UNIT 1 IMPLEMENTATION STRATEGIES

Structure **Page Nos.**

1.0 INTRODUCTION

The transformation, from design models to code, is an easy and simple feature of object-oriented design, but there are some features of the design models which do not directly map into programming language structures. This unit considers some of the most noticeable features, and discusses the various strategies that have to be adopted for their implementation.

The most significant feature of class diagrams is **association** that is not directly present in the programming languages structures. In this unit, we will describes the different ways of implementation of complex types of association, such as qualified associations and association classes.

The information contained in the dynamic models of an application is reflected in the code that **implements individual operations** for that application. Object interaction diagrams are used to describe the order in which messages are communicated in the execution of an operation, and this information is used as guidance for the implementation of individual operations.

State charts are used to describe constraints that must apply across all the operations of a class, and which can affect the implementation of all a class's operations. A **consistent strategy** should be adopted to check that all these **constraints are correctly** reflected in the implementation of the member functions of the class.

Basically, this unit will cover different aspects of implementation, which covers implementing, associations, implementing constraints, and implementing statecharts.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- explain how the design models are implemented in coding;
- explain how the unidirectional associations are implemented;
- explain how the bi-directional associations are implemented;
- describe how the associations are implemented as classes;
- show how the generalizations are mapped to the tables;
- explain how the constraints are implemented.

- explain how the statecharts are implemented, and
- use the concept of persistence in the implementation.

1.2 IMPLEMENTATION ASSOCIATIONS

Associations describe the properties of the **links** that exist between **different objects** when a system is **running**. A link from one object to another informs each object of the identity, or the location of the other object. Association enables the objects to send messages to each other using the **link as a kind of communication channel**. When the implementation of links is done then these features of links should also to be supported. You can implement a simple association by using **references to linked objects**.

The basic difference between links and references is that links are symmetrical whereas references refer only in one direction. If two objects are linked, a single link serves as a **channel for sending messages in either direction**. By using a reference, however, one object can send messages to another, but the other object is **not aware of** the object that is referring to it. So, it has no way of sending messages back to the first object. You can say that if a link has to support message passing in both directions, it will require **a pair of references** for the implementation for each direction.

But, the use of two references adds a considerable overhead for the implementation. While implementing both references it has to be checked that inverse references are consistently maintained. The implementation of associations is required to be implemented only in one direction because that particular link only needs to be traversed in one direction. The link used association for one direction can be implemented by a single reference, pointing in the direction of traversal.

But, the implementation of an association only in one direction involves a tradeoff between the present implementation strategy and the future modifications that may be incorporated in the design which may also affect the association.

You can take unidirectional and bi-directional associations in this way:

i) unidirectional implementation associations are those in which the decision has been taken to maintain the association in only one direction.

ii) in bi-directional implementations, association must be maintained in both directions.

In general, there are two distinct aspects to the implementation of associations.

i) It is necessary to define the data declarations that will enable the details of actual links to be stored. It will consist of defining data members in one class that can store references to objects of the associated class.

ii) It is necessary to consider the means by which these pointers will be manipulated by the rest of the application. The details of the underlying implementation of the association should be hidden from client code.

Now, let us see how unidirection implementation takes place.

1.3 UNIDIRECTIONAL IMPLEMENTATIONS

In this section we will discuss cases by taking in to consideration that an association will only be supported in one direction. This design decision can be represented on a class diagram by writing an arrow (→) on the association to show the required direction of traversal.

We will discuss the cases where the multiplicity of the type is:

- Optional
- One, and/or
- Many.

1.3.1 Optional Associations

The *Figure1* shows an association that is implemented in **one direction**. Every bank account can have a debit card issued to the account holder for use. But it is not necessary that **all the account holders** will take the debit card from the bank. It depends upon the bank as well as on the account holder to have a debit card.

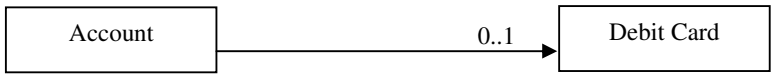

Figure 1: An optional association

This type of association you can implement using a simple reference variable, as shown below. This allows an account object to hold a reference to, at most, one debit card object. Cases where an account is not linked to a card are modeled by allowing the reference variable that will point to the **null**.

```
public class Account
{
public DebitCard getCard()
{
return theCard;
}
public void setCard(DebitCard card){
thecard = card;
}
public void removeCard()
{
theCard = null;
}
private DebitCard theCard
}
```

The above code assumes that a card may be supplied when an account is created. In addition, operations are provided to change the card linked to an account, or to remove a link to a card altogether.

