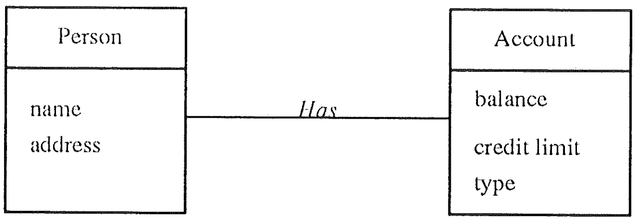
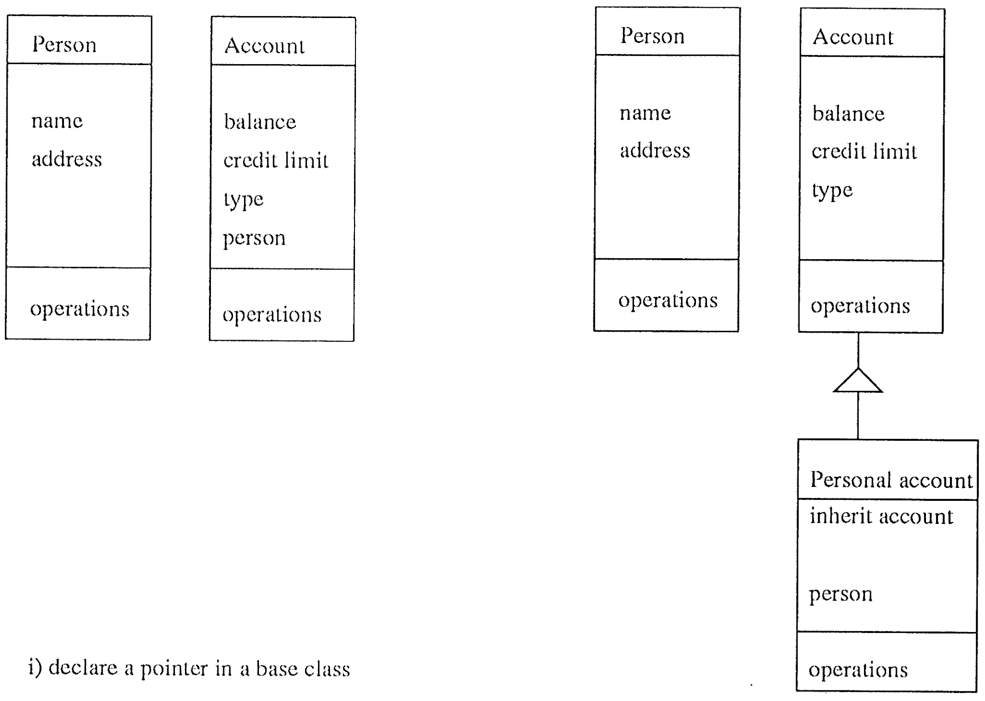


Figure 3.13: Simple banking application object model



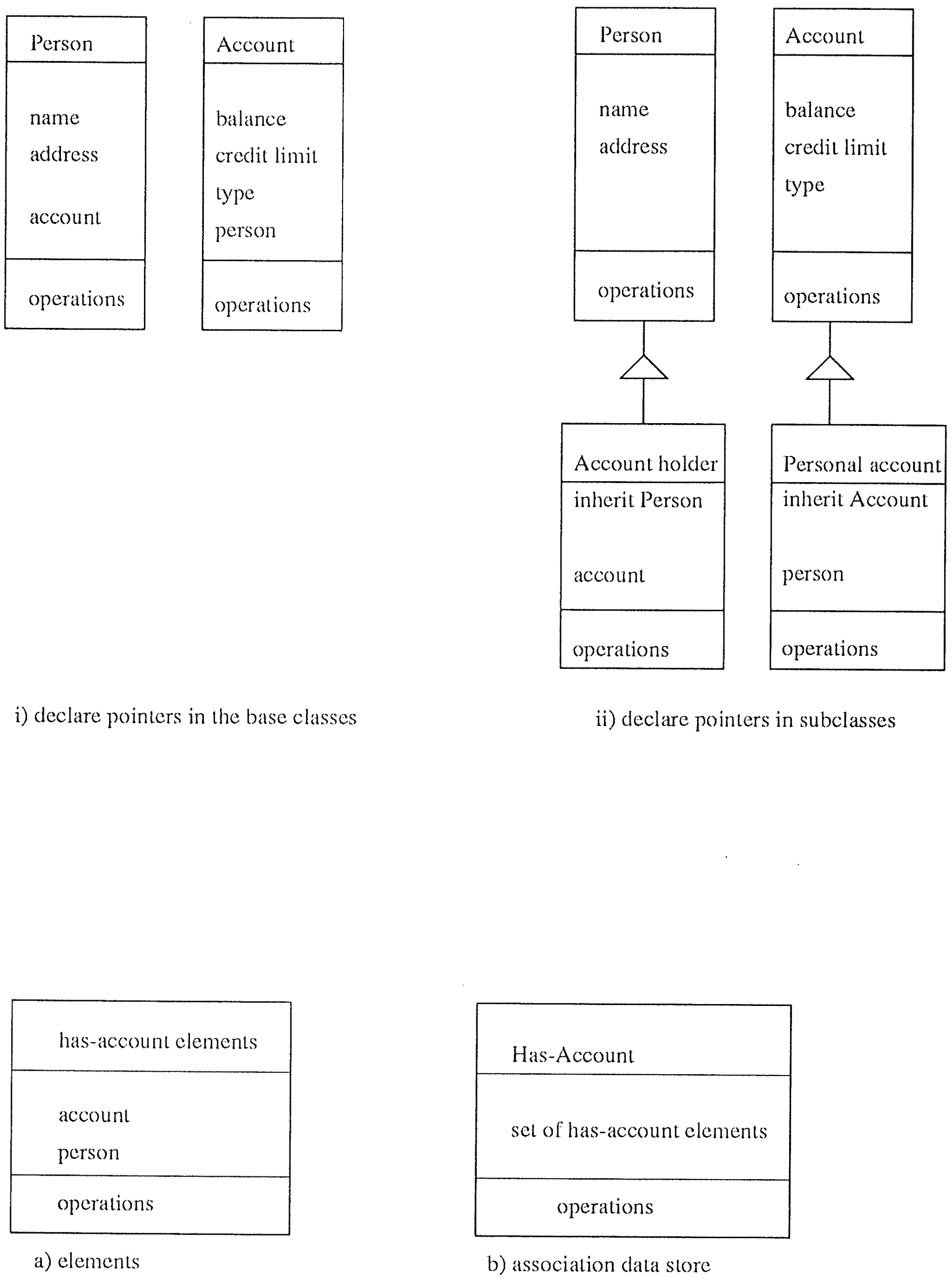
ii) declare a pointer in a subclass

Figure 3.14: Implementing one way associations

Visibility need only be provided in one direction. Of course, such a decision immediately removes some of the information about a relationship and reduces traceability. The model indicates that an account must have one person associated with it, so the account class would declare an instance of person. The model also indicates that the value of the instance of person must never be NIL, that is it must always be assigned to a person object. There are two alternative ways to implement associations to give one way visibility.

- Option 1—declare a pointer in the base class

  Implement the **Person** class with the attributes and methods defined in the analysis model. The **Account** class is then implemented with an instance of **Person** as an additional attribute. The methods required to access the person are also added to class **Account**. The implemented **Account** class does not then match the **Account** class in the analysis model as is shown in figure 3.14(i). The **Account** class is bound to the **Person** class and cannot be used in a system without including the **Person** class.



iii) use a set to represent the association

Figure 3.15: Implementing two way associations

- Option 2—declare a pointer in a subclass

  Implement both the **Person** and the **Account** classes as they appear on the analysis model. A new subclass of **Account** is then defined to add an instance of **Person** as an instance variable. This class could be called **Personal Account**. The classes **Person** and **Account** both match those in the analysis model but a new class has been added to implement the association. This gives three classes in the implementation, as shown in figure 3.14(ii), when only two were identified in the analysis model.

However, it is possible that the association must provide visibility in two directions. There are several possible ways of providing this.

- Option 1—use search routines

  The association itself could be implemented using a one way link as indicated above. Search routines would need to be provided to implement visibility in the other direction. This method of implementation may require lengthy searches and reduces the speed of the system.

- Option 2—declare pointers in each of the base classes

  Each class declares an instance of the other as an attribute in a reciprocal client-server relationship as shown in figure 3.15(i). This provides a one-to-one association between objects. The classes do not have the same attributes as those in the analysis model and are bound together.

- Option 3—declaring pointers in subclasses

  The base classes are implemented as identified in the analysis model. Subclasses are then declared to implement the association as shown in figure 3.15(ii). This results in four classes instead of two being required.

- Option 4—implement the association as a set

  A separate class can be added to implement the association. This class defines association objects as sets of pairs of objects. For example, the Person has Account relationship could be implemented as a set of has-account elements as shown in figure 3.15(iii). Each has-account element would contain a pointer to a **Person** and a pointer to an **Account**. This allows the classes modelled during analysis to be directly implemented. The classes are therefore traceable. However, the objects participating in a relationship have no knowledge of the relationship so the association is not traceable via the objects involved.

The availability of the different possible representations of a one-to-one association shows that this relationship is not traceable in the implementation. The situation is more complicated when one-to-many or many-to-many associations are implemented. In these cases, a data structure must be declared to store the objects at the many end of the association.

It was reported in section 2.2.1, that Lamb suggests that components be factored to provide small, simple components in order to enhance reusability. This suggests that it is better to implement the classes directly from the analysis model and then either add subclasses which implement the associations between the classes or define a set to represent the association.

As mentioned in section 3.2.3, OMT allocates attributes to associations if required and also identifies qualified associations. These also need to be implemented. The guidance given is to store the attribute in either class involved in a one-to-one association or in the class at the many end of a many-to-one association. In the case of a many-to-many association, a distinct association object should be introduced to allow the qualified associations to be implemented. The attributes identified as qualifiers of an association are implemented as attributes of the class they qualify.

The Responsibility Driven Approach does not model associations directly. They are converted into collaborations before they are modelled. Collaborations are unidirectional. There is no mention of how to deal with situations where a two way collaboration is required.

The availability of alternative mechanisms for implementing conceptual associations hampers traceability.

- **temporary association**

  The identity of individual objects involved in temporary associations is not significant. Temporary associations are implemented by the simple client relationship. An instance of one class becomes a parameter to a function provided by the other class. The required information can therefore be obtained from or supplied to the object but there is no persisting record of the actual instance involved.

  Parameters are required for reasons other than the implementation of a temporary association so this relationship is not readily traceable.

- **instantiates**

  This relationship occurs when one object is responsible for creating an instance of another object. This is another relationship implemented by the client-server relationship. The client class needs to declare an instance of the class to be created or instantiated. This relationship is not easily traceable.

- **dependency**

  This relationship is implemented by the client-server relationship. The example of this relationship given earlier, in section 3.3 is:

  A minimum balance in an account (A) is maintained by transferring funds from another account (B). This method is invoked on the basis of the balance in account A and causes a change in the value of the attribute balance in account B.

  Account A would declare an instance of account B to allow access to the funds stored in B. The reason for the relationship is hidden.

Table 3.6.2 summarises the above conversions. The brackets indicate that the relationship is not available in all object oriented languages. It shows that seven of the eight relationships identified during analysis are converted into the client-server relationship for implementation. Six of these seven relationships are implemented by the declaration of instance variables in the client class.

3.6.3 Subsystems

The implementation of subsystems is not directly supported by the languages investigated. However, it is possible to group classes together to form subsystems.

In Eiffel, this can be done by declaring the classes which form a subsystem in one directory. The grouping of classes in this way does not affect the interface of the classes. That is, the interfaces of the classes form the interface to the subsystem. It is possible to add a class to represent the subsystem and to use the Eiffel selective export feature to ensure that the features of the subsystem are accessed only via the subsystem class.

analysis relationship	programming construct used
is-a	inheritance
consists of (one)	simple client (or expanded client) via instance variable(or selective export,friend), inheritance
contains	simple client (or generic client) via instance variable
uses	simple client (or privileged client) via instance variable
conceptual association	simple client (or privileged client) via instance variable or extra class
temporary association	simple client via a procedure parameter
instantiates	simple client (or privileged client) via instance variable
dependency	simple client (or privileged client) via instance variable

Table 3.2: Conversion of relationships

The other three languages, C++, Oberon-2 and Modula-3, all allow more than one class to be declared in a file. This allows all the classes involved in a subsystem to be implemented in one file. A class to represent the subsystem can also be declared. This class can be the only class exported, so forming the interface to the subsystem.

All the methods available for implementing subsystems allow them to be identified in the implementation but involve restricting access to a class and its objects. This has the effect of reducing the reusability of the individual classes because they are bound to the class which implements the subsystem. The implementation of subsystems therefore improves traceability but reduces the reusability of classes.

3.7 Design gains and losses

This section completes the investigation into traceability of information through the process of object oriented development. Section 3.6 showed that the designer has to make many decisions about which mechanisms to use to represent the different types of information. This section discusses the effects of the different design decisions on different aspects affecting the reusability of the classes. The use of inheritance and the implementation of subsystems are discussed first. The problems caused by using the client-server relationship to implement many different types of implementation are then explained with particular emphasis being placed on the implementation of conceptual associations.

The inheritance relationship is available in both analysis and implementation but the two constructs are not used in the same way. In analysis, it is used for subtyping but in programming it is commonly used for code reuse. The difference in meaning impairs understanding of the design and may reduce the ability to extend the system.

In section 3.6.3, several ways of implementing subsystems were identified. The methods suggested are useful in a single system but prevent the classes which form the subsystem being reused in a different system. The designs used for implementing subsystems therefore improve traceability but reduce reusability.

Section 3.6.2 explained the decisions which designers were required to make in order to translate the information contained in the eight relationships identified during analysis into the two constructs provided by the implementation languages. Some of the information contained in the analysis model is thereby lost. For example, the use of the client-server relationship to implement seven different relationships makes it difficult to distinguish between attributes representing part of an object's structure and attributes representing the other relationships. Classes which are related because they are involved in these other relationships are bound together in the same way as classes which combine together to form a complex structure. The information to distinguish between these different constructs is lost with the result that the implemented classes become less easily recognisable as models of the objects which they represent. The loss of information occurs at different stages of development. The RDA method identifies the relationships but converts them into programming language constructs before modelling. Much information about the different relationships is therefore lost before the analysis model is produced, whereas in the Coad and Yourdon and OMT methods the information is lost during the design stage.

It is possible, when using Eiffel, C++ or Oberon-2 to implement the system, to distinguish between the structural components of an object and the associations in which it is involved. Eiffel, C++ and Oberon-2 (but not Modula-3) allow objects to be declared either as instances (expanded objects), or as pointers to an object. Structural components can be declared as expanded objects and associations as pointers. The use of expanded objects has the advantage of ensuring that one component cannot be part of two separate objects. For example, it would make it impossible for one wheel to belong to two or more cars. Of course, there are situations in which it is desirable for one object to be part of two separate objects. In a document processing system, for example, it is possible that one figure may be required in two documents. If figures are declared as expanded objects within a document on the grounds that figures form part of the structure of a document, two copies of the figure would be needed. This could cause problems if the figure was updated. The use of expanded objects allows a distinction to be made between structural components and other relationships between objects. There is some traceability of information. This does not provide an ideal implementation because both structural relationships and associations are implemented by variations of the client-server relationship so the classes involved are still bound together. Reusability of the classes is not improved.

Section 3.3 suggested that many of the relationships identified during analysis come into the category of conceptual associations. The representation of these relationships is therefore an important issue. Several different methods of implementing a one-to-one conceptual association between two objects were detailed in the previous section. The availability of several different implementation techniques impairs reusability by reducing the traceability of the information.

The different techniques can be thought of as three basic options. Each option can have different detailed interpretations. The three basic options are:

1. declare an attribute in one or both of the base classes involved in the association,
2. declare an attribute in a subclass or subclasses,
3. add a data structure to store pairs of associated objects.

Each one of these options affects the reusability of the implemented classes. It should be remembered that reuse has been defined in this thesis to include reuse in an extension of an existing system. It is therefore necessary to consider the implications of using each technique when an extension to an existing system is required. These implications are explained by considering an extension to the simple banking system discussed in section 3.6.2. The extension adds the ability for a person to own shares as shown in figure 3.16. Each of the basic options for implementation is discussed in turn.

