Implementing the conceptual associations by combining inheritance hierarchies introduces standardisation but leaves the developer to write all the code to implement each required association. This does not therefore provide information hiding but might provide an improvement over current techniques.

The use of role types allows anticipated enhancements to systems. This is achieved by partially separating the classes involved in an association. One of the classes is dependent only on the role type for its implementation and can therefore be reused in any system which includes the role type. This method appears to be useful for extending a system to add new participants in an existing role—an increase in functionality which may be anticipated. However, the classes taking an active part in the association, such as a person owning a car, require an additional instance variable and additional methods so become tightly coupled to the class taking the passive part in the role, such as the car being owned.

The use of the Sociable class technique appears to be potentially capable of providing a greater improvement in the reusability of components than the other methods selected for this comparison.

Chapter 6

Case Study

This chapter presents the results of a case study carried out to verify that the Sociable class technique is successful when applied to a complete system. The case study was chosen to demonstrate the use of a variety of associations in order to identify any problems encountered or further insights into the use of associations. The associations required by the system are one-to-one, one-to-many and many-to-many. The system also requires an association to be formed between two objects of the same class. The development of the one-to-many and many-to-many associations is reported in appendix F. The system is implemented in Eiffel v2.3 because this language provides the best implementation of the specifications of the association classes.

The case study is described in section 6.1. Section 6.2 describes the development of the system using the Sociable class design technique. Section 6.3 discusses the use of the Sociable class design technique. The final section of this chapter, section 6.4 summarises the conclusions drawn from the development of the case study.

6.1 Requirements specification

The kitchen garden system stores information about a variety of crops and insects which may be found in a garden. The system contains details of the user's garden to monitor the crops sown and determine that a defined crop rotation is being followed. All this information is loaded from files.

The system provides the following functions:

1. list the crops and insects which are stored in the system,
2. display the details of the garden,
3. find out which insects eat a certain crop,
4. find out what is eaten by a specified insect,
5. 'sow' a crop in a section of a plot within the constraints of the crop rotation rules (the user will be able to override the constraints if required),
6. list the crops which have been sown in each growing area,
7. find where a given crop has been sown.

6.2 Development

The analysis of the system, section 6.2.1, was carried out following the guidelines and notation given by Rumbaugh et al. [28]. The system was designed, section 6.2.2, using the Sociable class technique. The implementation of the system using Eiffel is discussed in section 6.2.3. Section 6.2.4 discusses the testing and use of the system.

6.2.1 Analysis

The analysis models, figures 6.1 - 6.3, use OMT notation. The features listed as attributes in the class boxes are considered to be intrinsic properties of the objects of that class.

The object model, figure 6.1, shows that this system requires six classes which are related by four different conceptual associations and two aggregation relationships. The 'defines_rules' association between **Garden** and **Crop rotation rules** is a one-to-one association. The 'sown_in' association between **Crop** and **Section** is a one-to-many association. This is because one crop can be sown many times throughout the season. Each sowing is in a separate section. A new section is defined for each crop sown. The 'eats_crop' association between **Crop** and **Insect** is a many-to-many association. The 'eats_insect' association between **Insect** and **Insect** is a many-to-many association.

The top level data flow diagram, figure 6.2, shows a perform transactions process. This process covers all the functions listed in section 6.1.

Figure 6.3 shows the final model of the required system after the object, dynamic and functional models have been combined. The model uses a combination of OMT object and functional model notation. The lines with arrows represent flows of data which were identified on the functional model. The arrow heads point to the class supplying the data. The constraints were also originally identified in the functional model.

The class **Transaction** and its three subclasses, shown in figure 6.3, are not considered part of the original object model, figure 6.1, because they represent operations applied to objects. Class **Date** is not included on the original object model because it represents the type of an attribute and is used in the same way as, for example, class **String** is used to represent the name of a crop. Class **Date** is not a library class so is included on the model of classes to be designed and implemented.

The three subclasses of class **Transaction** provide the functionality of the system. Class **Display** is responsible for displaying the data stored by the system, that is functions 1 and 2 in section 6.1. Class **Query** is responsible for allowing the user to find out about pests and predators, that is functions 3 and 4 in section 6.1. Class **Plant garden** is responsible for allowing the user to sow seeds and find out what is sown in the garden, that is functions 5, 6 and 7 in section 6.1.

6.2.2 Design

The system was designed using Sociable classes where applicable. The object model shows that objects of classes **Garden**, **Crop rotation rules**, **Insect**, **Crop**, and **Section** participate in associations. These classes are therefore designed as Sociable classes and inherit the ability to participate in associations from class **Social**. This can be seen in figure 6.4. Class **Garden plot** does not participate in any associations in the current system so is not designed as a Sociable class. However, it is easy to envisage extensions to the system which might require this class to participate in associations. The consequences of the decision not to design class **Garden plot** as a Sociable class are explained in section 6.3.3.

The associations involving objects of these classes are provided by instantiating the required library classes. The required classes are shown in figure 6.5.

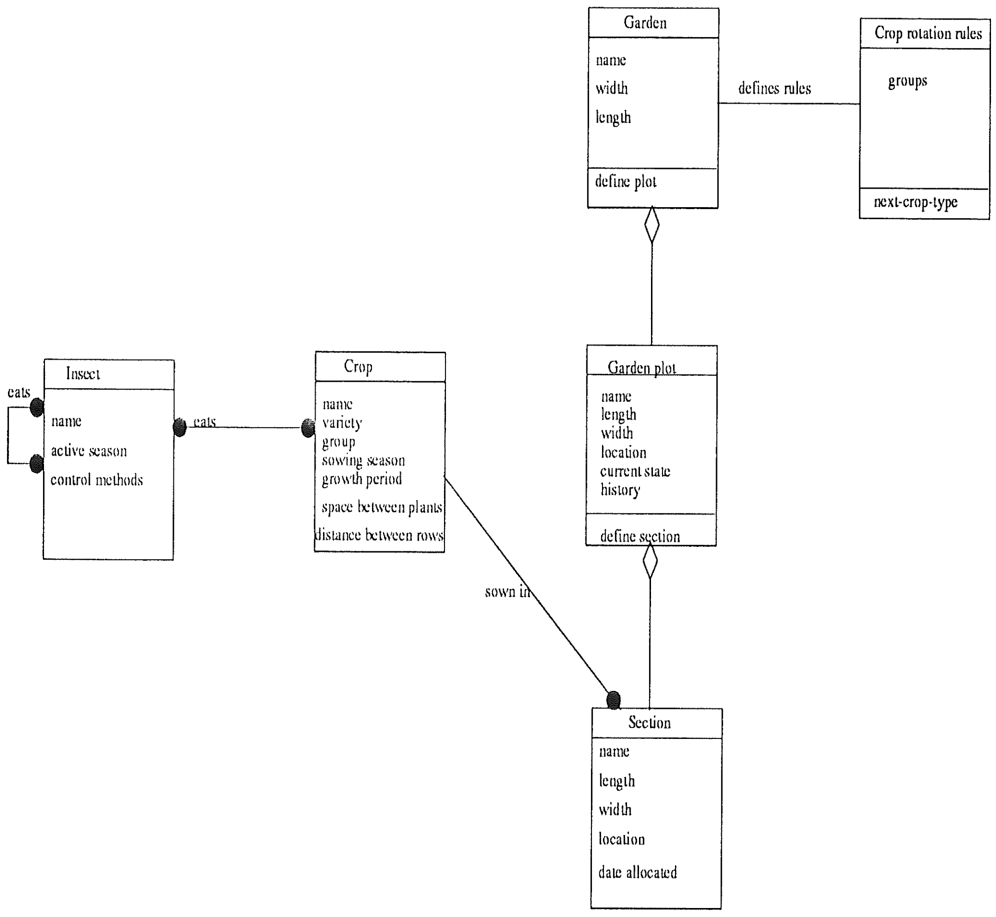

Figure 6.1: Object model for kitchen garden system

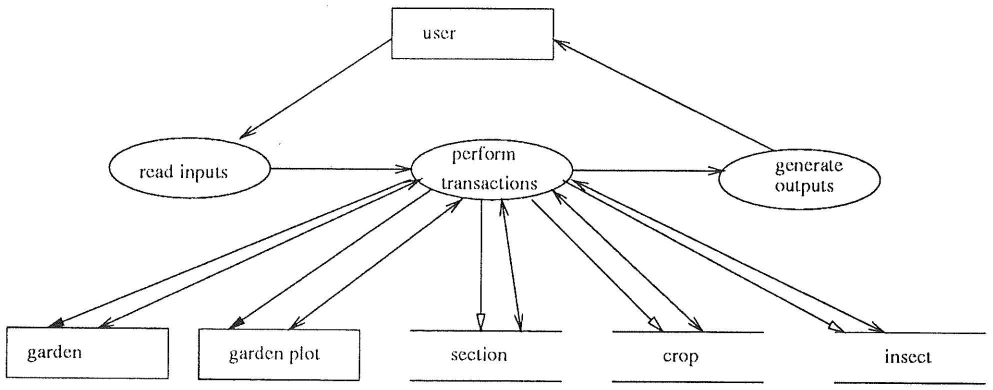

Figure 6.2: Top level data flow diagram for kitchen garden system

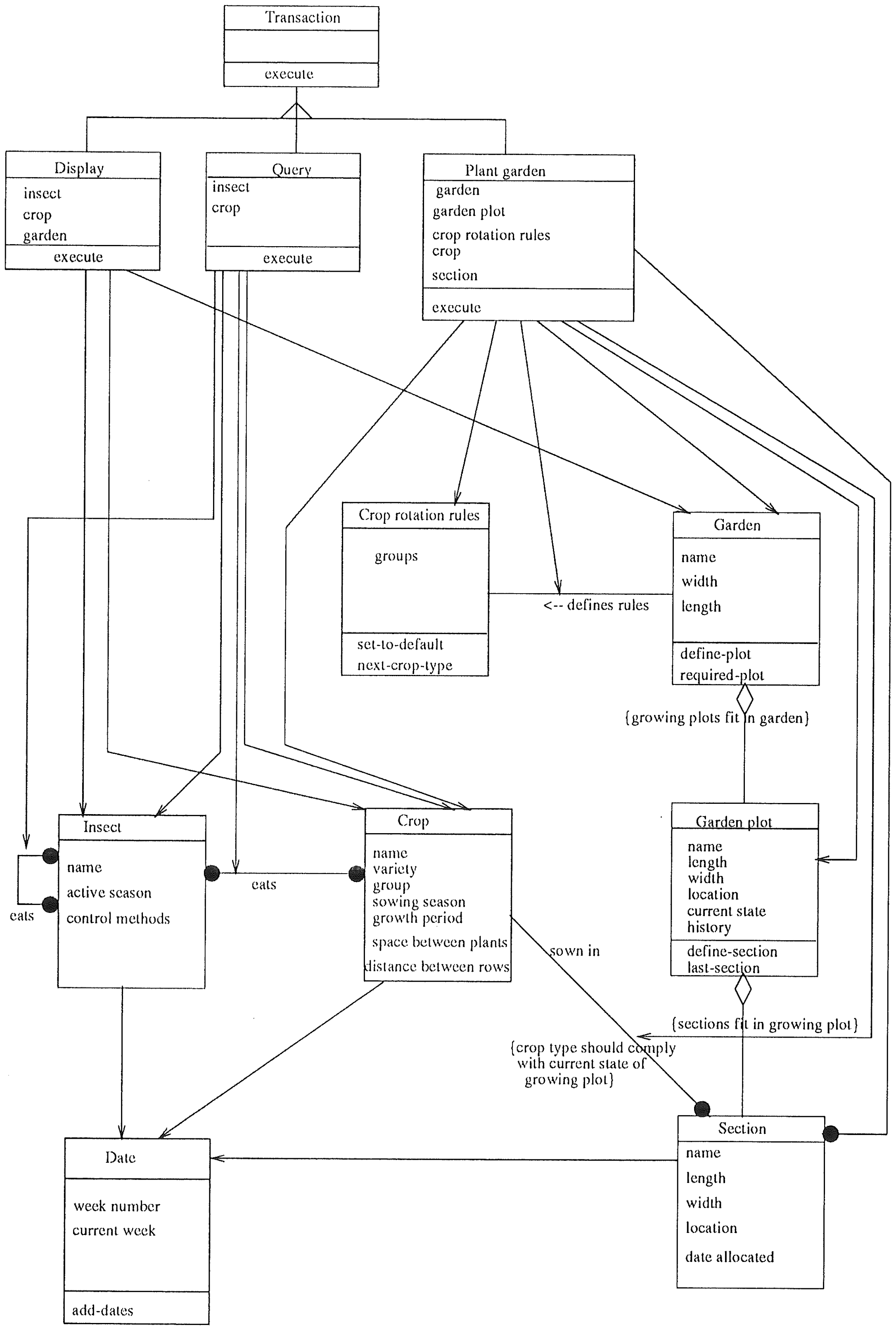

Figure 6.3: Final model for kitchen garden system

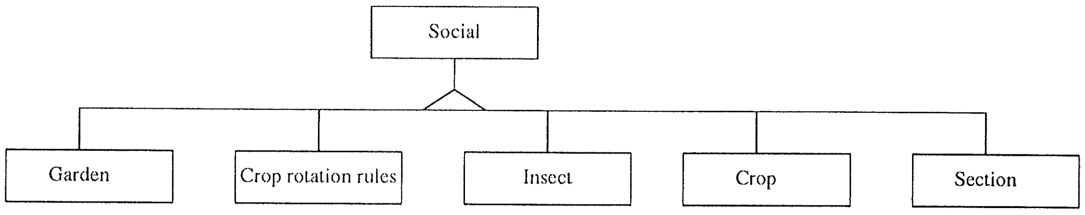

Figure 6.4: Sociable class hierarchy used in the kitchen garden system

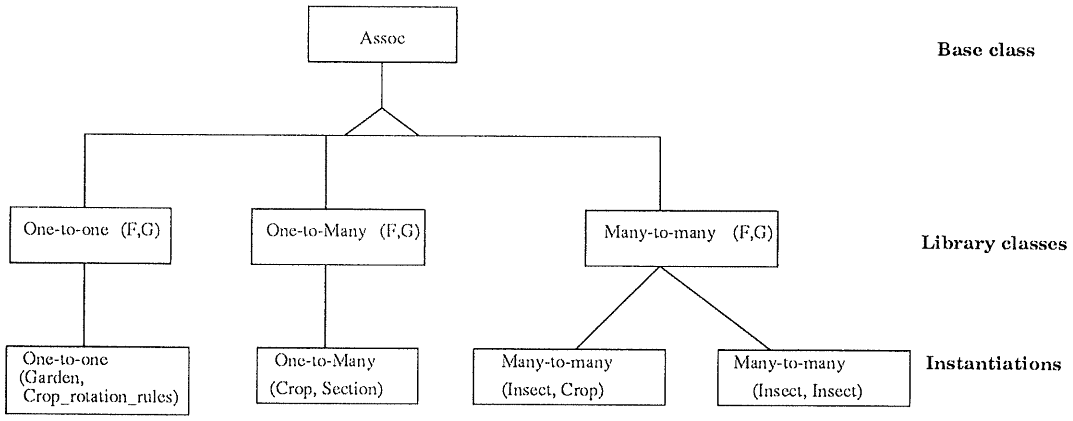
Assoc
Base class
One-to-one (F,G)
One-to-Many (F,G)
Many-to-many (F,G)
Library classes
One-to-one (Garden, Crop_rotation_rules)
One-to-Many (Crop, Section)
Many-to-many (Insect, Crop)
Many-to-many (Insect, Insect)
Instantiations

Figure 6.5: Association hierarchy used in the kitchen garden system

The classes **Garden** and **Garden plot** are aggregations of other classes. This relationship in the Sociable class design method is implemented by the client-server relationship, that is a data structure of the parts is declared as an attribute of the whole structure. This method of representation reflects the fact that sections and plots cannot exist without the garden in which they are located. This dependency is emphasised by the class **Garden** being responsible for allocating the plots. Class **Garden plot** is responsible for the allocation of sections.

The relationships involving the objects of class **Date** are one way 'uses' relationships. They are required to provide intrinsic properties of the objects. For example, the active season of an insect is intrinsic to that insect. The relationships involving class **Date** are therefore designed to use pointers, that is the client-server relationship, in the classes which require dates.

The relationships involving the subclasses of class **Transaction** are again one way relationships. These were also designed using the client-server relationship.

6.2.3 Implementation

The classes which were designed as Sociable classes are derived from class **Social**. The rest of their features were implemented as defined by the analysis model. The interfaces of these classes can be seen in appendix E.5. Objects of classes **Garden plot**, **Date** and the transaction hierarchy do not participate in associations so they do not inherit from **Social**.

The class used to coordinate the system is called **Root** and is responsible for setting up the required information and for the initial interaction with the user. The system is designed for use by one gardener so class **Root** declares one variable of class **Garden**. The system must know about many crops and insects so the class **Root** declares lists of these two objects. The information about crops, insects and the user's garden are stored in files which are read at the start of each session. Files are also used to store information about the required associations. The *set_up* method of class **Root** carries out this function. The code can be seen in appendix E.1.

