Improving Integration and Consistency in the OMT Methodology

Sinead Masterson B.Sc.

Submitted for the Award of
Master of Science

Dublin City University
School of Computer Applications
Professor J. A. Moynihan

January 1996

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Science, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Sinead Masterson.
10th January 1996.

Dedicated to my
beloved sister

Karen

(1969 - 1995)

"I balanced all, brought all to mind,
The years to come seemed waste of breath,
A waste of breath the years behind
In balance with this life, this death."

W.B. Yeats

Acknowledgements

Special thanks to my supervisor Professor Tony Moynihan for his invaluable ideas, suggestions and encouragement throughout my research, and for his benevolent supply of orange juice on our "coffee mornings".

Thanks also to Dr. David Sinclair for his sincere interest in my research, and for his constructive comments and helpful suggestions.

Endless thanks to my family for their support and encouragement during my poverty-stricken student days, and for their masterfully feigned understanding of OMT. Sincere apologies for boring you all to tears.

Thanks indeed to my "rich" friends who bought me drinks, and insisted I didn't return the favour. Pay back time is nigh !

And finally, extra special thanks to my partner in crime, Derek Doran, a truly exceptional friend and colleague. Someone who equals my noble appetite for truth, knowledge and ... cinema !!!, co-crusader in the quest for "The Ultimate Movie", and acclaimed inventor of the immortal phrase "Friday, 2:00 p.m., Savoy 1".

Table of Contents

Table of Figures

Abstract

Object Modeling Technique (OMT) by Rumbaugh et al, is a methodology for the analysis and design of object-oriented systems. The primary strength of the OMT methodology is that it allows a complete specification of a system, covering it's static structure, dynamic behaviour and functionality. It models each system from three related but different viewpoints - the functional model specifies what happens, the dynamic model specifies when it happens and the object model describes what it happens to. Each model uses a concise and understandable notation, and thus all three models can be appreciated to a large extent on their own.

However, as each of these models represents a different view of the system, they need to be integrated together in order to get the overall picture. Herein lies the crux of the problem as the diversity present in the OMT methodology is also paradoxically it's major weakness. Each model is developed more or less independently, and the inter-relationships between the three models are not explicit, resulting in a lack of integration, and subsequent lack of consistency between the object, dynamic and functional models.

The purpose of my research is two-fold. Firstly to discover and document the cause of this lack of integration and consistency, namely the apparent weakness of the functional model, caused by the unsuitability of using data flow diagrams to model the functionality of an object-oriented system; and the inadequate inter-model relationships, which are poorly defined and are neither supported by formal steps in the methodology nor a comprehensive illustrated example.

The second part of my research involves developing solutions to alleviate this lack of integration and consistency, in the form of a somewhat revised functional model which embraces the object-oriented paradigm; and improved inter-model relationships which extend the OMT methodology, to incorporate guidelines for constructing an integrated analysis model, as well as guidelines for checking the completed model for consistency.

The layout of the thesis is as follows. The first chapter introduces the OMT methodology by describing the object, dynamic and functional models as proposed by Rumbaugh. The second chapter documents the two major factors which are responsible for the unsatisfactory level of integration and consistency within OMT, while the third chapter proposes detailed solutions to each of these identified problems in the form of a revised functional model and inter-model relationships. The fourth chapter comprises a case study detailing how to construct the revised functional model, and how to integrate the three OMT models. And finally, the fifth chapter concludes the thesis by discussing the strengths and weaknesses of the revised OMT approach, as well as reviewing current research in OMT, and identifying some areas of potential future research for the methodology.

Chapter 1

Introduction to OMT

1.1 Overview

The Object Modeling Technique (OMT) by Rumbaugh, models a system from three related but different viewpoints, each of which captures important aspects of the system. The *object model* represents the static, structural aspects, the *dynamic model* represents the temporal, behavioural aspects, and the *functional model* represents the transformational aspects, of a system.

The three models separate a system into orthogonal views, and although each model can be understood by itself to a large extent, the models are not completely independent, as each model describes one aspect of the system but contains references to the other models. The object model describes data structures that the dynamic and functional models operate on, and operations in the object model correspond to events in the dynamic model and functions in the functional model.

All three models are necessary for a full understanding of a problem, although different problems place different emphasis on the three kinds of models. The balance of importance among the models varies according to the kind of application, for example, non-interactive programs have a trivial dynamic model and a large functional model, whereas databases often have a trivial functional model, since their purpose is to store and organize data, not to transform it.