1. Declare an attribute in one or both of the base classes

   This method can be used to implement the association to allow access in one direction only by adding an attribute to one of the base classes. The addition of an attribute and the extra operations required to access it reduces the correlation between the classes in the analysis model and those implemented. That form of implementation also loses the bidirectional nature of the relationship

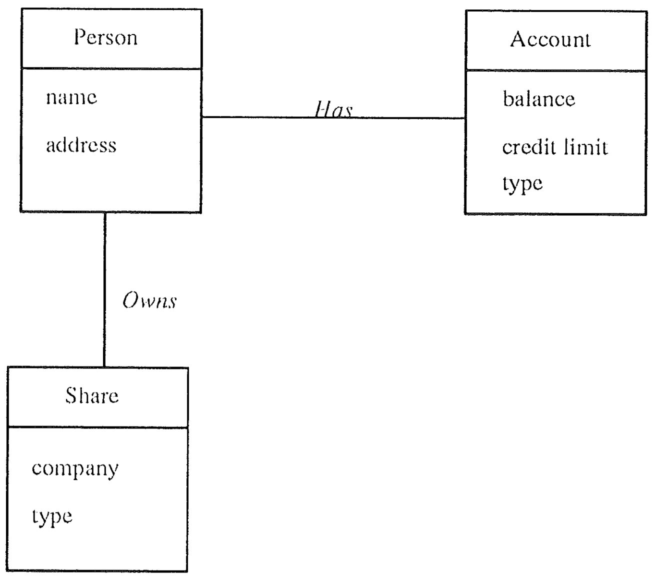


Figure 3.16: Object model of extended system

and requires that the values of the associated objects are changed from the implemented end. For example, if the association between a person and an account was implemented as a pointer from the account to the person, the attributes of the person would need to be changed via the account unless another association was provided to allow updates.

The decision to implement a one-way association also restricts the extensibility of the system. Using the above example, it is more difficult to add functions which require access to the account via the person object. A search routine would need to be added to traverse the association in the reverse direction. This type of routine is very expensive in computer time.

Another problem of this one-way implementation is that the class which implements the association is bound to the class with which it is associated. It can be used only if both classes are included in the system. The other class is, of course, independent of the association and can be reused as required.

The same basic method can also be used to implement a bidirectional association by declaring attributes in each of the base classes. This is also known as declaring converse pointers. The use of this method binds both classes together so that neither can be used in a system without the other. This restricts the reusability of both classes.

Another effect of this implementation is that the knowledge about the association is distributed between two classes of object. The information about the association is not encapsulated. This lack of encapsulation can lead to difficulties when maintaining the integrity of associations. The developer must ensure that both values are maintained together in order to avoid inconsistency. This is known as maintaining referential integrity. The difficulties incurred by implementing associations in both directions might be the reason that Shlaer and Mellor [59] state that associations must be unidirectional not bidirectional.

As mentioned previously, consideration must be given to the consequences of using each of the implementation techniques when a system is extended. For this example, two-way associations are implemented. The system has its functionality increased to provide the ability to own a share.

A share class must, of course, be added to the system. This share class must contain an attribute person to provide the association. It is then necessary to implement the association between a person and the share owned. This association is implemented by declaring a subclass of person as shown in figure 3.17(i). This subclass declares an instance of share and the operations required to access this attribute. It is also necessary to ensure that the new classes contain the code required to maintain consistency between the objects involved in the association. This extra subclass reduces the correlation between the analysis model and the implemented system and therefore reduces traceability. The number of classes in the system is also increased leading to increased complexity. Further extensions to the system can result in long inheritance hierarchies which can also reduce the understandability of the individual classes and, as mentioned in section 2.2.7, may lead to increased problems with corrective maintenance.

It can be seen that the implementation of an association by declaring pointers in the base classes impairs reusability in several ways. The classes in the analysis model and the implemented system do not correlate so the information is not readily traceable. The classes become bound together so that both classes may be required in a system even if only one is intentionally being reused. Implementing a two-way association does not provide encapsulation of the information about associations. This means that the code to maintain the correct values must be rewritten for each class. A further consequence is that extensions to the system can result in increased complexity due to the increase in the number of subclasses.

2. Declare an attribute in a subclass or subclasses

   This method of implementation avoids binding the base classes together and therefore makes them available for reuse in other systems. It was shown in figure 3.15(ii) that the number of classes increases and that the correlation between the analysis model and the implementation had been reduced. Thus this method reduces traceability and increases design complexity.

   The information about the association is again distributed if a two-way association is implemented. In this method, the information is distributed between subclasses instead of between base classes. The same problems will still be encountered.

   Figure 3.17(ii) shows the class hierarchy produced if the banking system is extended to provide the ability to own a share. The figure demonstrates that this method of representation produces longer hierarchies when system functionality in increased. This also reduces the traceability of information because the definition of a class is distributed between the class itself and all of its superclasses. This can increase the problems of understanding the full structure and capabilities of a class.

   The addition of attributes to subclasses rather than the base classes does provide some advantages in that the base classes are available for reuse but this method still distributes the information about associations between the classes involved and may result in long inheritance hierarchies.

3. Use a data structure to implement the association

   The association is implemented as a class containing a group of objects. Each of the objects represents a pair of objects involved in an association. The new object is independent of either of the participating objects. This method of implementation has the advantage that the classes of objects involved do not need to be changed in any way. The association is a distinct entity and provides encapsulation of the information about the association. The instances of the association are stored in a new data structure. This data store must be searched in order to find the required

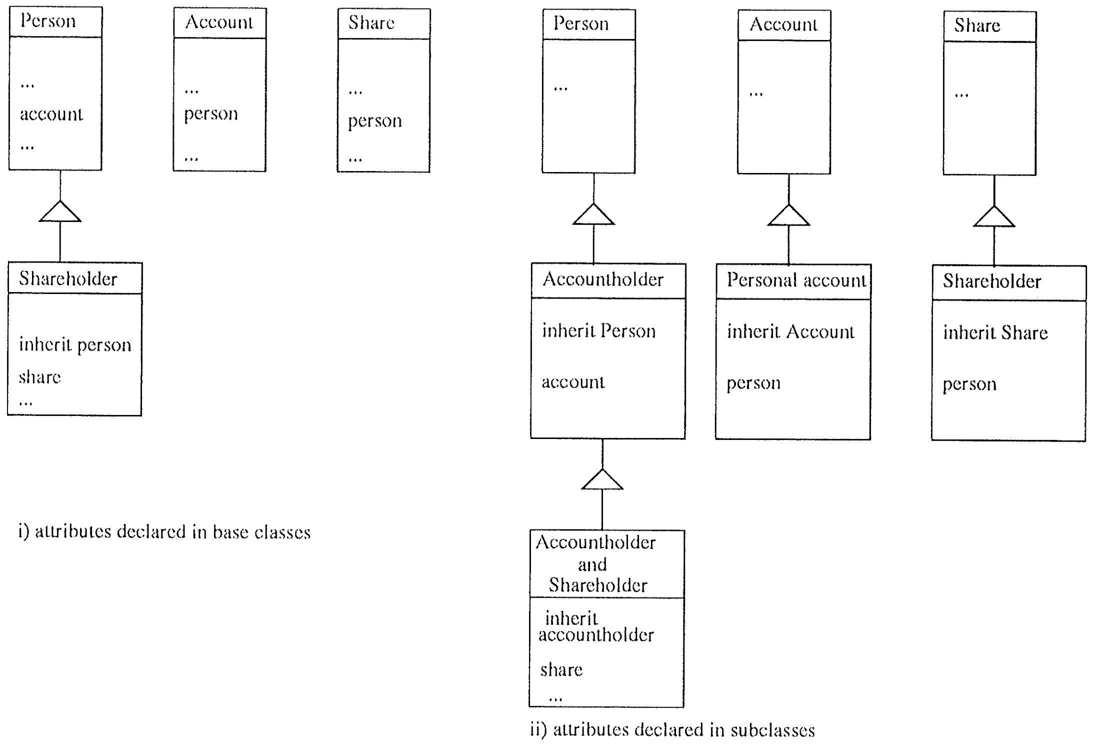


Figure 3.17: Extending a system

objects because the objects involved have no knowledge of the association. The search will increase the access time required but as pointed out by Rumbaugh et al. [28], if a hash table is used to store the details this decrease in efficiency will be minimal. However, two hash tables would be required to allow access from either end of the association. This type of implementation technique would be particularly suitable in situations where few objects are involved in a specific association.

The extension of a system does not require new subclasses to be declared to allow extra associations to be implemented which avoids the increased complexity and lack of traceability of associations caused by the other methods of implementation. However, with this approach the objects have no record of the associations in which they are involved. The objects no longer encapsulate all the knowledge about themselves. Some of the information about objects is distributed between the associations in which they are involved.

None of the three basic options for implementing associations between objects provides traceability of all the information about the associations. The first two options involve the declaration of pointers in the classes. These declarations could be accompanied by comments to explain the reason for their declaration. This would improve traceability but would not prevent the classes being bound together. The reusability of the classes would not be enhanced.

This section has shown that the translations required during the design process reduce the traceability of information and consequently reduce the reusability of software components.

3.8 Discussion

This section assesses the traceability of information during object oriented system development. Object oriented analysis, design and implementation use many of the same terms. All three stages of development use the terms class and object. The meanings of these terms are compatible. During analysis, the term class is used to identify a group of objects with the same structure and behaviour. A class also describes how to create the objects. During implementation, the term class is used to mean the template from which objects are produced. Objects at all stages are defined to be instances of classes and to combine data and behaviour. The class and object concepts are therefore traceable through the development process. The names of the classes of objects identified in the analysis stage of development are identifiable in the implementation giving continuity of representation.

Analysis identifies many different relationships between these classes and objects. These relationships are not traceable in the implementation of the system. They must be converted to the two relationships provided by programming languages, resulting in a loss of information. The stage at which this conversion occurs depends on the methodology being followed. The RDA methodology identifies many relationships but converts these to programming constructs before they are modelled. The model therefore transfers into implementation without the need for translation. The information is lost before the model is produced. The other two approaches examined, OMT and the Coad and Yourdon approach, model several different relationships retaining the information until the design stage converts them into the relationships available in implementation languages. The loss of information occurs during the design phase.

One of the most common relationships between objects is the conceptual association. The use of the client-server relationship to implement such associations may reduce the reusability of classes. Classes which are involved in associations are tightly coupled in the same way as classes which form the structure of another class. The developer is required to add attributes and operations, which effectively adds structure and behaviour, to the classes in the analysis model. The structure and behaviour of the classes are not then traceable from the analysis model to the implemented class. The class in the analysis model is simpler than the implemented class because the implemented class has become application specific. In addition, the implemented classes are less like the original entities they represent and therefore harder to understand. This makes the classes less reusable in another similar system because they can only be reused if they can be understood.

Traceability of information can, of course, be improved by the addition of comments to the code. These comments would give the details of the reason for including attributes in a class declaration, such as structural or to implement an association. This does not, however, improve the actual correlation between the classes in the analysis model and the implemented classes.

The tight coupling of the classes caused by using the client-server relationship to implement conceptual associations results in a reduction in reusability because these classes depend on each other for their implementation. As Kilian [52] notes, all the classes on which a class depends for its implementation must be included in the system. The reuse of a class which depends on many others for its implementation therefore brings a lot of extra classes with it into a new system, thus increasing the size of the new system and causing extra work for the developer in identifying the many dependencies.

Implementing conceptual associations by the addition of attributes to classes does not provide encapsulation of the information about associations. Rumbaugh [60] identifies a problem which arises because of the lack of encapsulation. The problem is that

"interactions are buried in the instance variables and methods of the classes, so that the overall structure of the system is not readily apparent."

The system is therefore less understandable than if the associations were implemented as separate constructs and traceable from the analysis to the implementation.

Difficulties also arise if the client-server relationship is used to implement conceptual associations when a system is extended. The number of classes involved in a system increases to provide new associations even if the classes to be associated already exist in the analysis model of the system. This is because a new subclass must be defined to implement a new association. If bidirectional implementation is required, two new subclasses must be declared. This may result in long inheritance hierarchies which increase the complexity of the implemented system and reduce the traceability of information.

It is possible to implement conceptual associations without using the client-server relationship to link the classes. This involves the use of a data structure to implement the association. The use of this technique stores the association separately from the objects involved with the result that the objects do not encapsulate all the knowledge about themselves. They encapsulate knowledge of their structure and behaviour but not of the associations in which they are involved.

3.9 Summary

This section summarizes the conclusions about the traceability of information from analysis to implementation. From the previous sections, it seems clear the class construct is present in both analysis and implementation. However, the two development stages define different relationships between the classes. The analysis methods identify more types of relationship between classes than can be represented in the programming languages. The information about the analysed relationships is not traceable in the implementation. This lack of traceability was shown to reduce the reusability of classes for one or more reasons. These reasons include:

- The classes implementing an association may become bound together in the same way as classes which represent the structure of a complex object. This reduces reusability because classes can only be reused if both classes are included in the system.
- The implemented classes may not correspond exactly with those in the analysis model. The lack of correspondence may make the selection of a class for reuse more difficult.
- The information about the association may be distributed between the classes involved in the association which makes the structure of the system difficult to understand. This can cause problems when maintaining or enhancing a system.
- Long inheritance hierarchies may be produced when classes are reused in an extension to an existing system. This increases the complexity of the system and again reduces the correlation between the analysis model and the implemented system. This may increase the difficulty of providing further enhancements of system functionality.
- All the information about objects may not be encapsulated with the objects themselves. It may be distributed between the objects and the sets representing relationships between objects.

It is agreed by Rumbaugh and Kilian that providing traceability of conceptual associations by their implementation as separate constructs in object oriented systems would both increase reusability and improve the clarity of system design. The design method presented in the next chapter provides traceability of conceptual associations from the analysis model to the implementation. The conceptual associations are provided as separate constructs. These become part of the structure of the objects involved in the association without modifying their class definition. The separate constructs improve the traceability of conceptual associations. This helps prevent the loss of information and improves the understandability of the classes.

Chapter 4

Sociable classes—a novel design technique for improving traceability

This chapter presents a novel design technique, called the Sociable classes technique. A short report of this technique is contained in [61]. The Sociable class technique provides a means of representing conceptual associations between instances of classes. The representation of conceptual associations improves the traceability of these relationships through the development process. The classes used in the system are implemented directly from the analysis model which improves the reusability of the classes.

A feasibility study into the use of the technique has been carried out. This involved implementing the basic functionality of the required classes in several strongly typed object oriented languages. The feasibility study indicates that the Sociable class design technique might be applicable in various object oriented environments. The design presented in this chapter was developed using the Eiffel programming language and then translated into the other languages. The implementations use current language constructs. It is not necessary to add new features to the languages in order to adopt the design technique but the different language features affect the implementation.

Section 4.1 outlines the problems caused by the lack of traceability of conceptual associations which the Sociable class design technique is attempting to overcome. Section 4.2 details the perceived requirements for a design technique which offers a solution to the problems. (Alternative methods are being developed by other researchers to represent conceptual associations between objects. These methods do not attempt to meet the same design requirements. They are described in chapter 5.) Two new types of classes are necessary for a design to meet the requirements listed in section 4.2. These two types of classes are defined in section 4.3. Section 4.4 gives a specification for these classes and for one subclass of each. Examples of the use of this technique in the development of a new system and the extension of an existing system are given in section 4.5. The features of system designed using the Sociable class design method are described in section 4.6. The language features required to implement these classes are defined in section 4.7. The results of the feasibility study are described in section 4.8. These implementations and other alternatives are discussed in section 4.9. The feasibility study indicates that the proposed design method might provide a viable method for improving the development and use of reusable components. The principle points made in this chapter are summarized in section 4.10.

4.1 Problems to be addressed

The design technique presented in this chapter attempts to address some of the problems which lead to a reduction in the reusability of components in object oriented system development. The problems result from the lack of traceability of different types of information. This lack of traceability arises from the necessity to map several fundamentally different relationships identified during object oriented analysis on to one relationship in object oriented programming languages. The problems are detailed in chapter 3. Briefly the problems are:

- classes which are involved in associations are tightly coupled in the same way as classes which form the structure of another class.
- the implemented objects become less like the entities in the analysis model which they represent and therefore more difficult to understand.
- the structure of the system is difficult to ascertain because the information about the associations is distributed between classes and not encapsulated.
- enhancement of system functionality may result in a large number of subclasses in long inheritance hierarchies.

The overall effect of the lack of traceability is that reusability is decreased. The design technique presented in this chapter provides a mechanism to implement conceptual associations as distinct constructs. This provides traceability of the information about the associations between objects from the analysis model to the implementation.

4.2 Design requirements for a solution

This section identifies the design features required to allow information about conceptual associations to be traceable through the development process into the implementation. A solution to the above problems could be provided by introducing mechanisms to allow different degrees of binding between objects which are not related in an inheritance hierarchy. The different degrees of binding would provide:

- groups of objects which are tightly bound because they represent complex structures or aggregations.
- objects which are loosely coupled to other objects because they take part in conceptual associations. Conceptual associations are defined, for the purposes of this thesis, as logical relationships between specific instances of objects, see 3.3.

The above design points are illustrated in figure 4.1 which shows a general object model using OMT notation. The class AB describes an assembly object formed from objects of classes A and B. This represents tight binding between objects forming a complex structure. Classes F and C are linked to the class AB via labelled lines. These lines represent associations between objects. The objects involved are loosely coupled.

