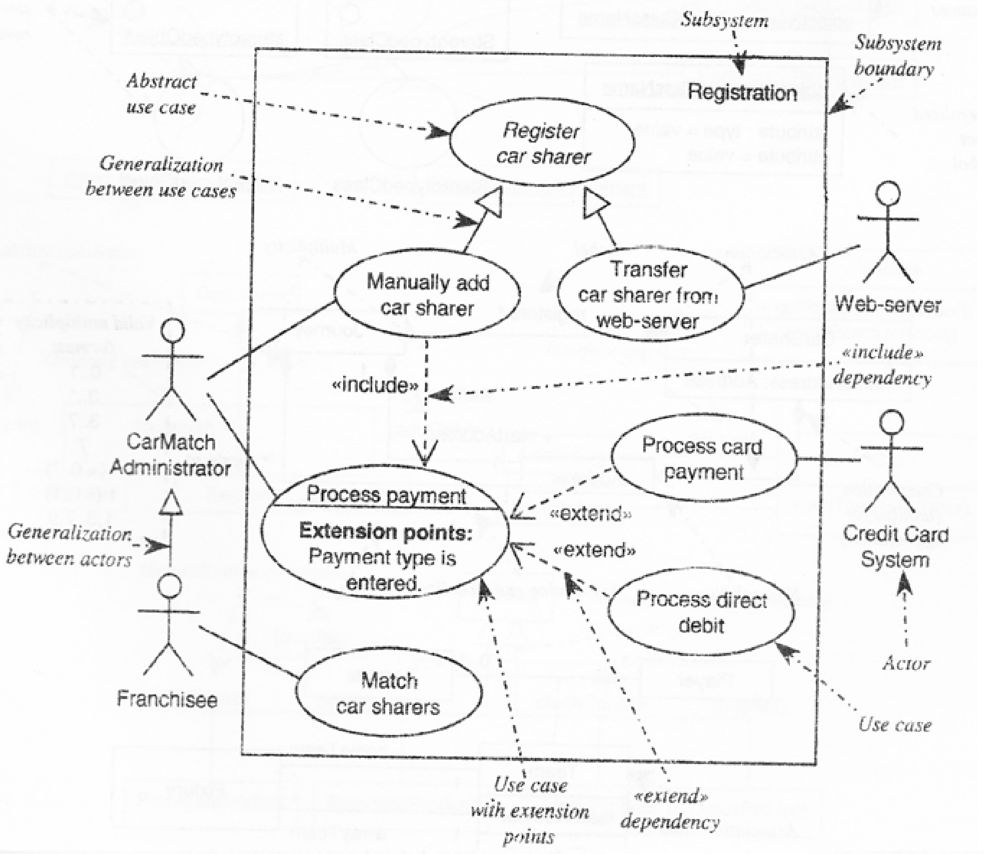
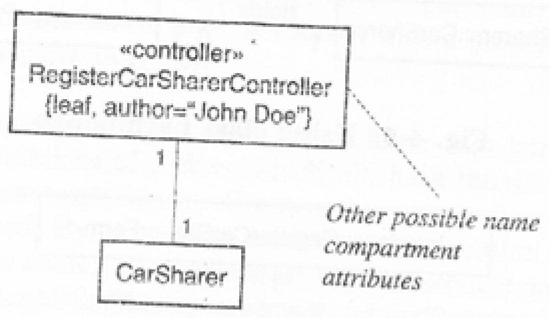


Aside

A controller class can be used to coordinate the interaction between classes in the core class model of an application and the interface of that application. Structuring the implementation in this way shields the interface implementation from many types of change to the underlying class model. Similarly, different interfaces can be commissioned without requiring reworking of the underlying class model.


«controller»
RegisterCarSharerController
{leaf, author="John Doe"}
1
1
CarSharer
Other possible name compartment attributes

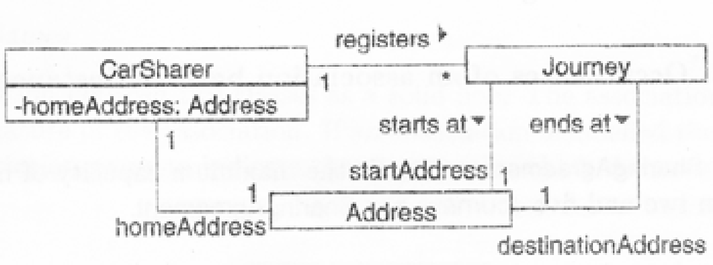
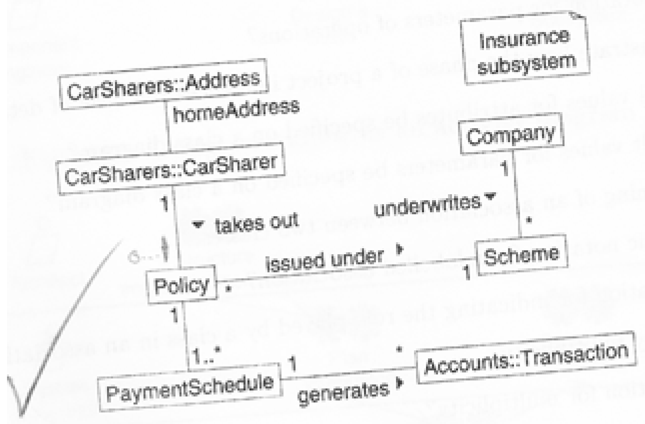
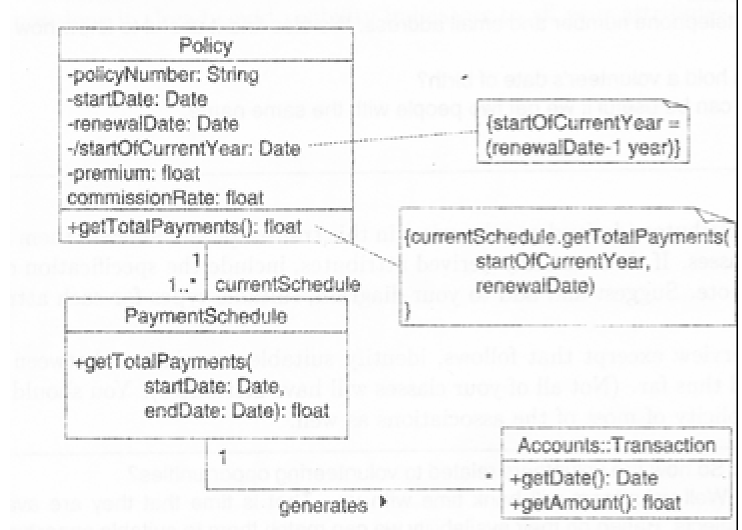
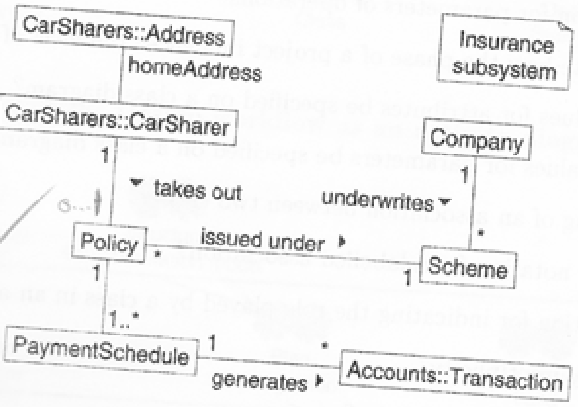