The associations between objects of the Sociable classes are implemented by instantiating the requisite association from the library. The association between **Garden** and **Crop rotation rules** is implemented by instantiating the **One-to-one2** association with **Garden** and **Crop rotation rules** . The two 'eats' associations are provided by instantiations of the **Many-to-many** associations. Of course, they must have different names. One was defined as 'eats-insect', the other as 'eats-crop'. The declaration of the associations is shown by the following extract from the class **Root**.

```
defines_rules : ONE_TO_ONE2[GARDEN,CROP_ROTATION_RULES];
sown_in : ONE_TO_MANY[CROP,SECTION];
eats_crop : MANY_TO_MANY[INSECT,CROP];
eats_insect : MANY_TO_MANY[INSECT,INSECT];
```

The three associations described so far are simple associations between objects. The required objects are associated by using the *associate* method of the associations. These associations are created by the **Root** class. The following extract of code shows how the required 'eats-crop' associations were formed.

```
read_pests is
local
  crop,pest : INTEGER;
do
  pests.create("pests");
  from pests.open_read;
  until pests.end_of_file
  loop
    pests.readint;
    pest := pests.lastint;pests.next_line;
    pests.readint;
    crop := pests.lastint;
```

– the next line of code forms an association between the required objects

```
    eats_crop.associate(insectlist.i_th(pest),croplist.i_th(crop));
    pests.next_line;
  end; --loop

end; -- read_pests
```

This code shows that only one line of code has to be written by the programmer to form an association between the required objects. The rest of the code is required to read the values from the file.

The fourth association, a one-to-many association between **Crop** and **Section** is more complex. The formation of these associations is part of the functionality and is discussed in the next paragraph.

The functionality of the system, listed above in section 6.1, is defined in the subclasses of class **Transaction**. Class **Display**, see appendix E.2, is responsible for functions 1 and 2, that is display the data stored by the system. **Display** simply asks the user to request the required information and then outputs it to the screen. The other **Transaction** subclasses implement functions which require associations to be traversed. Specifically, functions 3 and 4 which are implemented by the **Query** class, see appendix E.3, use the two 'eats' associations. The code used to display the insects which eat a certain crop is shown below.

```
print_insect_eats_crop is

local
  n : INTEGER;
  list : LINKED_LIST[INSECT];
  pest : INSECT;
  crop : CROP;
do
  io.putstring(" Which crop number do you wish to know about?"); io.new_line;
  print_shortcrops;
  io.putstring(" Enter the number corresponding to the crop");
  io.readint;
  n := io.lastint;io.new_line;
  crop := croplist.i_th(n);
```

– the next line of code retrieves the list of insects from the required crop

```
  crop.display;
  list := eats_crop.find_objects1(croplist.i_th(n));
  if list.void
  then
    io.putstring("has no known pests");
  else
   io.putstring("number of pests = ");
   io.putint(list.count);io.new_line;
    from    list.start
    until   list.off
    loop
       pest:= list.item;
       pest.display;
       list.forth;
    end; --loop
  end;--if
end; --print_insect_eats_crop
```

This code shows that the retrieval of the required information involves accessing the list of insects from the eats_crop association of the required crop using the *find_objects1* method supplied by the association and displaying each item in the list.

Function 5, sowing a crop, is more complex. This is because of the requirement to take the crop rotation rules into account when sowing a crop. Access to the history attribute of the growing plot and to the crop rotation rules of the garden is required before an association of this type is formed. It is also necessary to determine that the desired number of rows can be sown in the chosen part of the area. Thus, the 'sown_in' association encompasses 'dependency' relationship and requires the 'defines rules' association to be traversed and the *current state* feature of the growing area to be accessed. When it has been established that the section should and can be allocated, the association between section and crop is formed using the *associate* method of the **One-to-many** association. Associations of this type are added as a result of user input during a plant_garden transaction. It can be seen from the *sow-crop* method of class **Plant-garden** in appendix E.4 that most of the code is required to confirm and validate the user input. Again the programmer has to write one line of code to form the actual association.

Functions 6 and 7 access the 'sown in' associations created by function 4. The code to implement these functions can be seen in *crop_growing* and *area_growing* features of class **Plant_garden** in appendix E.4. These functions are provided by accessing the required association using the *find_object1* and *find_objects2* methods. The desired information is displayed on the screen. The code implementing these functions is very similar to the code shown above for displaying the insects which eat a crop so is not included here.

All the application specific code is contained in the root class or the subclasses of transaction. None of the base classes were modified to implement the associations. In order to provide the required associations, it was only necessary to declare the associations as features of the root class and write a line of code to form the required associations.

6.2.4 Testing

This section describes the problems encountered when testing the system and gives examples of the system in use after debugging.

The testing of the program was relatively easy. Only one problem was encountered. This occurred when the part of the system which uses the many-to-many association was being tested. The problem came in forming the associations correctly. This problem turned out to be partly due to an error in the code in class **One_to_many** and partly to poor design of class **Many_to_many**. Once these errors were corrected, the associations behaved as expected.

The following text shows the output from the system when a user asks which insects eat a chosen crop. The user first selects the crop from the list

```
Which crop do you wish to know about?
1 Crop name: BROAD BEAN variety: SUTTON
2 Crop name: BROAD BEAN variety: LONGPOD
3 Crop name: CARROT variety: NANTES
4 Crop name: CAULIFLOWER variety: LATEMAN
5 Crop name: LETTUCE variety: TOM THUMB
6 Crop name: CABBAGE variety: MINECOLE
7 Crop name: CABBAGE variety: PROSPERA
8 Crop name: PEA variety: KELVEDON WONDER

Enter the number of the crop: 4
```

and is then given information such as:

```
Crop name: CAULIFLOWER variety: LATEMAN
number of pests = 3
Insect name: GREENFLY
Insect name: WHITE-FLY
Insect name: CABBAGE ROOT FLY
```

Sowing a crop is a more complex process. The user must first select the crop form the displayed list.

```
Which crop do you wish to sow?
1 Crop name: BROAD BEAN variety: SUTTON
2 Crop name: BROAD BEAN variety: LONGPOD
3 Crop name: CARROT variety: NANTES
4 Crop name: CAULIFLOWER variety: LATEMAN
5 Crop name: LETTUCE variety: TOM THUMB
```

```
6 Crop name: CABBAGE variety: MINECOLE
7 Crop name: CABBAGE variety: PROSPERA
8 Crop name: PEA variety: KELVEDON WONDER

Enter the number of the crop: 1
```

The user is then asked where they wish to sow the seeds.

```
Where do you wish to sow the seeds? Choose from the following areas.

1:-
the area is called plot 1
its location is current position: x = 0 : y= 0
 The measurements are 3000 by 150
It is currently growing potato

2:-
the area is called plot 2
its location is current position: x = 200 : y= 0
 The measurements are 3000 by 150
It is currently growing brassica

3:-
the area is called plot 3
its location is current position: x = 400 : y= 0
 The measurements are 3000 by 150
It is currently growing legume

4:-
the area is called plot 4
its location is current position: x = 600 : y= 0
 The measurements are 3000 by 150
It is currently growing rootcrop

 Enter number of plot required: 1
```

If, as in this case, the user chooses a plot which should not contain the crop, the decision to continue has to be confirmed.

```
This crop grew here 2 years ago and should not be in this plot.
Do you wish to sow it now? yes to agree
```

If the user chooses to continue, questions are asked to find the number of rows and position within the plot to sow the seeds. In this example the user has chosen to continue and sows a number of rows which can be planted at the position.

```
yes
How many rows?
4
```

```
requires plot size
length  of section = 120
width of area = 150

The area details are:

the area is called plot 1
its location is current position: x = 0 : y= 0
 The measurements are 3000 by 150
It is currently growing potato.
Where in the plot do you wish to sow the seeds?

Enter the distance along the length of the plot.

200
```

After this successful input, an association between the crop and the section is created. It is then possible to ask where a crop has been sown. The user is first asked to select a crop.

```
 Which crop do you wish to know about?
1 Crop name: BROAD BEAN variety: SUTTON
2 Crop name: BROAD BEAN variety: LONGPOD
3 Crop name: CARROT variety: NANTES
4 Crop name: CAULIFLOWER variety: LATEMAN
5 Crop name: LETTUCE variety: TOM THUMB
6 Crop name: CABBAGE variety: MINECOLE
7 Crop name: CABBAGE variety: PROSPERA
8 Crop name: PEA variety: KELVEDON WONDER
Enter the number of the crop: 1
```

The user is then told in which plot and the position within the plot that the crop has been sown.

```
Crop name: BROAD BEAN variety: SUTTON
 is growing in plot 1 at current position: x = 0 : y= 200
```

The system was further tested with other input. The expected output was obtained.

6.3 Discussion

This section discusses three aspects of the development case study. Section 6.3.1 discusses the use of the design technique. Section 6.3.2 assesses the traceability of the information through the development process. Section 6.3.3 discusses the reusability of the classes involved in the system.

6.3.1 Use of the Sociable class technique

The use of the Sociable class technique presented no problems in the development of the case study. The required associations and class structures can be read from the final object model of the required system. The designer is not required to make decisions concerning the representation of each association. Such decisions in the design methods described in chapter 3.6 include:

- should the association be one-way or two-way?
- if one-way, which way?
- should subclasses be defined or should attributes be added to base classes?
- should a set be used?

A designer using the Sociable class method automatically defines two-way associations so either object can be used for accessing the association.

It is also easy to decide which classes should be declared as Sociable, namely all the classes which, according to the object model, participate in associations. However, as section 6.3.3 points out, it may be wiser to define more than the minimum number of classes as Sociable.

When a system has been designed using Sociable classes, the implementor of the system can implement an association by writing two lines of code in the driver class. One line declares the association, the other creates the association object. One line of code is also needed to form the association between the required objects.

Only the code required to provide the application specific functionality of the system must be written in order to assemble a system from the basic classes. This process is simpler and quicker than defining and coding new subclasses and distributing new information throughout pre-existing code.

6.3.2 Traceability of Information

The traceability of information is assessed by examining the information contained in the final analysis model of the required system and comparing this information with the information obtained by examining the feature declaration of each class.

The final object model, figure 6.3, shows that the system contains one or more objects of classes **Crop**, **Insect** , **Crop_rotation_rules**, **Transaction** , **Display**, **Query** and **Plant_garden**. It also shows that the system contains objects of class **Garden** which represents an assembly structure consisting of many **Garden plot** objects. Each of the plots consists of many **Section** objects. The model also shows that:

1. the following associations are required:
    - a one-to-one between **Garden** and **Crop_rotation_rules**,
    - a one-to-many between **Crop** and **Section**,
    - many-to-many between **Insect** and **Crop** and between **Insect** and **Insect**.
2. the following one directional uses relationships are required:
    - **Insect**, **Crop** and **Section** all use **Date**,
    - **Display** uses **Insect**, **Crop** and **Garden**,
    - **Query** uses **Insect** and **Crop**,
    - **Plant garden** uses **Crop**,**Crop_rotation_rules**, **Garden**, **Garden plot** and **Section**.

In the implemented system, the root class feature declarations show that the garden system involves objects of classes **Garden**, **Crop**, **Insect** , **Crop_rotation_rules**, **Transaction** , **Display**, **Query**, **Plant_garden** and an integer. The declarations also show that the following associations exist:

- a one-to-one between **Crop** and **Crop_rotation_rules**,

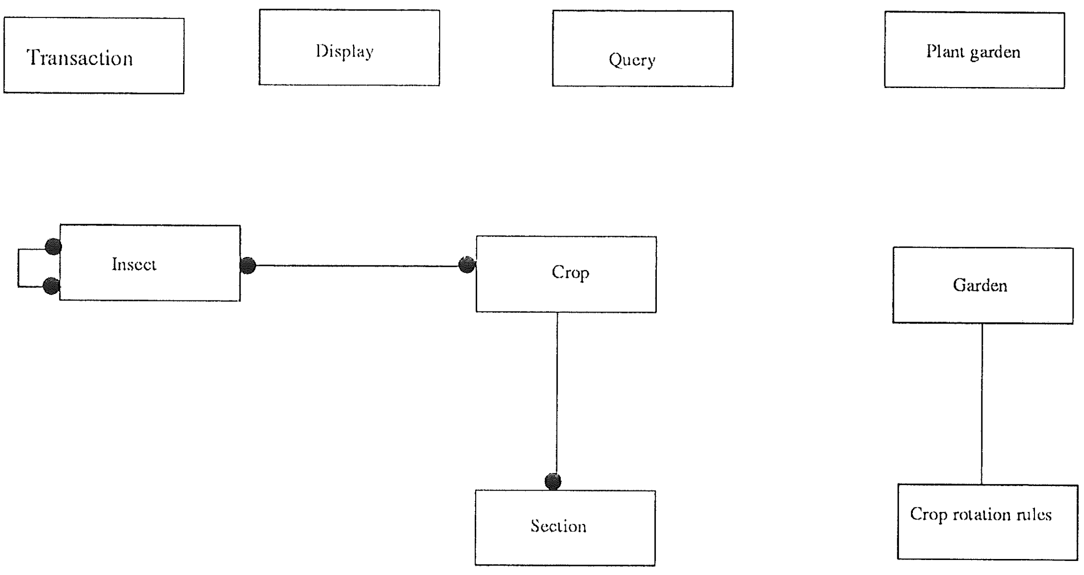

Figure 6.6: Model drawn from root class declarations

- a one-to-many between **Crop** and **Section**,
- many-to-many between **Insect** and **Crop** and between **Insect** and **Insect**.

This information is sufficient to generate the object model shown in figure 6.6.

Further information is required to form the complete model of the system. The structural features of each object are defined by the classes. The structural features of the classes are sufficient to generate a model containing the same information as 6.3.

Thus it is possible to trace information from the original analysis model to the code and to regenerate the analysis model from the class features.

6.3.3 Reuse of classes

The definition of classes **Garden**, **Crop rotation rules**, **Insect**, **Crop**, and **Section** as Sociable classes provides reuse benefits over the alternative methods of providing the required associations. The associations in which they are involved are relevant because of the system in which the classes are being used rather than being intrinsic properties of the objects themselves. For example:

- Class **Garden**.

  The association between class **Garden** and class **Crop rotation rules** is only required because the garden is being used for growing vegetables. The concept of crop rotation would not be relevant in an application concerning flower gardens. The use of an association allows the garden class to be reused more easily in such a system.

- Class **Crop**

  This class participates in two associations. The association with **Section** is again relevant because this is a planting system being developed rather than a system being used by a seed supplier.

The association with **Insect** is required because the gardener requires to know information about insects in the garden. This association is included as part of an original system but can be used to demonstrate a further benefit of using the Sociable class technique. It would have been possible to develop a kitchen garden system without requiring any information about pests or beneficial insects. This extra could be added at a later stage by including an insect class and declaring the required associations in the system. It would not be necessary to define a new subclass of class **Crop**.

Thus, all the classes which were designed as Sociable classes are available for reuse in an extension to this system or in another system without requiring subclasses to be developed. The classes encapsulate only the information which is relevant to their intrinsic properties. This information obviously includes the fact that objects of these classes can participate in associations. The classes do not contain information about the specific classes with which the objects can form associations so are not coupled with classes unnecessarily. The method results in reduced coupling and greater cohesion of information.

Class **Garden plot**, however, was not designed as a Sociable class because the model of the required system showed that it was not required to participate in associations. This decision restricts the extensibility of the system. Imagine that the system is to be extended to contain information about soil conditions, for example, which plots are infected with fungi. This would require an association between a newly introduced **Fungus** class and the class **Garden plot**. Three ways to implement the new association are:

- use one of the design techniques described in chapter 3.6.
- derive a new subclass of class **Garden plot** which defines a sociable class, class **Sociable Garden plot**.
- edit the source code of class **Garden plot** to include a line declaring it as a subclass of class **Social**.

The first two methods do not require the source code of class **Garden plot** to be edited but adversely affect the structure of the system. The third method involves editing the original class code. This is usually considered to be undesirable because the changes may affect existing systems which use the class. In the case of Sociable classes, the additional features gained by declaring **Social** as an extra superclass are completely separate from, and do not interact with, the existing features of the class. In Eiffel, the new features are not exported by the modified class so the interface of the class would not be affected by such a change. It seems probable therefore that the existing system and any other systems using the class would not be affected by the changes. The addition of "sociable" features to existing classes was identified in chapter 4 as an area requiring further research. The potential problems can be avoided by defining classes as Sociable even if, in the current version of the application, they do not require the functionality.

6.4 Conclusions about the use of associations

This case study has reinforced the conclusions drawn in chapter 4. As predicted, use of Sociable classes has enabled reduced coupling and provided greater cohesion of encapsulated information. The use of Sociable classes has improved the reusability of the classes involved in the system. The classes contain only the features required to model their intrinsic properties. The application specific code is all located in the functional classes.

In addition, it was demonstrated that other classes of associations can be implemented and that associations can be formed between objects of the same class. It was also suggested that in order to maximise the extensibility of a system, all classes defined in the initial object model should be declared as Sociable classes.

The case study required instances of one-to-many and many-to-many associations. The exact cardinality of these associations was undefined. The associations used provide the facilities required to make, access and delete such associations.

Other systems might require a specific cardinality such as one-to-two, one-to-three or two-to-five. A user of the general one-to-many and many-to-many associations would be required to ensure that the number of associations did not exceed the required specification. It may be possible to extend the capabilities of the existing one-to-many and many-to-many associations to include the ability to define the maximum and/or minimum cardinality required. Alternatively, a new class could be developed for each cardinality. Extending the functionality of the existing classes appears to be the better option as this would prevent an indefinite number of association classes being required. The exact requirements require further research.