In class based object oriented development, objects are instances of classes. Complex objects such as those defined by assembly class **AB** are produced by using the client-server relationship between classes to add the attributes and structure of a class. Attributes and structure are then tightly bound within the class to which they belong. For example, a car is an assembly of wheels, engine etc.. The client-server relationship is also used to represent associations between objects. The classes involved

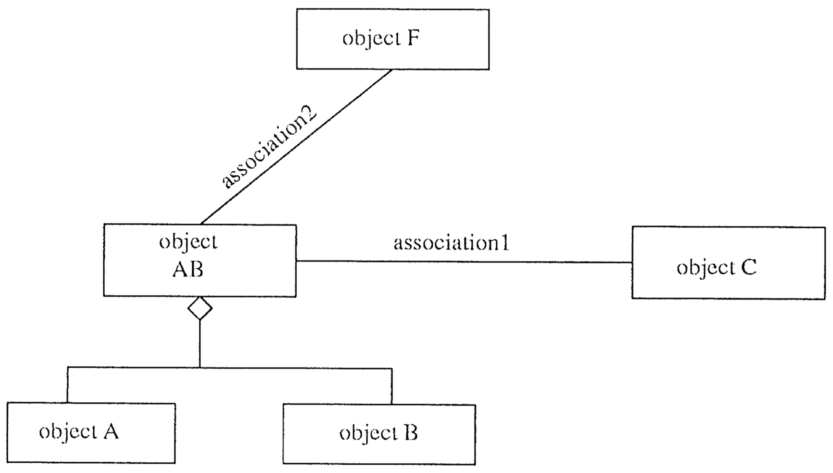


Figure 4.1: General object model

in associations are then tightly bound together even though only loose coupling between the objects is required.

In this suggested design, the client-server relationship is used to implement aggregations. Aggregations are then formed from classes which are tightly bound together. The loose coupling between objects is produced by adding associations between objects. For example, an association between a person and an account is added to implement the association 'a person has a bank account', or an association between a person and a car is added to implement the association 'a person owns a car'. It is not necessary for all objects of a class to be involved in all types of association.

In order for a design to meet the above requirements, the following facilities should be provided:

- a means of explicitly implementing associations between objects rather than between classes of objects.
- the ability for objects to take part in many different associations. These associations must be added to objects without changing the definition or implementation of the classes of which they are instances. Therefore, the classes should have no knowledge of the specific associations in which any or all of its objects are involved.
- the ability to add new logical relationships without producing subclasses of the classes involved.

In addition, to comply with the original aim of the project, the design must use current language features not require new features or a new language.

A system designed and implemented by following the above criteria would consist of classes which are easily recognisable as definitions of the objects identified in the analysis model. The classes would not be modified to provide associations with other classes of objects. New subclasses of objects would be introduced only when extra attributes or structure need to be added to existing classes. The implemented system would be simpler and therefore easier to understand, maintain and enhance. The classes would be readily available for use in other systems because application specific features would not have been added. Classes define only those properties which are intrinsic to all objects of the class. The objects themselves store the information which is specific to them.

The remainder of this chapter describes the Sociable class design technique which attempts to meet these design criteria.

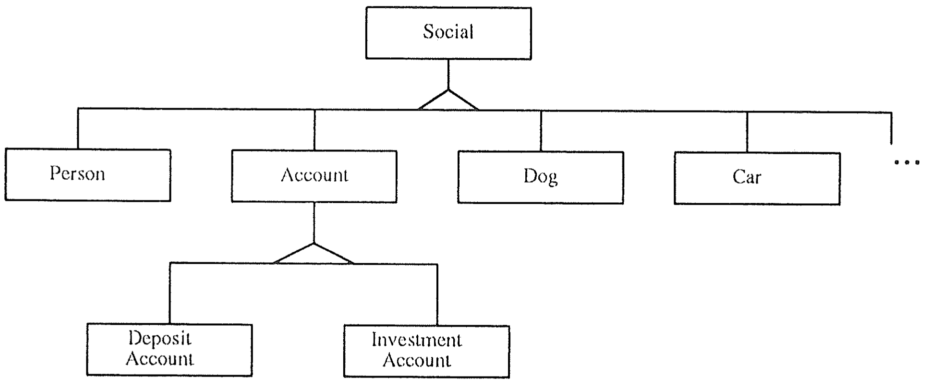


Figure 4.2: Sociable class hierarchy

4.3 The definition of Sociable classes and related constructs

The Sociable class design technique requires two types of classes to be developed. These classes are Sociable classes and Association classes. This section describes the current definition of each of these constructs. Other definitions of these constructs are possible. Some of these other possibilities are discussed in section 4.9.3

Sociable classes define objects which have the ability to take part in a potentially unlimited number of different associations. This ability is provided by declaring an abstract class, **Social**, which provides features to add, retrieve and delete associations from an object inheriting these features. These features are inherited by all Sociable classes which, therefore, define objects with the ability to take part in associations.

An example of a Sociable class hierarchy is shown in figure 4.2. The figure shows that all Sociable classes are derived from **Social** and that they can be used as the basis of further inheritance hierarchies. For example, classes **Deposit Account** and **Investment Account** are subclasses of the **Account** class. Sociable classes are identified in the analysis model by the fact that they take part in conceptual associations.

As stated above, the **Social** class provides objects with the ability to participate in associations. There are many different types of association such as one-to-one and one-to-many associations between objects. The different types of association are instances of subclasses of a second abstract class, **Assoc**. This class defines the ability to make and break associations between objects.

When an association is made, instances of associations become linked to the part of the object which was inherited from **Social**. Associations therefore become part of the objects involved in the association. They remain part of the object until the association is broken. When an association is broken, the object ceases to have any knowledge of that type of association. The lines labelled *association1* and *association2* in figure 4.1 are instances of **Assoc** subclasses.

Part of the association hierarchy is shown in Figure 4.3. Each of the subclasses defines a general type of association such as a one-to-one or a one-to-many association. The ability to make and break associations must be redefined specifically for each subclass. There must be one class of association for each general type. This ensures that the required type of association will always be available.

The general types of association, in the lower level of figure 4.3, implement the specific features provided by associations of that type. Each general association must provide the ability for a specific

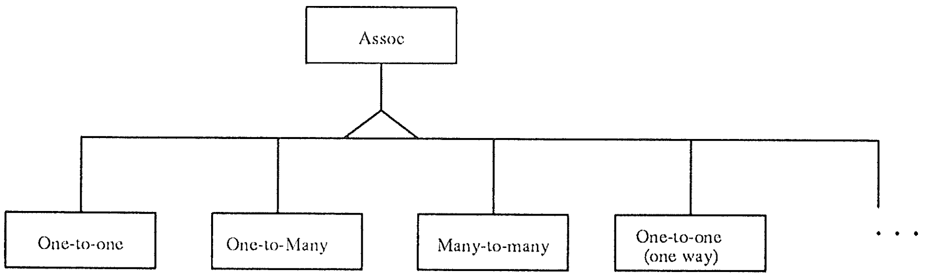


Figure 4.3: Association class hierarchy

association to:

- associate the required objects,
- know which objects it links,
- provide facilities for accessing the objects involved in the association,
- break an association between objects.

The general associations define the features required to form associations between objects from unspecified classes. In order to form specific associations between objects of specific classes, a new class of association must be derived from the general association. This is done by replacing the unspecified class names in the general implementation of the association by the specific class names required.

Objects of the classes involved in an association are linked by using the association's ability to associate the required objects. Associations between objects are produced by creating a new instance of the association. This new association is stored by the Sociable objects involved. Thus, an object encapsulates the knowledge of its own associations in compliance with object oriented principles. The association affects only those objects which are specifically linked. Other objects of the same class have no knowledge of that type of association.

The associations required in a system are identified by examining the analysis model. Each of the lines representing a conceptual association requires a specific association to be defined.

The general form of associations is shown in figure 4.4. This figure shows that one object of class **Social** can be involved in nought or many associations. Objects involved in a system would be expected to, but are not required to, participate in associations. The figure also shows that all associations must involve at least one object of class **Social**. This is because only Sociable classes have the ability to store associations and therefore to access them. Class **X** represents classes which are not derived from class **Social**. As the figure shows, it is possible for objects of these classes to participate in associations. The objects of class **X** cannot be used for accessing other objects with which they are associated because they do not have the required features. Associations involving objects defined by classes like Class **X** are one way links between objects.

This section has described the constructs required by the Sociable class technique to allow the implementation of associations between specific objects. To summarize, Sociable classes are used in conjunction with associations. The associations become part of the objects involved in the association. A specification for the above classes is given in section 4.4.

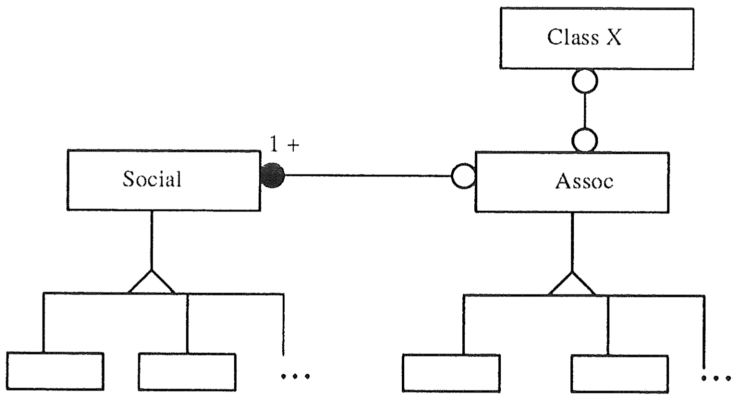


Figure 4.4: General associations

4.4 Class specification

This section contains a specification for classes required by the Sociable classes design technique. The classes specified are **Social**, **Person**, **Assoc** and **One_to_one2**. Classes **Social** and **Assoc** are the two abstract classes used by the Sociable classes design technique. **Person** is an example of a Sociable class. The **One_to_one2** class is a subclass of **Assoc** which is used to implement one to one associations which are traversable in two directions.

Implementations of this design have been produced in four object oriented programming languages. The different implementations are discussed in section 4.9.

4.4.1 Class Social

This class provides the ability to add, retrieve and delete associations from an object. In order to provide this ability, the class **Social** declares a collection of associations as a private attribute and provides features to access this attribute. The collection may be implemented by any appropriate data store such as a linked list or an array.

The specification shows that the features to access the private attribute are not made public. The only classes which can access the attribute are **Assoc** and its derivatives. The access is limited in this way to encapsulate knowledge of the implementation of associations. Classes which are derived from **Social** do not need to access any of the features.

Class Name:	**Social**
Description:	Abstract base class from which all Sociable classes are derived
Super classes:	None

Features:

```
        Private Attribute
                association : Collection of Assoc;
        Attribute exported to class Assoc
                association_found : BOOLEAN;
        Methods exported to class Assoc
                add_association(c:Assoc);
                access_association(c:Assoc);
                delete_association(c:Assoc);
```

Method descriptions

  add_association(c:Assoc);

    Adds an instance of class Assoc to the attribute, association;

    only one instance of each specific type of association must be allowed.

  access_association(c:Assoc);

    finds and returns the instance of the required type of association,

    set association_found to TRUE if found,

    VOID returned if NOT association_found.

  delete_association(c:Assoc);

    removes an instance of an association from the association attribute

4.4.2 A Sociable class

This section gives the specification of a Sociable class. The example chosen is a simple **Person** class. This class provides name, address and telephone number fields and defines the features required to access the attributes.

The specification shows that the only design addition required by the class **Person**, or any other Sociable class, is to declare **Social** as a superclass.

Class Name:	**Person**
Description:	Person with name address and telephone number
Super classes:	**Social**

Features:

  Public Attributes

    name :String

    address : String

    telephone_number: Integer

  Public methods

    add_name(String)—changes the name of the person

    add_address(String)—changes the address of the person

    add_telnumber(Integer)–changes the telephone number

  get_name:String—returns the name of the person

  get_address(String)—returns the address of the person

  get_telnumber(Integer)–returns the telephone number

4.4.3 Class Assoc

This base class is declared to allow all associations to be assigned to the same data structure in instances of the class **Social**. The *make_assoc* and *break_assoc* features are defined by class **Assoc** to allow the use of polymorphism when making and breaking associations. These two features represent the minimal functionality that must be provided by all subclasses. Both features require the objects which are to be associated or disassociated to be passed to the method. The objects are passed in a list because different types of association can involve different numbers of objects. The list can be redefined by derived classes to contain the desired number of objects. The variable number of objects involved in associations means that different numbers of features are required to access the objects participating in different forms of association. One access feature will be required for each object involved. These features cannot therefore be defined in the base class.

Class Name: Assoc
Description: Abstract base class from which all association classes are derived
Super classes: None
Features:

Method available to descendant classes and other instances of same class

```
make_assoc(objects : List of objects);
break_assoc(objects : List of objects);
```

Method descriptions

```
make_assoc(objects : List of objects);
        This feature must be redefined by subclasses.
        The objects listed have an association added to their
        collection of associations.
        The parameter is declared as a list of objects of any
        class to allow associations to involve any number of objects
        At least one of the objects must be derived from Social.
break_assoc(objects : List of objects);
        This feature must be redefined by subclasses.
        The objects listed have the association removed from their
        collection of associations.
```

4.4.4 Class One_to_one2

This class defines the general form of a two way association between two objects. A generic class is used because it provides the mechanism required to replace general classes with specific ones to form a new type of object. The general classes are listed in the interface of the class and are called formal generic parameters. The classes used to replace the formal generic parameters F and G must be Sociable classes.

In order to simplify the use of associations, the public methods *associate* and *disassociate* take two parameters, the two objects which are to be associated. The class **One_to_one2[F,G]** puts these objects into a list which is used by its inherited method, *make_assoc*, to associate the objects.

Class Name: **One_to_one2**
Class Interface: **One_to_one2[F,G]**
Description: Generic class to produce one to one associations between two objects derived from **Social**.
Forms a two way link.
Super classes: **Assoc**
Features:

Private Attributes

```
object1 : F;
object2 : G;
```

Public Methods

```
associate(object1 : F; object2 :G);
        makes an association between the two objects.
disassociate(object1 : F; object2 :G);
        breaks an association between the two objects.
find_object1(object2 :G) : F;
        returns an instance of the first generic parameter,
        VOID if no association exists.
```

```
                find_object2(object1 : F) : G;
                        returns an instance of the second generic parameter,
                        VOID if no association exists.
        Methods available to other instances of same class
                make_assoc(objects : List of objects);
                        redefined.
                break_assoc(objects : List of objects);
                        redefined.
Method descriptions
        associate(object1 : F; object2 :G);
                assign object1 and object2 to a list (l) — object 1 as first element
                object2 as second element,
                create a new instance of the association,
                call the make_assoc feature on the new association passing
                the list (l) as the parameter.
        disassociate(object1 : F; object2 :G);
                assign object1 and object2 to a list (l) — object 1 as first element
                object2 as second element,
                call the break_assoc feature passing the list (l) as the parameter.
        find_object1(object2 :G) : F;
                use the access_Association feature from class Social
                to obtain the required association if it exists.
                If the association exists, return the value of object1
                else return VOID.
        find_object2(object1 : F) : G;
                use the access_Association feature from class Social
                to obtain the required association if it exists.
                If the association exists, return the value of object2
                else return VOID.
        make_assoc(objects : List of objects);
                requires two objects in the list (list l from feature associate)
                — both objects must be instances of a Sociable class,
                first element must be of class F,
                second element must be of class G,
                first element assigned to attribute object1,
                second object assigned to object 2,
                current instance of association added to the list of associations in each object.
        break_assoc(objects : List of objects);
                requires two objects in the list (list l from feature disassociate)
                — both objects must be instances of a Sociable class,
                find the association between the two objects,
                remove this from each object's collection of associations,
```

4.4.5 Other classes of association

In order to make the technique useful in all applications, many other classes of association need to be provided. Figure 4.3 gives an idea of some of the general types of association needed. A class is needed for each different type of association. The exact number and nature of the other types of

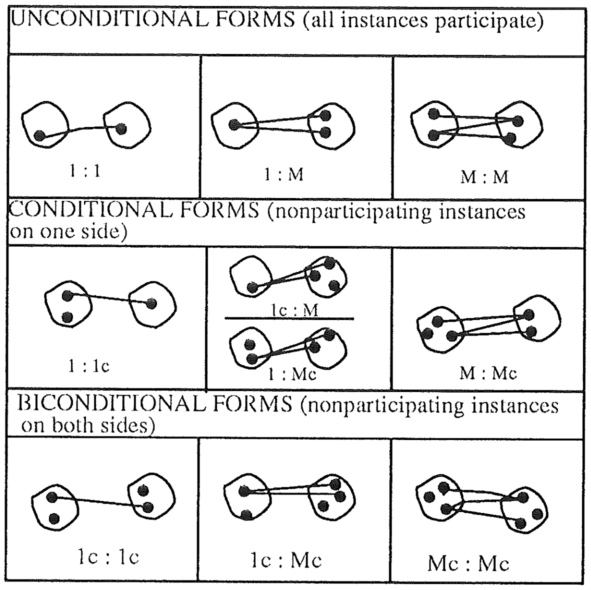

UNCONDITIONAL FORMS (all instances participate)
1 : 1
1 : M
M : M
CONDITIONAL FORMS (nonparticipating instances on one side)
1 : 1c
1c : M
1 : Mc
M : Mc
BICONDITIONAL FORMS (nonparticipating instances on both sides)
1c : 1c
1c : Mc
Mc : Mc

Figure 4.5: Ten forms of relationship from Shlaer and Mellor

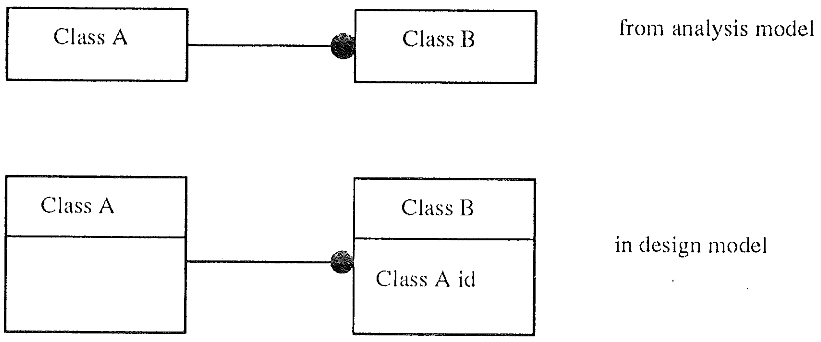
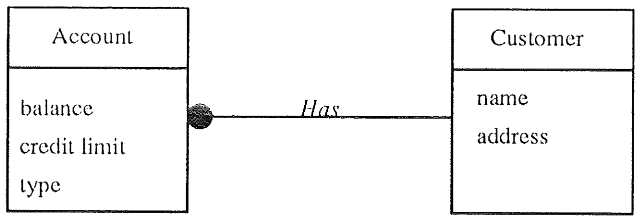


Figure 4.6: A formalized association from Shlaer and Mellor

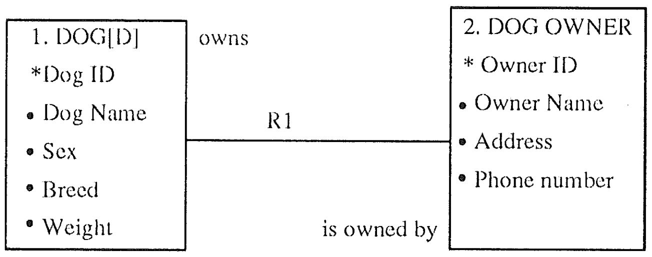
association requires further investigation. This section gives some initial ideas concerning the variety of associations which may be defined.

