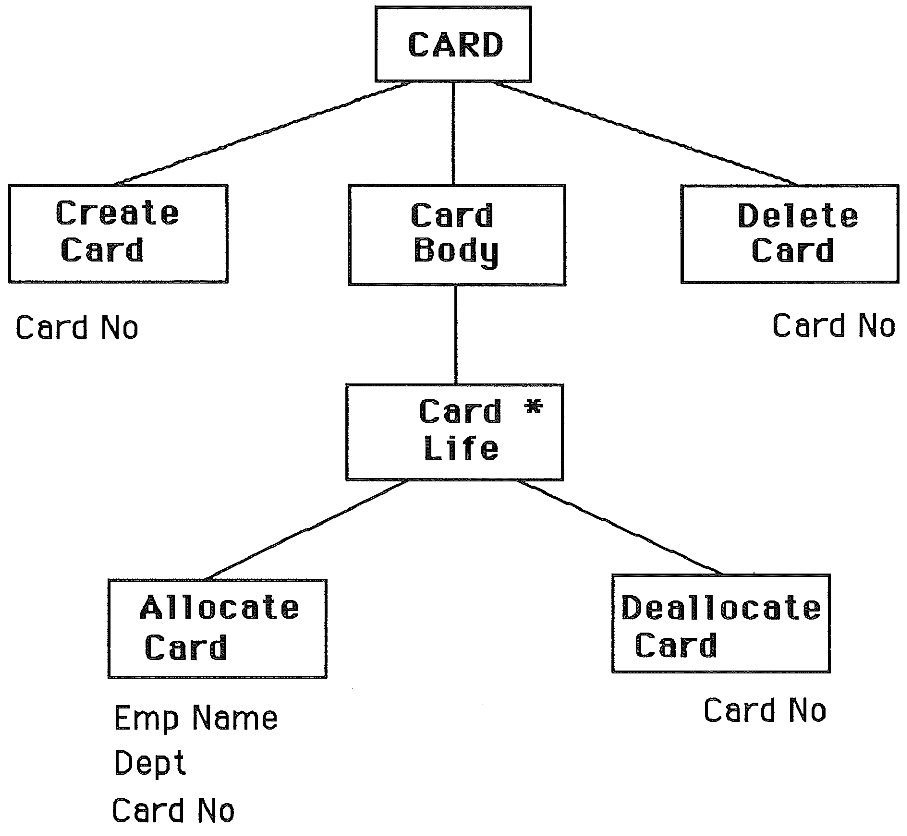
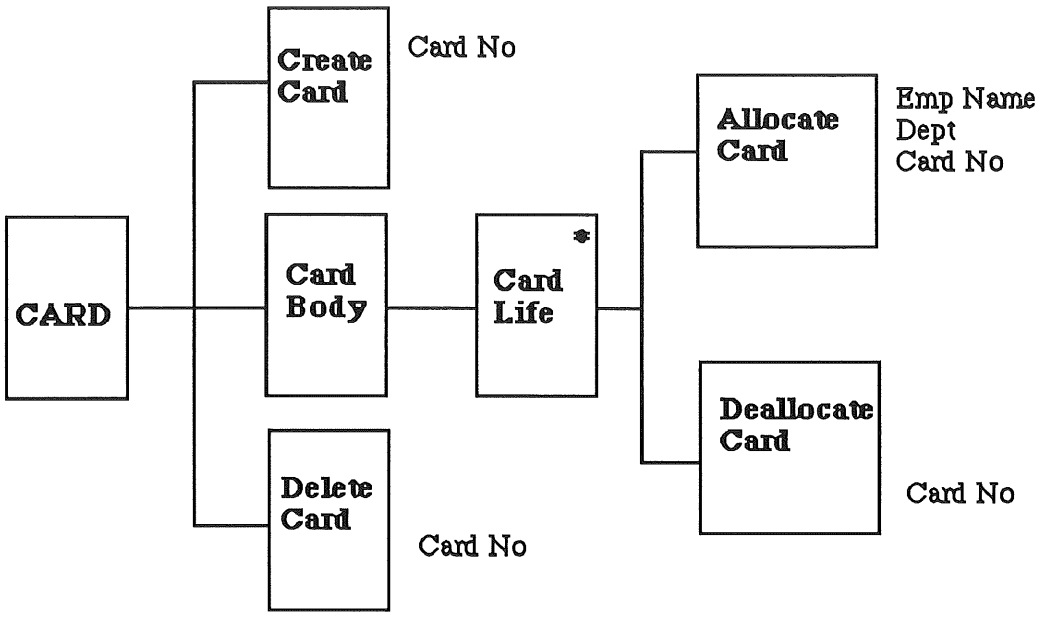