The case study has verified that the Sociable class design technique can be successfully applied to a significant system.

Chapter 7

Conclusion

This chapter assesses the usefulness of the Sociable class technique as a means of improving the reusability of components. Section 7.1 summarises the reasons for developing the Sociable class design method. The main reason is the loss of information during the analysis or design phase of development. Information is lost because several different types of information are represented by the same implementation construct and are not therefore traceable. The lack of traceability causes a reduction in the reusability of the classes and in the extensibility of the system. The traceability of information can be improved by using the Sociable class design technique.This design method and its use both in a feasibility study and in a complete case study are summarized in section 7.2. The feasibility study and case study indicate that the use of the Sociable class design method results in improvements in the reusability of classes and the extensibility of systems.Section 7.3 suggests further research which would be necessary to develop this design technique fully. Other areas in which traceability could be improved are also noted. Section 7.4 assesses the outcome of the project with respect to the original aims.

7.1 Reuse and traceability

Reuse was defined, in section 2.1, to mean **use, with or without modifying the component, in an extension to the existing system or in a different system**. This wide definition of reuse was adopted in order to maximize the benefits accrued by reusing components.

The reuse potential of components was shown in chapter 2 to be dependent on several factors. One of these factors is the traceability of information from the analysed requirements to the final product. Improving the traceability of information was shown to have the potential to led to improvements in several of the other factors, such as standardisation and understandability, which affect the reusability of components. The research therefore concentrated on identifying an area in which the traceability of information could be improved.

Chapter 3 investigated three object oriented development methods which follow the waterfall process model. It was found that, whereas object oriented analysis identifies several different types of information about the interactions between the components of a system, object oriented programming only allows two types of information to be represented. Two of the most common and therefore the most important types of information are the facts that one component is an assembly of other components, such as a car is composed of wheels, body and so on, or that a component has a conceptual association with other components in the system such as 'a person has an account'. Both these types of information are often represented by the same mechanism, the client-server construct, in object oriented programming languages. The same construct is used to represent many other types of relationship. This reduces the traceability of the relationship and results in a loss of information.

The use of the client-server construct to represent conceptual associations between objects was shown to reduce both the reusability of components in a new system and the extensibility of systems. This reduction in reusability occurs because the classes involved in the association become bound together, long inheritance hierarchies are produced when a system is extended and there is a reduced correlation between the problem domain and the implemented system. The information about the association is not encapsulated but is distributed between the classes involved.

An alternative representation is the use of data structures to implement conceptual associations. This was shown to cause different problems. This mechanism encapsulates the information about associations but distributes the information about the objects involved. It was decided that a separate mechanism should be developed to allow conceptual associations to be distinguishable as separate constructs. This would improve traceability and should enhance the reusability and extensibility of the system.

7.2 The Sociable class design method

The Sociable class design method, described in chapter 4, was developed to provide a separate construct to implement conceptual associations. The separate construct is used to provide traceability of the information from the analysis model to the implemented system. The method was designed to be applicable for development using a variety of current object oriented programming languages. The technique can be implemented in languages which provide the basic object oriented features of encapsulation, inheritance and polymorphism.

The design method requires two new class hierarchies to be defined. One type of class is required to define objects with the ability to take part in unspecified associations with other objects. Such classes are called Sociable classes and are derived from the base class **Social**. The other type of classes are associations. All these classes are derived from class **Assoc**. This class is used to provide compatibility between all the different types of association. The different basic types of association are provided by generic or template classes which are derived from class Assoc. One generic or template class is required for each type of association such as one-to-one or many-to-many. These classes encapsulate all the information required to make, break and access that type of association and are provided as library classes. Specific instances of association are produced by replacing the generic parameters with the names of the required classes.

The viability of the design method has been demonstrated by carrying out a feasibility study and a complete case study. The feasibility study involved the development of a small system which was then extended. This system was developed in four object oriented languages. The feasibility study showed that the design could be implemented in each of the languages. The implementation is most reliable in languages, such as Eiffel, which support generics and dynamic type checking of reverse assignments. Further evidence of the viability was provided by the development of a complete case study using Eiffel v2.3. The case study, reported in chapter 6, demonstrated that a variety of different types of association can be developed and that associations can be formed between objects of the same class.

A system designed using the Sociable class method consists of classes which define the basic, intrinsic properties of the objects. Classes do not define any application specific associations in which the objects may be involved. Application specific associations are defined by the application program which creates one instance of each association. This instance is used to associate specified objects. When a new association between specific objects is required a new instance of the association is added to the objects involved. Thus conceptual associations are stored by the objects involved. Associations are object specific not class specific. These properties maintain the object oriented nature of the system because

- the classes define the basic structure and properties of the objects,
- the objects encapsulate all their own information,
- the associations encapsulate information about themselves.

A system designed using this technique can be readily extended to provide increased functionality without requiring new subclasses to be declared. This increase in functionality includes the addition of features which were not anticipated when the system was originally developed.

It should be possible to develop a system from pre-tested units. The pre-tested units would be the application specific classes as defined by the analysis model plus the library classes used to provide the conceptual associations between objects. The application classes can be developed and tested individually. The library classes providing the conceptual associations would be thoroughly tested before inclusion in the library.

In chapter 5, the use of the Sociable class design technique was compared with other techniques which can be used to implement associations between objects. It was found that the Sociable class technique gives potentially more reuse than the other design methods. This greater reusability is brought about by improvements in traceability, information hiding, understandability, reliability and standardisation of representation of conceptual associations. The complexity of the system design is reduced when compared to current methods of implementation.

7.3 Future work

The Sociable class design technique presented in this thesis appears to provide a viable mechanism for the implementation of conceptual associations. It allows these associations to be visible in the implementation of the system. The present development has included the implementation of case studies using one-to-one, one-to-many and many-to-many associations between objects. Further research is needed to provide all the information required to complete the development of the method. Section 7.3.1 identifies some of the more important areas.

Sections 3.6-3.8 indicated that other information modelled during analysis was lost during the design stage and therefore not traceable in the implementation. Section 7.3.2 indicates some areas in which research might be beneficial.

7.3.1 The Sociable class technique

This section suggests areas in which more research is required to allow the Sociable class technique to be fully developed.

- The class **Social** provides a minimal set of operations. This may need to be extended to provide other capabilities. Some possibilities are:

  1. the ability to request that an object returns a list of other objects with which it is associated,
  2. the ability to request a list of the types of association, such as 'has-account' or 'has-share', in which it participates.

  The requirement for other capabilities should be investigated.

- The class **Social** declares a data structure to store the associations. Investigations should be carried out to determine the most efficient data structure to use.
- The technique requires dynamic type checking during the access of associations. This is a relatively slow process so should be investigated to allow developers to be certain that the processor time required does not significantly reduce the efficiency of the system.
- Research needs to be carried out to identify all the generic classes of association which should be developed as library classes. A full specification of all the required operations for each of these classes needs to be developed and implemented.
- The current implementations of the basic associations provide only the minimum functionality required. It may be possible to add code to the features to provide additional functionality. Some possibilities are:
    1. providing the ability to prevent duplication of the associations in the *associate* feature of that class. This feature is responsible for making associations between specific objects.
    2. providing the ability to maintain data consistency when the participants in an association change, for instance, to ensure that when a car changes ownership all the objects involved reflect the change. This would help solve one of the problems encountered when implementing two way associations.
    3. developing the library of associations to include classes for use in distributed systems. It may be possible to include in the association class the code necessary for accessing objects which are located on different machines.
    4. providing the ability to add attributes to associations.
- The possibility of improving the extensibility of existing systems by adding the ability to be "sociable" to the existing classes has not yet been investigated.

In addition to this development work, other related areas may be suitable for investigation. For example, associations between objects are very important in database applications. It might be possible to adapt the design technique for use in database implementations and provide some of the advantages which seem possible with the language implementations.

This research was restricted to the use of current object oriented languages. The design technique developed seems to have many potential advantages. The advantages might be increased by providing Sociable classes and associations as language constructs. A Sociable class could then be declared by using the keyword `social` in a similar way to the declaration of a generic or deferred class in Eiffel. The required associations could also be declared by using language keywords.

7.3.2 Improving traceability of other relationships

This section identifies other aspects of the development process which do not provide traceability.

It was shown in section 3.7 that the conceptual associations are only one of the many relationships between classes and objects which are implemented by the client-server relationship. These other relationships are not traceable. Improving the traceability of these relationships might also improve the reusability of the classes produced. Research could also be carried out to identify mechanisms to represent these relationships in programming languages.

A further area in which research might prove beneficial is the constructs used to represent subsystems. Functional and/or structural subsystems are often identified during analysis. It is not possible to represent this concept in object oriented systems while retaining the reusability of classes. A construct to enable the implementation of subsystems may improve extensibility by dividing the system into larger units than the individual classes.

7.4 Review of achievements

This section assesses the outcome of the project against the aims, defined in sections 1.1 and 2.3. The broad aim defined in section 1.1 was to enhance software reuse in object oriented development. The more specific aim was to improve the reusability of software components which can be developed in currently available object oriented languages. Section 2.3 identified a narrower field within the original aim. This led to the definition of the final objective which was to enhance the reusability of components by improving the traceability of information. Reuse was defined as **use, with or without modifying the component, in an extension to the existing system or in a different system.**

The Sociable class design method was developed and shown to be applicable to a variety of object oriented languages. The method was used in the development of a case study. The results of the case study demonstrate that this method improves traceability by improving the representation of conceptual associations. Three significantly different associations were developed for the case study showing that the method can represent a range of associations.

The improvement in traceability enhances the reusability of the components because:-

1. the implemented classes closely resemble those defined during analysis. This makes the classes easier to understand which should improve the ability to identify and reuse classes from a software library. This enables classes to be reused, with or without modification, in a different system.

2. the implemented system is less complex than the same system implemented by using the other techniques investigated. The simplicity of the system enables the functionality to be extended more easily, thus providing reuse, without modification of the components, in an extension to an existing system. This contrasts with the usual methods of development which require the declaration of new subclasses to implement associations between objects.

In addition, systems are simpler to implement because library classes are used to implement associations between objects. The classes required to implement this design can be implemented using current object oriented language features. However, the benefits of the design method might be further enhanced by the inclusion of the required classes as language constructs.

The Sociable class design method increases the traceability of information about relationships by improving the representation of conceptual associations in programming languages. The increased traceability improves the reusability of the software components. The Sociable class design method successfully meets the aims of the project.

Appendix A

Eiffel classes used to implement the Sociable class method

This appendix contains the code used to implement the classes used in the implementation of the Sociable class design method.

A.1 Class Assoc

This is the base class for all associations.

```
class ASSOC

EXPORT
   make_assoc{ASSOC} , declaredtype{SOCIAL}

feature

declaredtype:STRING; -- added to provide dynamic type checking

create (declared_type : STRING) is
   do
      declaredtype := declared_type.duplicate;
   end; --create

make_assoc(objects : LINKED_LIST[ANY]) is
-- the parameter is declared as a list of type ANY so that any variety of
-- objects can be associated. It is necessary to have at least one of the
-- objects derived from SOCIAL in order to use this class.

   do
io.putstring("this feature must be defined by subclasses");io.new_line;
   end; --make_assoc

end --ASSOC
```

A.2 Class One-to-one2

This class provides code to generate an association between 2 objects with access in either direction.

```
class ONE_TO_ONE2[a -> SOCIAL, b -> SOCIAL]

export
  find_object1, find_object2, associate, disassociate,
  make_assoc {ONE_TO_ONE2}, get_object1{ONE_TO_ONE2},
  get_object2{ONE_TO_ONE2}, make_list{ONE_TO_ONE2},
  find_assoc1{ONE_TO_ONE2},find_assoc2{ONE_TO_ONE2},
  break_assoc {ONE_TO_ONE2}
```

```
inherit
    ASSOC

rename create as basecreate
redefine make_assoc

feature
   x : a;
   y : b;
   done :BOOLEAN;

create (declared_type : STRING) is
  local
      type :STRING;
   do
      type := declared_type;
      basecreate(type);
  end;--create

associate(object1  :  A, object2 : B)
  -- makes an association between the two named objects.
  -- The class of the calling
  -- association  is the class of association produced.

  is

  local
      o1 : A;
      o2 : B;
      link : ONE_TO_ONE2[A,B];
      assoclist : LINKED_LIST[SOCIAL];

  do
      o1 := object1;
      o2 := object2;
      link:= current.deep_clone;
      assoclist.create;
      assoclist := link.make_list(o1, o2);
      link.make_assoc(assoclist);
  end; --associate

disassociate(object1 :  A, object2 : B) is
--breaks the association between two objects

  local
       o1 : A;
       o2 : B;
       link : ONE_TO_ONE2[A,B];
       assoclist : LINKED_LIST[SOCIAL];
  do
       o1 := object1;
       o2 := object2;
       link:= current.deep_clone;
       assoclist.create;
       assoclist := link.make_list(o1, o2);
       link.break_assoc(assoclist);
  end; --disassociate


find_object1( object2 : B): A is
  -- returns an instance of the first actual generic parameter.
  -- Void if no association exists.
  local
      p : B;
      as1 : ONE_TO_ONE2[A,B];
      r : A;
  do
```

```
      p := object2;
      as1 := find_assoc2(p);
      if not as1.void then
         Result := as1.get_object1;
      else
         Result := r;
      end; --if
  end; --findobject1

find_object2( object1 : A): B is
  -- returns an instance of the second actual generic parameter.
  -- Void if no association exists.

  local
      p : A;
      as1 : ONE_TO_ONE2[A,B];
      r : B;
  do
      p := object1;
      as1 := find_assoc1(p);
      if not as1.void then
         Result := as1.get_object2;
      else
         Result := r;
      end; --if
  end; --findobject2
------------------------------------------------------------------
-- private features

make_list(obj1 : a, obj2 : b) : LINKED_LIST[SOCIAL] is
  -- this feature must be called before make_assoc.the first named object must
  --conform to the first actual generic parameter. The second parameter must
  -- conform to the second actual generic parameter.
  -- This cannot be included as a precondition because x and y are void at
  -- this stage. It may be possible to add a create procedure to cure this
  -- problem.

  local
      list : LINKED_LIST[SOCIAL]
  do
      list.Create;
      x.Create;
      y.Create;
      list.put_right(obj2);
      list.put_right(obj1);
      done := TRUE;
      Result := list;
  ensure
       list.i_th(1).conforms_to(y),list.i_th(2).conforms_to(x)

  -- THESE POSTCONDITIONS CAUSE A SEGMENTATION FAULT IF X AND Y ARE VOID
  -- REFERENCES. THE TWO OBJECTS ARE THEREFORE CREATED ABOVE.
  end; --make_list

make_assoc(objects: LINKED_LIST[SOCIAL]) is

 require
       objects.count =2, done, objects.i_th(1).conforms_to(x),
      objects.i_th(2).conforms_to(y)
      -- the conforms_to checks should be superfluous.
  do
      x := objects.i_th(1);
      y := objects.i_th(2);
      x.addAssociation(current);
      y.addAssociation(current);
  end;--make_assoc
```

```
break_assoc(objects: LINKED_LIST[SOCIAL]) is
  --removes the association from the list of associations of both objects
  local
      p : B;
      as1 : ONE_TO_ONE2[A,B];
      r : A;
  do
      r := objects.i_th(1);
      p := objects.i_th(2);
      as1 := find_assoc2(p);--check that association exists
      if not as1.void then
      --should really get the association from object1 as well and check
      -- that they are the same
         p.deleteAssociation(as1);
        r.deleteAssociation(as1);
      end;--if
  end;--break_assoc

find_assoc1(object1 :A) :ONE_TO_ONE2[A,B] is
  -- returns an instance of the same class as association which has object1
  -- as the instance of its first actual generic parameter. Returns void if no
  -- instance of the association exists.
  local
      assoc : ONE_TO_ONE2[a,b];
      p : A;
      c : ASSOC;
  do
      p:= object1;
      c:= p.accessassociation(current);
      assoc ?=c;
      Result := assoc;
  end; --find_assoc1

find_assoc2(object2 :B) :ONE_TO_ONE2[A,B] is
  -- returns an instance of the same class of association which has object2
  -- as the instance of its first actual generic parameter. Returns void if no
  -- instance of the association exists.
  local
      assoc : ONE_TO_ONE2[a,b];
      p : B;
      c : ASSOC;
  do
      p:= object2;
      c:= p.accessassociation(current);
      assoc ?=c;
      Result := assoc;
  end; -- find_assoc2

get_object1 :  a  is
  do
      result:=  x
  end; --get_object1

get_object2 :  b is
  do
      result := y
  end; --get_object2

end -- ONE_TO_ONE2
```

A.3 Class Social

This class is the base class for all Sociable classes.

```
class SOCIAL
```

```
export
      addAssociation{ASSOC}, accessAssociation{ASSOC},
      association_found{ASSOC}, deleteAssociation{ASSOC}
feature
  associations : EXPANDED LINKED_LIST[ASSOC];
  association_found : BOOLEAN;

addAssociation ( c : ASSOC) is -- adds an instance of class
  -- association to  the list of associations.
  do
      associations.finish; -- go to the end of the list
      associations.add_right(c); -- add the association
  end; -- addAssociation

accessAssociation(c : ASSOC) : ASSOC is
  --find and return the instance of the required type of connection
  --  association_found false if not found and VOID returned.
  -- association_found true if found.
  local
      i: INTEGER; -- used for controlling loop
      x : ASSOC;

  do
      association_found := false;
      from i := 0;
      until i = associations.count or association_found

      loop
        i:= i+1; -- initially get element number 1
        x:= associations.i_th(i); -- get the ith
           -- element from the list
        if  x.declaredtype.equal( c.declaredtype)
             --reference semantics
             -- the element is the required type
        then
            association_found := true;
        else
            x.forget;
            -- resets value of x to VOID
        end;--if
      end; --loop
      Result :=  x; -- returns the value of x
  end; --  accessAssociation

deleteAssociation(c:ASSOC)is
  local
      i: INTEGER; -- used for controlling loop
      x : ASSOC;
      done : BOOLEAN;
  do
      association_found := false;
      from    i := 0;
      until i = associations.count or done
      loop
        i:= i+1; -- initially get element number 1
        x:= associations.i_th(i); -- get the ith
            -- element from the list
        if  x.declaredtype.equal( c.declaredtype)
             --reference semantics
              -- the element is the required type
        then
            done := true;
            associations.go(i);
            associations.remove;
        else
            x.forget;
```

```
              -- resets value of x to VOID
          end;--if
        end; --loop
    end;-- deleteAssociation

  end -- SOCIAL
```

A.4 Class Person

This class is an example of a Sociable class. It inherits from **Social** to obtain all the required features.

```
class PERSON
export

      first_name, address, telNo, assignName, assignAddress, assignTelNo
inherit SOCIAL

feature
  -- attributes identified during analysis
      first_name : STRING;
      address : STRING;
      telNo : INTEGER;
  -- exported assign features
assignName ( s:STRING) is
  do
      first_name := s.duplicate;
  end; --assignName

assignAddress ( s:STRING) is
  do
      address := s.duplicate;
  end; --assignAddress

AssignTelNo (i:INTEGER) is
  do
      telNo := i;
  end; -- assignTelNo

end --person
```

Appendix B

Modula-3 classes used to implement the Sociable class method

This appendix contains the code used to implement the classes used in the implementation of the Sociable class design method. Each class is implemented as a pair of modules. One of the modules represents the interface and the other presents the implementation.