An idea of the number of different classes required can be gained by identifying all the different types of associations between objects. Shlaer and Mellor [59] have identified ten different forms of associations between two objects. These are shown in figure 4.5. These ten forms are formalized into relationships which can only be identified from one of the participating objects not both.

It can be seen from figure 4.6 that objects of class A have no knowledge of their relationship with objects of class B. It is possible that some applications may require an association to be traversed in either one or both directions. Allowing for this possibility increases the number of possible types of association between two objects to twenty. The number of different types of associations is so large because they include both conditional and unconditional relationships. These conditions are application specific. Using the Sociable Class design technique, these application dependent conditions could be implemented by the programmer when the system is written. They need not be included in the declaration of the association or the classes involved in the association. This leaves six distinct relationships



i) part of banking application (ATM) object model from OMT method



ii) Graphical representation of *Dog owner owns dog* and *Dog is owned by dog owner* from Shlaer and Mellor.

Figure 4.7: Sample object models

between two objects. These are one-to-one, one-to-many and many-to -many, all traversable in either one or two directions.

However, more types of association can be identified. Some associations might be mandatory or involve more than two objects. Each of these associations would need a general type to describe them. Some of the associations identified by the OMT technique have link attributes associated with them. It might be possible to provide subclasses which contain one or more attributes. These possibilities suggest that a large number of classes of association need to be provided. The general types would be provided in a library and so be readily available to programmers. Only a subset of the possible association types would be needed by most applications.

4.5 Using Sociable classes

It was shown in chapter 3 that some object oriented development methods produce models which include details of associations between objects. For example, figure 4.7 shows examples from the descriptions of OMT [28] and Shlaer and Mellor object oriented analysis [59]. This diagram is also shown in chapter 3 but is repeated here for convenience.

These relationships can be implemented unambiguously by using Sociable classes and the standard types of associations described previously. The use of Sociable classes is demonstrated in two stages. Section 4.5.1 explains how to develop a new system and section 4.5.2 shows how to extend the system. The simple banking application described in chapter 3 is used to illustrate the process.

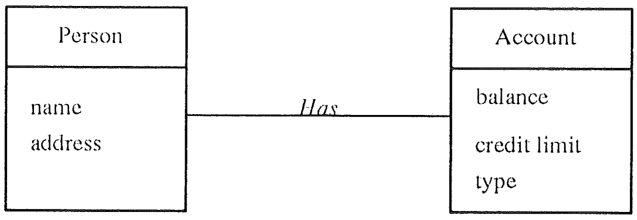


Figure 4.8: Simple banking application object model

4.5.1 Developing a new system

This section details the use of Sociable classes in the development of a simple banking system. The analysis model is shown in chapter 3 but is repeated here, see figure 4.8 for ease of reference. The implementation of the system should proceed as follows:

- implement the **Person** and the **Account** classes.

  Examination of the object model indicates that both these classes define objects which participate in associations so they are both declared as subclasses of class **Social**. The other features of the classes are implemented directly as defined by the analysis models without adding attributes to provide associations with other objects.

  Variables of these classes are declared in the application program in the usual way.

- implement the association.

  The general type of association required must be chosen. In this example the model defines a one to one relationship. This can be either a one way or a two way association. The desired representation is chosen, in this example a two way link is required. The general class for this type of association is identified from the class library. The new class of association is defined by substituting the general classes with the **Person** and **Account**. One variable of this new association is declared in the application program. For example, using Eiffel syntax

  `has_account : One_to_one2 (Person, Account);.`

  This variable is then used to create, access and delete associations between the required objects.

The application class is declared as follows, using Eiffel syntax:

```
class BANK

feature

   a,b : PERSON;
   x,y : ACCOUNT;
   has_account : ONE_TO_ONE2[PERSON,ACCOUNT];

Create is
do

--create and assign values to person and account variables
  a.Create;
  ...
  x.Create;
  ...
```

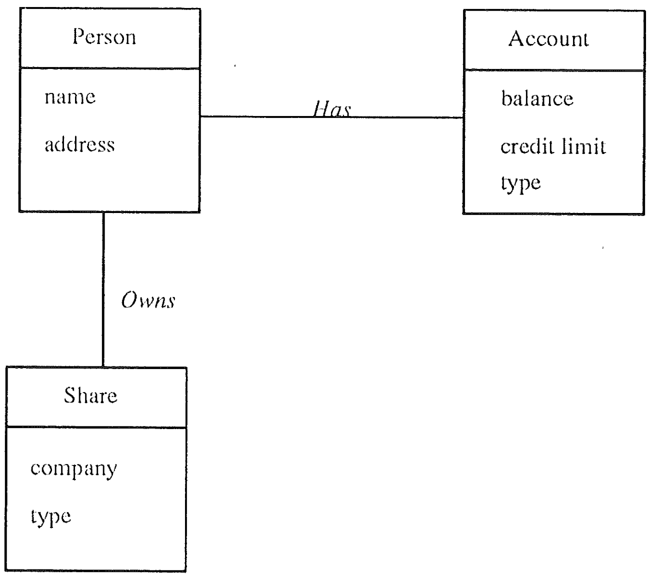


Figure 4.9: Object model of extended system

```
--create the association variable
  has_account.Create;


-- associate the required objects
  has_account.associate(a,x);

-- find the account belonging to person a
  y := has_account.find_object2(a);

--the account can the be accessed via object y

-- find the person owning account x
  b := has_account.find_object1(x);

-- the owner of account x can then be accessed via object b

-- rest of code

end;--Create
end --BANK
```

4.5.2 Extending a system

The same basic case study is used to describe the method of extending a system. The banking system might need to be extended to allow the bank to provide share dealing facilities as shown in figure 4.9.

The following steps are needed:

1. Implement the new **Share** class.

   This class defines objects which can participate in associations so the **Share** class is declared as a Sociable class, that is as a descendant of class **Social**,

   Instances of the new class are declared in the application program.

2. Implement the new association.

   Again, the object model is examined. It indicates that a one-to-one two-way association is required. This is produced by substituting the general classes with the **Person** and **Share** classes. One instance of this is declared in the application program. For example, using Eiffel syntax `has_share : One_to_one2 (Person, Share);`.

3. Add the new lines of code to create and access the instances of the new association.

The application program is declared as follows, using Eiffel syntax:

```
class BANK

feature

   a,b : PERSON;
   x,y : ACCOUNT;
   m,n : SHARE;
   has_account : ONE_TO_ONE2[PERSON,ACCOUNT];
   has_share : ONE_TO_ONE2[PERSON,SHARE];

Create is
do

--
END
```

The variables are created and accessed in the same way as in the original system.

4.6 Features of the Sociable class method

A system designed using the Sociable class technique consists of classes of objects. The objects can be linked by associations as well as by client-server relationships. The associations become part of the object involved in the association. These features have several benefits when considering the design and implementation of conceptual associations.

- The structure of the implemented system corresponds with the structure of the analysis model because

  1. the implemented classes are defined as they are identified during analysis. They are therefore recognisable as implementations of the 'real world' entities they represent.
  2. the conceptual associations between objects are implemented as distinct constructs.
  3. associations are formed between objects not between classes of objects. Only objects which are involved in a particular type of association have any knowledge of that type of association.

- The objects encapsulate all the knowledge about themselves because the associations become part of the objects.

- The associations belonging to an object are accessed via an instance of an association to maintain simplicity.

- Classes are only coupled together if they are related by an is-a or an is-part-of relationship.

- The code to implement associations is encapsulated and provided by reusable library classes.
- It is not necessary to declare a new subclass to allow objects of the class to participate in a new association. Therefore, extending a system will not result in long hierarchies so the complexity of the implemented system increases in proportion to the complexity of the required system.

These features suggest that a system implemented using the Sociable class design technique would be simpler and easier to understand than a system developed using the current techniques for implementing associations. The greater simplicity of the system should facilitate maintenance and enhancement activities.

4.7 Desirable language features

The language features required to implement the above specifications are inheritance, polymorphism, generics, dynamic type checking and the ability to make features available to selected classes of objects. The specification of class **Assoc** requires all classes to be derived from a common base class.

Inheritance is used to allow the classes to be derived from the base classes. Multiple inheritance would allow the Sociable characteristics to be added to pre-existing classes.

Polymorphism, used here to mean the ability of an object to have more than one type, is required to allow different types of association to be stored in the same data structure. Polymorphism is also required to allow different versions of the *make-assoc* feature to be defined.

Genericity is required to permit general classes to be declared from which specific classes of association can be defined. This is not a required feature of object oriented languages but is desirable for this design. Other language features can be used to mimic this behaviour as demonstrated in section 3.4.5. In some of the generic types, it is necessary that the classes named in the declaration can only be replaced by a restricted range of classes. For example, in the specification of class **One_to_one2** the actual parameters used to replace F and G must be derived from class **Social**. This is known as constrained genericity. If such a restriction is not available within the language, checks must be made to ensure that the requirement is met by all instantiations.

Dynamic type checking is required to permit the correct type of association to be retrieved from the collection. Dynamic type checking is not provided by all languages but can be implemented by adding a field to an object's class to contain the type identifier. The value of this type field can then be checked by the code to ensure that an instance of the correct type is retrieved from the data store. The correct instance must then be assigned to a variable of its dynamic type. This involves assigning variables in the opposite direction to that allowed by normal assignment rules.

It is desirable to make the features defined in class **Social** available only to objects of class **Assoc** to ensure that the correct access is made. Again, this feature is not provided by all object oriented languages so the mode of access would need to be clearly documented.

The ideal languages for implementing the Sociable class design method would provide inheritance, polymorphism, generics, dynamic type checking, the ability to make features available to selected classes of objects and a base class for all objects. The feasibility study assesses whether the specifications can be met by the chosen languages and also identifies alternative mechanisms where the language features are less than ideal.

4.8 Feasibility study

The feasibility study involved the implementation of the classes required by the Sociable class design method. Four programming languages were used. The languages are Eiffel v2.3.4, C++ (compiled using the gnu compiler, g++), Modula-3 v 2.11, and Oberon-2. All the languages support object oriented programming and therefore provide inheritance and polymorphism. There are significant differences in the details of the implementations of these features. These differences are highlighted in this section where necessary. They were explained more fully in section 3.4.

Eiffel was used for the original implementation of the design. The Oberon-2 and Modula-3 implementations are basically translations of the final Eiffel code. The translations were performed to investigate the applicability of the specification to different languages. The C++ version implements improved specifications. It is possible that better designs and implementations could be developed for the different languages. Some of the possibilities are discussed in section 4.9.

4.8.1 Eiffel

The Eiffel language provides inheritance, polymorphism, selective exports, generics with the ability to constrain generic types and a base class, **ANY**, from which all classes are derived. It also provides three techniques for dynamic type checking. These are:

1. the *conforms_to* function from class **ANY**,
2. the *reverse assignment attempt ?=*,
3. the *dynamic_type* function from the kernel class *INTERNAL*.

The provision of these features by the language should allow the implementation of Sociable classes directly as specified. Constrained genericity was used to ensure that all instantiations of class **One_to_one** involve only classes derived from **Social**. Selective export was used to make the features of class **Social** selectively available to class **Assoc** and its derivatives. It was discovered, however, that the dynamic type checking does not distinguish between different instantiations of generic classes. The evidence for this is explained fully in Appendix G.

The following comments describe the current Eiffel implementation of class **Assoc**, generic class **One_to_one2**, class **Social** and Sociable class **Person**. The classes are described in this order because the need to provide manual dynamic type checking class **Assoc** and its derivatives affects the implementation of class **Social**.

The code for the Eiffel implementation can be found in appendix A

- class **Assoc**

  The need to implement a manual form of dynamic type checking means that this class cannot implement the specifications in section 4.4 exactly. The code used to implement this class can be found in appendix A.1. Dynamic type checking is required to allow a specific type of association to be retrieved from a collection in order to provide access to a particular association. The type checking had to be implemented in the classes themselves. An extra field, *declaredtype : STRING*, was added to the base class **Assoc** to provide the required type checking. The *create* procedure for this class assigns the desired type name to the *declaredtype* field. In Eiffel, create features of base classes are not passed on to descendants so all subclasses must rename the base class create feature and call this create in their own create feature. All instances of a subclass have the same *declaredtype* because the client code creates one instance of each association. This instance is then used to create, access and delete all other instances as required. This implementation does not allow associations of one type to be created via an instance of the base class **Assoc** and so does not involve polymorphism and dynamic binding. The *make-assoc* and *break-assoc* features were included in the specification and implementation of class **Assoc** to allow polymorphic creation of associations between objects. This feature cannot be used in this implementation so the class **Assoc** could be simplified to provide a *declaredtype* field and access features only. A final point about this manual dynamic type checking is that the *declaredtype* feature is exported to class **Social** only.

The Eiffel implementation of class **Assoc** needs to export the make_assoc feature to itself, that is to class **Assoc**. This is because the object defines the scope of features. In order for an object to call a feature which is declared by the class on another object of the same class, the feature must be exported to the class itself.

- generic class **One_to_one2**

  This class is implemented as specified in section 4.4. As mentioned before, the create feature must be supplied with a parameter of type String which is used to define the dynamic type of a specific association.

  Again, some of the features must be exported to the class itself to allow access of one object by another object of the same class. The code for this class can be seen in appendix A.2.

- class **Social**

  The code used to implement class **Social** can be seen in appendix A.3. In this implementation of class **Social**, the collection is implemented as a LINKED LIST. In order to prevent duplicate entries, code should be added to check for duplicates in order to meet the specification. The list of associations is declared as an expanded field. This is to ensure that the list is created whenever an instance of the class is declared. Descendant classes do not need to create this data structure or to access it so do not require any knowledge of the inherited features.

  The code which implements the *accessAssociation* feature of this class has to check the value of the *dynamictype* field in order to find the required association. The *dynamictype* field contains a string variable. Strings in Eiffel are implemented as references to objects. The code to compare the fields must check the values of the fields not the value of the references. The function *equal*, provided by class **ANY**, is used for the comparison.

- class **Person**

  The class **Person** is an example of a Sociable class. The code for this class can be seen in appendix A.4. This class declaration states that class **Person** inherits from class **Social**. The other features of the class are derived from the analysis model.

The Eiffel language permits classes to be declared as *deferred* which allows features to be specified but not implemented. This type of class is commonly used for defining abstract base classes. In this system, neither of the abstract base classes can be declared as *deferred*. It is not permissible to declare instances of *deferred* classes because they contain features that are not implemented so *deferred* classes cannot have create features. The class **Assoc** declares a create feature which is used to assign the required value to the dynamic type field of the objects to ensure that all descendants provide this field. All the features of class **Social** are fully implemented. The class does not contain deferred features so cannot be a deferred class either.

The Eiffel implementation fulfills the functional requirements but has required an additional field to be added to the abstract base class **Assoc**. This field is required to distinguish between the different instantiations of the generic associations. The requirement for this feature means that this implementation cannot use the polymorphic features provided by class **Assoc**.

4.8.2 Modula-3

Modula-3 does not provide all the desirable language features. In particular, Modula-3 provides single inheritance only so the class **Social** must be at the base of the class hierarchy if it is to be used as described in the specification. The ability to define abstract classes is not supported. Generic types are provided but constrained generics is not supported. There is no selective export but each module can supply several interfaces. Modula-3 does support dynamic type checking.