You can see that in this implementation, different cards are linked to an account at different times during its lifetime. Associations with this type of property are called **mutable associations**. On the other hand, immutable associations are those that require that a link to one object cannot be replaced by a link to a separate object, i.e., the link cannot be changed. In this case, we will take the assumption that only one card has been issued to a particular bank account.

If the association between accounts and debit cards cannot be changed, then the following declaration of the account class should be used. This provides an operation to add a card to an account, and only allows a card to be added if **no card is already held**. Once allocated, a card cannot be changed or even removed, and the relevant operations have been removed from the interface.

```
public class Account
{
public Debitcard getCard( )
{
return theCard ;
}
public void setCard(DebitCard card)
```

```
{
if (theCard != null)
{
// throw ImmutableAssociationError
}
theCard = card ;
}
private DebitCard the Card ;
}
```

1.3.2 One-to-One Associations

Some of the properties of associations can be implemented **directly** by **providing suitable declarations** of data members in the relevant classes. Other **semantic features** of an association can be enforced by **providing only a limited range of operations** in the class's **interface**, or by including code in the implementation of member functions that ensures that the necessary constraints are maintained.

Now, let us consider the association shown below in *Figure 2*. This association describes a situation where **bank accounts** must have a **guarantor** who will pay any debts incurred by the account holder in exceptional conditions. It may frequently be needed to find out the guarantor of an account, but it is not necessary to find the details of the account of the guarantor for which s/he is responsible. So, the implementation of the association will be only in the direction **from account to guarantor**.

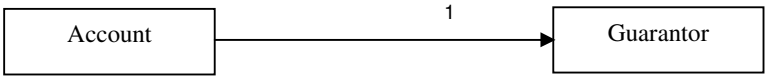

Figure 2: A one-to-one association

You may implement the account class in the following manner if the above assumptions are taken into consideration. The constructor will throw an exception if a null reference to a guarantor object is provided, and no operation is provided in the class's interface to update the reference held.

```
public class Account
{
public Account (Guarantor g)
{
if ( g == null) {
// throw Null Link Error
}
The Guarantor = g;
}
public Guarantor get Guarantor()
{
return the Guarantor;
}
private Guarantor the Guarantor;
}
```

The above code implements the association between account and guarantor objects as an **immutable association**. If the association is mutable, in that case the guarantor of an account could be changed, and a suitable function could be added to this class provided that feature. It should also check that the new guarantor reference **is –null** or not.

1.3.3 Associations with Multiplicity 'Many'

Figure 3 shows an association with multiplicity '**many'** for which a **unidirectional** implementation is required. In a bank, **each manager is responsible** for looking after a **number of accounts**, but it is not necessary to find out the manager of **a particular account**. In this association, a manager object could be linked not just to one, but potentially to many account objects.

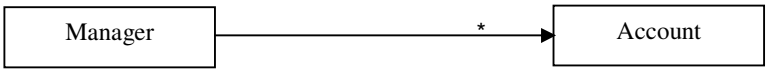

Figure 3: An association with multiplicity 'many'

In order to implement this association, the manager object **must maintain multiple pointers**, **one** to **each linked account**. A data structure is required to store all the pointers. Apart from this, it may be the case that the interface of the manager class will provide operations to maintain the collection of these pointers.

This type of association can be implemented by using of a suitable container class from a class library. The class declares a **vector of accounts** as a private data member, and the functions to add and remove accounts from this collection simply call the corresponding functions defined in the interface of the vector class.

```
public class Manager
{
public void addAccount(Account acc)
{
theAccounts.addElement(acc) ;
}
public void removeAccount(Account acc){
theAccounts.removeElement(acc) ;
}
private Vector theAccounts
}
```

In implementation, there can only be, at most, one link between **a manager** and **any particular account**. In this implementation, however, there is no reason why many pointers to the same account object could not be stored in the vector held by the manager. A correct implementation of the "***addAccount***" function should check whether the account that is added is linked to the other manager object or not.

☞ Check Your Progress 1

1) What is the most significant feature that does not directly map into programming language structures? Why?

……

……

……

2) What are object interaction diagrams?

……

……

……

3) What is the difference between link and reference?

……

……

……

4) What are the two distinct aspects to the implementation of associations?