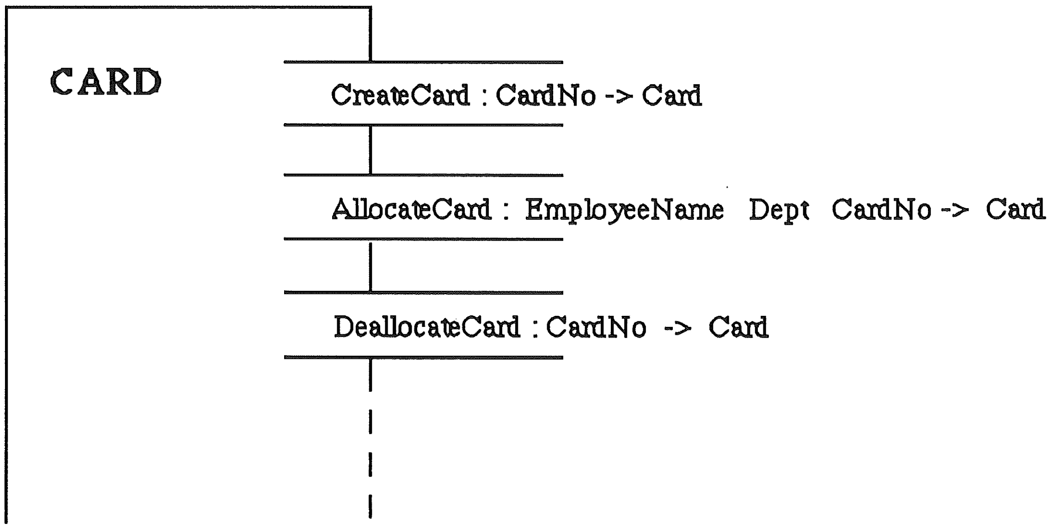
Figure 7 : The CARD Entity Life History

From the diagram we can see that, during its life in the system, a card may be allocated and be deallocated. The * symbol in the Card Life box tells us that this sequence of events can happen any number of times or not at all.

The entity life history describes the class Card in terms of the events or actions which may effect any occurence of the card object. If we rotate the diagram through 90^{o}, we can see the similarities with the CARD abstract data type diagram.

Figure 8 shows a comparison of the rotated entity life history from Figure 7 with the original abstract data type diagram from Figure 1.





CARD
CreateCard : CardNo -> Card
AllocateCard : EmployeeName Dept CardNo -> Card
DeallocateCard : CardNo -> Card

Figure 8 : Comparison of CARD entity life history and CARD abstract data type

By comparing the two diagrams in Figure 8 we can see the underlying similarity in the techniques of the entity life history and the abstract data type. Both techniques model the data object CARD from the same viewpoint: in terms of the actions or operations that can be performed on it.

In highlighting this similarity between entity life histories and abstract data types we are not advocating a choice between the two techniques. Both are necessary for sucessful design. There are important differences in the models produced which each give the designer separate insights into the problem.

In the entity life history all the actions identified are time-ordered. In the example in Figure 7 a card cannot be allocated before it is created, or deleted before it is deallocated. This time-ordering constraint can help the customer and the developer when they are trying to capture all the actions belonging to a particular data object. It is relatively straightforward for a customer to think about actions in the order they are performed. Time ordering makes the entity life history a useful vehicle for discussions with the customer about what happens in the system.