Each class involved in the system is implemented by one module pair. The objects are implemented as opaque object types. This means that all objects declared by a client module must have their type declared as the type made publicly available in the interface module. In all classes the public type is declared as T. Objects are therefore declared as `x : Modulename.T`. The code used for the current implementation can be seen in appendix B. The implementation of each class is discussed in turn.

- class **Assoc**

  The modules used to implement this class can be seen in appendix B.1. This class defines one method *makeAssoc*. This feature is defined to specify the minimum functionality of all classes. Of course, a breakAssoc feature is required by the specification but has not yet been implemented. The *makeAssoc* feature requires an `REF ARRAY OF ROOT` as a parameter. ROOT is the base class of all objects in Modula-3.

- generic class **One-to-one2**

  The modules used to implement this class can be seen in appendix B.2. Modula-3 provides generic types. The generic class declaration acts as a template for the instantiated types. A generic class is not compiled until actual generic parameters are provided. This means that different code is produced for each instantiation. It is not possible to limit the interfaces which can be used to instantiate a generic interface so it is not possible to ensure that only derivatives of class **Social** are supplied as actual interfaces. However, the procedures supplied by the generic class use features from class **Social** so attempts to compile new instantiations with classes without these features fail. The inability to constrain the interfaces does not seem to be a problem.

  In the implementation of class **One_to_one2**, the procedure *make_one_to_one2* is assigned to the *make_assoc* method inherited from class **Assoc**. This inherited method requires a list of type REF ARRAY [1..5] OF ROOT as one of its parameters. The compiler does not allow changes in the type of parameters so this parameter must remain as an ARRAY of ROOT in all subclasses of **Assoc**. In class **One_to_one2**, the actual type of parameter required is Social.T which is a subtype of ROOT. This supertype parameter does not pose a significant problem however because, as mentioned earlier, the implementation of the procedure requires features which are supplied by Social.T. The class will not compile if classes other than descendants of Social.T are used. Unless, of course, that class supplies features with identical signatures to those of Social.T. The requirement for class **One_to_one2** to be instantiated with classes derived from Social must be clearly documented.

  The above problem could be avoided by redeclaring the *make-assoc* feature in each subclass. The redeclared feature would specify the required type of parameter. This was not done because redeclaration in Modula-3 does not provide polymorphic features.

- class **Social**

  The modules used to implement this class can be seen in appendix B.3.

  Dynamic type checking is available for all traced types. Traced type is the default type. (Untraced types are usually only needed for systems programming so are not considered here.) When objects are created, they are automatically tagged with a type tag. This tag is permanently assigned to an object. The type tag can be accessed by any of the type checking features. These are `ISTYPE`, `NARROW`, `TYPECASE` and `TYPECODE`. In the implementation of class **Social**, the `TYPECODE` feature is used to ensure that the correct type is returned from the procedure *access_Assoc*.

  The collection of associations declared in **Social** is an ARRAY. This is not a reference type so does not need to be dynamically created. The descendant classes only need to import the interface to **Social** and list the type Social.T as an ancestor. The collection was implemented as an ARRAY to provide a simple means of representation. Again, checks must be made to prevent duplicate associations being added.

  The deleteAssociation method has not yet been implemented.

- class **Person**

  The modules used to implement this class can be seen in appendix B.4. Both modules implementing this class must import the module Social. The interface module declares class **Person** to be a descendant of class **Social**. The other features are implemented directly as defined by the analysis model.

The implementation fulfills all the functionality of the design. It falls short of the specification in that the requirement for features of class **Social** to be available only to instances of class **Assoc** is not met. This must therefore be documented in the class description. The inability to restrict the classes supplied as actual parameters to the generic modules also reduces the safety of the implementation in this language compared to Eiffel. It would be possible for programmers to use the generic classes without using class Social but it would require a lot of extra work.

4.8.3 Oberon-2

Oberon-2 does not provide all the desirable language features. The features supplied by Oberon-2 are inheritance, polymorphism and dynamic type checking. However, the dynamic type checking provided is only useful for distinguishing between known subtypes. Genericity, the ability to make features available to selected classes only and a common base class for all classes are not supported. In common with Modula-3, Oberon-2 provides single inheritance so class **Social** must be the base class for the hierarchy. Despite these deficiencies, a reliable implementation can be produced.

The code used for the current implementation can be seen in appendix C. The implementation of each class is discussed in turn.

- class **Assoc**

  A type field was added to class **Assoc** to provide dynamic type checking of unknown types of associations. This feature is exported as a read-only field. The value of this field is checked by the *access_Association* feature in class **Social** to ensure that the correct type of association is accessed. The only other feature of this class is a procedure to assign a value to the type field. The implementation of this class can be seen in appendix C.1.

There are two ways to overcome the problem of not having a common base class. The base class is required by the *make_assoc* and *break_assoc* features in class **Assoc**. These features require a list of objects as a parameter to allow subclasses to redefine the list to fit their own function. These features also permit features of the class **Assoc** to be called polymorphically. The easiest solution is to remove the *make_assoc* and *break_assoc* features from class **Assoc**. These methods are included in the specification of class **Assoc** to define the minimum behaviour required by associations and allow for the use of polymorphism when creating associations. As stated in section 4.8.1, the requirement for a type field means that polymorphism cannot be used so these features do not need to be included in class **Assoc**.

An alternative method is to declare a class **Object** from which class **Social** is derived. Class **Object** is implemented as an abstract class with no features, that is it is a pointer to an empty record. This class is then used as the type of the parameter required by the features in class **Assoc**.

The current implementation of class **Assoc** does not provide either *make_assoc* and *break_assoc* features.

- class **One_to_one2**

  The implementation of this class can be seen in appendix C.2.

  The specification of class **One_to_one2** requires the use of generics which are not available in Oberon-2. This problem was overcome by writing a template for the class. The template class must be edited by programmers when an instance of a new type of this association is required. In order to minimize the number of changes that need to be made, the template uses the ability to rename imported modules. In class **One_to_one2**, the module Social is imported twice. It is renamed as both A and B. The module is written in terms of the classes exported by modules A and B. The specification of class **One_to_one2** shows that an instance of each class involved in the association must be declared. All modules used to replace Social in the import list must declare a common type if the number of editing changes are to be limited. Hence, class **Social** and all Sociable classes define the type as T.

  When specific associations of type **One_to_one2** are required they can be produced by copying the template and changing four words. These are

  - the module name at the beginning and end of the module,
  - the two occurrences of the word Social in the import list of the module.

  The new module can then be compiled producing the code required to implement the association. Classes which are not derived from Social are highly unlikely to have the correct features so cannot be used to replace Social in the import list.

  The type of reuse involved in using such templates is often known as white box reuse because the source code is changed. It is possible to introduce errors when editing code in this way. Such changes are therefore not as safe as the black box reuse involved in Eiffel generics where the source code is unchanged. The underlying operating system could be used to ensure that the only changes made were to the four words required which would improve the safety of the template class.

- class **Social**

  Class **Social** is implemented in a module called Social and declared as type T which is a pointer to a record called SocialDesc. All Sociable classes are implemented by a module with the same name as the class being implemented. The type declared is type T. Only one type called T can be exported by a module so in this implementation modules become equivalent to classes. The *AddAssociation* and *AccessAssociation* features are implemented as type-bound procedures. The current implementation of this class can be seen in appendix C.3.

- class **Person**

  This class is implemented in a module called Person. The class is declared as type T. This type is declared as a pointer to a record called PersonDesc which is derived from the record SocialDesc.

  The extra attributes are those defined by the analysis model. The current implementation of this class can be seen in appendix C.4.

4.8.4 C++

C++ does not provide all the language features listed, in section 4.7, as desirable for implementing the Sociable class design method. It provides multiple inheritance, generics via templates and polymorphism but does not provide dynamic type checking or a base class from which all user defined classes are derived. I am grateful to Bob Dickerson for writing the C++ implementation.

The current implementation can be seen in appendix D. The implementation of each class is discussed in turn.

- class **Assoc**

  The class **Assoc** has been simplified. In the specification of class **Assoc**, the *make_assoc* and *break_assoc* features are included to permit dynamic binding and polymorphism. The specification of these features requires the provision of a base class from which all objects are derived. These two class features are not defined in the C++ implementation. There are two reasons for this. The first is that C++ does not provide a mechanism for determining the dynamic type of objects. A tag field must be used. This means that the associations cannot be created by an instance of the base class so polymorphism cannot be used. The second reason is that C++ does not provide a base class for all user defined classes. The class **Assoc**, as defined in C++, provides a field to store the name of the association. This is a tag field used to determine the dynamic type of the association. The current implementation of this class can be seen in appendix D.1.

- class **One_to_one2**

  Some improvements have been made to the class **One_to_one2** with respect to its specification. The functionality and data as defined in the specification has been divided into two separate classes. Both classes are template classes. The code can be seen in appendix D.2. One class called **One_to_one2**, provides the functionality as defined by the methods in the specification and provides the name of the association in the field inherited from class **Assoc**. A second class, **One_to_one2node**, is used to provide the data about the association, that is it provides fields to store pointers to the objects involved and a field to store the name of the association. When the *associate* method of the instance of class **One_to_one2** is called, an instance of class **One_to_one2node** is created. This instance is added to the list of associations of the object to be associated. The change to the original specification of class **One_to_one2** removes data areas from the object of the **One_to_one2** class declared in the application program. These data areas in this object were unused. Data was assigned to these attributes in the instances created by the object declared in the application program only. The same improvements could probably be made to the implementations in the other languages.

- class **Social**

  Class **Social** declares a variable which is a linked list of Assocs and provides the required access features. The Sociable classes are derived from **Social** using public inheritance. This ensures that all subclasses of **Social** display the behaviour they inherit from **Social**.

  The code used to implement this class can be seen in appendix D.3

- class **Person**

  This class uses the public mode of inheritance to inherit the characteristics of class **Social**. The other features are declared as defined by the analysis model. The code used to implement this class can be seen in appendix D.4.

4.9 Discussion

This section discusses the main advantages and disadvantages of the Sociable class technique as implemented in the feasibility study. Alternative implementations are also discussed.

4.9.1 The technique

The Sociable class technique for implementing conceptual associations allows these relationships to be traced from the analysis model to the implementation. It also permits associations to be added to objects without changing the definition of the class from which they are produced. The classes can therefore define the intrinsic data and behaviour required by objects of that class without the need to consider the actual application in which the objects are to be used. Sociable Classes provide the ability to take part in associations as part of their intrinsic behaviour. It should then be possible to assemble systems by declaring associations between classes and writing the application specific code. The classes and generic associations can be thoroughly tested prior to system assembly. The only part of the system which should require testing is the actual linking provided by the associations and the code used to provide the application dependent functionality.

In the systems developed for the feasibility study, the Sociable classes are newly developed classes derived from class **Social**. It may be possible to add this ability to existing classes and gain benefits when an existing system is extended. The use of multiple inheritance is the simplest method of converting an ordinary class into a Sociable class. For example a **Sociable-Person** class could be derived from classes **Social** and **Person**. This would be possible in Eiffel and C++. The other two languages used in the feasibility study, Oberon-2 and Modula-3, provide single inheritance. It might be possible to add the functionality of class **Social** to an existing class **Person**. This might be done by deriving the **Sociable-Person** class from **Person** and adding an attribute of type Social. This is a possible area for further research.

4.9.2 Current implementations

This section discusses the current implementations of the Sociable class design. Similarities and differences between the implementations are noted. The limitations of the current implementations and possible ways to overcome them are discussed.

The implementations in the different languages have different characteristics. The Eiffel language provides most of the required features. The only feature lacking is the in-built ability to distinguish between different instantiations of generic types. The Eiffel version implements the specifications most closely and provides the most reliable implementation.

Modula-3 and Oberon-2 both require the use of templates to provide the generic classes. Modula-3 provides generic templates as a built-in concept. However, the code is not compiled until the type is instantiated with actual classes. This appears to mean that the source code must be supplied. The Oberon-2 templates also require that the source code is supplied. The result is that programmers can change the source code and therefore potentially reduce the reliability of the system.

The use of templates ensures that different code is produced for each instantiation. Modula-3 provides the required dynamic type checking. It is possible to check that two objects have the same type. This means that it would be possible to use the *make_assoc* and *break_assoc* features of class **Assoc** in a polymorphic procedure as originally intended. This might allow a different, more efficient implementation of the design to be produced when using Modula-3.

The C++, Modula-3 and Oberon-2 implementations produce different code for each instantiation of class **One_to_one2**. This means that it is possible to have two different types of association between the same classes. For example, a 'parent of' and a 'married to' association might be required in the same system. In languages, such as Eiffel, where the same code is used, a type identifier would be needed to distinguish between the two types of association. This suggests that a type identifier is a necessary feature in Eiffel association classes.

The class **One_to_one2** has been specified to make, access and break two-way associations between objects. Two-way associations can lead to problems of data consistency when changes are made to the objects which are associated. For example, if a person changes their bank account both objects must reflect the change. It may be possible for the **One_to_one2** class and the other classes representing two-way associations to provide this functionality and further reduce the risk of errors in the application code. Other functionality might also be provided. For instance, many applications are developed as distributed systems. This results in associations being formed between objects located on different processors. Accesses to these objects require more complex code than accesses to objects on the same site. It may be possible to include at least some of the complexity in the generic classes defining the types of association.

The ability to take part in associations is provided by deriving Sociable classes from a base class called **Social**. This class declares a data structure in which to store the instances of associations. This appears to be the only method of providing this ability. The data structure chosen can be any structure which stores elements. The structure used in the feasibility study is either an array or a linked list. The choice of structure was made on the basis of the availability of the structure in the language or its libraries. Clearly, the choice of data structure will affect the efficiency of the system. The performance characteristics should be examined. The most efficient representation would depend on factors such as the maximum number of associations entered into by any one object. It is anticipated that the number will be small because each object in a system is likely to be involved in only a few different types of associations.

In the implementations described earlier, the classes representing the different types of association, such as one-to-one, are library classes derived from class **Assoc** and encapsulate the information needed to implement associations. Ideally, these sub-classes of **Assoc** are generic or template classes. The required associations are formed by instantiating these generic classes. In languages such as Oberon-2 which does not provide generics, this implementation of the Sociable class technique requires class definitions for each type of association to be written as textual templates. These definitions must be edited to produce the required association. Such implementations which require the editing of source code are not as reliable as implementations using the generic facilities provided by a language.

In order to form an association involving objects of one or more Sociable Classes, an instance of the required association is declared and created. This instance is used to make, access and delete associations of that type from the required objects. The knowledge of associations is encapsulated within associations and can only be accessed via an association. Encapsulating the knowledge in this way has some disadvantages. Some of which are:

1. A layer of indirection is introduced during access to associations. This can be seen from the example in section 4.5. The *find_object2* method of the **has_account** object is invoked. This method requests the required information from the named object and returns the value into a variable of correct type. The speed of execution of the system is therefore decreased when compared with a system employing direct access.

2. It is not possible, at the moment, to request that an object returns a list of all the objects with which it is associated. Nor is it possible to request that an object can list the nature of the associations in which it is involved. If these are found to be necessary features, it may be possible to add such ability to class **Social** and hence to all Sociable classes.

3. The features *find_object1* and *find_object2* are used to access objects involved in an association. Object1 and object2 refer to the order in which the classes were declared in the instantiation of the association. The names of these features are not as intuitive as, for instance *get_person*. The use of the *find_object1* and *find_object2* features is simplified by giving the association a name which highlights the order. For instance, the name `has-account` implies a relationship going from a person to an account. The generic class is therefore instantiated with person as the first object and account as the second. It would be possible to use the inheritance and renaming facilities in Eiffel to change the names of the access features. The process is more difficult in the other languages. The use of renaming is a process which could be examined.

These disadvantages suggest that it might be better to provide access to the encapsulated knowledge in a different way. Alternative methods of implementation are discussed in the next section.

The feasibility study indicates that the Sociable class design method would provide a viable mechanism for implementing associations as visible constructs. This should improve both the reusability of the developed classes and the extensibility of the system. Improvements may also be possible in the reliability of systems if more functionality can be incorporated into the association classes.

4.9.3 Alternative implementations

The following section discusses alternative ways to implement the Sociable class design.

The difficulties associated with the indirect access of the objects involved and the naming of the features might be solved if the associations were accessed via the objects involved which could be thought a more natural approach. For example, it is possibly more natural to think of adding funds to a person account using code such as `person.account.addFunds(10)` rather than to ask an association to return the account and then add the funds.

Attempts were made to provide access to an association via the objects involved. This required Sociable Classes to possess an additional attribute to hold the association being accessed. This attribute was provided by declaring an attribute as `active_association : Assoc` in the base class **Social**. This attribute was declared as the base type to allow any association type to be assigned to it. When an association needed to be accessed, the object was requested to activate a specific association by retrieving the association from its list and assigning it to the `active_association` attribute. That is a message such as

```
    person.activate(has_account)
```

was sent to the object. The intention was to access the features of the active association via the object by using code such as,

```
    account := person.active_association.find_object2;
```

This was not possible because the static type checking systems only permit access to the features declared in the base class **Assoc**. The features of specific associations, such as the *find_object2* method of class **One_to_one2**, cannot be accessed.