……

……

5) Discuss how the one-to-one associations have to be implemented.

……

……

1.4 BI-DIRECTIONAL IMPLEMENTATIONS

In a bi-directional implementation of an association, a **pair of references** should **implement each link**. The declarations and code required to support such implementations are essentially the same as discussed in the unidirectional implementation. The only difference is that **suitable fields** have to be declared in **both classes** participating in the association.

On problem with bi-directional implementation is that the extra complexity in its implementations arises due to the need to keep the two pointers that implement a link consistent at the run-time.

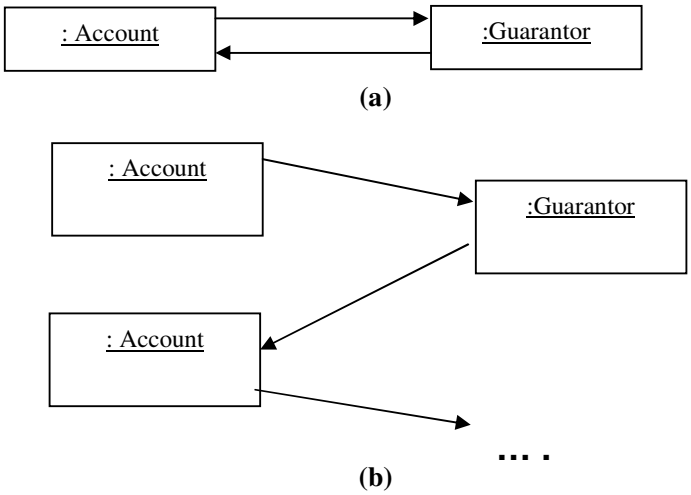

Figure 4: Referential integrity and its violation

It is necessary that the guarantor of an account must guarantee the same account type of the property of an association should hold between the accounts and guarantors as shown in *Figure 4(a). Figure 4(b)* violates this property, the top account **object holds a reference** to a guarantor object which in turn **holds a reference** to **entirely a different** account. These two references cannot be understood as being an implementation of a single link. This is a case of not being in referential integrity.

It should be clear from the above example that **referential integrity** cannot be ensured by simply giving appropriate definitions of data members in the relevant classes.

The unidirectional implementations will not support every association needs for all possible forms of manipulation of links. The behaviour that is required to be supported will depend on the **details of individual applications,** and will be defined by the **active operational interfaces** of the classes participating in the association.

1.4.1 One-to-One and Optional Associations

Assume that we are now required to provide a bi-directional implementation of the association discussed in previous section of this unit. Let us also assume that the association is immutable in the debit card to account direction, i.e., **once a debit card is linked to an account**, it must stay linked to that account until the end of its

lifetime. An account, on the other hand, can have different cards associated with it at different times, to cater for situations where the account holder loses a card.

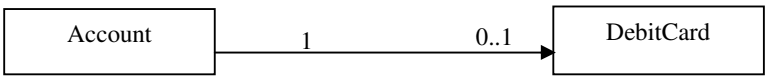

Figure 5: A bi-directional one-to-one association

We may think of this association as a combination of a mutable and optional association in the left-to-right direction with an immutable association in the other. A simple approach to its implementation would be to simply combine the implementations of those given in unidirectional implementation as shown below:

```
public class Account
{
public DebitCard getCard() {...}
public void setCard(DebitCard card) {...}
public void removeCard( ) {...}
private DebitCard theCard ;
}
public class DebitCard
{
public DebitCard(Account a) { ...}
public Account getAccount() { ...}
private Account theAccount;
}
```

The above implementation provides the data members to store the bi-directional links. But the operations available **maintain the two directions** of the link **independently,** e.g., the pointer from the account to the card is set up in the account constructor and the pointer from card to account in the card constructor.

For example, to create a link between a new debit card and an account, two separate operations are required, first to create the card and second to link it to the account. The link from card to account is created when the card itself is created. Code to implement can be as follows:

```
Account accl = new Account() ;
Debitcard card1 = new Debitcard(acc1);
acc1.setCard(card1) ;
```

It is necessary to ensure that these two operations are always performed together. However, as two separate statements are needed, there is a possibility that one might be omitted, or an erroneous parameter supplied, leading to an inconsistent data structure. In the following example, the debit card is **initialized** with the **wrong** account.

```
Account acc1= new Account (), acc2 = new Account ();
DebitCard card1 = new DebitCard(acc2) ;
accl.setCard(card1);
```

A better solution is to give only one of the classes the responsibility of maintaining the association. A link between two objects could then be created by means of a single function call, and encapsulation could be used to ensure that only trusted functions have the ability to directly manipulate links.