Fig. 4-18: Role names on associations

Find classes and associations
1-who does what?
2-restrictions?
3-object lifecycle




Policy
-policyNumber: String
-startDate: Date
-renewalDate: Date
-/startOfCurrentYear: Date
-premium: float
commissionRate: float
+getTotalPayments(): float
{startOfCurrentYear = (renewalDate-1 year)}
{currentSchedule.getTotalPayments(startOfCurrentYear, renewalDate)}
1
1..*
currentSchedule
PaymentSchedule
+getTotalPayments(startDate: Date, endDate: Date): float
1
Accounts::Transaction
+getDate(): Date
+getAmount(): float
*
generates



Company → Scheme
Initiate Policy object
(Assign Policy to CS
Set Payment Schedule
For Policy
That will be Recorded later
Noordel

Fig. 4-28: Class diagra

5.3.2 Composition

Aggregation then, implies a whole-part structure between two classes. This is also the job of the composition notation. However, a composition association also implies *coincident lifetime* (Object Management Group, 1999*b*, p.3-74). A coincident lifetime means that when the whole end of the association is created, then the part components are also created. When the whole end is deleted, the part components are also deleted. In other words, in composition a part cannot exist without being part of a whole.

a part is capable of existence outside of whole-part associ-

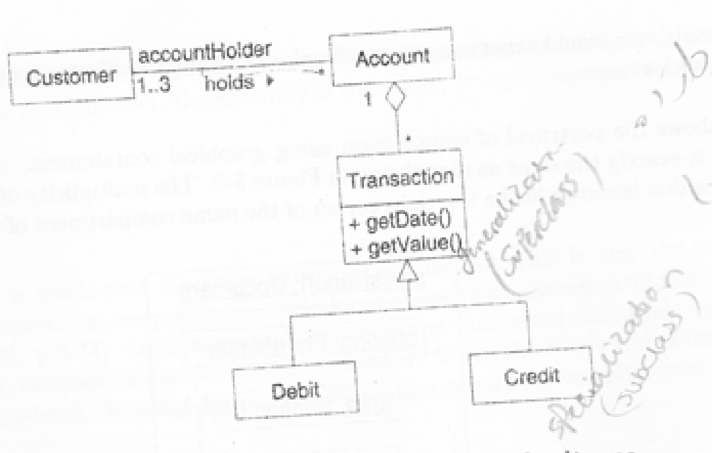


Fig. 5-5: Extended ATM example with inheritance

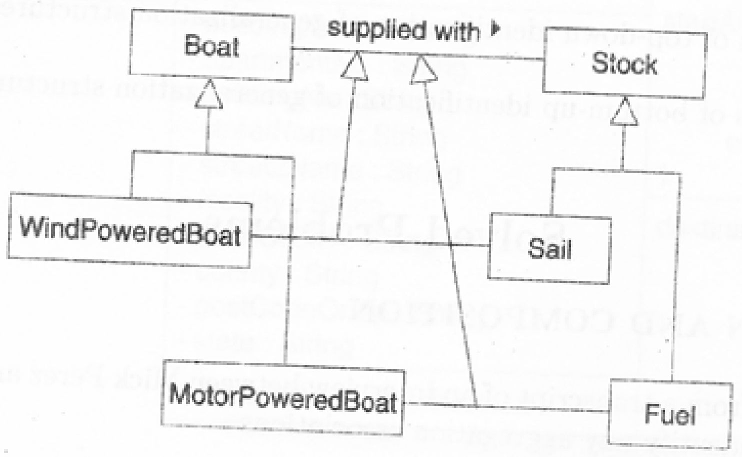
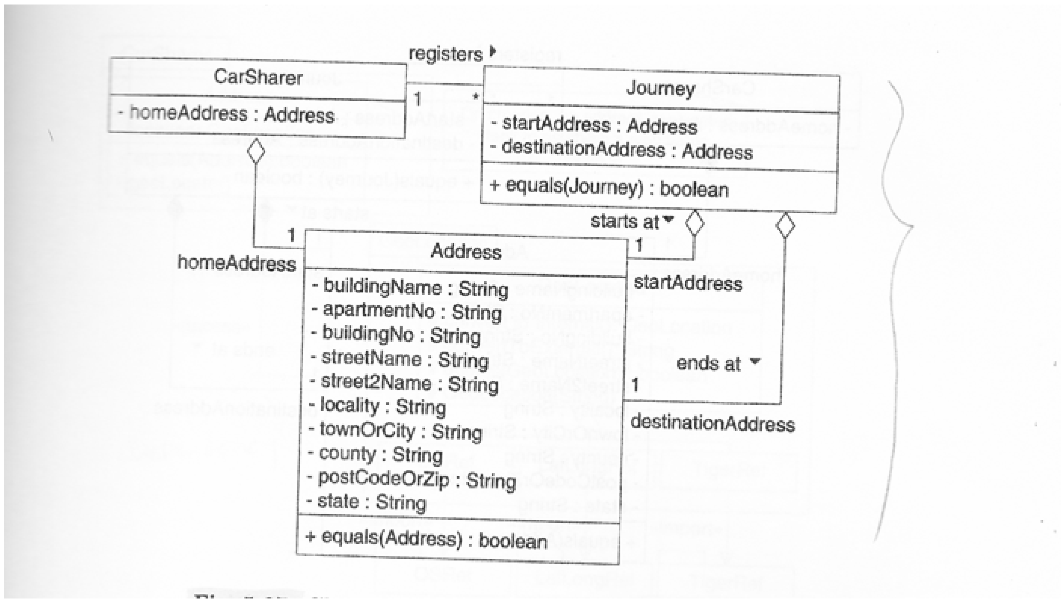