Section 3.4.4 described mechanisms for accessing the features available in the dynamic type of an object but not provided by the static type. These features are designed for use with whole objects not on the fields of objects, so cannot be used to access features in the different types of association. It was not therefore possible to access an association by using code such as,

```
    account := person.active_association.find_object2;
```

There are two alternative ways of approaching this problem. The first is to redefine the type system to allow dynamic type checking on the types of the fields. This solution does not allow the method to be used with current languages so is not considered in this thesis but is a solution worthy of consideration as an extension to existing languages. (The dynamic type checking and reverse assignment in Modula-3 may allow this but the feature is specific to the language so has not been investigated.) The other solution is to find a mechanism to work round the restriction. The only way currently available which provides access to the features of the active association is to return the association into a variable of the correct type and then to ask the association to return the instance required. The code required, using Eiffel syntax, might be:

```
has_account_association ?= person.active_association;
if not has_account_association.void
then
      account := has_account_association.find_object2(person);
-- code to access account as required
else
-- error code
end;
```

This adds another step to the process. Accessing the required object now requires three stages:

- ask the known object to return the association into a variable of the correct type
- ask this association to return the required object
- access the object

The code required to access associations in this way is more complex than the code required when the knowledge was encapsulated in and accessed via the associations. This method also requires the object to hand control of its associations to a client module which breaks the principle of encapsulation.

The conclusion drawn from this is that although it is possibly more natural to access an association via the objects involved rather than via an instance of the association, present type checking systems do not allow this to be done easily and reliably. However, it may be possible to incorporate the idea of Sociable Classes into a language and then provide a type checking system which allows access to any association in which an object is involved. This alternative implementation would have the effect of making the object interfaces more complex so may not be an improvement.

The design and implementation of the different types of associations presented in this chapter require the use of templates or generics. Generics is not available in all languages and the alternative method of using source code templates is a potential source of errors. An alternative implementation has been investigated. This implementation requires Sociable classes to be declared as described earlier. The base class **Assoc** is again used. The difference is in the way each association is implemented. Each association required in a system is implemented as a direct descendant of **Assoc**. This class provides the features required to associate, access and disassociate the required objects. For example, in a **has-account** class, a *find-person* feature is supplied to access the owner of a specified account. All the code to implement an association has to be written by the application developer. It cannot be supplied as a library class.

The second alternative implementation encapsulates the code required for the implementation of each association. This implementation has all except one of the features of the previous implementation. The missing feature is the provision of library classes of the code required to implement the different basic types of conceptual association. This second alternative implementation has the advantage that the names of these features are more clearly identifiable than in the previous implementation. The original implementation of associations specified in section 4.4 is felt to be the better of these two possibilities because more of the code is supplied in reusable library classes. This should provide more reliable systems.

4.10 Summary

This chapter has presented a design technique which allows the implementation of conceptual associations as they appear in the analysis model. The information contained in the associations is therefore traceable through the development process. This avoids some of the problems encountered when extending systems reusing classes in new systems. The technique involves providing objects with the ability to participate in an unknown number of associations. This ability is provided by declaring the classes from which they are produced as subclasses of class **Social**. These subclasses are called Sociable classes. Associations between objects are produced by using instantiations of generic classes. The associations required in a system are produced by instantiating the correct generic association with the required classes. Associations can then be added to objects as required during system execution.

Sociable classes define the intrinsic properties of objects such as the structure and ability to participate in associations. The client-server relationship is used to implement the structural attributes and properties of classes of objects. Inheritance is used to provide the ability to participate in associations. Each object stores its own associations and so encapsulates all its own data and maintains an object oriented structure.

The generic associations encapsulate knowledge about that type of association. This encapsulates the information about associations. An association belonging to an object is accessed via an instance of the the association not via the object itself.

The main features of the Sociable class technique are:

- Implemented classes define the intrinsic structure and properties of objects.
- Associations are formed between objects not classes.
- Classes are less coupled.
- Objects encapsulate all the information about themselves. The structure is defined by their class. The associations in which an object participates are stored in a list which forms part of their structure.

- New associations can be implemented without adding new subclasses to define the object involved.

- The association classes encapsulate all the knowledge about associations. This knowledge is not spread between two classes or stored in one of the classes involved.

- Associations can be traced from the analysis model to the implementation.

- Systems can be assembled from pre-tested units reducing the amount of system testing required.

The feasibility study indicates that the design can be implemented in a variety of object oriented languages. The most reliable implementation is produced using Eiffel. The other implementations require more documentation and require developers to follow more guidelines when using the method. These implementations therefore provide more opportunities for mistakes.

The results of the feasibility study suggest that this design method represents a possible alternative mechanism for implementing conceptual associations between objects. During the development of the method and the feasibility study, several areas have been identified in which more research is required. The following paragraphs summarize possible research into aspects of the different classes.

- Class **Social**

  1. This class requires a data store to hold the instances of associations in which the object is involved. This data store could be represented by a variety of structures. The most efficient structure depends on many factors including the number of elements to be stored. These factors should be identified and investigated in order to select the most efficient representation.
  2. The process of retrieving associations from the data store involves dynamic type checking. The effect of this on the efficiency of the completed system should be investigated.
  3. Class **Social** as defined in section 4.4.1 provides the ability to add, access and delete associations. It may be advantageous to increase this functionality, for example, to allow an object to return a list of the associations or a list of the types of association in which it is involved.
  4. The possibility of adding the features of Sociable classes to existing classes could be investigated.

- Class **Assoc**

  This class is specified to allow the *make-assoc* and *break-assoc* features to be polymorphic. The use of these features complicates the implementation of the generic classes. This ability has not been used in the feasibility study. It might be possible to simplify this class by removing the polymorphic functions with the result that the implementation of the generic classes could be simplified. The C++ and Oberon-2 implementations of class **Assoc** do not provide *make-assoc* and *break-assoc* features.

- Types of association

  Section 4.4.5 identified the existence of many basic types of association. The exact number and nature of these should be established to allow the required classes to be specified and implemented.

The specification of class **One_to_one** provides basic make, access and break functions. The possibility of increasing the functionality to ensure the integrity of associations and provide facilities to simplify the implementation of distributed systems could be investigated.

It might also be possible to develop classes of associations which also define attributes which are required to qualify the association.

The design method appears to have the potential to improve the reuse of components identified during analysis. This improvement applies both to extensions of existing systems and to new systems. In addition, Sociable classes and the different types of associations can be implemented using current object oriented languages. It is necessary to develop library classes but not to introduce new features into the languages.

Chapter 5

A comparison of the Sociable class technique with other techniques

This chapter compares the Sociable class technique with several other design techniques which can be used to represent conceptual associations. The purpose of the comparison is to evaluate the effect of each method on the reusability of the components developed. The techniques are selected from various branches of object oriented system development and include current programming techniques used in class based languages, database design methods as well as research methods. The Sociable class technique was developed to improve the traceability of information from the analysis model to the implementation of the system. On the analysis model, conceptual associations are bidirectional structures. This information should be retained in the implementation so all the methods used in the comparison produce bidirectional associations. The methods selected for comparison are:

- The Sociable class design technique
- The addition of attributes to the base classes
- The addition of attributes to subclasses
- The use of a data structure to represent the association
- Object oriented database design methods [62]
- A combination of object oriented and logic paradigms [63]
- The combination of inheritance hierarchies [64]
- The DSM system [60, 65]
- The use of role types [52]
- Design patterns [66]

The Sociable class technique was described in chapter 4. The Eiffel implementation is used for comparison in this evaluation. The use of attributes in base classes or subclasses and the use of data structures to represent associations are current design methods and were described in chapter 3. The other techniques are explained in section 5.1. The comparison is carried out to assess the viability of different approaches and their effect on the reusability of classes.

Traceability was identified in chapter 2 as an important feature in improving the reusability of software components. Improving traceability of conceptual associations was the major consideration when developing the Sociable class technique. Traceability is therefore the main criterion by which the techniques are compared. The method used for the comparison and the results are presented in section 5.2. It is important that an improvement in one factor which increases reusability does not adversely affect other factors and lead to a reduction in reusability. The techniques are therefore compared against other features which are important for improving reusability. The methods used and results obtained are presented in section 5.3

The results of all the comparisons are discussed in section 5.4. to identify the effect of the design techniques on the reusability of the resulting components. Section 5.5 draws conclusions about which approaches present the most opportunities for the development and use of reusable components.

5.1 Alternative design techniques

This section describes the design techniques which are used in this comparison but have not been described earlier. These extra design techniques were selected from literature and discussions with colleagues. All of the methods appear to offer possibilities for improving the implementation of associations. The methods selected for use in this comparison are:

- Object oriented database design methods [62]
- A combination of object oriented and logic paradigms [63]
- The combination of inheritance hierarchies [64]
- The DSM system [60, 65]
- The use of role types [52]
- Design patterns [66]

This section explains the mechanisms used by each of the methods and their relevance to the representation of associations. The nature and number of classes and other constructs required by each method to implement the simple banking application and its extension are identified. The number of classes and other constructs identified includes only those specified by the programmer. It does not include any of the library classes needed to support the specified classes.

5.1.1 Object oriented databases

Object oriented database design techniques are included in this comparison because relationships between objects are very important in database applications. Some object oriented databases such as ONTOS [62] are based on object oriented data models which are similar to the OMT models [28] described in chapter 3. It was considered that the mechanisms used to represent relationships in such databases may be suitable for adaptation into programming language constructs.

The term relationship is used in object oriented databases to encompass the relationships identified as ‘consists of’, ‘contains’ and ‘conceptual associations’ in section 3.3. The ‘consists of’ relationship is provided by user defined aggregation relationships and is thus implemented by a distinct construct.

One mechanism is used to represent both the ‘contains’ relationship and ‘conceptual associations’. Binary forms are represented [62] by adding reference attributes to each type in the same way as in programming languages. Many object oriented databases such as ONTOS provide referential integrity to ensure that an update to an attribute in one of the objects is reflected in the other object. This avoids the problems encountered when using converse pointers in object oriented programming languages.

Ternary relationships are implemented by creating new object types or classes to represent the relationships. The classes participating in the relationship are then amended to contain a reference to a relationship object and so become bound to the class which defines the relationship. One benefit derived from using this approach rather than simply defining a data structure to represent the association is that the relationship becomes part of the objects involved and is accessed via any of the objects.

The approach used for implementing ternary associations in object oriented databases can be used in object oriented programming languages for the representation of binary associations. This is used for comparison in this chapter. This design technique is referred to as the object oriented data base method in this thesis.

The simple banking application implemented by following this method would require four classes **Person**, **Account**, **has-account** and a root or driver class. The system would consist of four classes. The extension of the system would require three additional classes for **Share**, **Shareholder** and **own-shares**. The extended system would consist of seven classes.

5.1.2 Combining object oriented and logic paradigms

This method was devised to provide a more effective means of implementing associations between objects. It uses concepts not readily available in object oriented programming but is included to ascertain whether the ideas could be adapted or translated for use in object oriented programming languages.

Gottfried Razek [63] has carried out work which aims to combine the object oriented paradigm with the logic paradigm. He states that there are two types of knowledge concerning objects which represent aspects of the real world. One type of knowledge concerns information about objects. The object oriented paradigm is considered to provide a model of this information. The other type of information concerns the relationships between objects. The relationships referred to in the cited paper correspond to the conceptual association defined in section 3.3. Such relationships are modelled by the logic paradigm. Razek proposes that an integrated model can be produced by combining the two paradigms.

Razek developed an experimental language called OOLog. It is an extension of Prolog which is a language based on logic. A simple Prolog system [67] stores a series of facts as a database. The Prolog system has search routines built in which allow it to respond to queries about the facts by searching the store and locating the required answer.

The OOLog language is a prototype based language not a class based language. In a prototype based language everything is an object. Objects are used as prototypes or templates from which new objects are created. For instance, if the system requires person variables, a prototype person object is first declared. The required state variables and access methods are declared for this prototype. The person variables are then created from the prototype object. Any object can redefine any of the methods inherited from its prototype. There are no classes involved in this type of language.

OOLog is clearly very different from the class based languages discussed in chapter 3 and used for the implementation of Sociable classes in chapter 4. Despite these differences the approach suggested by Razek is included in this comparison because it was felt that similar concepts might be applicable to class based languages. The method is described first in terms of the prototype object language and then converted into class based programming terms. The grammar and syntax of OOLog are not discussed here as it is the principles of the approach which are important rather than the details of the implementation.

The mechanism used to provide associations between objects in OOLog involves creating prototype relation objects. These relation objects store tuples (lists of objects involved in a specific relation) in state variables and provide query, add and delete methods. The tuples stored can contain any number of objects so this prototype can be used as the basis for relationships between any number of objects. Specific relationships are created from the prototype. The methods provided by the template can be overridden to include constraints such as ' a person may have no more than two accounts'.

The effect of creating a specific relationship is to define a data structure to store instances of relations between objects. The data structure is accessed by the objects involved. This access is achieved by adding methods to the prototypes of the objects which will be involved in the relationship. The following example describes how the `has-account` association defined in chapter 4 might be implemented in OOLog.

- Objects are defined which act as templates for person and account variables.
- The `has-account` relation is created using the relation object as a prototype.
- The query, add and delete methods are redefined as necessary, for example, to define the actual prototype objects involved in the relationship.
- New methods are added to the person and account template objects to give access to the has-account relationship.

This method therefore provides associations as separate data structures but accesses the data structures via the objects involved in the association. The methods which access the data structures are simple to write because they rely on the underlying logic paradigm to return the result of the call.

Translating the above scheme into a class based language the prototype object, relation, would be a class, **Relation**, which has a set of tuples as an attribute. Subclasses of class **Relation** would be defined to provide associations between the required objects, such as the one between **Person** and **Account**. These subclasses could be provided by generic classes in the same way as in the Sociable class method to simplify the provision of specific relationships. Subclasses of the generic classes could be defined to provide any constraints such as ' a person may have no more than two accounts'.

Classes **Person** and **Account** would then be extended to provide access to the **has-account** association. The code required to implement the methods would be more complex in the class based languages than in OOLog because the object oriented paradigm does not provide the required mechanisms.

The simple banking application implemented by following this method would require four classes **Person**, **Account**, **has-account** and a root or driver class. The system would consist of four classes. The extension of the system would require three additional classes for **Share**, **Shareholder** and **own-shares**. The extended system would consist of seven classes.

This translation of the OOLog mechanism is used for comparison in this chapter and is referred to as the OOLog method. This design method is similar to the method used in object oriented databases in that the client code used to access the relationships would be very similar. However, the underlying implementations are different. The OOLog design creates a data structure to store the instances of the relationship whereas the database design stores the relationships with the objects involved.

5.1.3 Combination of inheritance hierarchies

Ossher and Harrison [64] propose that additional behaviour could be added to objects by combining base class hierarchies with extension hierarchies. The base class hierarchy consists of the class declaration as originally defined by the programmer. The extension hierarchy consists of the code which is required to add behaviour to a class. The code for each hierarchy is stored separately in a library. Systems are produced by merging the base classes with the required extensions. It is stated that this method can also be used to add instance variables and to modify existing behaviour, for example, to correct an error in the code. The classes used to produce the objects can, therefore, be extended by adding new behaviour without adding new subclasses.

The ability to add state variables and behaviour by merging base classes with extensions might provide an alternative method for implementing associations between objects. As an example, consider the implementation of the 'person has account' association. The person and the account classes could be provided as base classes. These would be implemented exactly as defined by the analysis model. The extensions required to implement the association could be declared as extensions to both classes. The system would be assembled by combining the required base and extension hierarchies. The implemented system would have the following characteristics:

- The person and account classes would appear to the developer to have had the extra attributes added to them.
- The association would be accessed via either object.
- The final implementation would be the same as if the association was implemented by adding pointers to the base class.

A difficulty with merging base classes and extensions has been identified by Ossher. This difficulty relates to possible conflicts between the extension and the base class. Such conflicts are most likely to occur if the extensions access existing methods or attributes. It appears that conflicts should not be a problem when implementing the above association because the new behaviour will not affect any existing attributes or methods. Neither should conflicts occur when a system is extended by the addition of new associations.

The simple banking application implemented by following this method would require two base classes, **Person** and **Account**, two class extensions, **has-account** and **account-has-owner** and a root or driver class; that is the system would consist of five parts. The extension of the system would require one additional class, **Share**, and two additional extensions **own-shares** and **share-has-owner**. That is the extended system would consist of eight parts.

Hierarchy combination appears to provide a viable mechanism for implementing conceptual associations. However, this method requires compiler support so cannot be readily applied to existing languages and therefore cannot be assessed.

5.1.4 DSM

The DSM [60, 65] is a programming development system. It is an extension of C to provide object oriented features. The system also provides other features. The feature which is important in this thesis is that it supports inter-object relations. Relations are declared by using a keyword sequence, `DEFINE <name> RELATION`. The declaration of a relation between two classes of objects generates access features for the relation. These features are added automatically to each class involved in the association thus modifying the class interface without changing the class definition. The original class is unchanged and available for reuse. However, as far as the programmer is concerned, the class definition appears to have been modified. The objects in the system have different properties from those defined by their classes. These additional features seem to be restricted to the features required to access the relationship. New relationships can be added to a system without adding subclasses. The relations can be accessed, that is created, questioned, modified or deleted, via the objects concerned.