The choice of proper class in this regard is very important.

The choice of which class to give the maintenance responsibility depends upon the other aspects of the overall design. In this case, it is likely that there would be an operation on the account class to create a new debit card for the account. Also, it would provide a strong argument for making, the account class responsible for maintaining the association. If this is the case, then the classes could be redefined as follows.

Implementation

```
public class Account
{
public DebitCard getCard()
{
return theCard;
}
public void addCard()
{
theCard = new DebitCard(this) ;
}
private DebitCard theCard ;
}
public class DebitCard
{
DebitCard(Account a) {  theAccount = a ; }
public Account getAccount( ) {
return theAccount;
}
private Account theAccount ;
}
```

You can create debit cards by the 'addCard' operation in the account class. The implementation of this operation dynamically creates a new debit card object, passing the address of the current account as an initializer. The constructor in the debit card class uses this initializer to set up the pointer back to the account object creating the card. A single call to the add card operation is now guaranteed to set up a bi-directional link correctly.

In the above example, implementation is simple because the association declared is immutable in one direction. If, both directions of an association are mutable then a lot of situations will arise in which links can be altered. Then a correct implementation will be required to ensure that all the above things are correctly handled.

Take the assumption that a customer could hold many accounts but s/he has only one debit card. S/he had the flexibility to select which account has to be debited when the card is used. The association between accounts and debit cards would now be mutable in both directions. It would be reasonable for the card class to provide an operation to change the account with which card was associated.

There are many manipulations involved in above discussed case. First, the existing link between the **card and its account must be broken**. Second, a new link must be established between the new account and the card. Finally, this should only happen if the new account is not already linked to a card.

The card class must call functions in the account class to update pointers, as shown in the implementation of the '**changeAccount**' operation given below.

```
public class Account
{
public DebitCard getCard() { ...}
public void addCard(DebitCard c) { ... }
public void removeCard()
{
theCard = null ;
}
private DebitCard theCard ;
}
public class DebitCard
{
```

```
public DebitCard(Account a) {…}
public Account getAccount() {…}
public void changeAccount(Account newacc)
{
if (newacc.getCard() != null) {
// throw AccountAlreadyHasACard
}
theAccount.removeCard() ;
newacc.addCard(this) ;
}
private Account theAccount;
}
```

1.4.2 One-to-Many Associations

The bi-directional implementation of associations involving multiplicities of **many** raises the same problems as discussed in last sub-section. For example, *Figure 6* shows an association specifying that customers can hold many accounts, each of which is held by a single customer.

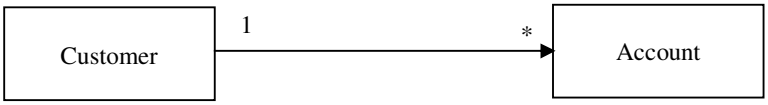

Figure 6: Customers holding accounts

The customer class could contain a data member to store a collection of pointers to accounts, and additionally each account should store a single pointer to a customer. The responsibility of maintaining the links of this should be given to the customer class.

1.4.3 Immutable Associations

Take the assumption that the association between accounts and guarantors was intended to be immutable and are traversed in both the directions. As *Figure 7* shows, the relevant class diagram which preserves the restriction that each guarantor can only guarantee one account.

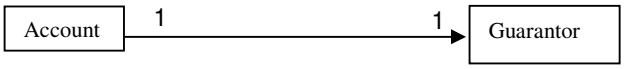

Figure 7: An immutable one-to-one association

Each class will define a data member **holding a reference to an object of the other class**. The class declarations might be as follows:

```
public class Account
{
public Account(Guarantor g)
{
theGuarantor = g;
}
public Guarantor getGuarantor()
{
return theGuaranter;
}
private Guarantor the Guarantor ;
}
public class Guarantor
{
public Guarantor(Account a)
{
theAccount = a;
```

```
}
public Account getAccount()
{
return theAccount;
}
private Account theAccount ;
}
```

You can see, in above declarations, that they introduce a circularity. When an account is created, it must be supplied with an already existing guarantor object, and likewise when a guarantor is created it must be supplied with an already existing account. It might be thought that this could be achieved by creating the two objects simultaneously, as shown in the following code:

```
Account a = new Account (new Guarantor (a));
Guarantor g = a.getGuarantor();
```

Associations are decided at the time of describing the requirements of the systems. Associations can be implemented in association classes and in the next section, we will discuss the implementation of association as classes.