The abstract data type is, in this sense, a higher level abstraction. Time ordering can be imposed on it externally, but the diagram itself has no ordering concept. The abstract data type is thus more flexible and more versatile than the entity life history. If we look again at the card in our example, we can see from the entity life history that a card is first created and then allocated in a separate action. This implies that valid cards can exist in this system, although they are not allocated to anyone. This time-ordering constraint on the actions which create and allocate a card can be reinforced in a design using abstract data types, by imposing an external order on the abstract data type operations CreateCard and AllocateCard. This is what is required for the particular system in our example. We can, however, imagine a situation where all cards are allocated as they are created. The same abstract data type model can be used for this scenario, with a different external order imposed on the operations.

A further difference between the entity life history and the abstract data type concerns the way in which the actions or operations are described. The actions in an entity life history can include attributes, such as Card No, Emp Name, and Dept, but these are not an integral part of the diagram and are often omitted. In the abstract data type diagram the full signature of each operation is given, showing both the parameters and the result. The abstract data type model gives a much more precise picture of the operations than the entity life history.

It should be stressed that the differences that we have identified between the entity life history and the abstract data type are merely differences of detail. By comparing the two diagrams of the CARD data object in Figure 8, we can see that the same view of modelling the data underpins both techniques. From the comparison we can feel a certain amount of confidence that modelling a problem area initially with entity life histories and then moving into abstract data types and object-oriented design would prove to be a natural progression.

Refining Entity Life Histories.

Up until now we have only been looking at object-oriented design in its initial stage, as a technique based on abstract data types. As such, we have shown that there is a strong correspondence between the underlying concepts of entity life histories and those of object-oriented design. There are, however, further aspects of object-oriented design which are fundamental to successful system development. One of the most important of these is the concept of inheritance and consequent reusability.

The concept of potential reusability is one of the main advantages of object-oriented design. The goal of avoiding reinvention of the wheel with each system development seems more attainable with object-oriented design than with systems designed using process-based techniques. One of the central tennets of reusability is the idea of inheritance. In object-oriented terms this means that one data object can inherit and utilise the properties of another.

Although entity life histories appear to be the most useful technique for modelling individual data objects, they do not, at present, provide a method of describing links between these objects. We therefore have no obvious way of defining the inheritance relationship between objects at an early stage in the development of a system. However, it is possible to extend an entity life history by adding new actions to the structure, thus creating a new, more sophisticated data object.

As an example, we can enhance our original CARD entity life history to include a PIN number for the user. The PIN number is added when the card is allocated to an employee and can be changed as many times as necessary during the time the card is allocated. Figure 9 shows the extended entity life history, CARD+.