B.1 Class Assoc

This is the base class for all associations.

```
INTERFACE Assoc;

TYPE     T <: Assoc;
         Assoc = OBJECT

         METHODS
         makeAssociation(list : REF ARRAY [1..5] OF ROOT)
         END;
END Assoc.

         ****************************************************

MODULE  Assoc;
IMPORT Wr, Stdio;

REVEAL
       T =  Assoc BRANDED OBJECT
OVERRIDES
       makeAssociation := makeAssoc;
END;


PROCEDURE makeAssoc(self :T; list : REF ARRAY [1..5] OF ROOT) =

BEGIN
       Wr.PutText(Stdio.stdout, " This must be
             redefined by subtypes.\n");
       Wr.Close (Stdio.stdout);
END makeAssoc;

BEGIN
END Assoc.
```

B.2 Class One-to-one2

This class provides code to generate an association between 2 objects with access in either direction.

```
GENERIC INTERFACE one_to_one2(F,G);
IMPORT  Assoc;

TYPE
      T<: one_to_one2;
      one_to_one2 = Assoc.T OBJECT
METHODS
      associate(object1: F.T; object2 : G.T);
      find_object1(object2 : G.T) : F.T;
      find_object2(object1 : F.T) : G.T;
END;
END one_to_one2.
           ****************************************************


GENERIC MODULE one_to_one2(F, G);
(* Really need two parameters of type sociable. The parameters are the module
name not the type name but the compiler will not allow the same module to be
 imported twice.
*)

IMPORT Wr, Assoc;

REVEAL
      T = one_to_one2 BRANDED OBJECT
        ob1 : F.T;
      ob2 : G.T;
OVERRIDES
      makeAssociation := makeone_to_one2;
      associate := assct;
      find_object1 := f_object1;
      find_object2 := f_object2;
END;

PROCEDURE makeone_to_one2( self : T; list :REF ARRAY [1..5] OF ROOT) =
(* The compiler will not accept an array of Social.T as compatible
with array of ROOT in the base class. *)

 BEGIN
      self.ob1 := list[1];
      self.ob2 := list[2];
      self.ob1.addAssociation( self );
      self.ob2.addAssociation( self );
 END makeone_to_one2;

PROCEDURE assct(self:T; object1: F.T; object2 : G.T) =
 VAR
      link := NEW (T);
      assoclist := NEW (REF ARRAY [1..5] OF ROOT);
  BEGIN
  (* make a new copy of the association*)
  (* These lines may not be necessary*)
      link.ob1 := self.ob1;
      link.ob2 := self.ob2;
  (* assign the required objects to the assoclist*)
      assoclist[1] := object1;
      assoclist[2] := object2;
  (*make the association between the objects*)
      link.makeAssociation(assoclist);
END assct;


PROCEDURE f_object1(self:T; object2 : G.T) : F.T =
  VAR
```

```
        as1 : T;
    BEGIN
        (* find the required instance of the association*)
        as1 := find_assoc1(self, object2);
        IF NOT as1=NIL (* if the association exists*)
        THEN (* return the first object of the association*)
          RETURN as1.ob1
        ELSE
          RETURN NIL
        END (*IF*)
END  f_object1;

PROCEDURE f_object2( self:T; object1 : F.T) : G.T =
  VAR
        as1 : T;
  BEGIN
        (* find the required instance of the association*)
        as1 := find_assoc2(self,object1);
        IF NOT as1=NIL (* if the association exists*)
        THEN (* return the second object of the association*)
          RETURN as1.ob2
        ELSE
          RETURN NIL
        END (*IF*)
END  f_object2;

PROCEDURE find_assoc2(self:T; object1 : F.T) : T =
  VAR
        assoc :T;
  BEGIN
        RETURN   object1.accessAssociation(self);
END find_assoc2;

PROCEDURE find_assoc1(self:T; object2 :G.T) : T =
  VAR
        assoc :T;
  BEGIN
        RETURN   object2.accessAssociation(self);
END find_assoc1;

BEGIN

END one_to_one2.
```

B.3 Class Social

This class is the base class for all Sociable classes.

```
INTERFACE Social;
IMPORT Assoc;

TYPE
        T <: Social;
        Social = OBJECT
METHODS
        addAssociation ( c:Assoc.T );  (* add a new instance of type
               connector to the list of connections*)
        accessAssociation (  c: Assoc.T ) :Assoc.T; (* return the instance of
              the required type of connection , if none found NIL is returned*)
        deleteAssociation ( c:Assoc.T );(*delete association c from the list
               of connections*)
END;

END Social.
           ****************************************************
```

```
MODULE  Social;
IMPORT Assoc;

CONST  maxAssociations = 10;

REVEAL
  T = Social BRANDED OBJECT
      associations : ARRAY [1..maxAssociations] OF Assoc.Assoc;
OVERRIDES
      addAssociation := aAssoc;
      accessAssociation := accAssoc;
      deleteAssociation := delAssoc;
END;


PROCEDURE aAssoc (self :T;  c: Assoc.T ) =
  (* add a new instance of type connector to the list of connections*)
VAR
       i : INTEGER;
BEGIN
        (*  find next empty space in array *)
        i:=0;
        REPEAT
           INC (i);
        UNTIL self.associations[i] = NIL;
        self.associations[i] := c;
END aAssoc;


PROCEDURE  accAssoc ( self : T;  c: Assoc.T ) : Assoc.T =  (* return the
instance of the required type of connection , if none found NIL is returned*)

VAR
        i := 0 ;
        x : Assoc.T;
BEGIN
       x := NIL;
       REPEAT  (* iterate through the array of associations
                        until the requested type is found.*)
                INC (i);
                x := self.associations[i];
        UNTIL  i =  maxAssociations OR  TYPECODE (x) =TYPECODE (c);
        RETURN x;
END accAssoc;

PROCEDURE delAssoc(self :T; c :Assoc.T) =

VAR
        i := 0 ;
        x : Assoc.T;
BEGIN
        x := NIL;
       REPEAT  (* iterate through the array of associations
                        until the requested type is found.*)
                INC (i);
                x := self.associations[i];
        UNTIL  i =  maxAssociations OR  TYPECODE (x) =TYPECODE (c);
        IF TYPECODE (x) =TYPECODE (c) THEN
                self.associations[i] :=NIL
        END;(*IF*)
END delAssoc;

BEGIN

END Social.
```

B.4 Class Person

This class is an example of a Sociable class. It inherits from **Social** to obtain all the required features.

```
(* This is the interface for the class person with associations*)

INTERFACE Person;
IMPORT Text , Social;
TYPE
      T <: Person;
      Person = Social.T OBJECT
METHODS
  newPerson();
  addName(s:Text.T);
  getName():Text.T;
  addAddress(s:Text.T);
  getAddress():Text.T;
  addTelNo(n:INTEGER);
  getTelNo(): INTEGER;
END;

END Person.

           ****************************************************
 (* This is the implementation for the class person with associations*)

MODULE Person;
IMPORT  Social;

REVEAL
      T = Person BRANDED OBJECT
        name: TEXT;
        address : TEXT;
        telNo : INTEGER;
OVERRIDES
  addName := addN;
  getName := getN;
  addAddress := addA;
  getAddress:= getA;
  addTelNo := addT;
  getTelNo := getT;
END;

PROCEDURE addN (self :T ; s:TEXT)  =
BEGIN
      self.name := s;

END addN;

PROCEDURE getN (self :T):TEXT =
BEGIN
      RETURN self.name;
END getN;


PROCEDURE addA(self :T ; s:TEXT) =
BEGIN
      self.address := s;
END    addA;


PROCEDURE getA(self :T):TEXT =
BEGIN
      RETURN self.address;
END getA;
```

```
PROCEDURE addT(self :T ; n:INTEGER) =
BEGIN
      self.telNo := n;
END    addT;


PROCEDURE getT(self :T): INTEGER =
BEGIN
      RETURN self.telNo;
END getT;

BEGIN

END Person.
```

Appendix C

Oberon-2 classes used to implement the Sociable class method

This appendix contains the code used to implement the classes used in the implementation of the Sociable class design method.

C.1 Class Assoc

This is the base class for all associations.

```
MODULE Assoc; (* Record plus type bound procedures*)

TYPE
  string *= ARRAY 32 OF CHAR;
  Assoc* = POINTER TO AssocDesc;
  AssocDesc * = RECORD
            (*exported in order to read-only export the fields *)
              declaredType  - : ARRAY 32 OF CHAR;
              END(*record*);

PROCEDURE(a : Assoc) assignType*( s: string);
BEGIN
   COPY (s,a.declaredType);
END assignType;

END Assoc.
```

C.2 Class One-to-one2

This class provides code to generate an association between 2 objects with access in either direction. This is a template for all associations of this type. In order to define a specific association, make a copy of this file with the required name. In the IMPORT statement replace the two occurrences of Social with the required module names. The module must be renamed both at the top of the file and after the final END.

```
MODULE OneToOne2;

IMPORT Assoc, Object, A := Social, B := Social;

TYPE T* = POINTER TO OneToOne2Desc;
 OneToOne2Desc = RECORD(Assoc.AssocDesc)


 obj1 : A.T;
 obj2 : B.T;
 END(* RECORD*);
```

```
PROCEDURE (a : T) associate*(object1    :A.T; object2 : B.T);
VAR
     link : T;
     type : Assoc.string;
BEGIN
      NEW (link);
      COPY (a.declaredType,type);
     link.assignType(type);
     link.obj1 := object1;
     link.obj2 := object2;
     link.obj1.AddAssociation(link);
     link.obj2.AddAssociation(link);
END  associate;

PROCEDURE (a : T) findObject1*(object2 : B.T) : A.T;
VAR
     as1 : T;
     o : B.T;
     b :Assoc.Assoc;
BEGIN
(* Find the required instance of the association*)
     o := object2;
     b := o^.AccessAssociation(a);
     WITH b :T DO
       as1 := b;
     END(*WITH*);
     RETURN as1^.obj1
END  findObject1;

PROCEDURE (a : T) findObject2*(object1 : A.T) : B.T;
VAR
     as2 : T;
     o : A.T;
     b :Assoc.Assoc;
BEGIN
  (* Find the required instance of the association*)
     o := object1;
     b := o^.AccessAssociation(a);
     WITH b :T DO
       as2 := b;
     END(*WITH*);
     RETURN as2^.obj2
END  findObject2;
END OneToOne2.
```

C.3 Class Social

This class is the base class for all Sociable classes.

```
MODULE Social; (* Record plus type bound procedures*)

IMPORT  Object, Assoc;
TYPE
     T* = POINTER TO SocialDesc;
     SocialDesc * = RECORD (Object.ObjectDesc)
         (*exported in order to read-only export the fields *)
          associations - :  ARRAY 10 OF Assoc.Assoc;
     END(*record*);

PROCEDURE(s : T) AddAssociation* ( a : Assoc.Assoc);
  (* add a new instance of association to the array of associations*)
VAR
     c : INTEGER;
```

```
BEGIN
     c :=0;
     REPEAT
          INC(c);
     UNTIL     s.associations[c] =NIL;
     s.associations[c] := a;
END AddAssociation;

PROCEDURE (s : T) AccessAssociation*(a: Assoc.Assoc) : Assoc.Assoc;
  (* return the instance of association  of the type requested.
  NIL returned if not found*)
TYPE
     string =ARRAY 32 OF CHAR;
VAR
     c : INTEGER;
     x :  Assoc.Assoc;
     current, required : string ;
BEGIN
     c := 0;
     COPY (a.declaredType,required );
     REPEAT
       INC(c);
       x :=  s.associations[c] ;
       COPY (x.declaredType,current );
     UNTIL   ( c= 10 )   OR    (x.declaredType = a.declaredType);
     RETURN x;
END AccessAssociation;

END Social.
```

C.4 Class Person

This class is an example of a Sociable class. It inherits from **Social** to obtain all the required features.

```
MODULE Persons; (* Record plus type bound procedures*)
IMPORT Social;
TYPE
     T* = POINTER TO PersonDesc;
          PersonDesc * = RECORD (Social.SocialDesc)
          (*exported in order to read-only export the fields *)
               name- : ARRAY 32 OF CHAR;
               address- : ARRAY 32 OF CHAR;
               telNo- : INTEGER;
          END(*record*);

PROCEDURE(p:T) AssignName*(n : ARRAY OF CHAR);
BEGIN
     COPY (n, p.name);
END AssignName;

PROCEDURE (p:T) AssignAddress* (n : ARRAY OF CHAR );
BEGIN
     COPY (n, p.address);
END AssignAddress;

PROCEDURE (p:T) AssignTelNo* (i :INTEGER);
BEGIN
     p.telNo := i;
END AssignTelNo;

END Persons.
```

Appendix D

C++ classes used to implement the Sociable class method

This appendix contains the code used to implement the classes used in the implementation of the Sociable class design method.

D.1 Class Assoc

This is the base class for all associations.

```
#ifndef ASSOC_H
#define ASSOC_H

/* #define NULL ((void *)0) */
#include <iostream.h>

class Social;
typedef char *AssocType;

class Assoc {
    int itag;
    char *ctag;
  public:
    Assoc(void) {  }
    virtual AssocType mytype_dyn(void) { return "error";}
    int sametype(AssocType id) { return strcmp(id,mytype_dyn())==0; }
};

class AListElem {
    Assoc *val; AListElem *nxt;
    AListElem(Assoc *v, AListElem *n) { val=v; nxt=n;}
    friend class AssocList;
};

class AssocList {
  private:
    AListElem *l, *iterp;
  public:
    AssocList(void) { l=NULL; iterp=NULL; }
    void add(Assoc *a) { l = new AListElem(a,l); }
    int empty(void) { return l==NULL; }
    void iterset(void) { iterp=l; }
    void iterstep(void) { if(iterp!=NULL) iterp=iterp->nxt; }
    int iterdone(void) { return iterp==NULL; }
    Assoc *iterget(void) { if(iterp!=NULL) return iterp->val; }
};

#endif ASSOC_H
```

D.2 Class One-to-one2

This class provides code to generate an association between 2 objects with access in either direction. This implementation uses two classes to implement the specification of class **One-to-one2**. Class **OneToOne2Node** is used to provide the actual links between the objects. Class **OneToOne2** provides the functionality required to make and access the associations.

```
#ifndef ONETOONE2_H
#define ONETOONE2_H

#include <iostream.h>
extern "C" {
extern char *strdup(char *);
}
#include "assoc.h"

template<class A, class B>
class OneToOne2Node : Assoc {
  private:
    A *a;
    B *b;

    char *typename;
  public:
    OneToOne2Node(A *aa, B *bb,  char *tn) :Assoc() {
 a = aa; b = bb;  typename = strdup(tn);
    }
    char *mytype_dyn(void) { return typename; }
    friend class OneToOne2<A,B>;
};

//template<class A, class B>char *OneToOne2Node<A,B>::typename = 0;

   *******************************************************************

template<class A, class B>
class OneToOne2 {
    char *assoctypename;
  public:
    OneToOne2(char *typn) {
// OneToOne2Node<A,B>::typename = strdup(typn);
 assoctypename = strdup(typn);
    }
    void associate(A *aa, B *bb) {
 OneToOne2Node<A,B> *tmp = new OneToOne2Node<A,B>(aa,bb,assoctypename);
 aa->addassoc(tmp); bb->addassoc(tmp);
    }
    A *geta(B *bb) {
 AssocList *os = bb->findassocs(assoctypename);
 os->iterset();

 return ((OneToOne2Node<A,B> *)(os->iterget()))->a;

 return NULL;
    }
    B *getb(A *aa) {
 AssocList *os = aa->findassocs(assoctypename);
 os->iterset();

 return ((OneToOne2Node<A,B> *)(os->iterget()))->b;

 return NULL;
    }
};


#endif ONETOONE2_H
```

D.3 Class Social

This class is the base class for all Sociable classes. It is implemented in two files. the header file is printed first.

```
#ifndef SOCIAL_H
#define SOCIAL_H

#include "assoc.h"

class Social {
  private:
    AssocList assocs;
  public:
    Social(void) :assocs() { }
    void addassoc(Assoc *a);
    AssocList *findassocs(AssocType id);
};


#endif SOCIAL_H

       ****************************************************

#include "social.h"

void Social::addassoc(Assoc *a) {
    assocs.add(a);
}

AssocList *Social::findassocs(AssocType id) {
    AssocList *res = new AssocList;
    assocs.iterset();
    while(! assocs.iterdone()) {
 if(assocs.iterget()->sametype(id)) {
     res->add(assocs.iterget());
 }
 assocs.iterstep();
    }
    return res;
}
```

D.4 Class Person

This class is an example of a Sociable class. It inherits from **Social** to obtain all the required features.

```
#ifndef PERSON_H
#define PERSON_H

#include <iostream.h>
#include "social.h"

class Person : public Social {
  private:
    char *name;
    int age;
  public:
    Person(char *n, int a) :Social() {
 name = new char[strlen(n)+1];
 strcpy(name,n); age=a;
    }
    const char *who(void) { return name; }
    const char *getname(void) { return name; }
    int getage(void) { return age; }
    void show(void) { cout << "Person{" << name << ", " << age << "}"; }
};

#endif PERSON_H
```

Appendix E

Case study implementation

This appendix contains the code skeletons for the classes used in the implementation of the case study detailed in chapter 6. The code shown is sufficient to allow the reader to understand the classes without reading the details. All the code which accesses associations is shown in the skeletons. Section E.1 contains the main controlling class for the system. Sections E.2- E.4 contain the classes which provide the application specific functionality. Section E.5 contains the Sociable classes and section E.6 contains the code for the other classes required by the system.