1.2 The Object Model

The object model forms the backbone of the OMT methodology as it is responsible for capturing the static structure of objects, and their attributes, operations, and inter-relationships.

1.2.1 Object Diagrams

The object model is represented graphically using *object diagrams*, which provide a formal graphic notation for modeling objects, classes and their relationships to one another. It is often said that a picture speaks a thousand words. Object diagrams do not deviate from this rule, as they are concise, precise diagrams, which are as easy to formulate as they are to understand. There are two types of object diagrams : *class diagrams* which describe classes, and *instance diagrams* which describe objects.

1.2.2 Classes and Objects

Classes are represented on class diagrams. A class diagram is a template. It provides a schema of attributes and operations, which each object instance of that class must conform to. The OMT symbol for a class is a box with up to three sections. The first section contains the name of the class in boldface, the second section contains the attributes of that class, (the type and default value are optional), and the third section contains the operation of that class, (the argument list and result type are also optional).

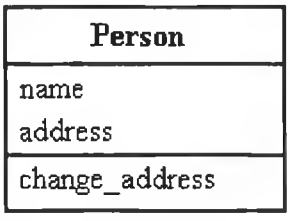


Fig 1.1 Class Diagram

Objects are represented on instance diagrams. An instance diagram describes a single instance of a particular class, by assigning values to the attributes of the class. The OMT symbol for an object instance is a rounded box with the class name in parenthesis and boldface at the top of the box, and the values assigned to the attributes of the class listed beneath it.

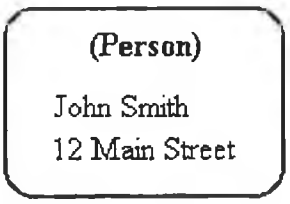


Fig 1.2 Instance Diagram

1.2.3 Links and Associations

Links and associations are the means for establishing relationships among objects and classes respectively. An association describes the relationship between two or more classes and a link is the instance of that association. Thus an association describes a set of potential links in the same way as a class describes a set of potential objects. All the links in an association connect objects from the same class, but the link is not part of either object by itself, but depends on both objects together. Using OMT an association is represented by a line between classes, and a link is represented by a line between objects, with the association name written in italics above the line.

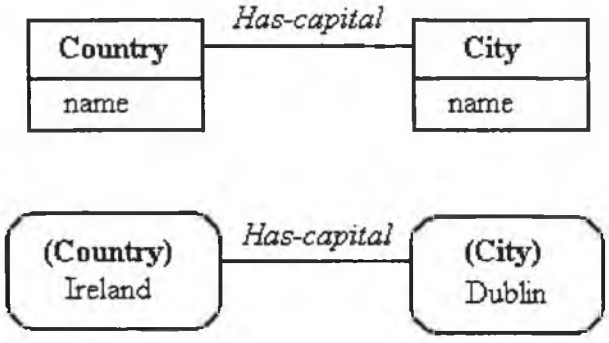
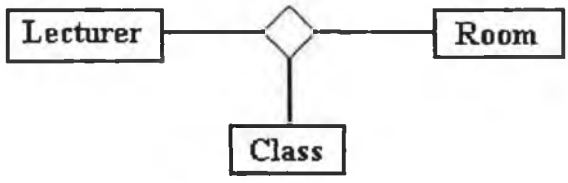


Fig 1.3 Binary Association

In addition to binary associations, ternary associations can also exist. These associations cannot be expressed as binary associations without the loss of information. For example, a lecturer may teach many classes of students in the same room, a lecturer may also teach the same class in many different rooms, and each class of students may have many lecturers. The OMT symbol for a ternary association and n-ary associations is a diamond with lines connecting the related classes. The name of the association is written next to the diamond, but can be left unnamed if it can be easily deduced from the classes it connects.



Lecturer	Class	Room
John Smith	CA3	TG01
John Smith	CA3	CG04
John Smith	CA4	CG04
Sarah Miles	CA4	CG12
Sarah Miles	CA3	TG01

Fig 1.4 Ternary Association

1.2.3.1 Multiplicity

Multiplicity specifies how many instances of one class may relate to a single instance of an associated class. OMT has many symbols which denote different kinds of multiplicity, but all symbols are placed at the end of the association line. A line without multiplicity symbols indicates a one-to-one association. A solid ball denotes many (i.e zero or more), a hollow ball denotes optional (i.e. zero or one), but in general multiplicity is specified with a number or a set of intervals, such as 1 (exactly 1), 1+ (one or more), 3-5 (three to five inclusive), and 2,4,5 (two, four or five only).