Fig. 5-26: Specialized association linking subclass to subclass



As the association and the association class are one and the same concept, there can only be one occurrence of an association class per occurrence of an association. For example, consider the Contractor—Project association and Assignment association class suggested in Table 6-1. This association is shown in Figure 6-22.

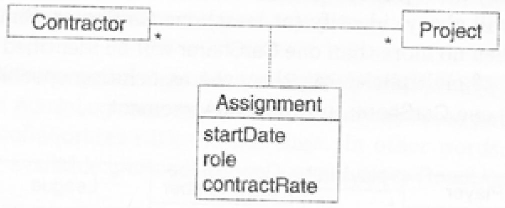


Fig. 6-22: Assignment association class between Contractor and Project

Modelling this association class as an equivalent set of normal classes and associations would require this 'one association class per association' constraint to be enforced by the multiplicities used. The class diagram shown in Figure 6-23 is not equivalent to Figure 6-22 as it would be possible to create different occurrences of the Assignment class for the same occurrences of Contractor and Project.

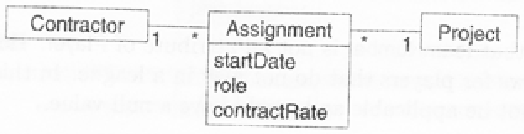


Fig. 6-23: Assignment modelled as class

Thus an association class can be used where the 'one association occurrence, one association class instance' constraint should be enforced. If this is not the case, for example, if a history of assignments

Of one of conractor to one project is required , then the association class should be modeled as a normal class with appropriate association to the other class.

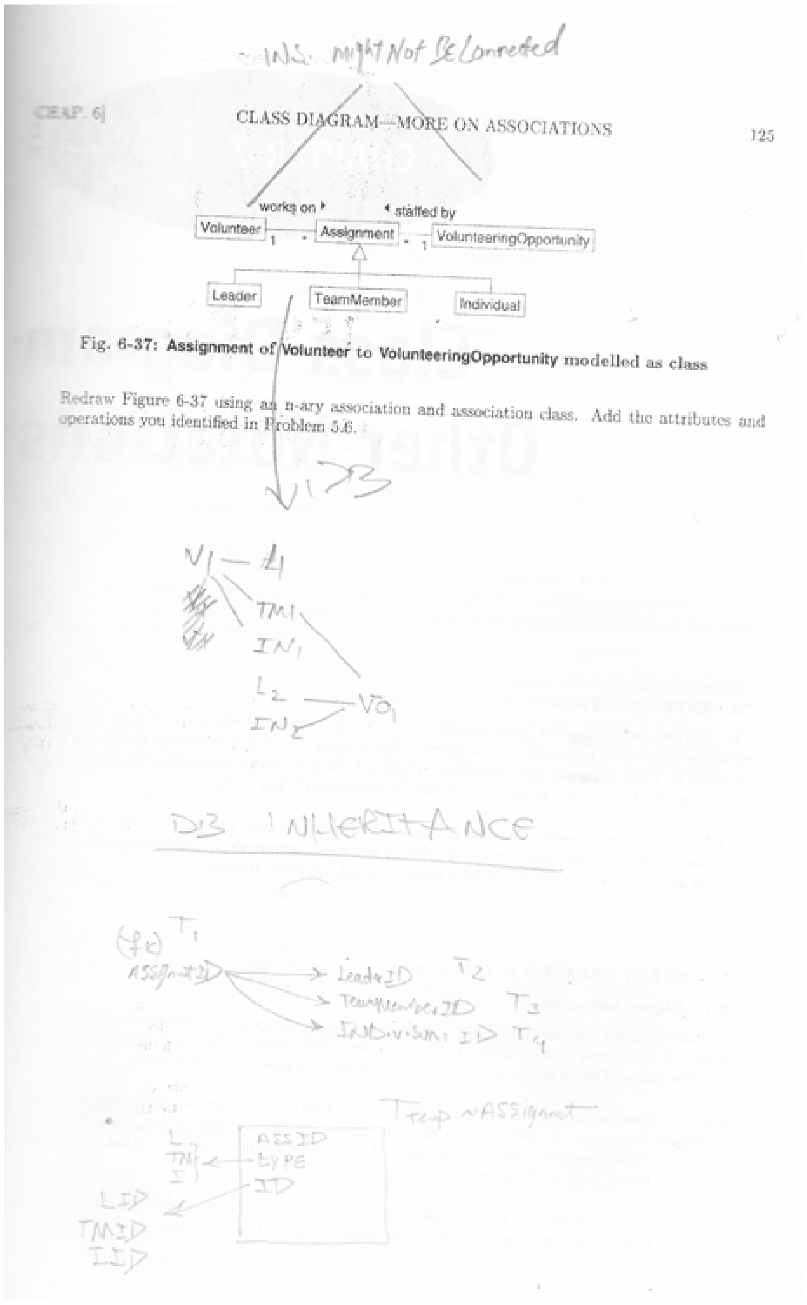
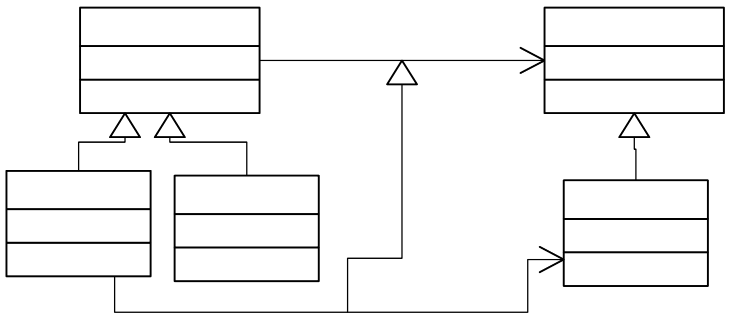