1.5 IMPLEMENTING ASSOCIATIONS AS CLASSES

It is not possible to handle association classes with a simple implementation of associations based on references. For example, consider the diagram shown below which shows that many students can be registered as taking modules, and that a mark is associated with each such registration.

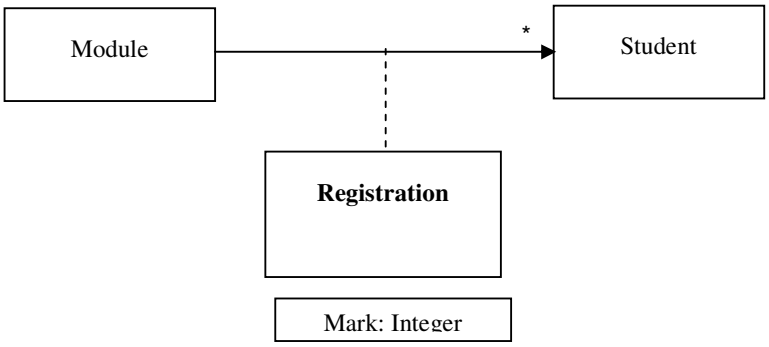

Figure 8: An association class

The association class shown in *Figure 8* needs to be implemented **as a class**, to provide a place to store the attribute values representing mark. The links corresponding to the association cannot then be implemented as references between the module and student classes.

A common approach, in this case, is to transform the association class into a simple class linked to the two original classes with two new associations, as shown in *Figure 9* below. In this diagram, the fact that many students can take a module is modeled by stating that the module can be linked to many objects of the registration class, each of which is further linked to a unique student, namely, the student for whom the registration applies.

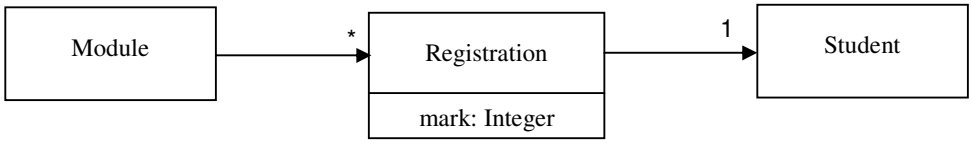

Figure 9: Transforming the association into a class

As a result of this transformation, neither of the associations in *Figure 9* have link attributes and so, they can be given straightforward implementations using references.

The original association shown in *Figure 8* would naturally have been maintained by the module class, which might provide operations to add a link to a student, and to record a mark stored for a student. Despite the fact that the association is now being implemented **as a class**, the interface presented to client code should **remain unchanged**. This implies that the module class must maintain both the registration class in *Figure 8* and *Figure 9*. Also, you can see that there are two new associations. Therefore, the operation to add a student to a module must create a new object and two new links.

The implementation of the registration class is very simple. It must store a reference to the linked student object and the mark obtained by that student. No need to provide any exclusive operational interface.

```
class Registration
{
Registration (Student st)
{student = st; mark= 0}
private Student student ;
private int mark;
}
```

The relevant parts of the definition of the module class are written below. Here, you can see that the implementation given in this example does not perform any of the validation discussed above.

```
public class Module
{
public void enrol(Student st)
{ registrations.addElement(new Registration(st));
}
private Vector registrations;
}
```

Associations, are implemented as a class whenever a class diagram contains l**ink attributes,** or **associations modeled as classes. Many-to-many association as a class** can be implemented in many situations. A simple pointer-based implementation will not be able to handle many- to-many association efficiently, whereas if the association had been implemented as a class, it would then be almost insignificant to add link attributes.

Bi-directional implementations of associations that are implemented, as a class do **not add new problems** to the above-mentioned situation. This make it easily implementable.

☞ Check Your Progress 2

1) Discuss how the bi-directional Implementations are made.

……

……

2) Explain how the associations are implemented as classes.

……

……

1.6 IMPLEMENTING CONSTRAINTS

Class constraints are used to describe relationships that must be between the attribute values of an instance of the class; and preconditions and post-conditions specify what must be true before and after an operation are called. Once these are implemented by

including code in the class which checks these conditions at the appropriate times; then the applications become more reliable and robust.