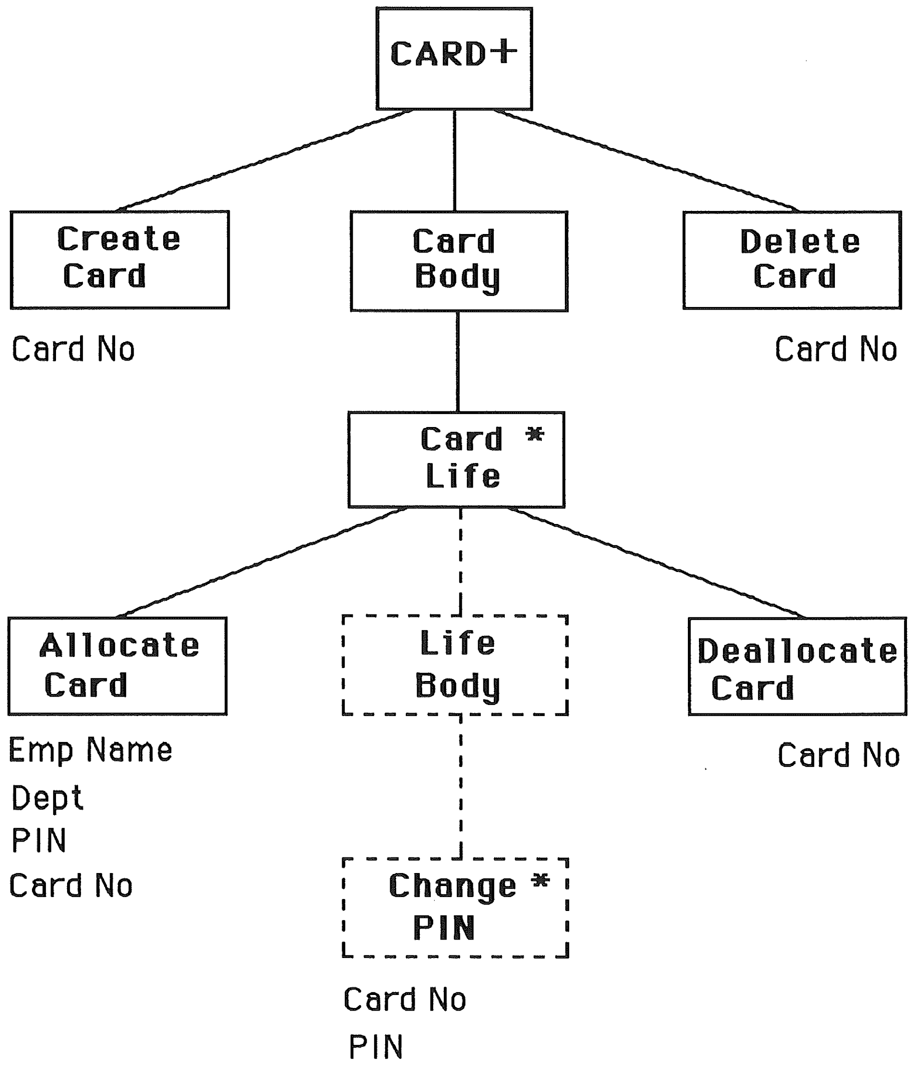


Figure 9 : The CARD+ Entity Life History.

The dotted lines in Figure 9 indicate additions to the original CARD entity life history. The diagram shows how the data object CARD+ has been created from CARD, inheriting all the actions and attributes of the CARD data object.

The technique of extending an entity life history can be used to create several different types of CARD object. Figure 10 shows a more sophisticated version of a card, ENTRY CARD, which can be used to allow access to restricted areas of a building. In the ENTRY CARD entity life history the attributes of the action Allocate Card include the room numbers which the employee has permission to enter. The Change Rooms Access action shows that these room numbers can be changed if the employee is given permission to enter a different area. As in Figure 9, the dotted lines indicate additions to the original CARD entity life history.