E.1 Root class

This class coordinates the system functionality.

```
class ROOT

feature

-- DATA
garden : GARDEN;

croplist : LINKED_LIST[CROP];
crop_rotation : CROP_ROTATION_RULES;
insectlist :LINKED_LIST[INSECT];

-- ASSOCIATIONS
defines_rules : ONE_TO_ONE2[GARDEN,CROP_ROTATION_RULES];
sown_in : ONE_TO_MANY[CROP,SECTION];
eats_crop : MANY_TO_MANY[INSECT,CROP];
eats_insect : MANY_TO_MANY[INSECT,INSECT];

-- FUNCTIONALITY
session : TRANSACTION;
display : DISPLAY;
query : QUERY;
plant_garden : PLANT_GARDEN;

choice : INTEGER;

-- STORAGE
gardens,crops,insects,predators,pests:FILE;


create is
local

do
   io.putstring(" Welcome to the garden system");
   io.new_line;
   io.new_line;
   io.new_line;
   set_up;     --reads in data
   display.create(garden,croplist,insectlist);
   query.create(croplist,insectlist,eats_crop, eats_insect);
    plant_garden.create(garden,croplist, crop_rotation,defines_rules,
                       sown_in);
   from       get_choice;
   until choice = 10
   loop
      inspect choice
        when   1 then   session := display
        when   2 then   session := query
        when   3 then    session := plant_garden
      end; -- inspect
      session.execute;
      get_choice;
   end; -- loop
end; --create


set_up is
do
```

–read information from files to set up the required data and associations

```
end;--set_up


get_choice is
do
```

– display menu of choices
   – get their choice

```
end;--get_choice


read_gardens is
```

– read the details from the file into the garden variable

```
end; --readgardens


read_crops is
```

– read the details from the file into the list of crops

```
end; -- readcrops


read_insects is
```

– read the details from the file into the list of insects

```
end; --read_insects


read_predators is
```

– read the details from the file to form the associations between insects

```
local
  predator,pest : INTEGER;
do
  predators.create("predators");
  from predators.open_read;
  until predators.end_of_file
  loop
    predators.readint;
    predator := predators.lastint;predators.next_line;
    predators.readint;
    pest := predators.lastint;
    eats_insect.associate(insectlist.i_th(predator),insectlist.i_th(pest));
    predators.next_line;
  end; --loop
end; -- read_predators


read_pests is
```

– read the details from the file to form the associations between insects and crops

```
local
  crop,pest : INTEGER;
do
  pests.create("pests");
  from pests.open_read;
  until pests.end_of_file
  loop
    pests.readint;
    pest := pests.lastint;pests.next_line;
    pests.readint;
    crop := pests.lastint;
    eats_crop.associate(insectlist.i_th(pest),croplist.i_th(crop));
    pests.next_line;
  end; --loop
end; -- read_pests

end --root
```

E.2 Class Display

This class is responsible for displaying the data stored by the system.

```
class DISPLAY
export execute

inherit transaction
feature
  choice : INTEGER;
  garden : GARDEN;
  croplist : LINKED_LIST[CROP];
  insectlist :LINKED_LIST[INSECT];


create(g: GARDEN;c_l: LINKED_LIST[CROP];i_l :LINKED_LIST[INSECT]) is
do
  garden:= g;
  croplist := c_l;
  insectlist := i_l;
end; --create


execute  is
do
io.putstring(" Welcome to the display section");
   io.new_line;
   from       get_choice;
   until choice = 10
   loop
      inspect choice
        when   1 then   display_garden
        when   2 then   display_crops
        when   3 then   display_insects
      end; --inspect
      get_choice;
   end; --loop
end; --execute


get_choice is
do
```

– display menu of choices
   – get their choice

```
end;--get_choice


display_garden is
do
```

– display the garden details

```
end; --display_garden


display_crops is
do
```

– display the list of crops

```
end; --display_crops


display_insects is
do
```

– display the list of insects

```
end; --display_insects

end --display
```

E.3 Class Query

This class allows the user to find out about pests and predators.

```
class QUERY
export execute

inherit transaction
feature

  choice : INTEGER;
  croplist : LINKED_LIST[CROP];
  insectlist :LINKED_LIST[INSECT];

  eats_crop : MANY_TO_MANY[INSECT,CROP];
  eats_insect : MANY_TO_MANY[INSECT,INSECT];


create(c_l : LINKED_LIST[CROP];i_l :LINKED_LIST[INSECT];
e_c : MANY_TO_MANY[INSECT,CROP];  e_i : MANY_TO_MANY[INSECT,INSECT])  is
do
  croplist := c_l;
  insectlist := i_l;
  eats_crop := e_c;
  eats_insect := e_i;
end; -- create


execute is
do
```

```
   io.putstring(" Welcome to the question section");
   io.new_line;
   from      get_choice;
   until choice = 10
   loop
      inspect choice
        when   1 then   print_insect_eats_crop
        when   2 then   print_insect_eaten_by_insects
        when   3 then   insect_eats
        when   4 then   crops_eaten_by_insect
      end; --inspect
      get_choice;
   end; --loop
end; --create



get_choice is
do

– display menu of choices
   – get their choice

end;--get_choice

print_shortcrops is

– display name and variety of all crops

end; --   print_shortcrops

print_shortinsects is

– display names of all insects

end; --   print_shortinsects

print_insect_eats_crop is

local
  n : INTEGER;
  list : LINKED_LIST[INSECT];
  pest : INSECT;
  crop : CROP;
do

– get users choice of crop
   –use eats_crop association to return list of insects eating the crop

  list := eats_crop.find_objects1(croplist.i_th(n));
  crop.display;
  if list.void
  then
    io.putstring(" has no known pests");
  else

– display list of pests

  end;--if
end; --print_insect_eats_crop

print_insect_eaten_by_insects is
local
  n : INTEGER;
  list : LINKED_LIST[INSECT];
  predator,pest : INSECT;
do

– get users choice of insect
   –use eats_insect association to return list of insects eating the insect
  list := eats_insect.find_objects1(insectlist.i_th(predator);
  predator.display;
  if list.void
  then
    io.putstring("has no known predators");
  else
```

– display list of pests

```
  end;--if

end; --print_insect_eaten_by_insects

insect_eats is
local
  n : INTEGER;
  list1 : LINKED_LIST[CROP];
  list2 : LINKED_LIST[INSECT];
  insect,pest : INSECT;
  plant :  CROP;
do
```

– get users choice of insect
  –use eats_crop association to return list of crops eaten by the insect

```
  list1 := eats_crop.find_objects2(insectlist.i_th(n));
  if list1.void
  then
    io.putstring("No known crops eaten");
  else
```

– display list of crops eaten

```
  end;--if
```

–use eats_crop association to return list of crops eaten by the insect

```
  list2 := eats_insect.find_objects2(insectlist.i_th(n));
  if list2.void
  then
    io.putstring("No known insects eaten");io.new_line;
  else
```

– display list of crops eaten

```
  end;--if
end; -- insect_eats

crops_eaten_by_insect is
local
  n : INTEGER;
  list : LINKED_LIST[CROP];
  plant : CROP;
  insect : INSECT;
do
```

– get users choice of insect
  –use eats_crop association to return list of crops eaten by the insect

```
  list := eats_crop.find_objects2(insectlist.i_th(n));
  insect.display;
  if list.void
  then
    io.putstring("No known crops eaten");
  else
```

– display list of crops eaten

```
  end;--if
end; --crops_eaten_by_insect
end --query
```

E.4 Class Plant_garden

This class is responsible for allowing the user to sow seeds and find out where crops are growing.

```
class PLANT_GARDEN

export execute

inherit TRANSACTION

feature

choice :INTEGER;
garden: GARDEN;
croplist : LINKED_LIST[CROP];
crop_rotation : CROP_ROTATION_RULES;
defines_rules : ONE_TO_ONE2[GARDEN,CROP_ROTATION_RULES];
sown_in : ONE_TO_MANY[CROP,SECTION];

create(g: GARDEN; c : LINKED_LIST[CROP], c_r : CROP_ROTATION_RULES;
d_r : ONE_TO_ONE2[GARDEN,CROP_ROTATION_RULES];
s_i : ONE_TO_MANY[CROP,SECTION])

is
local

do
    garden := g;
    croplist :=c;
    defines_rules:= d_r;
    crop_rotation := c_r;
    sown_in := s_i;

end; --create

execute is
do
      io.putstring(" Welcome to the planting section");
      io.new_line;
      io.new_line;
      io.new_line;
  from get_choice
  until choice = 10
  loop
      inspect choice
        when   1 then   sow_crop
        when   2 then   crop_growing
        when   3 then   area_growing
      end; --inspect
      get_choice;
   end; --loop
end; --create

get_choice is
do
```

– display menu of choices
   – get their choice

```
end;--get_choice

sow_crop is
local
  plant : CROP;
  area : GARDENPLOT;
  next_type: STRING;
  sow : BOOLEAN;
  section : SECTION;
  place,rows : INTEGER;
  length : INTEGER;
do

– find out the crop to sow
    – find out where to sow it
    – what is the next crop type to go in this area?
    – does selected crop group coincide with current state of plot?

  if plant.group.equal(area.current_state)

– yes then crop can be sown

  then
    sow := TRUE;

– would this comply with the desired crop rotation?
    – tell user it should be in this area next year.
    –ask if wish to sow anyway if answer yes

    then
     sow := TRUE;
    end; --if

– if crop grew here last year
    – tell user crop grew here last year and should not be in this plot
    –ask if wish to sow anyway if answer yes

    then
     sow := TRUE
    end; --if

  end;--if

    – if user has decided to sow the crop
    – ask for the number of rows required and the position in the plot
    – if plot can be allocated
    – form an association between crop and section

    sown_in.associate(plant,section);
end; --sow_crop

crop_growing is
local
  plant : CROP;
  area : GARDENPLOT;
  next_type: STRING;
  sow : BOOLEAN;
  section : SECTION;
  place,rows : INTEGER;
  length : REAL;
  sectionlist : LINKED_LIST[SECTION];
do

– ask user which crop they wish to find – use the sown_in association to return the list of sections growing the crop

  sectionlist := sown_in.find_objects2(plant);

– if crop has been sown
    – display the list of places

end;--crop_growing


area_growing is
local
  plot : GARDENPLOT;
  sections : ARRAY_LIST[SECTION];
  crop : CROP;
do
```

– find out which plot is required
    –if sections have been planted use sown_in association to find out which crop is growing in each section

```
     crop := sown_in.find_object1(sections.item);
```

– display the list of crops

```
end; --area_growing

end --plant_garden
```

E.5 Sociable classes

This section contains the code skeletons of the Sociable classes used in the kitchen garden system.

E.5.1 Class Crop

```
class  CROP export

display_details, assign_name,
assign_variety, assign_group, assign_sowing_season,
assign_growth_period, assign_space_between_plan,
assign_distance_between_r, crop_name, variety, group,
sowing_season, growth_period, space_between_plants,
distance_between_rows, display

inherit SOCIAL

feature
crop_name,variety,group : STRING;
sowing_season : ARRAY[DATE];
growth_period :DATE;
space_between_plants,distance_between_rows : INTEGER;

create is
do
  sowing_season.create(1,2);
end; --create
```

– assign features for all the above features

```
display_details is
```

– display all the details of the crop

```
end; --display_details

display is
```

– display the name and variety of the crop

```
end; --display
end --class CROP
```

E.5.2 Class Garden

```
class GARDEN export

garden_name, display,
assign_garden_name, define_plot, width, length,
required_plot, last_plot

inherit SOCIAL

feature
--attributes
```

```
garden_name: STRING;

length,width: INTEGER;

areas: ARRAY_LIST [GARDENPLOT];


create(gname:STRING; len, wid : INTEGER) is
 do
        assign_garden_name(gname);
        define_size(len,wid);
        areas.Create;
end; --create
```

– assign features for all the above features

```
display  is
do
```

– display the garden details

```
 end; -- display


define_plot(aname:STRING; len,wid,coord1,coord2 :INTEGER) is
local
        newarea : GARDENPLOT;
        done, correct :BOOLEAN;
do
```

–create a new area
  – add it to the areas feature if valid
  – output error message if not valid.

```
end; -- define_gardenplot


display_area_details is
```

–display details of all the garden plots.

```
end; -- display_area_details


required_plot: GARDENPLOT is
```

– return the required plot

```
end; --required_plot


last_plot :GARDENPLOT is
```

– return the last plot in the list

```
end; --last_plot

end --class GARDEN
```

E.5.3 Class Section

```
class  SECTION export

area_name, a_length, a_width, assign_name,
assign_length, assign_width, location,
assign_location, set_date

repeat NAMED_AREA
inherit SOCIAL;NAMED_AREA
```

```
feature
location :POINT1;
date_allocated : DATE;

create(sname:STRING;length,width:INTEGER) is
```

–assign parameters to features

```
end;--create


assign_location(x,y:INTEGER)   is
```

– assign value to location

```
end; --assign_location

set_date(week: DATE) is
```

– assign date to date_allocated

```
end; --set_date
end --section
```

E.5.4 Class Insect

```
class INSECT export

insect_name, active_season, control_methods,
assign_insect_name, assign_active_season,
assign_control_methods, display, display_details

inherit SOCIAL

feature
insect_name, control_methods : STRING;
active_season : ARRAY[DATE];

create is
do
  active_season.create(1,2);
end; --create
```

  – assign methods for above features

```
display_details is
```

  – display all the details of the insect

```
end; --display_details

display is
```

– display insect name

```
end; --display
end -- insect
```

E.5.5 Class Crop_rotation_rules

```
class CROP_ROTATION_RULES export

set_to_default, next_crop_type, error
inherit SOCIAL
feature
```

```
types : ARRAY_LIST[STRING];
max_value : INTEGER;     -- used to hold the number of crops in the rotation
rules_assigned,error :BOOLEAN;

Create is
do
        types.create;
end; --Create

set_to_default is
```

– add required default values

```
end;--set_to_default

next_crop_type ( type1: STRING) : STRING is
```

–return the next crop type in the series

```
end; -- next_crop_type

end --crop_rotation_rules
```

E.6 Other classes required

E.6.1 Class Gardenplot

```
class  GARDENPLOT export

area_name, display_details, set_history, history,
sections, last_section, location, assign_name,
assign_length, assign_width, assign_location,
current_state, a_width, define_section,
completed
feature
        area_name:STRING;
        a_length,a_width:INTEGER;
        location :POINT1;
        done :BOOLEAN;
        sections : ARRAY_LIST[SECTION];
        history : ARRAY[STRING];
        completed : BOOLEAN;

create(aname: STRING;a_len,a_wid,coord1,coord2 :INTEGER)  is
```

– assign parameters to features

```
display_details is
```

– display details of plot

```
end; --display_details

set_history(index:INTEGER, value : STRING)is
```

– assign values to history variables

```
end; --set_history

last_section:SECTION is
```

– return the most recently allocated section

```
end; --last_section


define_section(aname:STRING;len,wid,coord1,coord2: INTEGER) is
```

```
– create a new section and allocate it to sections if valid
    – output error message if cannot be allocated

end; -- define_section

current_state : STRING is

–return first item in the history variable

end; --current_state

end --class GROWINGAREA
```

E.6.2 Class Date

```
class DATE export

weekno, current_week, assign_week_number,
add_date, display
feature
weekNo : INT;

current_week : DATE is

– return a date

end; --current_week

assign_week_number(v : INTEGER) is

– convert an integer to a date

end; --assign_week_number

display is

    – display a date

end; -- display

end --date
```

Appendix F

Association library classes

This appendix describes two library classes. These classes provide one-to-many and many-to-many associations between two objects. The interfaces of these classes are as similar as possible to the interface of class **One-to-one**. The only difference between the interfaces is in the *find-object* methods. These changes are required because of the cardinality of the associations. Specifically, class **One-to-one** provides *find-object1* and *find-object2* methods, class **One-to-many** provides *find-object1* and *find-objects2* methods and class **Many-to-many** provides *find-objects1* and *find-objects2* methods. This similarity of interface simplifies the use ofassociation classes in the development of a system while the differences serve to emphasise the cardinality of the association.