It is necessary that all the preconditions that are specified for an operation should be explicitly checked in an implementation. Preconditions state properties of an operation's parameters that must be satisfied if the operation is to be able to run to completion successfully. It is the responsibility of the caller of the operation to ensure that the precondition is satisfied when an operation is called.

If an operation does not check its parameter values, then, there is a possibility that wrong or meaningless values will go undetected, and this results in unpredictable **run-time errors**. A better strategy is for an operation to check its precondition and to raise an exception if a violation of the precondition is detected. The following example provides a possible implementation of the withdraw operation of the savings account class.

```
public class SavingsAccount
{
public void withdraw(double amt)
{
if (amt >= balance)
{
// throw PreconditionUnsatisfied
}
balance = balance - amt ;
}
private double balance ;
}
```

In general**, any constraint can be checked at run-time** by writing code that will validate the current state of the model. But such checks increase overhead. That is why, except for the case of precondition checking, constraints are rarely implemented explicitly.

Statecharts are diagrams that specify an object's responses to the events it might detect during its lifetime. We will discuss statechart in next section.

1.7 IMPLEMENTING STATE CHARTS

It is very important to apply proper strategy in final implementation of the systems. Here we will see some strategies for implementations.

A Basic Implementation Strategy

This approach models the different states in the statechart explicitly by means of an enumeration in the class to which the statechart applies. The current state that an object is in, is recorded by a special data member of the class that can take on values from this enumeration.

Member functions that can have different effects depending on the state of the object are implemented as **switch statements**, each case of which represents one possible state of the object. The implementation of each case corresponds to a single transition on the statechart. It should check any applicable conditions, perform any actions, and if necessary change the state of the object by assigning a new value to the data member which records the current state.

This implementation of this approach is simple and can be generally applied but it has some disadvantages also. It does not provide much flexibility in the case where a new state is added to the statechart. The implementation of every member function of the class would have to be updated in such a case, **even if they were unaffected by the change**. Also, the strategy assumes that most functions have some effect in the

majority of the states of the object. If this is not the case, the switch statements will be full of '**empty cases**' and the implementation will contain a lot of redundant code.

An Alternative Approach

An alternative implementation of statecharts can avoid these problems at the cost of **representation of individual states**. Rather than representing the current state of an object by the value of a data member in the object itself, this approach represents states as objects. Each instance of the class described by the statechart maintains a pointer to its current state, which is an instance of one of the state classes. The state classes are arranged in a generalization hierarchy so that different states can be referred, by the same pointer.

Now, let us see a class diagram illustrating the structure of this implementation. You can see in *Figure10* below, which shows the classes that would be declared to implement the creation tool class from the diagram editor.

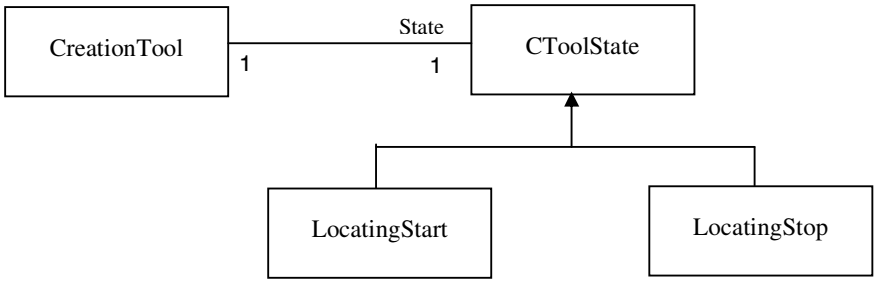

Figure 10: Representing states with classes

A field in the creation tool class will hold a reference to an object of type 'CtoolState'. This is an abstract class, so at run-time the object referred to will be an instance of one of the subclasses 'LocatingStart' or 'LocatingStop'. In this way, a creation tool always holds a reference to an object representing its current state.