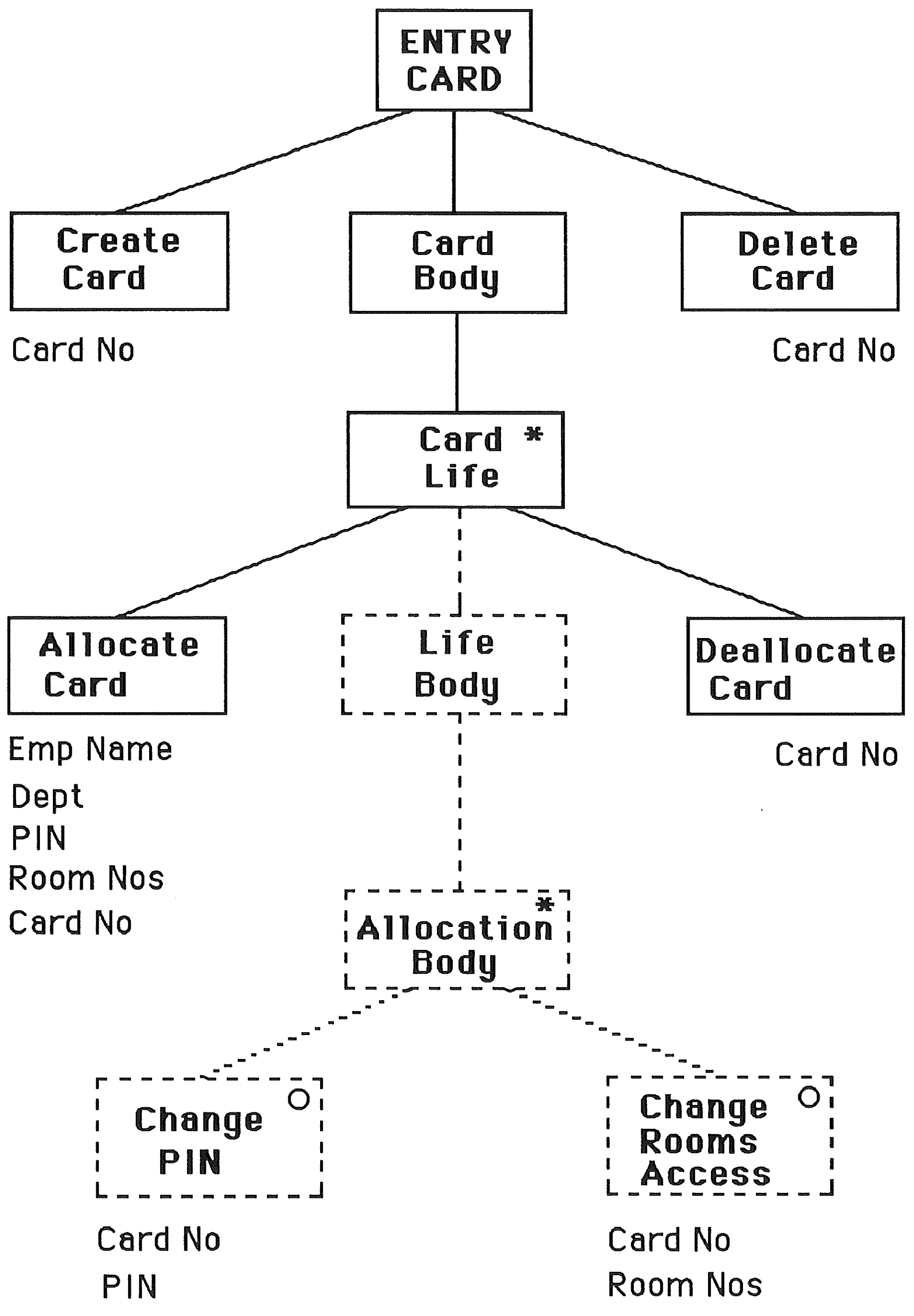


Figure 10 : The ENTRY CARD Entity Life History.

Conclusion

In this paper we have highlighted a basic incompatibility between some structured modelling techniques and object-oriented design. We have shown that, among the most popular techniques, entity life histories provide the best mapping from an initial model to an abstract data type which can itself define a class of objects. We have also shown how entity life history diagrams can be extended to model simple inheritance.

The technique of the entity life history provides a useful diagrammatic representation of data objects which can be transformed into abstract data types. However, the entity life history is an informal technique, with all the inherent problems of imprecision and ambiguity. In order to define more rigourously the abstract data types or the entity life histories themselves, it is possible to use a formal specification language such as OBJ or CCS. The benefits and problems of this step will be the subject of a further paper.

References

Booch 1991
Booch, Grady
Object-Oriented Design with Applications
Benjamin Cummings 1991

Britton and Doake 1988
Britton,C.E. and Doake, M.J.
Taming the Abstract Data Type
MSc. Project
Division of Computer Science, Hatfield Polytechnic 1988

Buchanan 1990
The Phantom of the Object
Buchanan, K.M.
MSc. Project
Division of Computer Science, Hatfield Polytechnic 1990

Downs,Clare and Coe 1988
Downs,E., Clare,P. and Coe,I.
Structured Systems Analysis and Design Method
Prentice-Hall 1988

Jackson 1983
Jackson, Michael
System Development
Prentice-Hall 1983

Meyer 1988
Meyer,B.
Object-Oriented Software Construction
Prentice Hall 1988