The many-to-many association is developed as a class containing two one-to-many associations. The specification of class **One-to-many** includes the extra features and interface methods required to develop class **Many-to-many**. Section F.1 gives the specification and discusses the development of a one-to-many association. Class **Many-to-many** is specified and described in section F.2. The code used to implement both library classes is given in section F.3.

F.1 Class One-to-many

The class used to implement this association is designed in the same way as the class **One-to-one** described in chapter 4. This class implements a general one-to-many relationship. It is intended to be used whenever the exact cardinality of the required relationship is not known or is undefined.

F.1.1 Class specification

```
Class Name:       One_to_many
Class Interface:  One_to_many[F,G]
Description:      Generic class to produce one to many associations between two
                  objects which are derived from Social.
                  Forms a two way link.
Super classes:    Assoc
Features:
        Private Attributes
                  object1 : F;
                  objects2 : LINKED_LIST[G];
        Public Methods
                  associate(object1 : F; object2 :G);
                          makes an association between the two objects
                          if consistent with a one to many association.
                  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_objects2(object1 : F) : LINKED_LIST[G];
                          returns a list of the second generic parameter,
                          VOID if no association exists.
        Features available to other instances of same class
                  make_new_assoc(objects : List of objects);
                  y;
```

```
                add_assoc(object:G);
                break_assoc(objects : List of objects);
                getobject1;
                getobjects2;
                make_list;
                find_assoc1;
                find_assoc2;
                add_to_y;

        Features available to class Many_to_many
                make_onenew_assoc(objects : List of objects);
                add_to_y(object:G);
                find_assoc1;
                make_list(objects : List of objects);
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–if valid,
                call the make_new_assoc feature on the new association passing
                the list (l) as the parameter.
                if not valid return an error message
        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_objects2(object1 : F) : LINKED_LIST[G];
                use the access_Association feature from class Social
                to obtain the required association if it exists.
                If the association exists, return the list of object2
                else return VOID.
        make_new_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 added to objects2,
                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 object2's collection of associations,
                if last object2 removed delete the association from object1
        find_assoc1(object1 :A) :ONE_TO_MANY[A,B];
                returns an instance of the same class as association which has object1
                as the instance of its first actual generic parameter. Returns void if no
                instance of the association exists.
        find_assoc2(object2 :B) :ONE_TO_MANY[A,B];
                returns an instance of the same class of association which has object2
                as the instance of its second actual generic parameter. Returns void if no
                - instance of the association exists.
        add_assoc(object2 : B);
                add object 2 to end of list of object 2s
                add association to object2
        add_to_y(object2 : B)
                add a new object to feature y
        make_oneway_assoc(object1:A,object2:B);
                create a new association and attach it to object1
```

F.1.2 Design issues

In order to implement this class, several decisions had to be taken.

1. Data Structure

   A data structure is required to store the objects at the 'many' end of the association. It was decided that a linked list should be used as this does not involve placing any unnecessary limit on the number of objects which can be associated.

2. Retrieving the objects

   There are alternative ways of retrieving the objects from the 'many' end of the association. For example, it would be possible to find the first, second, third etc. object from the list of instances. This requires the user to be certain of the order in which the objects were added to the list and would require the interface of the class to provide methods to access each individual object in the list. Such access could be provided either by methods such as *find-1st-object2*, *find-2nd-object2* etc. or by one *find-object2* method.

   Methods such as *find-1st-object2* would require a single parameter, an instance of the object at the 'one' end of the association. This corresponds with the parameter list in the *find-object2* method provided by the interface of class **One-to-one**. However, this class, class **One_to_many**, is designed to allow a single object to be associated with an indefinite number of other objects. An indefinite number of access methods would also be required resulting in a large and complex class interface.

   The provision of a single *find-object2* method requires this method to be supplied with a second parameter to provide the numerical index of the object being retrieved. This results in methods from different classes having methods which perform the same function requiring different parameters. Such a situation would make the use of associations more complex.

   It was decided that the simplest and best method of retrieving the 'many' objects involved in this type of association was to provide one method which returned all the 'many' objects in a list structure. The method is *find-objects2* which requires a single parameter, that is an instance of the object at the 'one' end of the association. The parameter list of this method corresponds with the parameter list of the *find-object2* method provided by the interface of class **One-to-one**.

   The use of a *find-objects2* method has two advantages. Firstly, the method emphasises the cardinality of the association, thus ensuring that the programmer is aware of the type association being implemented. Secondly, a small simple class interface is provided.

   The developer is left with the task of extracting the required object from the list. (The same problem would be encountered by developers using the conventional methods of implementing one-to-many associations.)

F.2 Class Many-to-many

This class implements a general many-to-many relationship. It is intended to be used whenever the exact cardinality of the required relationship is not known or is undefined.

The implementation of a many-to-many associaitons requires a clear understanding of the exact meaning of a many-to-many relationship. It is clear from figure F.1 that:

- objects of class A are participating in one-to-many relationships with objects of class B.
- objects of class A are participating in one-to-many relationships with objects of class B.

However, the situation is more complex than that. Normally in a one-to-many relationship each object of class B can be associated with only one object of class A. That this is not the case with the individual parts of a many-to-many relationship can be seen from figure F.1. This class of association is implemented as two separate and distinct one-to-many associations. One association represents each direction.

F.2.1 Class specification

Class Name:	**Many_to_many**
Class Interface:	**Many_to_many[F,G]**
Description:	Generic class to produce one to many associations between two

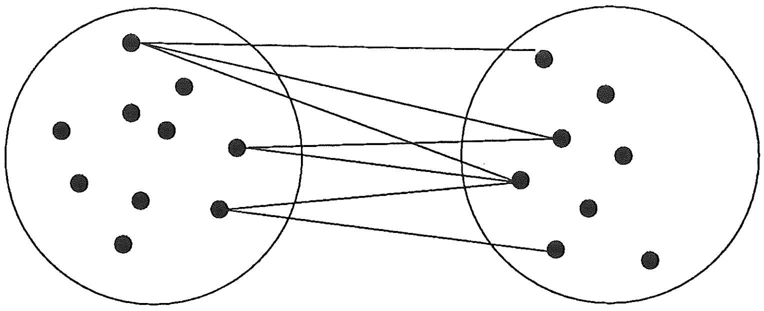

Figure F.1: Many to many relationships

```
                  objects derived from Social.
                  Forms a two way link.
Super classes:    Assoc
Features:
        Private Attributes
                  firstlink : ONE_TO_MANY[F,G];
                  secondlink : ONE_TO_MANY[G,F];
        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_objects1(object2 :G) :LINKED_LIST[F];
                          returns a list of the first generic parameter,
                          VOID if no association exists.
                  find_objects2(object1 : F) : LINKED_LIST[G];
                          returns a list of the second generic parameter,
                          VOID if no association exists.
        Method descriptions
                  associate(object1 : F; object2 :G);
                          if required make new association(s) an attach to the objects
                          else add objects to existing associations.
                          Use make_one_way method and add_assoc method from class One_to_many.
                  disassociate(object1 : F; object2 :G);
                          use the disassociate method from class One_to_many.
                  find_object1s(object2 :G) : F;
                          use the find_objects2 method on the secondlink.
                  find_objects2(object1 : F) : LINKED_LIST[G];
                          use the find_objects2 method on the firstlink.
```

F.2.2 Design issues

This class uses the class **One-to-many** for its implementation. Class **Many-to-many** declares two one-to-many associations. It is not possible for class **Many-to-many** to manipulate these one-to-many associations using only the public features of class **One-to-many**. This is because the *associate* method of class **One-to-many** ensures that the objects at the 'many' end of the association are associated with only one of the objects at the 'one' end of the association. This is clearly violates the requirements of a many-to-many relationship. The *associate* method of class **Many-to-many** needs to bypass the checks.

The checks can be bypassed by using the *make_new_assoc* feature of class **One-to-many** which creates a new one-to-many association. However, using this feature would result in each association being attached to both objects even though access is only required in one direction when used as part of a many-to-many association. Redundant information would be stored. Using the *make_new_assoc* could also mean that an object has a very long list of associations, several of which would be the same type. In order to prevent both the storage of redundant information and the long list of associations, a new method was added to the class **One-to-many**. This method is called *make_oneway_association* which, as its name suggests, implements a one way association. The method is exported to class **Many-to-many**. Implementing the many-to-many associations as described above ensures that the list of associations for each object is kept to a minimum.

It was also necessary to allow class **Many-to-many** to access some other features of class **One-to-many**. The *find_assoc1* method was required to provide access to the associations of each object. The *add_to_y* method was required to add new objects to existing associations. The *make_list* method was required to create the parameter for the *make_oneway_association* method.

F.3 Code

This section contains the code used to provide one-to-many and many-to-many associations.

F.3.1 Class One-to-many

```
class ONE_TO_MANY[a -> SOCIAL, b -> SOCIAL]


export

find_object1, find_objects2, associate, disassociate,

make_new_assoc {ONE_TO_MANY}, get_object1{ONE_TO_MANY},
 get_objects2{ONE_TO_MANY}, make_list{ONE_TO_MANY,MANY_TO_MANY},
find_assoc1{ONE_TO_MANY,MANY_TO_MANY},Find_assoc2{ONE_TO_MANY},
break_assoc {ONE_TO_MANY}, add_assoc {ONE_TO_MANY},
y{ONE_TO_MANY},add_to_y{ONE_TO_MANY,MANY_TO_MANY},
make_oneway_assoc{MANY_TO_MANY}

inherit
        ASSOC

rename create as basecreate
--redefine make_assoc

feature
x : A;
y : LINKED_LIST[B];
done :BOOLEAN;


create (declared_type : STRING) is
local
        type :STRING;

do
        type := declared_type;
        basecreate(type);
end;--create




associate(object1 :  A, object2 : B)

-- makes an association between the two named objects.
-- The type of the calling
-- association  is the type of association produced.

is

local
        o1 : A;
        o2 : B;
        link1, link2 : ONE_TO_MANY[A,B];
        assoclist : LINKED_LIST[SOCIAL];

do

        o1 := object1;
        o2 := object2;

        link1 := current.find_assoc1(o1);
        link2:= current.find_assoc2(o2);
        if link1.void and link2.void
         -- no association of this type exists for object1 or object2
        --so create a new one and attach to both objects.

        then
                link1:= current.deep_clone;
                assoclist.create;
                assoclist := link1.make_list(o1, o2);
                link1.make_new_assoc(assoclist);


        elsif (not link1.void) and link2.void then
                --association of this type already exists for o1 but not o2
                --so add o2 to existing list
                link1.add_assoc(o2);

        elsif not link1.void and not link2.void then
                -- both objects have this type of association

                if link1 = link2 -- links are same
                then
                io.putstring(" association already exists");
                else  -- links not the same
                     io.putstring(o2.generator);
                     io.putstring(" already associated with a ");
                     io.putstring(o1.generator);
                     io.new_line;
                end;--if
        elsif link1.void and not link2.void then
         -- object 1 not involved in assocn but o2 is -illegal in 1:M
                io.putstring(o1.generator);
                io.putstring(" not in an association but ");
                io.putstring(o2.generator);
                io.putstring(" already associated with a ");
                io.putstring(o1.generator);
                io.new_line;
        end; --if

end; --associate

disassociate(object1 :  A, object2 : B) is
--breaks the association between two objects


local
        o1 : A;
        o2 : B;
        link : ONE_TO_MANY[A,B];
        assoclist : LINKED_LIST[SOCIAL];

do
        o1 := object1;  -- assign values to lacal vars.
        o2 := object2;
        link:= current.deep_clone;
        assoclist.create;
        assoclist := link.make_list(o1, o2);
        link.break_assoc(assoclist);
```

```
end; --disassociate


find_object1( object2 : B): A is
-- returns an instance of the first actual generic parameter.
-- Void if no association exists.
local
        p : B;
        as1 : ONE_TO_MANY[A,B];
        r : A;
do
        p := object2;
        as1 := find_assoc2(p);   --retrieve required assoc from object
        if not as1.void then
                Result := as1.get_object1;
        else
                Result := r;
        end; --if
end; --findobject1




find_objects2( object1 : A): LINKED_LIST[B] is

-- returns a list of instances of the second actual generic parameter.
-- Void if no association exists.

local
        p : A;
        as1 : ONE_TO_MANY[A,B];
        r : LINKED_LIST[B];
do
        p := object1;
        as1 := find_assoc1(p);    --retrieve required assoc from object
        if not as1.void then
                Result := as1.get_objects2;
        else
                Result := r;
        end; --if

end; --findobjects2



-----------------------------------------------------------------
-- private features


make_list(obj1 : a, obj2 : b) : LINKED_LIST[SOCIAL]

-- this feature must be called before make_assoc.the first named object must
--conform to the first actual generic parameter. The second parameter must
-- conform to the second actual generic parameter.

is
local
list : LINKED_LIST[SOCIAL]

do

list.Create;
x.Create;
y.Create;
list.put_right(obj2);
list.put_right(obj1);
done := TRUE;
Result := list;

end; --make_list


make_new_assoc(objects: LINKED_LIST[SOCIAL])

is
do
        x := objects.i_th(1);
--      y.finish;
        y.add_right(objects.i_th(2));

        x.addAssociation(current);
        objects.i_th(2).addAssociation(current);


end;--make_new_assoc

make_oneway_assoc(object1:A,object2:B) is

do
        y.create;
        x := object1;
        y.add_right(object2);
        x.addAssociation(current);
end;--make_oneway_assoc


add_assoc(object2 : B) is

do -- add object 2 to end of list of object 2s
       add_to_y(object2);
        -- add association to object2
        object2.addAssociation(current);
end;-- add_assoc

add_to_y(object2 : B) is
do
        y.finish;
        y.add_right(object2);
end;--add_to_y
break_assoc(objects: LINKED_LIST[SOCIAL])
--removes the association from the list of associations of objects as needed

is
local
        p : B;
        as1 : ONE_TO_MANY[A,B];
        r : A;
do
        r := objects.i_th(1);
        p := objects.i_th(2);

        as1 := find_assoc2(p);--check that association exists
        if not as1.void then
        --remove assoc from p and remove p from list
                p.deleteAssociation(as1);
                as1.y.start;
                as1.y.search_equal(p);
                as1.y.remove;
                        if as1.y.count =0 -- list of associations is empty
                        then
                        r.deleteAssociation(as1);
                        end; --if
        end;--if
end;--break_assoc

find_assoc1(object1 :A) :ONE_TO_MANY[A,B]
-- returns an instance of the same class as association which has object1
-- as the instance of its first actual generic parameter. Returns void if no
-- instance of the association exists.

is
local
        assoc : ONE_TO_MANY[a,b];
        p : A;
        c : ASSOC;
do
        p:= object1;
        c:= p.accessassociation(current);
        assoc ?=c;
        Result := assoc;

end; --find_assoc1

find_assoc2(object2 :B) :ONE_TO_MANY[A,B]
-- returns an instance of the same class of association which has object2
-- as the instance of its second actual generic parameter. Returns void if no
-- instance of the association exists.

is
local
        assoc : ONE_TO_MANY[a,b];
        p : B;
        c : ASSOC;
do
        p:= object2;
        c:= p.accessassociation(current);
        assoc ?=c;
        Result := assoc;
end; -- find_assoc2

get_object1 :  a
is
do
        result:=  x
end; --get_object1


get_objects2 :  LINKED_LIST[B]
is
do
        result := y
end; --get_objects2




end -- ONE-TO-MANY
```