In the small banking system, the Person and Account classes would be declared as defined in the analysis model. A relation would be defined to form the association between the two classes. The declaration would take the form:

```
DEFINE has_account RELATION
person 1-1 account
```

The Person and Account classes would then automatically have features added to them to allow access to the relation. For example, the Person class would have a get_account feature added to it. This feature would return the account belonging to the person into another account variable. This variable would then be used to access the person's account. It appears that features of the account object cannot be accessed directly via the person object.

When a relationship is defined two hash tables are created. Each table represents one direction of the association, for example, one table implements *Person has Account* and another implements *Account owned by Person*. Hash tables are used to provide maximum efficiency for accesses to the tables of relations. The implementation adds data structures which are not present in the analysis model but these structures are hidden from the programmer. The system appears to the programmer, to involve objects and relationships which reflect the application.

From the above description it can be seen that the simple banking application implemented using the DSM system would require three classes, **Person**, **Account** and a root or driver class, and one relation, **has-account**. The system would consist of four parts. The extension of the system would require one additional class, **Share**, and one additional relation, **own-shares**. The extended system would consist of six parts.

The implementation of relations in DSM appears to create similar structures to those employed in the OOLog design in that both provide data stores of objects involved in the association and add access methods to the class interfaces. DSM was specifically designed to implement both the 'consists-of' relationship and conceptual associations. The concepts used cannot be added to other languages without changing the nature of the languages by adding features to the compiler or implementing a preprocessor to supply the necessary features.

There are some similarities between the DSM system and the Sociable class technique. The similarities are that systems developed by either method can be extended without the addition of subclasses and that relation between objects are visible in the implemented system. However, in DSM, relationships are used to implement both the 'consists of' relationship and conceptual associations. The relation construct was not designed to be used to distinguish between these two different types of relationship. However, this distinction could be achieved by representing the 'consists of' relationship by instance variables. Instance variables in DSM are only used to optimise relationships in which traversal is important in one direction only. In the following evaluation it is assumed that the DSM `RELATION` construct is used to implement conceptual associations and instance variables are used to represent the 'consists of' relationship. This distinction between the representation of the two types of relationship allows greater traceability of information.

There are other significant differences between the Sociable class technique and the DSM system.

- In the Sociable class technique, the objects themselves store their associations whereas, in DSM, the associations between objects are stored in tables. The objects in DSM do not, therefore, encapsulate all the information about themselves.

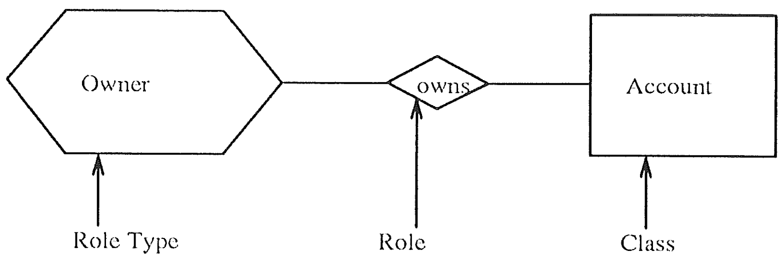


Figure 5.1: Model showing role types involved in the simple banking system

- In the Sociable class technique, only objects involved in a conceptual association know of the existence of that type of association. In DSM, the features required to access the relation construct are added automatically to the class definition. All objects of that class therefore know about the association.
- The Sociable class design technique can be applied to a variety of languages. The DSM system consists of a pre-processor and a subroutine library for the C language. It might be simpler to write the library classes required by the Sociable class technique than to develop a pre-processor for each language.

The DSM system provides the ability to implement relations as separate constructs but as described by its developers does not distinguish between the 'consists-of' relationship and conceptual associations. If the `RELATION` construct is used only for conceptual associations, the DSM system appears to meet some of the requirements listed in section 4.2. Specifically it provides a means to add new logical relationships without requiring new subclasses to be defined. However, the classes involved do have knowledge of the specific associations in which any or all of its objects are involved. The objects do not store their own associations. Also, a new feature has been added to an existing language. Similar additions might not be possible in other languages.

5.1.5 Role-types

Kilian [52] suggests that the ability to define role types should be added to object oriented programming languages. The role types would not be part of the normal type hierarchy but would specify the minimum behaviour which must be displayed by objects if they are to perform that role. Any class which has the required features as part of its interface would be usable where the role type is expected. Compatibility with the role type could be checked statically during compilation.

The use of role types during the development of a system can be demonstrated using the simple banking application. The relationship between a person and the account is first generalised to an owner relationship, that is , the 'person has account' relationship is generalised to 'owner has account'. The relationship between an owner and an account is shown in figure 5.1. The owner type is an example of a role type. This role type specifies that any owner of an account must provide a *Get-Name* feature which returns a string variable.

The **Account** class then defines the *Assign-owner* feature. This feature requires a parameter which is a variable of type **Owner**. The *Assign-owner* feature can then be used by any class which has a feature *Get-Name* returning a string variable. The Account class is available for use by unanticipated classes which provide the required interface. It is not bound to a specific class and its descendants.

As mentioned above, a class which represents potential owners, such as *Person*, must supply the required *Get-Name* feature. In order to provide a two-way association, the definition of the **Person** class uses the *Assign-owner* feature provided by **Account**. The **Person** class is therefore bound to the **Account** class and is not available for reuse without the **Account** class.

The simple banking application implemented by following this method would require three classes, **Person**, **Account** and a root or driver class, and one role type, **Owner**. The system would consist of four parts. The extension of the system would require two additional classes, **Share** and **Shareholder**(an extension of person). The extended system would consist of six parts.

The use of role types appears to remove the binding between classes in one direction only, so does not improve the reusability of all classes involved. The provision of role types in languages such as those discussed in chapter 3 would require changes to the type systems.

5.1.6 Design patterns

Design patterns [66] have been proposed by Gamma et al. as a mechanism for identifying reusable micro-architectures. They are used to identify common abstract structures found in object oriented designs. The use of design patterns would allow system designs to become standardised.

Design patterns are language independent. They describe the structure and interactions of a class or group of classes, the functions the class or group provides and how the pattern can be used in a system.

The catalogue of design patterns was examined for patterns which could be used to add attributes and features to a class. It was found that all the patterns involved redefinition of features, as defined in section 3.4.3. This redefinition allowed new attributes to be added within an existing feature. None of the patterns added features to the interface of the classes. Additional interface features are necessary in the current implementations of conceptual associations.

It was considered that the design patterns currently available could not be adapted to be used in the implementation of associations. However, it is possible to think of the Sociable class technique as a design pattern. For example, the technique is language independent. It relies on the common factor that object oriented systems contain classes of objects which can take part in associations. These classes are represented by Sociable classes. The interactions between the classes are the various types of associations which are provided by the generic library classes. The Sociable class technique is not a micro-architecture but a property of the whole of the system.

5.2 Traceability

Traceability of information was identified in chapter 3 as an important feature of systems which, if present, allows users of classes to understand their structure and behaviour. The understanding obtained promotes the possibility of reuse. This section compares the traceability of the information about conceptual associations provided by the Sociable class technique with the other implementation techniques. Traceability has been assessed by comparing the object model produced during analysis with an object model derived from the implementation. The correspondence between the analysis model and the implemented system is used as the measure of traceability.

5.2.1 Results

Use of the Sociable classes technique allows conceptual associations to be visible in the implementation. The implemented classes contain the same attributes and behaviour as those detailed in the analysis model. The object model derived from the implementation should be the same as the original analysis model. This technique provides traceability.

As stated previously in chapter 3, implementing associations by adding attributes to either base classes or subclasses does not provide traceability. An object model derived from an implementation would be very different from the original analysis model. The implementation of associations by defining a data structure of elements provides a separate construct for associations but the objects have no knowledge of the association. The association construct is therefore traceable but the objects have lost some of their traceability to the analysis model.

In object oriented database and OOLog design methods, associations are visible as separate constructs in the implementation. However, the class definitions are changed so object models derived from the implementations would be different from the original analysis model.

The implementation of conceptual associations by using a combination of inheritance hierarchies would give some traceability of the base classes in a system. The classes which form the base hierarchy would be traceable back to the classes in the object model. The code in the extension hierarchy could be traced back to the conceptual associations. The traceability of the associations is performed indirectly by examining the hierarchies. However, if the extension hierarchy is used for other purposes such as to modify existing behaviour, the traceability of associations via the extension hierarchy is lost. Traceability could be maintained by storing the code for conceptual associations in a separate extension hierarchy.

The DSM system was designed to allow visibility of relationships. Assuming that instance variables are used to implement the 'consists-of' relationship and the `RELATION` construct used to implement conceptual associations, the DSM system does provide traceability. The object model derived from the implementation should be the same as the original analysis model. However, the classes in the implemented system appear to have more features than the class in the analysis model.

The introduction of role types into a system would not allow conceptual associations to be visible in the implementation. None of the classes involved in a conceptual association would correspond to the analysis model. The passive class, in the banking system example this would be the **account** class, has an instance of the role type added. The implementations of the active class or classes, such as the **Person** class, involved in conceptual associations would contain extra attributes and methods. The object model derived from the implementation would be different from the original analysis model. Designing with role types does not appear to provide traceability of conceptual associations.

5.2.2 Conclusion

The results of the evaluation show that differing degrees of traceability are obtained by the different design techniques. Traceability of the information concerning conceptual associations is maintained by the Sociable class technique. Traceability can be maintained by hierarchy combination and DSM. The use of data stores gives some traceability but this is not as complete as with the other methods because the objects have no knowledge of the associations in which they are involved.

5.3 Other factors

This section compares the systems produced by the different techniques for implementing associations against other criteria which affect the reusability of components. The choice of the criteria is based on the factors which were identified in chapter 2 as affecting reusability. The criteria chosen for this evaluation are:

- Information hiding
- Understandability
- Standardisation
- Reliability
- Design complexity
- Product efficiency
- Extensibility
- Language support

The design methods do not directly address the problem of component compatibility. This factor is excluded from the comparison. It is assumed that the same method is used for developing and extending the system which avoids compatibility problems.

It should be remembered that the evaluation is based on the design techniques used to implement conceptual associations. Any differences between the analysis models and the implementation which are needed for other reasons are not considered. The criteria are evaluated in order to give an indication of the features of the design methods. It is beyond the scope of the thesis to investigate fully all the facets of the design methods. A simple design metric used in traditional engineering [27] has been used for this evaluation. In the future, more detailed comparisons could be made by using metrics such as the law of Demeter [29] to compare design efficiency. Product efficiency could be more accurately assessed by finding out the CPU time required for accessing associations. However, only methods which can currently be implemented could be investigated in this way.

Each of the selected criteria are compared in separate sections. Each section first defines how the criteria is evaluated and then presents the results and conclusions drawn from the comparison.

5.3.1 Information hiding

This feature is assessed by ascertaining the amount of information which must be understood by the programmer when using the design technique. Complete information hiding, in this context, implies that the method provides conceptual associations as encapsulated units. The conceptual associations are then implemented automatically by the system when declared by the programmer. The details of the implementation of associations are hidden from the programmer. Lack of information hiding in this context is defined to mean that the programmer is responsible for all the coding required to implement conceptual associations and no details of the implementation are hidden.

The method used to assess information hiding is to identify the translations that are required to transform analysis models into programming constructs and the amount of new code that must be written when implementing an association. It gives an indication of the difficulties which may be encountered by programmers using the design techniques.

Results

The Sociable class design technique allows programmers to define which associations are required. The features required to access the associations are provided by the generic association classes. The programmer does not need to write code to implement the associations.

Current programming language design techniques and database design techniques require programmers to decide on the mechanism to be used for implementing conceptual associations. The programmers must translate conceptual associations into one of several possible representations. The programmer is responsible for writing all the code required to access each association in the system.

Using the translation of the OOLog design gives automatic generation of the data structures to store the relationships but leaves the programmer to add the code required to access the data store and hence access the relationship.

The implementation of conceptual associations by combining base and extension hierarchies requires the programmer to write all the code to create and access the associations and therefore does not provide encapsulation.

The DSM system provides relations as semantic constructs and provides automatic access to relations via the objects involved. The mechanism used to implement associations is therefore transparent to users of the language.

A designer using role types must first define the role to be played in an association. This role type is then defined. Classes which can take on the role must supply the correct interface and also define the code needed to perform the role. The programmer is responsible for writing all the code concerned with the association.

Conclusion

Information hiding is provided only by the Sociable class technique and DSM.

5.3.2 Understandability

In this context understandability is used to mean the ease with which implementation classes can be recognised as implementations of classes identified during analysis. The names of the classes are assumed to be the same in both instances giving initial understandability. This evaluation examines the details of the classes such as the attributes and methods provided. This criterion is evaluated by establishing the correlation between the classes defined in the analysis model and the implemented classes.

Results

Designing a system using Sociable classes allows the classes in the analysis model to be implemented directly without any changes. The lines in the analysis model which represent associations between objects of the class are implemented by instantiations of the required type of association.

It was shown in section 3.6 that current design methods do not provide a good correlation between the analysis model and the implemented system. Following such methods has one of the following results.

1. The base classes have additional attributes and methods because the client-server relationship has been used to implement conceptual associations between objects.

2. Subclasses of the classes in the analysis model are defined to add the extra attributes needed to implement conceptual associations between objects. These subclasses cannot have the same name as the classes in the analysis model.

3. Relation classes are added to store the information. The implemented classes are identical to those in the analysis model. The objects of these classes have no knowledge of any associations.

The use of the database or OOLog technique results in extra relation classes and modified class definitions. The implemented classes do not correlate with those in the analysis model.

Hierarchy combination utilises two hierarchies of classes. The base classes would be direct implementations of the classes in the analysis models. The extension hierarchy would contain the additional code required to implement associations. The base classes would be readily understood.

The DSM system is designed to allow classes to be implemented directly from the analysis model. The code required to access associations is automatically generated and added to the class definitions when the `DEFINE <name> RELATION` construct is used.

The use of role type allows some of the classes to be implemented directly from the analysis models. Other classes however require the addition of attributes and methods so do not correlate to the analysis models.

Conclusion

The Sociable class design technique and the DSM system allow a direct correlation between the classes in the analysis model and the implemented classes. These two methods result in increased understandability of the classes when compared with the other methods.

5.3.3 Standardisation

The standardisation of the representation of conceptual associations is also taken into account. The number of ways which can be used to implement associations affects the understandability of the system. A standard method of implementing associations ensures that systems are simpler and easier to understand.

Results

The Sociable class design method provides standard representations for all types of conceptual associations as library classes.

Currently, three distinct methods of implementation are used. Variations also occur within the three methods. The implementation of conceptual associations is not standard at present.

The object oriented database, OOLog, hierarchy combination, role types and the DSM system all provide a standard representation for conceptual associations.

Conclusion

Standardisation could be improved by using the Sociable class design method, the object oriented database method, the OOLog method, hierarchy combination, role types or the DSM system. However, these standards are all different and are not compatible with each other.

5.3.4 Reliability

Two main aspects of reliability are considered. They are:

1. Type checking

   In section 3.4, it was stated that type checking is considered an essential requirement for the production of reusable code. This is, therefore, a major consideration when comparing the design choices. All the languages investigated during the research provide static type checking. The investigation into this criterion concentrates on establishing any requirements for dynamic type checking which affects the speed of execution of the system.

2. Testing requirements

   This criterion is assessed by comparing the nature and number of tests that need to be carried out when using the different design methods. The aspect of design being considered is concerned with the representation of conceptual associations. The classes which are to be associated are assumed to be available as pre-tested units.

Results

1. Type checking

   Consideration of type checking requirements suggests that the only design technique which requires dynamic type checking is the Sociable class technique. Dynamic type checking in that technique is required to retrieve the correct type of association from a heterogeneous data store. The requirement for dynamic type checking reduces the efficiency of the product, that is the implemented system.

2. Testing requirements

   The testing requirements for the code written to implement conceptual associations are closely related to the amount of information hiding in the design method used. The methods which give information hiding, that is Sociable class design technique and the DSM system, will require less testing. It is assumed that the underlying constructs have been fully tested. The only testing required when assembling such systems should be to ensure that each association is declared and accessed correctly.

   The other methods which do not provide information hiding will require more testing. For instance, implementing the association by declaring converse pointers in the classes involved requires the use of inheritance to add subclasses. The subclass must be tested to ensure that

   - the new code does not conflict with the base class code, for example, to ensure that the new feature names do not correspond with existing feature names.
   - the code implementing the association is correct.

Conclusion

Dynamic type checking is required by the Sociable class design technique and may reduce its efficiency.

The testing required when using the DSM system or the Sociable class technique should be less than when using the other methods.