The classes representing states provide implementations of the operations declared in the creation tool interface. When a creation tool receives a message, it simply passes it on to the object representing its current state, which contains a suitable implementation. A partial definition of the creation tool class could be given as follows.

```
public class CreationTool
{
public void press()
{
state.press() ;
}
private CToolState state;
}
```

The interface of the 'CtoolState' class must include all the messages that will be passed on from tools. Default implementations can be provided in 'CtoolState', however, so that individual states need only define those operations which evoke some interesting behaviour in that state. The following code gives a partial declaration of the 'CtoolState' class.

```
public abstract class CToolState
{
public abstract void press();
}
```

Subclasses that represent individual states must now redefine the operations that interest them. In the case of 'press', the press event can be detected in the 'LocatingStart' state, and in response the tool should change state. A possible

definition of the function is given below. For reasons that are discussed below, only pseudocode implementations of the functions are given.

```
public class LocatingStart extends CToolState
{
public void press()
{
set start position to current;
draw faint image of shape;
set current state to 'LocatingStop' ;
}
}
```

As the press event cannot be received in the 'LocatingStop' state, as the mouse button is already depressed, no definition of this function is required in the class 'LocatingStop'. The default implementation of the function inherited from 'CTooIState' is quite adequate.

It is important that any implementation must be sensitive to the **fact that an object can be in different states at different times**, and that the effect of operations is dependent on the **current state**. The simple implementation makes this all explicit, and the programmer must write switch statements to detect the current state of the object. In the more sophisticated implementation outlined above, detection of the current state is performed implicitly by the dynamic binding performed when a virtual function from the general 'CtoolState' class is called.

The sophisticated implementation has several advantages over the simple one. For example, the classes that represent individual states only need to define the operations that are relevant to them, and can inherit default implementations of the others from the general state class. This can considerably simplify the implementation of a statechart, especially in the case where many operations are only applicable in a small subset of the object's states.

The sophisticated approach is also more maintainable than the simple one. For example, if the statechart is extended to include extra states, these can simply be added as new subclasses of the general state class, and existing code which is not relevant to the change will be unaffected. This contrasts with the simple implementation, where adding a state requires the implementation of every member function to be updated.

There are costs associated with the sophisticated approach, however, which mostly stop from the fact that the implementation of the member functions of the state classes often needs to update the state of the object itself.

1.8 PERSISTENCY

Persistent data is data which has a longer lifetime than the program created it. In the context of an object oriented program, this means that it must be possible to save the objects created in one run of a program and to reload them at a later date. The user should not have to create all the data used by a program from scratch every time the program is run. The usefulness of the program will be rather limited if it is not possible to save diagrams to disk and to continue working on them at a later date.

Enabling data to be **stored on a permanent storage medium** provides persistency. The most common techniques used are to store data in files, or to make use of a back-end database system.

Identifying Persistent Data

The basic problem is that it may not be always clear from a model exactly what data needs to be persistent. Models in UML are not restricted to describing permanent data,

or database schemas, and as a result a single model can combine persistent and transient data.

The only notation that UML provides for persistency is a tagged value ‘persistence’. This has two values ‘persistent’ and ‘transient’ and can be applied to classes, associations and attributes.

For example, the diagram and element classes in the diagram editor need to be persistent. This is the data that the user would expect to be able to save and reload at a later date. The tool class, on the other hand, does not need to be persistent. Tools represent transient features of the user’s interaction with the editor, and it would not be felt to be a major shortcoming if the tool that was being last used was not available when the program was restarted.

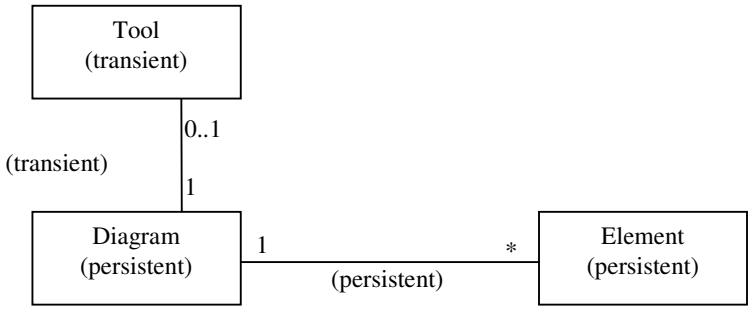

Figure 11: Persistent and transient classes

The primary unit of persistency is the class. Associations between two persistent classes are normally persistent, so that information about the links between persistent objects is stored. An **association between a persistent and a transient class will be transient**, as the instances at one end of the association are not being stored. Attributes normally have the same persistence value as their enclosing class, though it is sometimes natural to have a transient attribute within a persistent class.

Dealing with Object Identities

References are transient, and the persistency of an object model cannot be obtained simply by copying the values of these references to disk. The problem is that references are normally implemented using the address of the object referred to in memory, and therefore an object’s identity depends on where free memory happened to be available when it was created.