INS. might Not Be Connected
works on
staffed by
Volunteer
Assignment
VolunteeringOpportunity
1
*
*
1
Leader
TeamMember
Individual
V1 DB
V1 — L1
TM1
IN1
L2
VO1
IN2
DB INHERITANCE
(fk)
T1
ASSignmentID
LeaderID
T2
TeamMemberID
T3
INDividual ID
T4
Temp ~ASSignment
ASS ID
type
ID
L
TM
I
LID
TMID
IID

Fig. 6-37: **Assignment of Volunteer to VolunteeringOpportunity** modelled as class

Redraw Figure 6-37 using an n-ary association and association class. Add the attributes and operations you identified in Problem 5.6.

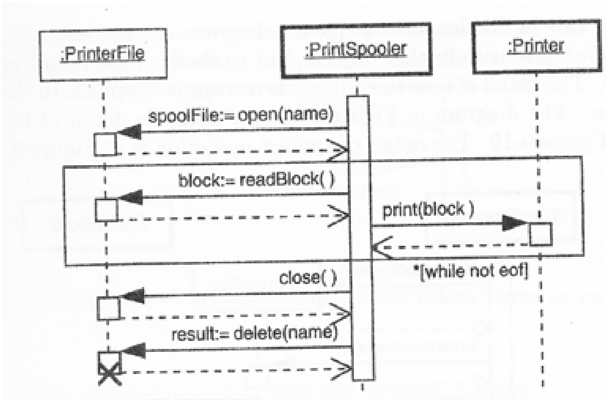
1-Designs can be at [conceptual-implementation-Design] Mode
2-in database-respective object of one class would be a record.

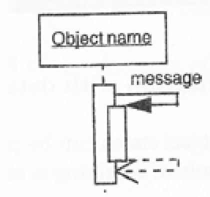


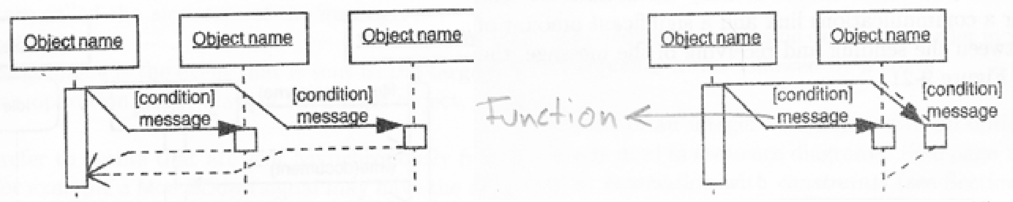
We cannot tell whether the ability of one object to send a message to another is based on a permanent association in the class diagram or a temporary link («local» or «parameter») created for the purpose of this interaction.

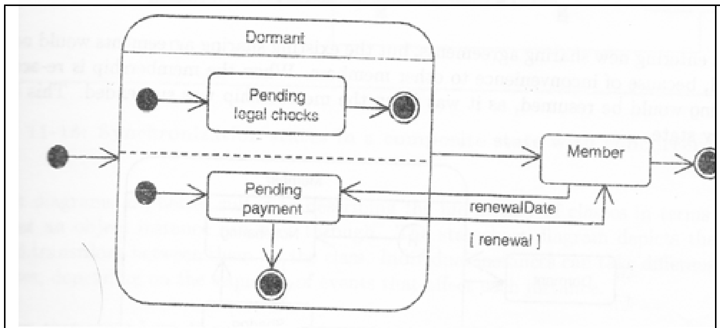
9.3 PURPOSE OF THE TECHNIQUE

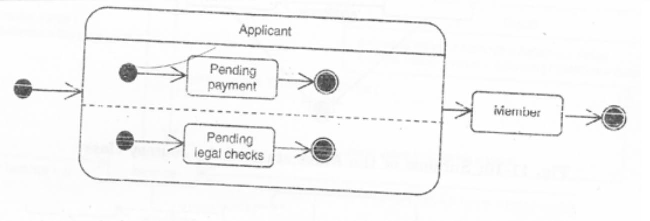
Sequence diagrams are used to model the interaction between object instances in the context of a collaboration. The collaboration is implicit in a sequence diagram, rather than explicitly represented as in a collaboration diagram. Instances are used rather than roles, but it must be remembered that each instance is playing a role that has been defined in a collaboration.