F.3.2 Class Many-to-many

```
class MANY_TO_MANY[a -> SOCIAL, b -> SOCIAL]


export

find_objects1, find_objects2, associate, disassociate,

inherit
        ASSOC
```

```
rename create as basecreate
--redefine make_assoc

feature
Istlink : ONE_TO_MANY[A,B];
sndlink : ONE_TO_MANY[B,A];


create (declared_type : STRING) is
local
        type,type1,type2 :STRING;

do
        type := declared_type;
        basecreate(type);
        type1 :=type;
        type1.append("1");
        Istlink.create(type1); --assign type to required 1:M links
        type2 :=type;
        type2.append("2");
        sndlink.create(type2);
end;--create


associate(object1 :  A, object2 : B)

-- makes an association between the two named objects.
-- The class of the calling
-- association  is the class of association produced.

is

local
        o1 : A;
        o2 : B;
        link1 : ONE_TO_MANY[A,B];
        link2 :  ONE_TO_MANY[B,A];
        list : LINKED_LIST[SOCIAL];
do
        o1 := object1;  ---assign parameters to local vars
        o2 := object2;
        list.create;

        -- retrieve associations from o1 and o2
        link1 := current.Istlink.find_assoc1(o1);
        link2 := current.sndlink.find_assoc1(o2);

        if link1.void and link2.void -- no association of this type exists
                                --for object1 or object2
        then

        --make associations between the objects
                link1 := Istlink.deep_clone;
                link1.make_oneway_assoc(o1,o2);
                link2 := sndlink.deep_clone;
                link2.make_oneway_assoc(o2,o1);

        elsif (not link1.void) and link2.void
         --association of this type already exists for object1, add o2 to its
        -- list and make new 1:M and attach it to o2
        then
                link1.add_to_y(o2);
                list:= sndlink.make_list(o2,o1);
                link2 := sndlink.deep_clone;
                link2.make_oneway_assoc(o2,o1);
        elsif link1.void and not link2.void
        -- association of this type already exists for object2, add o1 to its
        -- list and make new 1:M and attach it to o1
                then


                link2.add_to_y(o1);
                list:= Istlink.make_list(o1,o2);
                link1 := Istlink.deep_clone;
                link1.make_oneway_assoc(o1,o2);



        elsif not link1.void and not link2.void then
        -- associations exist for both objects
                      link1.add_to_y(o2);
                      link2.add_to_y(o1);

else

        end; --if


end; --associate

disassociate(object1 :  A, object2 : B) is
--breaks the association between two objects


local
        o1 : A;
        o2 : B;
do
        o1 := object1;
        o2 := object2;
--break link from each end
        Istlink.disassociate(o1,o2);
        sndlink.disassociate(o2,o1);

end; --disassociate


find_objects1( object2 : B): LINKED_LIST[A] is
-- returns an instance of the first actual generic parameter.
-- Void if no association exists.
local
        p : B;
        as1 : ONE_TO_MANY[B,A];
        r :LINKED_LIST[A];
do
        p := object2;
        Result := sndlink.find_objects2(p);

end; --findobject1




find_objects2( object1 : A): LINKED_LIST[B] is

-- returns an instance of the second actual generic parameter.
-- Void if no association exists.

local
        p : A;
        as1 : ONE_TO_MANY[A,B];
        r : LINKED_LIST[B];
do
        p := object1;

                Result := Istlink.find_objects2(p);
end; --findobjects2
end -- MANY-TO-MANY
```

Appendix G

An investigation into the type checking of generic types in ISE Eiffel v2.3

This appendix describes the results of an investigation carried out into the dynamic type checking features provided by Eiffel. Version 2.3 was used. The investigation arose from problems encountered when developing a system which required the retrieval of different generic type objects from a collection of base type objects. It was found that the Eiffel dynamic type checking features did not distinguish between some different generic types. As a consequence of this, a tag field had to be added to the generic types so that this could be used to retrieve the required type of object from the data structure.

The tests were carried out using linked lists and arrays from the Eiffel library. Static tests were carried out using the ISE compiler. In order to anticipate the results of the tests, the Eiffel [2, 51] definitions of the terms class, type and object are explained in section G.1. The features used to ascertain the dynamic type of objects are explained in section G.2. Section G.3 describes the tests carried out. This section includes the expected and actual results for each test. The information in sections G.1 and G.2 is used to determine the expected results. The final section discusses the results and draws conclusions about the use of the dynamic type checking features.

G.1 Eiffel terminology

This section explains the Eiffel use of the terms class, type and object. Briefly the definitions are:

class the textual description of a type

type describes the structure and capabilities of objects which can be created.

object an instance of both a class and a type

The Eiffel interpretation of classes and types binds the two concepts together because the type includes information about structure as well as capabilities. The type of an object is usually determined by the class from which it is generated.

A generic class is declared with formal generic parameters and defines a template for a set of types not one specific type. Each member of the set of types is formed by providing actual generic parameters for each of the formal generic parameters in the class definition. The types produced are called generically derived types.

The class definition used to produce a generically derived type is called its base class. One generic base class can instantiate many generically derived types. Each generically derived type has a different type and is its own base type. Each generically derived type shares the same general structure and functionality as other types derived from the same generic base class but differs in the details of both.

These definitions mean that for an object produced by a non-generic class, the type and generating class are the same. However, instantiating generic classes produces a set of objects with different types but with the same base class.

G.2 Features used in the investigation

Two features which can be used to distinguish between different dynamic types were investigated. They are:

1. the *reverse assignment attempt*, ?=,

2. the *conforms_to* function from class **ANY**,

Each of these functions has a different purpose. The *reverse assignment attempt* causes an assignment to be made if and only if the dynamic type of the target of the assignment conforms to the dynamic type of the source. The syntax of the *reverse assignment attempt* is

```
        customer ?= people_list.get(i);
```

where `customer` is of class CUSTOMER, `people_list` is a list containing variables conforming to class PERSON, `get(i)` is the feature which retrieves the i-th element from the list.

If the dynamic type of the i-th element is the same as, or conforms to, the static type of the customer variable, the assignment is made. If not, the value of the variable is Void. After a successful assignment, the features of the subclass can be accessed. This means that any features declared by the class from which the variable customer was generated can be called.

The *conforms_to* function is declared by class **ANY** which is the ultimate base class of all Eiffel classes. Features from class **ANY** are made available to all descendants so the *conforms_to* feature can be used to test for conformance between two objects. The *conforms_to* feature is defined as supplying the answer to the following question:

Is dynamic type of current object a descendant of the dynamic type of other?

The term descendant can also mean the type itself. Thus an object of class **PERSON** conforms to another object of class **PERSON**.

A call such as `p.conforms_to(c)` , where p is the *current object* and c is *other*, returning true does not permit access to any extra features present in c through variable p.

Another feature provided by the Eiffel language was used in the tests. The feature *generator* was used to ascertain the generating class of the variables.

G.3 Tests carried out

A small system was implemented. The root class contained the following variables:

- a list of people containing elements of type PERSON,
- a list of accounts containing elements of type ACCOUNT,
- a list of anything containing elements of type ANY,
- an array containing elements of type PERSON,
- a variable of each type of element.

Elements were added to the list of people and list of accounts and several tests carried out. The tests were carried out in two groups. The first group consisted of three static tests using the compiler. The second group were five dynamic tests which used the features explained in section G.2. The code used is given in section G.5.

The Eiffel definitions of types and classes given above were used to determine the expected results of the test. These definitions indicate that each instantiation of a generic class has a different type but the same base class. Each instantiation is its own base type. The type and class of objects of generic types are not the same. For non-generic classes the type and class of an object are the same.

The results of the tests are presented in two groups. The results of the static tests are presented first.

Static tests

The type checking system is expected to ensure that incorrect types are not assigned. The compiler which performs static type checking should reject attempts to assign incompatible types.

- **Test 1** Assign people list to accounts list.

  Expected results

  The compiler is expected to reject attempts to assign an instance of one type to an instance of a different type even if they have the same base class. This code should not compile.

  Actual results

  The code did not compile.

- **Test 2** Reverse assign the people list to the accounts list.

  Expected results

  The compiler is expected to reject attempts to use the *reverse assignment attempt* to assign variables from different branches of the type hierarchy. This code should not compile.

  Actual results

  The code did not compile.

- **Test 3** Test the people list for conformance with the accounts list.

  Expected results

  The compiler is expected to reject attempts to use the *conforms_to* feature to test for conformance between variables from different branches of the generic type hierarchy. This code should not compile.

  Actual results

  The code did not compile.

Dynamic tests

The accounts list was assigned to the list of anything, so the list of anything now contains the accounts list. These tests are designed to ascertain whether the dynamic type checking system prevents runtime type errors when generic types are used.

- **Test 4** Test the list of anything, now containing the accounts list, for conformance with the people list.

  Expected results

  These two objects are different instantiations of the generic class LINKED_LIST. According to the definitions, each is a different type and is its own base type. The *conforms_to* feature should return False.

  Actual results

  Returned True

- **Test 5** Test the list of people for conformance to the array of people.

  Expected results

  These two objects are instantiations of different generic classes. The *conforms_to* feature should return False.

  Actual results

  Returned False

- **Test 6** Reverse assign the list of anything, containing the accounts list to the people list.

  Expected results

  These objects have different dynamic types, the *reverse assignment attempt* should not assign objects of one type to objects of a different type. The assignment should not be successful.

  Actual results

  This feature call was successful. The list of accounts was assigned to the list of people. The people list now contained elements of type ACCOUNT. This means that a variable of one generically derived type was assigned to a variable of another type with the same base class.

- **Test 7** After successfully performing the above assignment, assign elements from the people list, now a list with elements of type ACCOUNT, to objects of type PERSON.

  Expected results

  The system should refuse to assign ACCOUNT elements from the list to objects of type PERSON.

  Actual results

  The elements in the list were assigned to objects of type PERSON even though they were actually ACCOUNT objects.

  The system considered these ACCOUNT objects to be instances of the type PERSON. Attempts to access features of the objects resulted in either a segmentation fault or spurious values being reported.

- **Test 8** After the above assignments were made, the generating class of each variable was determined.

  Expected results

  1. All the linked list variables should be generated by class Linked List.
  2. The array of People should be generated by class Array.
  3. The elements should be generated by the class of their declared type.

  Actual results

  This showed that all the variables with the same generic base class were generated by the same class as expected.

  This test also showed that after **Test 7** a person variable, declared as p : PERSON, extracted from the person list was generated by class ACCOUNT.

G.4 Discussion and Conclusion

The results show that when objects are generated from a generic class where a set of types are derived from one class, the dynamic type checking does not give the expected results although the static type checking does.

The above results indicate that different instantiations of one generic class have different static types but the same dynamic type. This implies that the Eiffel type system uses one set of information for static type checking and another for dynamic type checking. Consequently it is possible to make assignments at runtime which are rejected as type errors if attempted statically.

The results of the tests using *generator* indicate that the behaviour of the *reverse assignment attempt* and *conforms_to* may to be allied to the class from which the object is generated.

The conclusion drawn from the above results is that the dynamic type checking mechanisms may be testing the generating class of an object rather than the actual type of an object. Thus performing dynamic class checking rather than dynamic type checking. In the case of non generic types the result is the same because the generating class defines one type only. However, in the case of generic types one class can define many types and the distinction between dynamic type checking and dynamic class checking becomes significant. Dynamic class checking means that it is possible to assign a variable of one generically derived type to a variable of a different generically derived type. The result is that it is possible for an Eiffel program to abort with a run time type error or to use spurious values for variables.

These test results suggests that Eiffel dynamic type checking features should only be used with non generic types. The dynamic type of generically derived objects requires the addition of tag fields which can be used to ascertain the dynamic type.

G.5 Code of root class for generic type conformance tests

```
class CONFCODE


feature
 Create is
local
```

```
 x : ANY;
 p :PERSON;
 a :ACCOUNT;

 all : LINKED_LIST[ANY];
 people :LINKED_LIST [PERSON];
 accounts : LINKED_LIST[ACCOUNT];


 apeople : ARRAY[PERSON];
 w,y :ANY;



do

-- create all the required lists.

 people.Create;
 accounts.Create;


-- add entries to people,accounts.
 p.Create;
 p.assignName("Audrey");
 p.assignAddress("hut 220");
 people.put_right(p);
 p.forget;

 p.Create;
 p.assignName("Mary"''B220'');
 p.assignAddress("B220");
 people.put_right(p);
 p.forget;
    a.Create;
 a.open(1);
 a.addFunds(1.00);
 accounts.put_right(a);
 a.forget;

 a.Create;
 a.open(2);
 a.addFunds(1.00);
 accounts.put_right(a);

-------------------------------------------------------------


--print out the lists

io.putstring("people list");
io.new_line;
 p:=people.i_th(1);
 p.print;
io.new_line;
 p:=people.i_th(2);
 p.print;
 io.new_line;
io.new_line;
io.putstring("accounts list");
io.new_line;

 a := accounts.i_th(1);
 a.print;
io.new_line;
 a := accounts.i_th(2);
 a.print;
```

```
io.new_line;



-------------------------------------------------------
-- check the dynamic type checking features
-------------------------------------------------------
-------------------------------------------------------
-- tests with generic lists
io.putstring(" TESTING GENERIC LISTS");
io.new_line;
--*****************************************************
--Test 1

--accounts:= people;

--Type mismatch: PERSON is not a descendant class of ACCOUNT
--Type mismatch: LINKED_LIST[PERSON] is not a descendant class of
--LINKED_LIST[ACCOUNT]
--*********************************************************
--Test 2
--accounts ?= people;
-- Type mismatch: ACCOUNT is not a descendant class of PERSON
-- Type mismatch: LINKED_LIST[ACCOUNT] is not a descendant class of
-- LINKED_LIST[PERSON]
-- Type mismatch: LINKED_LIST[ACCOUNT] does not conform to LINKED_L
--IST[PERSON]

--*********************************************************
--Test 3

--if accounts.conforms_to(people)
--then io.putstring ("people and accounts conform");
--else io.putstring(" no conformance");
--end ;--if
-- COMPILATION ERROR
--"test", Type mismatch: PERSON is not a descendant class of ACCOUNT
--"test", Type mismatch: LINKED_LIST[PERSON] is not
--a descendant class of LINKED_LIST[ACCOUNT]
--"test", Incorrect type for argument number 1 of conforms_to

--*********************************************************
--**********************************************************
io.putstring("-------------------------------------------------------");

io.new_line;
io.putstring("ASSIGNING  ACCOUNTS TO ALL.");io.new_line;

 all := accounts;

io.putstring(" all now contains;");
io.new_line;
io.new_line;
        x:=all.i_th(1);
        x.print;
io.new_line;
        x:=all.i_th(2);
        x.print;
        io.new_line;
--**********************************************************

--Test 4
io.new_line;io.new_line;io.new_line;
io.putstring("-------------------------------------------------------");
io.new_line;
io.putstring("TESTING all.conforms_to(people)");io.new_line;
if
```

```
all.conforms_to(people)
then

io.putstring("accounts conforms to people");io.new_line;
else
io.putstring("accounts does not conform to people");
end; --if

--**************************************************************

----------------------------------------------------------------
-- Test 5 testing compatibilty of arrays with linked lists
---------------------------------------------------------------
io.new_line;io.new_line;io.new_line;
io.putstring("-----------------------------------------------------");
io.new_line;
io.putstring("TEST 5");io.new_line;
apeople.create(1,10);
apeople.put(people.i_th(1),2);
apeople.put(people.i_th(2),1);
io.putstring("elements of people list
                           assigned to apeople - printing apeople");
io.new_line;
apeople.print;
p:= apeople.item(1);
p.print;
io.new_line;
p:= apeople.item(2);
p.print;
io.new_line;
--if apeople.conforms_to(people)
--then
--io.putstring(" apeople conforms to people");
--end; --if
--DOES NOT COMPILE BECAUSE LINKED_LIST
--[PERSON] NOT DESCENDANT OF ARRAY[PERSON].



y:= apeople;
io.putstring("variable y: ANY now contains apeople");io.new_line;
io.putstring("TESTING y.conforms_to(people)");io.new_line;
if y.conforms_to(people)
then
io.putstring(" apeople conforms to people");
else
io.putstring(" apeople does not conform to people");
end; --if
io.new_line;
io.new_line;
io.new_line;

w:= people;

y:= apeople;
io.putstring("variable y:ANY contains apeople");io.new_line;
io.putstring("variable w:ANY now contains people");io.new_line;
io.putstring("TESTING y.conforms_to(w)");io.new_line;
if y.conforms_to(w)
then
io.putstring(" apeople conforms to people");
else
io.putstring(" apeople does not conform to people");
end; --if
io.new_line;
```

```
--************************************************************
--Test  6

io.new_line;io.new_line;io.new_line;
io.putstring("-----------------------------------------------------");
io.new_line;
io.putstring("TEST 6");io.new_line;

io.putstring("TESTING people ?= all");
io.new_line;
people ?= all;

io.new_line;
--*******************************************************************
--Test  7

io.new_line;io.new_line;io.new_line;
io.putstring("-----------------------------------------------------");
io.new_line;
io.putstring("TEST 7");io.new_line;

io.putstring("people list after reverse assignment attempt ");
io.new_line;
 p := people.i_th(1);
 p.print;
io.new_line;
 p := people.i_th(2);
 p.print;
io.new_line;

io.new_line;
io.putstring("accessing the features of person objects");
io.new_line;
io.putstring("people.i_th(2) telNo ");

io.new_line;
 io.putint(p.telNo);
io.new_line;io.new_line;
 io.putstring("calling p.assignName(Tom)");

io.new_line;
 p.assignName("Tom");
 io.putstring("calling p.print");io.new_line;
p.print;
io.new_line;
io.new_line;
io.putstring("people.i_th(2) telNo ");

io.new_line;
       io.putint(p.telNo);
-----------------------------------------------------------------

--********************************************************************
--Test 8
io.new_line;io.new_line;io.new_line;
io.putstring("-----------------------------------------------------");
io.new_line;
io.putstring("TEST 8");io.new_line;

io.putstring("Name of current objects generating class i.e.the type of");
io.new_line;
io.putstring(" which it is a direct instance");io.new_line;

io.putstring("people :LINKED_LIST [PERSON]; --  ");
io.putstring(people.generator);io.new_line;
io.putstring(" accounts : LINKED_LIST[ACCOUNT]; --  ");
io.putstring(accounts.generator);io.new_line;

io.putstring("apeople : ARRAY[PERSON]; --  ");
io.putstring(apeople.generator);io.new_line;
io.putstring("all : LINKED_LIST[ANY]; --  ");
io.putstring(all.generator);io.new_line;
io.putstring("a :ACCOUNT; --  ");io.putstring(a.generator);
io.new_line;
io.putstring(" p :PERSON; --  ");io.putstring(p.generator);
io.new_line;


end; --create
end -- confcode
```