5.3.5 Design complexity

Several different methods of assessing design efficiency or design complexity were identified in chapter 2. None of the methods is accepted as a definitive metric for assessing object oriented software. The basic principles underlying measurements of engineering design complexity rely on a combination of several factors. These include:

1. the number of parts,

2. the interfaces between the parts,

3. the number of types of parts,

These factors can be assessed for object oriented systems so they have been used in this comparison. Each of these factors is informally assessed by analysing the perceived implementation requirements of the simple banking application using each design method. The features of the designs which are concerned with the implementation of conceptual associations are considered in the assessment. The following design metric suggested by Pugh [27] is used to formalize the comparison. The following formula [27] is used to compare the complexity of the designs.

$$complexity\ factor = \sqrt[3]{Np * Nt * Ni}$$

where Np is the number of parts, Nt is the number of types of parts and Ni is the number of interfaces. It is suggested by the metrics used in engineering, that simple designs use fewer parts, fewer types of parts and have smaller interfaces. A lower figure for the complexity factor indicates a less complex design. This metric is intended to give an indication of design complexity not to give a definitive answer.

Results

Table 5.1 gives the results of assessing the different design methods with respect to the number of parts, size of interface and the types of parts.

The number of parts in the system includes the class required as the driver or root of the system. It can be seen that the number of parts varies from 3 to 5. The simple system used is too small to show significant differences. Larger systems would need to be developed to show significant difference in this factor.

In order to arrive at a figure for the number of interfaces, some simplifying assumptions were made.

- The classes are all the same size with the same original number of features in the interface. This number is assumed to be 10.
- When a class is extended by the addition of a pointer to a variable of another class its interface is assumed to be extended to provide access to all the features of that class. This doubles the size of its interface. The figure used is the maximum size of interface of any one part. The figure for the number of interfaces is therefore 10 or 20.

The size of the class interfaces differs between the various design methods. Most of the design methods result in an increased class interface. The only methods which do not increase the class interface and can be used with current object oriented languages are the Sociable class technique and the use of data stores to represent the association. The identification of the interface size of the classes implemented in the DSM system is problematical. The classes as implemented have the same sized interface as the classes in the analysis model but when they are included in the system the interface is increased. This increase occurs because the `DEFINE <name> RELATION` construct automatically adds the required access features to the classes involved. The results table, Table 5.1, shows the values for the interface size of the actual classes not as they appear in the system. Because of the automatic increase in class interface, the real figures for the design complexity of the system implemented in DSM are probably rather higher than the figure recorded.

Pure object oriented system designs consist only of classes. The number of types of parts used in such systems is therefore one. Systems designed using Sociable classes, attributes in base classes, attributes in subclasses, data store, OO database or OOLog translation are considered to be pure object oriented systems and to be comprised only of classes. Systems designed using DSM consist of classes and relations. Systems designed using role types consist of classes and role types. Systems designed

Method	No. of parts	max interface size	types of parts	complexity
Sociable classes	4	10	classes	3.42
attributes in base classes	3	20	classes	3.91
attributes in subclasses	5	20	classes	4.64
data store	4	10	classes	3.42
OO database	4	20	classes	4.31
OOLog translation	4	20	classes	4.31
hierarchy combination	5	10	classes + extensions	4.64
DSM	4	10	classes + relations	4.31
role types	4	20	classes + role types	5.43

Table 5.1: Design complexity

using hierarchy combination consist of classes and extensions. The last three design techniques all have two types of parts.

The figures for the design complexity are given in the last column of table 5.1. These figures give an approximate measure of complexity only. More and larger systems would need to be examined fully to establish the validity of these results. The comparison used above considers the interface provided by the class. No estimate is made for the number of classes required to support the classes in each design method.

Conclusion

The results of this comparison using the complexity factor as a metric suggests that the use of the Sociable class technique and the use of data structures to represent the associations result in the least complex designs.

5.3.6 Extensibility

One of the perceived benefits of using object oriented techniques is that systems are extensible. This criterion is included to ensure that mechanisms aimed at improving traceability and reusability do not impair extensibility.

Extensibility is assessed by defining the increase in complexity when a system is extended. The assessment is based on the extension of the simple banking system described in chapters 3 and 4.

Results

It was shown in section 4.5.2 that extending a system developed using the Sociable class technique requires the programmer to declare a new association and if necessary include a new class in the system. Subclasses of the classes involved in the association do not need to be declared. The complexity of the implemented system, as measured by the number of parts involved in the system, increases in proportion to the complexity of the required system.

It was mentioned in section 3.6 that the use of the client-server relationship to add attributes when implementing associations can result in deep inheritance hierarchies if a system is extended. These hierarchies can be difficult to understand and increase the complexity of the system. Similar increases in complexity would be produced by following the object oriented database and OOLog techniques because methods to access the new relation objects are added to the classes involved. This requires declaring a new subclass of the class.

Method	No. of parts	max interface size	types of parts	complexity
Sociable classes	6	10	classes	3.91
attributes in base classes	5	30	classes	5.31
attributes in subclasses	8	30	classes	6.21
data store	6	10	classes	3.91
OO database	7	30	classes	5.94
OOLog translation	7	30	classes	5.94
hierarchy combination	8	10	classes + extensions	5.43
DSM	6	10	classes + relations	4.93
role types	6	30	classes + role types	7.11

Table 5.2: Design complexity of extended system

The extension of a system by using data structures to store associations is achieved by the declaration of another structure to store the newly associated objects. It is not necessary to implement subclasses of the classes involved in the association.

Extending a system which implements conceptual associations by combining inheritance hierarchies appears to require the addition of one extension for each new association and a new base class to the base hierarchy if the association involves objects of classes not already in the system.

Systems implemented using the language DSM are readily extended without adding subclasses to implement the extra associations. A new relationship is defined. The features to access this relationship are automatically added to the classes involved. The classes as stored in the library therefore keep their original interface size. As mentioned in section 5.3.5, the size of the interface of a DSM class is a difficult measure to assess.

The addition of role types to languages allows for anticipated changes. This can be demonstrated by using the simple banking system as an example. The role type 'owner' could be used to generalize the association between person and account. This allows new classes of owners, such as companies, to be introduced into the system without changing any of the classes already involved. However, if the system is extended to provide the ability for a person to own shares, the person class must be extended to add the required features. This would require a new subclass and result in an increase in complexity. The role type 'owner' could of course be reused when implementing the Share class.

Table 5.2 shows the design complexity for the banking system extended to include the ability of customers to own shares. The figures obtained here use the same assumptions that were made when comparing the original complexity. Again the figures obtained are intended to give a general indication rather than be definitive values.

Table 5.3 compares the design complexity of the original and extended system. The final column in the results table shows the percentage increase in complexity after the system is extended.

Conclusion

Examination of the values in the Table 5.3 shows that the increase in complexity is least when using the Sociable class technique, a data store, hierarchy combination or the DSM system. The increase in complexity when extending a system using these methods is 17% or less. The other design methods result in at least a 30% increase in complexity of the implemented system.

Method	original system	extended system	% change
Sociable classes	3.42	3.91	14.3
attributes in base classes	3.91	5.31	35.8
attributes in subclasses	4.64	6.21	33.8
data store	3.42	3.91	14.3
OO database	4.31	5.94	37.8
OOLog translation	4.31	5.94	37.8
hierarchy combination	4.64	5.43	17.0
DSM	4.31	4.93	14.4
role types	5.43	7.11	30.9

Table 5.3: Comparative design complexity of original and extended system

5.3.7 Product efficiency

This criterion gives an indication of the processing efficiency of the different designs; that is the speed of execution of systems.

Product efficiency is evaluated by considering the number and nature of the feature calls required to add money to a known person's account. This takes into account the type of data structures to be accessed after the person object has been selected. Many of the design methods are theoretical and not possible with current languages so it is not possible to obtain an accurate comparison of all designs.

Results

The use of Sociable classes in the design of systems requires access of a data structure to locate the correct association before the account can be retrieved and the required feature accessed. Dynamic binding is also required. The data structure storing the association is expected to be small because objects will be involved in a limited number of associations. As was mentioned in chapter 4, further work needs to be carried out to establish the most efficient storage structure. The requirement for dynamic binding also reduces the efficiency of the system.

Systems implemented by adding attributes to the base classes or subclasses, result in feature calls such as `person.account.addFunds(amount)`. Such feature calls involve two levels of redirection but no searching of data structures. Such methods are currently used for implementation and are therefore considered sufficiently efficient.

Using either a data structure to store the relations or the OOLog method require a search of the data structure used to represent the association. The efficiency of these design methods depend on the data structure used and the number of entries stored. This search is in addition to the call to the required feature of the account object. These methods of implementation would be expected to lower the efficiency of the system. This reduction in efficiency is not necessarily significant.

The object oriented database method accesses values of the relations via the objects involved. The access time and hence product efficiency would be expected to be similar to that obtained with the other methods currently used.

Use of hierarchy combination might be expected to produce similar results to current methods because after the hierarchies are combined the systems should be very similar.

The DSM system uses tables of relations but the author of the language claims in [60] that hash tables provides an efficient mechanism for representing associations.

Role types are a compile time notion so would also be expected to give similar efficiency to current methods.

Conclusion

From this simple comparison, it appears that most of the design methods would produce efficient systems. A system designed using the Sociable class technique is likely to be the least efficient.

5.3.8 Language support

This criterion is included in order to assess the general applicability of the design technique to current languages. This is considered important because techniques which rely on new language features cannot readily be adopted by industrial users. This criterion is a simple measure of whether the method can or cannot be used in current object oriented languages.

Results

Many of the design methods described in section 5.1 can be employed using current language constructs. The Sociable class design technique has been shown to be usable with four current languages. The provision of generics in a language simplifies the implementation of associations and increases the reliability of the system but is not a prerequisite. This technique also requires the ability to assign a subclass variable contained in a superclass variable to a variable of its own class. Clearly, current design techniques used in language and database implementations require only current object oriented features. The OOLog method can be translated for use in current object oriented programming languages.

The use of combination of inheritance hierarchies and role types requires compiler support so cannot be used with current object oriented languages.

DSM has the features required to implement conceptual associations built in to the language. The concepts used require language support which is not available in other object oriented languages.

Conclusion

The Sociable class design method, current design methods, the OO database method and the OOlog translation can all be implemented using current object oriented languages. Hierarchy combination, DSM and role types all require extra support from the languages and so cannot be used with current object oriented languages.

5.4 Discussion

This section discusses the results in the wider context of reusability. The effect of the different design approaches on the reuse potential of the classes is assessed. Table 5.4 summarises the results of the comparison.

The form of the results recorded in the table requires explanation. Many of the results are given as yes or no. This indicates that the features are, or are not, considered to be shown by a design method. In all cases, a yes answer shows that the design method has a feature which is required for enabling reuse. In the 'language' column, yes signifies that the method can be used with the class based languages discussed in section 3.4. The results for testing are not included because the only indication of the testing requirement was found to be the provision of information hiding. The results for design complexity are copied from tables 5.1 - 5.3. A lower number for design complexity indicates a simpler

Method	traceability	info hiding	understandability	standardisation
Sociable classes	yes	yes	yes	yes
attributes in base classes	no	no	no	no
attributes in subclasses	no	no	no	no
data stores	yes	no	no	yes
OO database	no	no	no	yes
OOLog translation	no	no	no	yes
hierarchy combination	yes	no	no	yes
DSM	yes	yes	yes	yes
role types	no	no	no	yes

	complexity				
Method	original	extended	% increase	efficiency	language
Sociable classes	3.42	3.91	14.3	?	yes
attributes in base classes	3.91	5.31	35.8	yes	yes
attributes in subclasses	4.64	6.21	33.8	yes	yes
data stores	3.42	3.91	14.3	yes	yes
OO database	4.31	5.94	37.8	yes	yes
OOLog translation	4.31	5.94	37.8	?	yes
hierarchy combination	4.64	5.43	17.0	?	no
DSM	4.31	4.93	14.4	yes	no
role types	5.43	7.11	30.9	?	no

Table 5.4: Results summary

design. The results for the extension of a system are the percentage change in complexity from table 5.3. The results of the comparison of product efficiency gives some entries as ? because this feature has not been assessed. Design methods which are currently used are assumed to give an acceptable level of efficiency.

The discussion is based on the premise that reusability of components can be improved by increasing traceability, information hiding, understandability, standardisation, reliability and product efficiency whilst at the same time decreasing design complexity. It should be remembered that the DSM system is assumed to have been modified slightly. The `DEFINE <name> RELATION` is used only for the implementation of conceptual associations. A further assumption concerns the combination of inheritance hierarchies. The assumption is that the extensions used to implement conceptual associations are kept in a separate hierarchy.

The results of this evaluation indicate that the provision of conceptual associations as separate constructs demonstrates some of those characteristics. This can be seen by comparing the results of using the Sociable class design technique, the addition of data stores to represent the association, the combination of inheritance hierarchies or the DSM system which allow conceptual associations to be implemented separately from the classes involved in the association. The characteristics shown by each of these four methods are:

1. Improved traceability

   The implemented classes provide the attributes and behaviour defined in the analysis model. The classes can therefore be traced from the analysis through to the implementation and identified as representations of the same entities. The associations are visible in the design as distinct constructs and are not embedded in the classes. They are also traceable.

The OOLog technique and object oriented database design method also provide the associations as separate constructs but both methods require changes to be made in the base classes thus not improving traceability. The other design methods embed the associations in the classes. The classes are not readily identifiable as implementations of the classes in the analysis model and the associations are distributed between classes.

2. Simpler design

   The results show that these methods result in classes with interfaces which match the corresponding classes in the analysis model. The other methods all result in classes with increased interface size. Classes with smaller interfaces are easier to understand resulting in a simpler system design.

   Both the original system and the extended system under consideration are small. The design complexity of the extended system indicates that the increase in design complexity using the Sociable class design technique, the addition of data stores to represent the association, the combination of inheritance hierarchies or the DSM system is less than with the other methods. The percentage increase in complexity for each of these methods is 17% or less. It seems likely that the design complexity figures for larger systems developed using one of these techniques would be significantly less than the corresponding figures for systems developed using other methods.

   The complexity of the design as measured in this comparison shows that the Sociable class technique and the use of a data store to represent the association result in the simplest designs. This is because those techniques use one type of part whereas the other techniques use two types of parts.

3. Understandability

   The improved traceability and reduced design complexity provided by the explicit implementation of associations results in greater understandability of the classes and the whole system.

Other desired characteristics are shown by two of the methods. DSM and Sociable classes both provide:

1. Increased information hiding

   The Sociable class technique encapsulates the knowledge about associations in generic classes. An instantiation of one of these generic classes encapsulates the knowledge about a specific association. The instances of the association are stored with the related objects.

   The DSM system automatically generates the code required to access the associations and attaches this code to the relevant classes. The information required to implement associations is encapsulated within the `DEFINE <name> RELATION` construct.

   The programmer has very little code to write in order to provide associations by either of these two methods. In all the other methods the programmer is responsible for writing all the code to implement and access each association.

2. Increased standardisation

   Both the Sociable class technique and the DSM system provide the basic relationships as predefined constructs. This allows a standard implementation of associations which also contributes to the understandability of the design.

   The use of role types, hierarchy combination, the object oriented database and OOLog methods all implement associations in a standard way but require the programmer to write all the code.

3. Increased reliability

   Both the Sociable class technique and the DSM system improve reliability by providing pre-tested design constructs. The use of these design constructs reduces the amount of code to be written and therefore reduces the amount of testing required.

It can be seen from the above discussion, that both the Sociable class design method and the DSM system provide many features which improve traceability and may enhance reusability. However, only the Sociable class technique has all these features combined with the ability to be implemented in current object oriented languages.

The only factor not yet addressed is the question of product efficiency. The structures used to implement the relation construct in the DSM system have been stated by Rumbaugh to give an efficient system. Thus providing all the factors for increasing the reusability of design components. The efficiency of designs implemented using the Sociable class technique has not been ascertained. This is an area for future work.

5.5 Conclusions

The major conclusion drawn from this comparison is that the use of the Sociable class design technique has the potential to enhance the possibilities for reuse. This potential enhancement is achieved by increased traceability which leads to improved information hiding, understandability, reliability and standardisation of representation of conceptual associations. In addition, the complexity of the designs is reduced when compared to current methods of implementation and the system can be readily extended to provide increased functionality. The increase in functionality includes the addition of features which were not anticipated when the system was originally developed. The other methods do not provide all these features.

The DSM system provides most of the above features but produces a more complex design because a separate type of part is used to implement the associations and the access features are automatically added to the interfaces. The DSM system also has the disadvantage that the principles cannot be applied to the current languages without the need for pre-processors or changes in the type systems. Preliminary investigations indicate that the Sociable class technique can be used with a variety of current languages.

Both the Sociable class design technique and the DSM system allow conceptual associations to be visible in the implementation and do not require the definitions of the classes to be changed when associations are added. These two methods therefore avoid binding classes together which are involved in a conceptual association that is relevant because of the application and not because of the intrinsic properties of the classes. Class implementations written using either of these two techniques are available for reuse in a different system. However the use of the relation construct in the DSM system to implement both the 'consists of' relationship and conceptual associations loses some of the information about structures. This could be overcome by using instance variables to represent the structure of objects and relations to represent conceptual associations between objects. Both of these methods allow unanticipated changes to be made in the system's functionality without a disproportionate increase in the complexity of the implemented system.

The object oriented database and OOLog methods do not appear to be significantly better than current methods. The only benefit is that if one of the methods were adopted there would be a standard representation of associations. This benefit would also be obtained by using the declaration of sets of associated objects as the standard mechanism for the implementation of all associations.