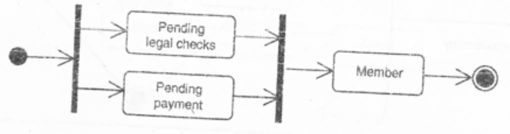


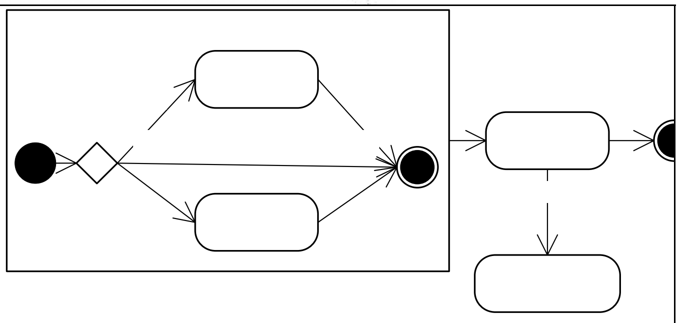












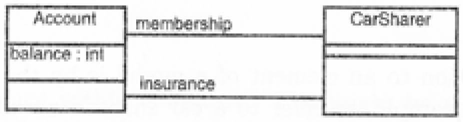
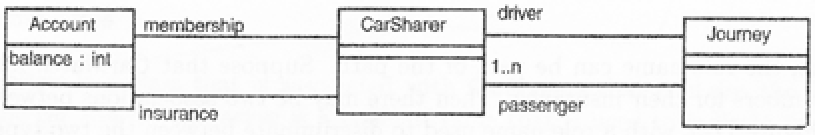


Fig. 12-5: Two relationships between Account and CarSharer delineated by roles

```
context CarSharer inv:
    insurance.balance >= -500
```

It is possible to navigate across a number of associations. So, for example, if it is made a condition that the driver in a journey has paid his or her insurance, using the associations in Figure 12-6 we can express this as follows.

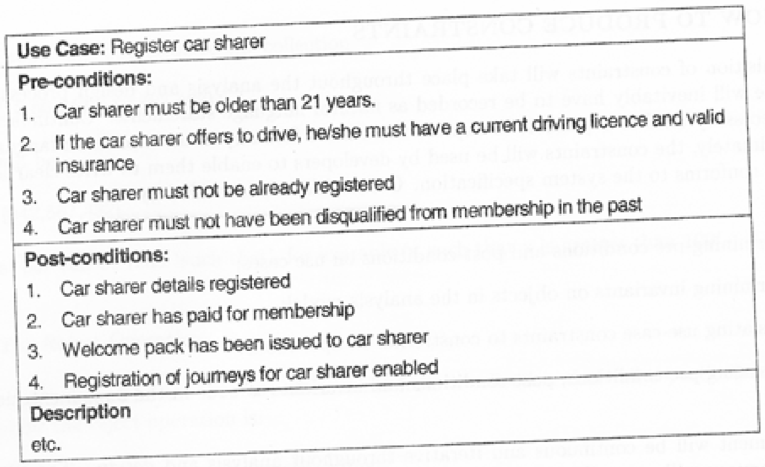
```
context Journey inv:
    driver.insurance.balance >= 0
```



//

12.4 HOW TO PRODUCE CONSTRAINTS

- determining pre-conditions and post-conditions on use cases;
- determining invariants on objects in the analysis model;
- translating use-case constraints to constraints on operations;
- translating pre-conditions, post-conditions and invariants to code in the implementation.



Use Case: Register car sharer

Pre-conditions:
1. Car sharer must be older than 21 years.
2. If the car sharer offers to drive, he/she must have a current driving licence and valid insurance
3. Car sharer must not be already registered
4. Car sharer must not have been disqualified from membership in the past

Post-conditions:
1. Car sharer details registered
2. Car sharer has paid for membership
3. Welcome pack has been issued to car sharer
4. Registration of journeys for car sharer enabled

Description

etc.

Fig. 12-10: The use case description Register car sharer

```
context CarSharer inv:
    self.age() >= 21.
```

The constraint that car sharers must not have been disqualified from membership in the past implies that we need to keep a register of car sharers who have been disqualified. The class diagram in Figure 12-11 indicates the object relationships to support the Register car sharer use case. Now we can write a set of pre-conditions on the register operation on RCSControl as follows:

```
context RCSControl::register(name,address)
    self.CarSharer -> forall(self.name <> name and self.address <> address) and
    self.DisqualifiedCarSharer -> forall(self.name <> name
            and self.address <> address) .
```

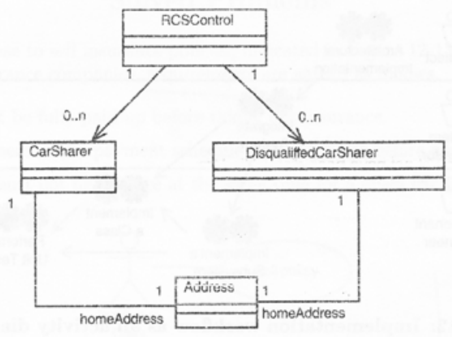


Fig. 12-11: Part of the class diagram to implement the use case Register car sharer

Solved Problems

12.1 Consider the use case to sell insurance policies, indicated in Figure 12-13. As part of the discussions with the insurance companies, requirements are agreed as follows.