G.6 Results of generic type conformance tests

```
people list
"Mary"
"B220"
0

"Audrey"
"hut 220"
0


accounts list
2
1.000000

1
1.000000

 TESTING GENERIC LISTS
-------------------------------------------------------
ASSIGNING  ACCOUNTS TO ALL.
 all now contains;

2
1.000000

1
1.000000

-------------------------------------------------------
TESTING all.conforms_to(people)
accounts conforms to people

-------------------------------------------------------
TEST 5
elements of people list assigned to apeople - printing apeople

1
10
"Audrey"
"hut 220"
0

"Mary"
"B220"
0

variable y: ANY now contains apeople
```

```
TESTING y.conforms_to(people)
 apeople does not conform to people


variable y:ANY contains apeople
variable w:ANY now contains people
TESTING y.conforms_to(w)
 apeople does not conform to people

-----------------------------------------------------
TEST 6
TESTING people ?= all

-----------------------------------------------------
TEST 7
people list after reverse assignment attempt
2
1.000000

1
1.000000


accessing the features of person objects
people.i_th(2) telNo
16777231

calling p.assignName(Tom)
calling p.print
559096
0.000000


people.i_th(2) telNo
16777231

-----------------------------------------------------
TEST 8
Name of current objects generating class i.e.the type of
 which it is a direct instance
people :LINKED_LIST [PERSON]; --  LINKED_LIST
 accounts : LINKED_LIST[ACCOUNT]; --  LINKED_LIST
apeople : ARRAY[PERSON]; --  ARRAY
all : LINKED_LIST[ANY]; --  LINKED_LIST
a :ACCOUNT; --  ACCOUNT
p :PERSON; --  ACCOUNT
```

Appendix H

Published paper

The following paper is published in Microprocessing and Microprogramming 40 (1994) 811-814

ELSEVIER

Microprocessing and Microprogramming 40 (1994) 811–814

Microprocessing and Microprogramming

Implementing Associations between Objects

Audrey Mayes, Bob Dickerson and Carol Britton
School of Information Sciences, University of Hertfordshire, College Lane,Hatfield, Herts AL10 9AB, UK
Tel 0707 284763 Fax 0707 284303
email comrjam@herts.ac.uk or comqcb@herts.ac.uk

1. Introduction

This paper presents an alternative design method for the implementation of *conceptual associations* identified during the analysis of a problem.

Object-oriented development methods such as OMT [1], identify associations between objects. Some of these associations represent aggregations of objects such as a wheel is part of a car. Other associations represent conceptual links between objects such as a person has an account. These types of association are shown in figure 1.

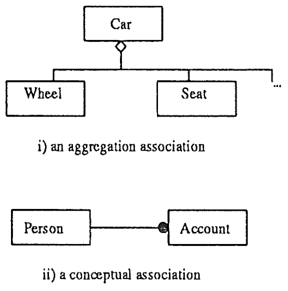

Figure 1. Associations between objects

These two types of association clearly represent different concepts. However, both are usually implemented in object-oriented programming languages by using the client-server relationship. The class representing an aggregation declares an instance of the classes which represent its parts. A class involved in a conceptual association declares an instance of the classes with which it is associated. The addition of an association usually requires the declaration of new subclasses of the classes involved. The result of using the client-server relationship to add conceptual associations is that the implemented objects become less like the objects identified during analysis and are bound together in the same way as aggregations. The effects are:

- the classes used to define the objects are application specific and therefore less reusable [2].
- the structure of the system is difficult to understand.

It is agreed by Kilian [2] and Rumbaugh [3] that the provision of associations as separate constructs in object oriented systems would increase reusability and improve the clarity of system design.

The design method presented here allows conceptual associations from the analysis model to be implemented directly by using Sociable Classes. These Sociable Classes have the ability to participate in associations without the addition of attributes.Conceptual associations are provided as instances of generic classes. New associations can be added as required by introducing a new instantiation of the required type of association to the system. It is not necessary to define new subclasses of the participating objects. The conceptual associations retain the object-oriented structure by storing the associations with the objects. Relational tables of associations are not added to the system.The design technique overcomes some of the problems mentioned above.

2. Basis of the design

In order to differentiate between conceptual associations and aggregations, a mechanism must be provided to allow different degrees of binding between objects. The different degrees of binding would result in:

- groups of objects which are tightly bound because they represent aggregations, such as a car.
- objects which are loosely coupled to other objects because they take part in conceptual associations, such as a person has an account.

In this suggested design, the client-server relationship is used to implement aggregations. The loose coupling between objects is produced by adding conceptual associations between objects. For example, an association is added between a person and an account to implement the association 'a person has a bank account'. It is not necessary for all objects of a class to be involved in all types of association.

In order for a design to provide loosely coupled objects, the following facilities should be available:

- a means of explicitly implementing associations between 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.

The next section describes the classes used in the Sociable Class design technique which attempts to meet the above criteria.

3. Sociable Classes and related constructs.

Sociable Classes define objects which have the ability to take part in a potentially unlimited number of different associations. The design technique using Sociable Classes requires the declaration of two abstract base classes, **Social** and **Assoc**. Sociable Classes are subclasses of **Social**. The associations between objects are formed by instances of subclasses of class **Assoc**.

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 not separate entities stored in a data structure. 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.

One instance of each type of association required in the system is declared and used to create, access and delete all other instances of that type of association.

3.1. 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.

The features to access the private attribute are not made publicly available. 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. All Sociable Classes are derived from **Social** and do not need to access any of its features.

3.2. 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 *associate* and *disassociate* features are defined by class **Assoc**. These two features represent the minimal functionality that must be provided by all subclasses. 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.

3.3. Sociable Classes

Sociable classes are implemented by declaring the required classes as subclasses of **Social**. The features defined by the analysis model are then added. The following extract of code gives two examples of the definition of Sociable Classes using Eiffel notation [4].

```
class PERSON
export        first_name,...

inherit SOCIAL

feature
        first_name : STRING;

...

end --person

class ACCOUNT

export
        accountNumber,...
inherit SOCIAL

feature
        accountNumber: INTEGER

...

end --account
```

3.4. Associations

The many different types of associations required in systems are provided by subclasses of **Assoc**. Each subclass defines a general type of association, such as a one-to-one bidirectional association. Figure 2 shows part of the association hierarchy. The subclasses of **Assoc**, in the lower level of figure 2, are generic classes which supply

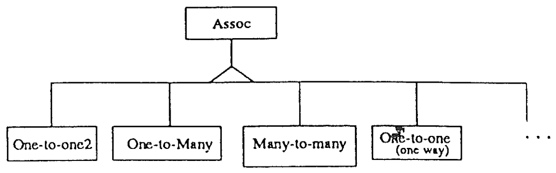

Figure 2. Association class hierarchy

the implementation of features provided by associations of that type.

For example, class **One_to_one2** implements one-to-one bidirectional associations. That is, it implements associations between two objects and allows the association to be traversed via either object. This generic class requires two formal generic parameters, one for each of the objects involved in the association. It provides implementations for the *associate* and *disassociate* features inherited from class **Assoc**. It also defines the features required to access the objects involved in a specific association. The classes used to replace the formal generic parameters must be Sociable Classes. Using Eiffel notation, the interface of this class is **One_to_one2 [A→SOCIAL, B→SOCIAL]**. An association between a person and an account is declared as `has-account : One_to_one2(Person,Account)`.

4. Using Sociable classes

This section gives an example of the use of the Sociable class design technique. The association being implemented is shown in Figure 3. The classes **Person** and **Account** are declared as indicated in Section 3.3. The root or main class is declared as follows:

```
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;
  ...

--create the association variable
  has-account.Create(hasaccount);

--(the parameter is required for
-- reasons beyond the scope
-- of this paper)

-- 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
...

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

5. Conclusion

A system designed and implemented by using the above technique would consist of classes which are recognisable as definitions of the objects identified in the analysis model. The client-server relationship is used to implement the structure and attributes of each class. The classes would not be modified by the addition of extra attributes 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.

Further research is being carried out into the use of other languages for implementation and into the possibility of including the constructs defined in this paper in a language definition. This work forms part of a PhD thesis to be submitted Sept 94.

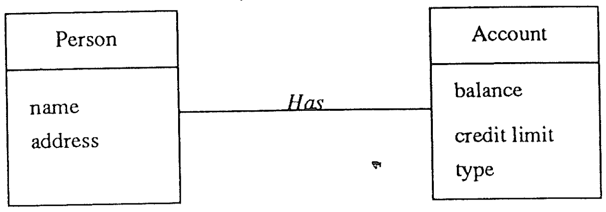

Figure 3. Simple banking application object model

REFERENCES

1. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modelling and Design.* Prentice-Hall International Editions, Englewood Cliffs, New Jersey, 1991.
2. Michael Kilian. A Note on Type Composition and Reusability. *OOPS Messenger*, 2(3), 7 1991.
3. J. Rumbaugh. Relations and semantic constructs in an object-oriented language. *OOPSLA'87*, 1987.
4. B. Meyer. *Object-oriented Software Construction.* Prentice Hall, Hemel Hempstead, 1988.

Bibliography

[1] T. Caper Jones. Reusability in Programming: A Survey of the State of the Art. *IEEE Transactions on Sofware Engineering* , 10(5), September 1984.

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

[3] Wayne C. Lim. Effects of Reuse on Quality, Productivity, and Economics. *IEEE Software*, 11(5), September 1994.

[4] B.J. Cox. Planning the Software Industrial Revolution. *Byte*, 10 1990.

[5] P.A.V. Hall, editor. *Software Reuse and Reverse Engineering in Practice.* Chapman and Hall, London, 1992.

[6] V.R. Basili and H.D. Rombach. Support for comprehensive reuse. *Software Engineering Journal*, 9 1991.

[7] T. Biggerstaff and A. Perlis, editors. *Software Reusability, Vol 1.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[8] William B. Frakes. Success Factors of Systematic Reuse. *IEEE Software*, 11(5), September 1994.

[9] J. A. Mayes. A survey of the current state of reuse in a software environment. TR 146, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1992.

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

[11] I. Sommerville. *Software Engineering.* Addison Wesley, third edition, 1989.

[12] D. A. Lamb. *Software Engineering: planning for change.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[13] P. Naur. Programming as Theory Building. *Microprocessing and Microprogramming*, 15, 1985.

[14] K. S. Rubin. Reuse in Software engineering: An Object-Oriented Perspective. In *COMPCON Spring '90 Thirty-Fifth IEEE Computer Society International Conference*, 1990.

[15] J. A. Mayes and C. Britton. Are there any parallels between object-oriented system development and other branches of engineering? TR 139, Hatfield Polytechnic, College Lane, Hatfield, Herts AL10 9AB, 1992.

[16] J. A. Mayes. A comparison of development methods used in traditional engineering and software engineering. TR 147, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1992.

[17] M. Lenz, H.Schmid, and P. Wolf. Software Reuse through Building Blocks. *IEEE Software*, 29(4), 4 1987.

[18] M. Stovsky and B. Weide. The Role of Traditional Engineering Design Techniques in Software Engineering. In *SEKE Proceedings 2nd International Conference on Software Engineering and Knowledge Engineering*, 1990.

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

[20] Thomas A. Standish. An Essay on Software Reuse. *IEEE Transactions on Software Engineering*, 10(5), September 1984.

[21] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application sofware maintenance. *Commun. Ass. Comput. Mach.*, 21, June 1978.

[22] L. Berlin. When Objects Collide: Experiences with Reusing Multiple Class Heirarchies. In *ECOOP/OOPSLA '90 Conference on Object-Oriented Programming: Systems,Languages, and Applications*, 1990.

[23] D. J. Leech and B.T. Turner. *Engineering Design for Profit.* Ellis Horwood, 1985.

[24] E.Tjalve, M.M. Andreasen, and F. Frackmann Schmidt. *Engineering Graphic Modelling.* Newnes-Butterworths, London, 1979.

[25] R. D'Ippolito and C.Plinta. Software Development Using Models. In *Proceedings 5th International Workshop on Software Specification and Design*, 1989.

[26] Editorial. Scaling up: a research agenda for software engineering. *Communications of the ACM*, 33(3), 1990.

[27] S. Pugh. *Total Design.* Addison-Wesley Publishers Ltd., Wokingham, England, 1990.

[28] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modelling and Design.* Prentice-Hall International Editions, Englewood Cliffs, New Jersey, 1991.

[29] K.J. Leiberherr and I.M. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Software*, September 1989.

[30] J. Nino. Object Oriented Models for Software Reusability. *IEEE Proceedings- 1990 Southeastcon*, 1990.

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

[32] P. Coad and E. Yourdon. *Object-Oriented Design.* Prentice Hall, Inc, Englewood Cliffs, New Jersey, 1991.

[33] Russel Winder. *Developing C++ Software.* John Wiley and Sons Ltd., Chichester, West Sussex., 1991.

[34] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, Reading,Massachusetts, 1990.

[35] Martin Reiser and Nicholas Wirth. *Programming in Oberon.* Addison-Wesley Publishing Company, New York, 1992.

[36] N. Wirth. The Programming Language Oberon. *Software - Practice and Experience*, 18, July 1988.

[37] Samuel P. Harbison. *Modula-3* . Prentice Hall , Englewood Cliffs, New Jersey, 1992.

[38] Brian Henderson-Sellers and Julian M. Edwards. The Object-Oriented Systems Life Cycle. *Communications of the ACM*, 33(8), September 1990.

[39] J. A. Mayes. Experience of using Coad and Yourdon Object-oriented Analysis and Design. TR 148, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1992.

[40] J.A.Mayes. The Responsibility Driven Object-oriented Design Method advocated by Wirfs-Brock, Wilkerson and Weiner. TR 149, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1992.

[41] N. Ingles. Object-Oriented Design Methods. Unpublished BSc Project Report,University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1993.

[42] Marcus Schulz. Object-Oriented Development of an application using C++. Unpublished BSc Project Report, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1994.

[43] I. D. Game. Comparison between Structured Methods and the Rumbaugh Object Modelling Technique. Unpublished MSc Report,University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1994.

[44] Mehmet Aksit and Lodewijk Bergmans. Obstacles in Object-Oriented Software Development. *OOPSLA 92*, 1992.

[45] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. *Sigplan Notices*, 25(10), October 1990.

[46] Audrey Mayes and Mary Buchanan. A comparison of Eiffel, C++ and Oberon-2. Technical Report 191, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1994.

[47] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger* , 1(1), 1990.

[48] Colin Atkinson. *Object Oriented Reuse, Concurrency and Distribution– An Ada based approach.* ACM Press , New York, 1991.

[49] Setrag Khoshafian and Razmik Abnous. *Object Orientation: Concepts, Languages, Databases, User Interfaces.* Wiley, New York, 1990.

[50] H. Mossenbock and N. Wirth. Differences between Oberon and Oberon-2. *Structured Programming*, 4 1991.

[51] B. Meyer. *Eiffel: The Language.* Prentice Hall, Hemel Hempstead, 1992.

[52] Michael Kilian. A Note on Type Composition and Reusability. *OOPS Messenger*, 2(3), 7 1991.

[53] Audrey Mayes and Mary Buchanan. The Oberon-2 Language and Environment. Technical Report 190, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1994.

[54] Laszlo Boszormenyi. A Comparison of Modula-3 and Oberon-2. *Structured Programming*, 14, 1993.

[55] Mary Buchanan. Overloading and Polymorphism in the interpretation of Inheritance in C++. Technical Report 202, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1994.

[56] William R. Cook, Walter L. Hill, and Perter S. Canning. Inheritance Is Not Subtyping. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, January 1990.

[57] N. Wirth. From Modula to Oberon. *Software - Practice and experience*, 7 1988.

[58] Derek Coleman et al. *Object-oriented development: The fusion method.* Prentice Hall Inc, Englewood Cliffs, New Jersey 07632, 1994.

[59] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles, Modelling the World in States.* Yourdon Press, Englewood Cliffs, New Jersey, 1992.

[60] J. Rumbaugh. Relations and semantic constructs in an object-oriented language. *OOPSLA'87*, 1987.

[61] J. Audrey Mayes, Bob Dickerson and Carol Britton. Implementing Associations between Objects. *Microprocessing and Microprogramming* , 40, 1994.

[62] R.G.G. Cattell. *Object data management: object-oriented and extended relational database systems.* Addison-Wesley Publishing Company, Inc , Reasing, Massachusetts, 1994.

[63] G. Razek. Combining Objects and Relations. *ACM SIGPLAN Notices*, 27, December 1992.

[64] Harold Ossher and William Harrison. Combination of Inheritance Hierarchies. *OOPSLA'92*, 1992.

[65] A.Shah, J. Rumbaugh, J. Hamel, and R. Borsari. DSM: an object relationship modelling language. *OOPSLA'89 as ACM Sigplan*, 24, 11 1989.

[66] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. A Catalog of Object-Oriented Design Patterns. Unpublished IBM report, 1994.

[67] Danny Crookes. *Introduction to Programming in Prolog* . Prentice Hall International (UK) Ltd, Hemel Hempstead, Hertfordshire. UK, 1988.