A more common approach, however, is to adopt some form of encoding whereby references and object identities are stored in such a way that they can be consistently recreated.

Serialization

Serialization is a generic term used for mechanisms that enable objects and object structures to be converted into a portable form, removing the volatility created by object addresses. As we have already studied in MCS-024, serialization is provided in Java by means of an interface called ‘Serializable’. This interface defines no methods, so in order to make a class serializable it is sufficient to declare that it implements this interface. Once this has been done, the methods ‘writeObject’ and ‘readObject’ can be used to transfer objects to and from streams, and persistence can then easily be implemented. Serialization therefore provides a convenient and straightforward way of making data persistent. It is most appropriate when the amount of data involved is relatively small. If larger amounts of data are to be stored, serialization may no longer be appropriate.

☞ Check Your Progress 3

1) What do you mean by persistence? How you will make your data persistent?

……

……

……

2) What is serialization? Where it is used and why?

……

……

……

3) What are the different strategies of implementation of the Statecharts?

……

……

……

1.9 SUMMARY

A simple strategy for implementing associations uses references to model links. Implementations can be either unidirectional or bi-directional, depending on how the association needs to be navigated. Bi-directional implementations of associations need to maintain the referential integrity of the references implementing a link. A strategy for robustness is to assign the responsibility of maintaining references to only one of the classes involved. Also, we have seen that Association classes should be implemented as classes. This involves introducing additional associations connecting the association class to the original classes. Implementing an association as a class is a general strategy for the implementation of associations that can increase the ability of a system to withstand future modifications at the expense of a more complex implementation.

We saw that Constraints in a model can be explicitly checked in code, but often it is only worth doing this for the preconditions of operations. A sophisticated technique for implementing statecharts was described, which represented each state by a class. This offers considerable benefits, at the cost of a significantly more complex implementation. Persistent data is data that must be preserved after the program which created it has finished running. Providing persistence for object structures is not easy, because of the use of references to implement links between objects. Small amounts of data can be saved using the technique of serialization, provided by many object-oriented programming libraries.

1.10 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Association, because there are complex types of association, such as qualified associations and association classes which are very difficult to implement.

2) Object interaction diagrams are used to **describe the order** in which messages are communicated in the execution of an operation, and this information is used as guidance for the implementation of individual operations.

3) The basic difference between links and references is that **links are symmetrical** whereas references **refer only in one direction**. If two objects are linked, a single link serves **as a channel** for sending messages in either direction. By using a reference, however, one object can send messages to another, **but the other object is not aware of the object that is referring to it**.

4 i) It is necessary to define the data declarations that will enable the details of actual links to be stored. It will consist of defining data members in one class that can store references to objects of the associated class.

ii) It is necessary to consider the means by which these pointers will be manipulated by the rest of the application. The details of the underlying implementation of the association should be hidden from client code.

5) Some of the properties of associations can be implemented directly by providing suitable declarations of data members in the relevant classes. Other semantic features of an association can be enforced by providing only a limited range of operations in the class's interface, or by including code in the implementation of member functions that ensures that the necessary constraints are maintained.

Check Your Progress 2

1) In a bi-directional implementation of an association, **a pair of references** should implement **each link**. The declarations and code required to support such implementations are same as needed in the unidirectional implementation. The only difference is that suitable fields have to be declared in both classes participating in the association.

2) A common approach in this case is to **transform** the **association class** into a simple class linked to the two original classes with two new associations, as shown in *Figure 9*.

Check Your Progress 3

1) Persistent data is data, which has a longer lifetime than the program that created it. Enabling data to be stored on a permanent storage medium provides persistency. The most common techniques used are to store data is in the form of files or to make use of a back-end database system.

2) *Serialization* is a generic term used for mechanisms that enable objects and object structures to be converted into a portable form, removing the volatility created by object addresses.

   Serialization provides a convenient and straightforward way of making data persistent. It is most appropriate when the amount of data involved is relatively small. If larger amounts of data are to be stored, serialization may no longer be appropriate.

3) There are two strategies of implementation for state charts:

   i) **Basic Implementation Strategy**

   This approach models the different states in the statechart explicitly by means of an enumeration in the class to which the statechart applies. It represents the current state of an object by the value of a data member in the object itself.

   ii) **An Alternative Approach**

   This approach represents states as objects. Each instance of the class described by the statechart maintains a pointer to its current state, which is an instance of one of the state classes. The state classes are arranged in a generalization hierarchy so that the same pointer can refer different states.