- Members must be fully paid up before taking out insurance.
- The first payment of the payment schedule must be made before cover is granted.
- Two policies must not be in force at the same time for a given car sharer.

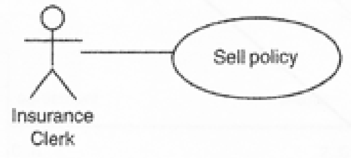


Fig. 12-13: The use case Sell policy

SCOPE ~~w/ Args~~
context → OPERATION
OR
ACTOR

Write OCL constraints to handle these requirements.

These are added to the use case as pre-conditions and recorded as indicated in Figure 12-10 above. Then, as part of the design process a partial class diagram is devised as in Figure 12-14. Here a control object, SPControl, has been introduced to manage the use case transactions, and it has three operations (among others) to gather insurance details, to create a new policy and to create a new payment schedule. The first pre-condition should sensibly be checked before insurance details are gathered—it is not appropriate to gather information when there is something that is easily checked that could block the transaction. We would therefore put this as a pre-condition on the gatherInsuranceDetails operation, thus:

```
context SPControl::gatherInsuranceDetails( )
    pre: CarSharer.membership.balance >= 0.
```

Thus we have used the CarSharer that is linked to SPControl, and traced through the membership role to the account that holds the outstanding balance on membership fees, and checked that the balance is greater than zero.

The second requirement is sensibly mapped onto the createPaymentSchedule operation. Because dates are not basic types, an operation will have to be available on the date type to check that it is before another date, using the other date as the argument (this operation must return a boolean value). The constraint becomes:

```
context SPControl::createPaymentSchedule(startDate, regularDate, amount, frequency)
    pre: startDate.before(Policy.startDate).
```

The third requirement is an invariant on CarSharer, which can be expressed as 'the renewal date of a policy must fall before the start date of any later policy', and this becomes in OCL a set operation:

```
context CarSharer
    inv: Policy->forall( p1,p2 | p1<>p2 and p1.startDate.before(p2.startDate)
            implies p1.renewalDate.before(p2.startDate)).
```

MORE DETAIL

In considering this, we need to place a constraint on Policy that makes sure that renewal dates fall after start dates, thus:

```
context Policy
    inv: startDate.before(renewalDate).
```

On further consideration of the design of the system, the invariant on CarSharer is ensured by placing a pre-condition on the createPolicy operation that can be expressed as:

```
context SPControl::createPolicy(startDate, renewalDate)
    pre: CarSharer.Policy -> forall( p | p.startDate.before(self.startDate)
            implies p.renewalDate.before(self.startDate)).
```

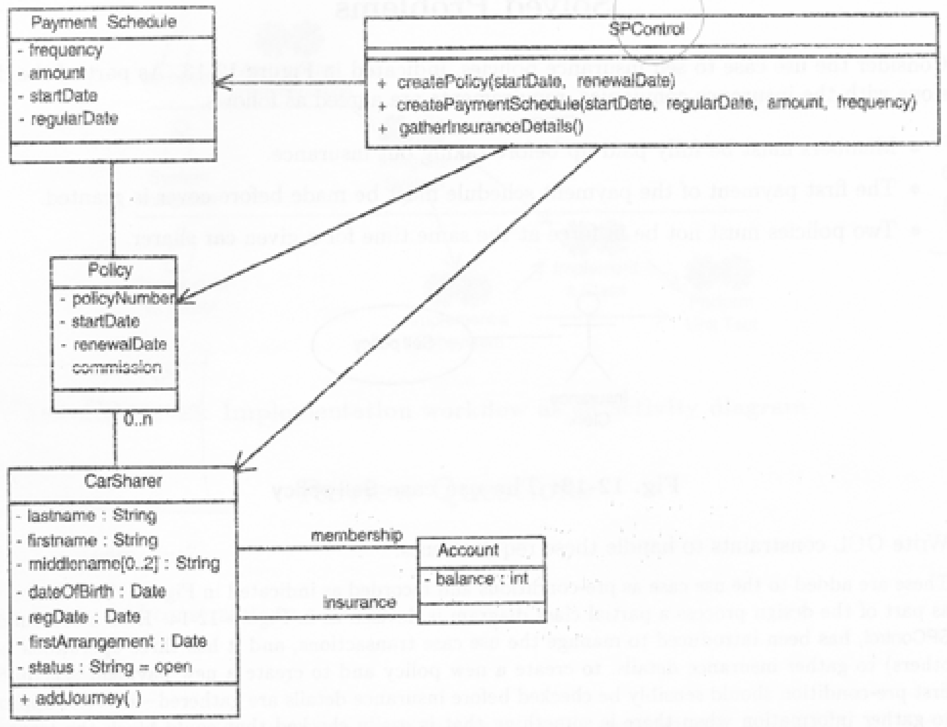


Fig. 12-14: Partial class diagram to implement the use case Sell policy

The translation of the pre-conditions into code would depend very much on the programming language chosen. However, in each case the operation would inevitably begin with some simple tests to check that the pre-conditions were true. If a pre-condition is false, then the operation would either exit with some return value set, or raise an exception.

X **12.2** Consider the use case Match car sharers. The requirements for sharing include the condition that to create a sharing agreement, there must be at least one driver, and at least two people in the agreement. Write OCL constraints to handle these requirements.

Available
List of journeys & CarSharers

This would be a post-condition of the use case Match car sharers. We might then design a control class MCSControl with an operation to choose sharers for a particular journey and then create a sharing agreement. The object model is indicated in Figure 12-15. The requirement would mean that the chooseSharers operation must return a list of car sharers among whom is a driver. We clearly need a means of checking if a car sharer can drive, and we might elect to put a canDrive attribute in the object. The requirement can be expressed as a post-condition:

Sharer

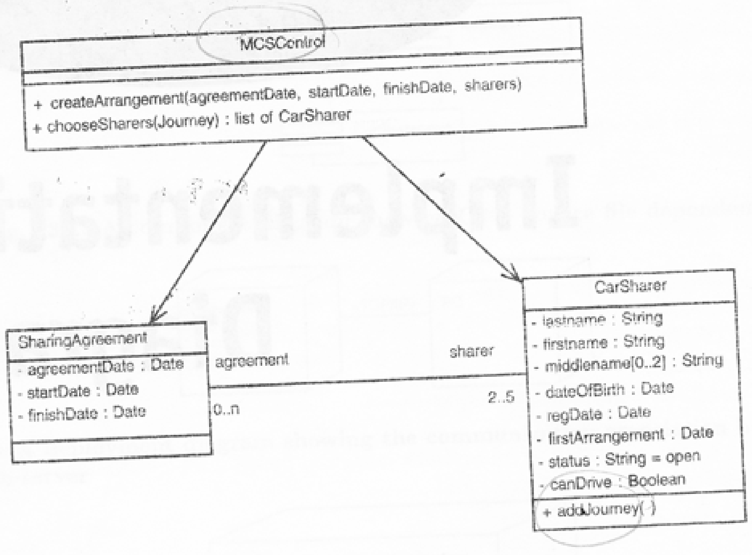
```
context MCSControl::chooseSharers(journey): list of CarSharer
BODY    post: self->exists (x | x.canDrive) and (self->size) >=2.
```

This would then become a pre-condition on the createArrangement operation, expressed thus:

```
→   context MCSControl::createArrangement(agreementDate, startDate, finishDate, sharers)
      PRE ~~post~~:sharers -> exists (x | x.canDrive) and (sharers ->size) >=2.
```

Meeting these pre-conditions and post-conditions would be instrumental in implementing the invariant on a sharing agreement as follows:

```
    context SharingAgreement
→       inv: CarSharer->exists (x| x.canDrive) and (CarSharers->size) >=2.
```



Generel

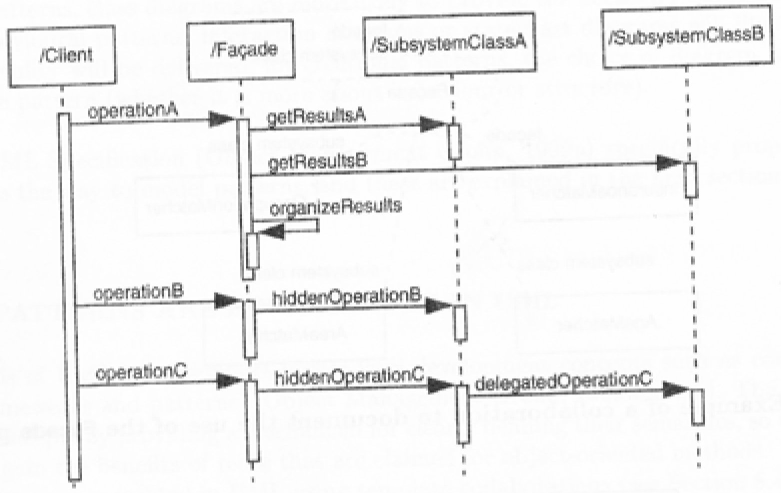


Fig. 16-6: Typical interactions using the Façade pattern

X

SUB SYS Detail

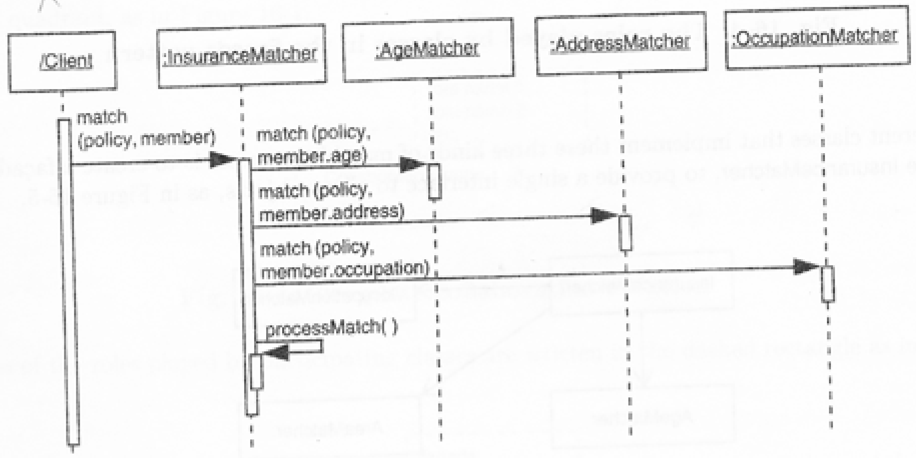
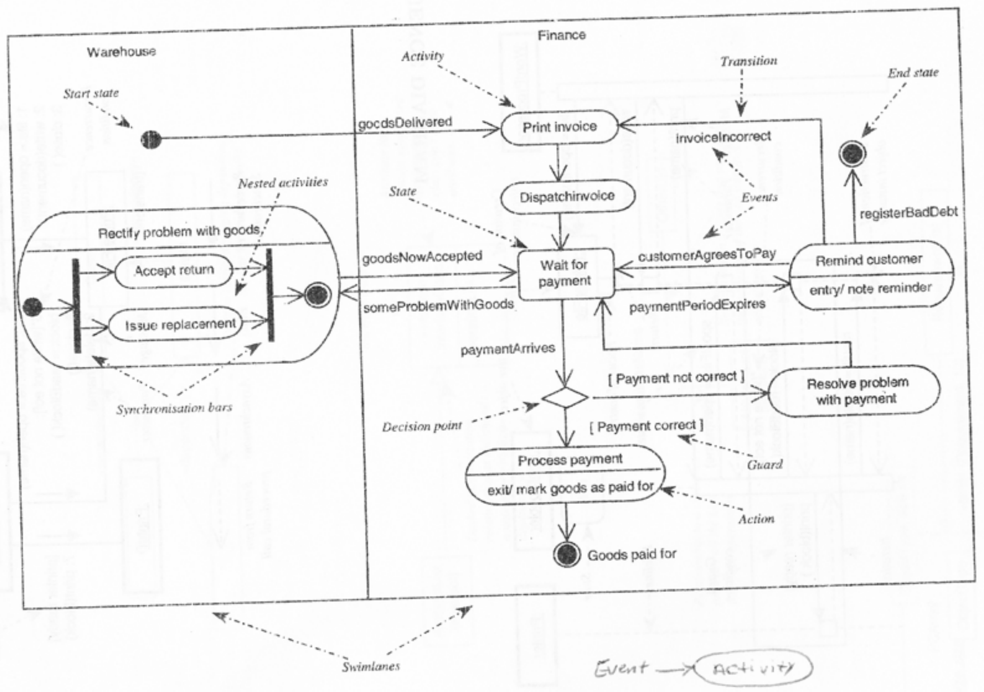


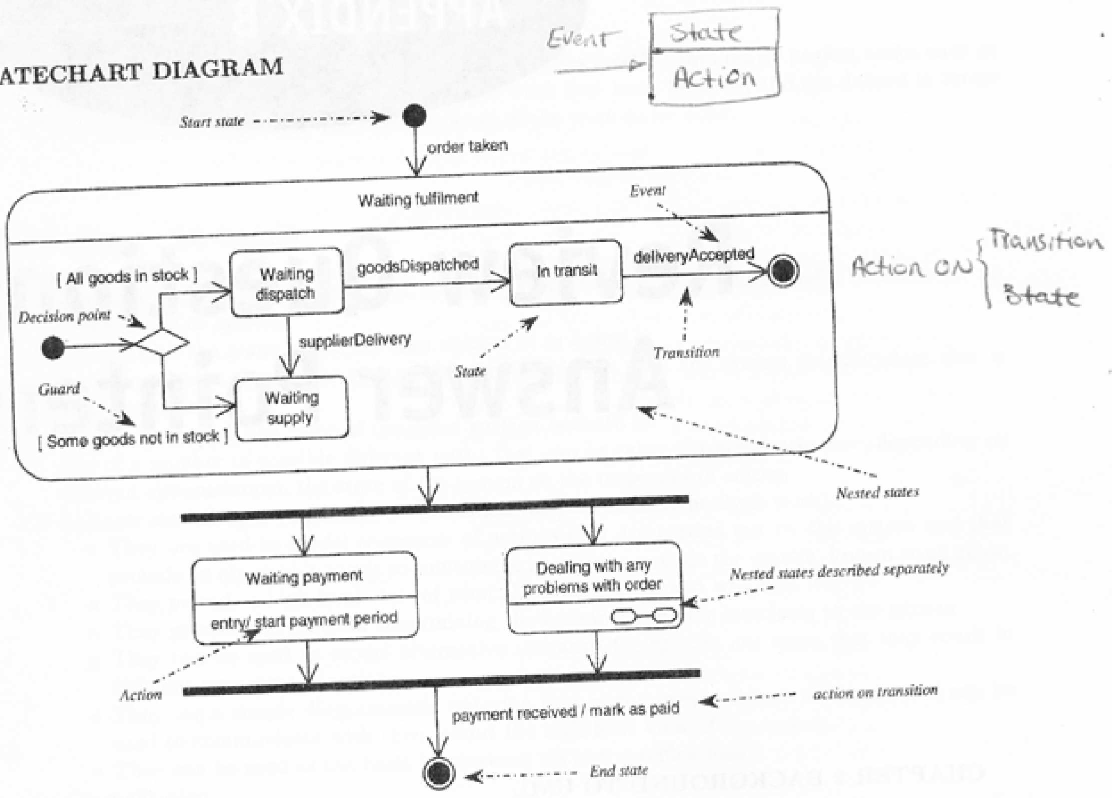
Fig. 16-7: Sequence diagram for the Façade pattern

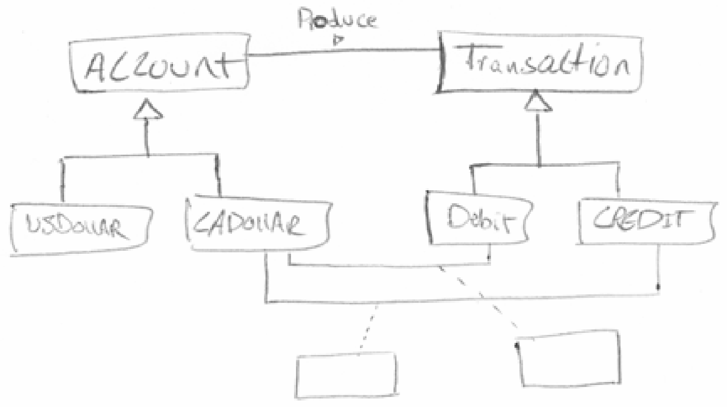
Diagram and design sequence
1-usecase (processes)
2-Activity Diagram (business flow)
3-class diagram (static View classes and database structure)
4-Sequence Diagram (Dynamic View)
5-state chart diagram

ACTIVITY DIAGRAM



STATECHART DIAGRAM





⇓ DB (IF ABSTRACT OR NOT?)