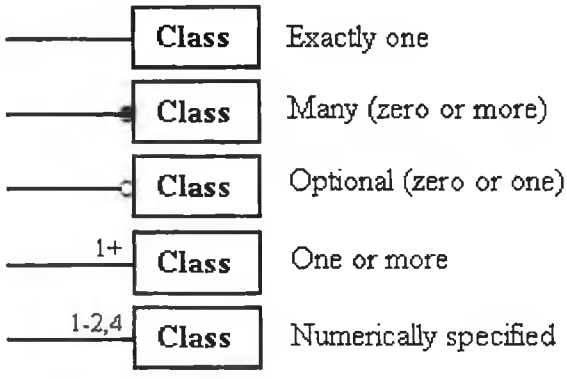


Fig 1.5 Multiplicity Symbols

For example, each athlete can run in more than one race, and each race can have many athletes participating in it.

Athlete	Race
Jane White	100m
Jane White	200m
Greta Halpin	200m
Greta Halpin	400m
Deirdre Sinclair	100m
Karen Smith	400m

Fig 1.6 Multiplicity

1.2.3.2 Link Attributes

In the same way that an attribute is a property of all objects in a class, a link attribute is a property of all links in an association. Remember that the link is not part of either object, but depends on both objects together, hence the link attribute is a property of the link and cannot be attached to either object. The OMT notation for a link attribute is a box attached to the association by a loop, with one or more link attributes in the second region of the box. Associations with many-to-many multiplicity often have link attributes. For example, each company employs many people, and each person can work for many companies. As a result of being employed by a company each person receives a salary and a job title, thus salary and job title are invariably attributes of the link *works-for*.

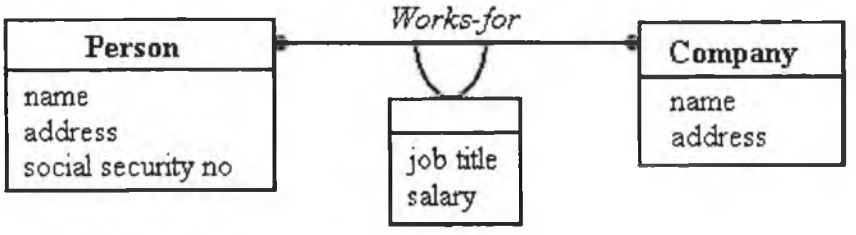


Fig 1.7 Link Attributes

1.2.3.3 Associations as Classes

The notation for a link attribute is a special case of an association which was modeled as a class. The association class can have a name and operations in addition to attributes. Each link in the association is an instance of the class, thus emphasising the similarites between links and objects, associations and classes.

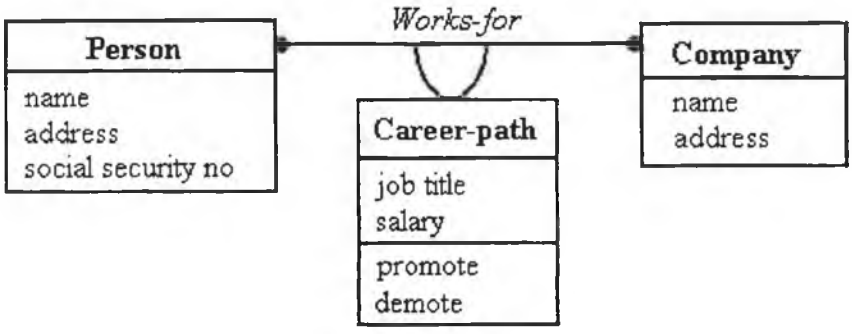


Fig 1.8 Association as a Class

1.2.3.4 Role Names

A role name is a name that uniquely identifies one end of an association. For example, using the *works-for* association between the Person and Company classes, the person assumes the role of ***employee***, and the company assumes the role of ***employer***. In OMT a role name is written next to the class which plays the role on the association. Role names are necessary between two objects of the same class. For example, ***boss*** and ***worker***, are both part of the *manages* association, which links two instances of the person class.

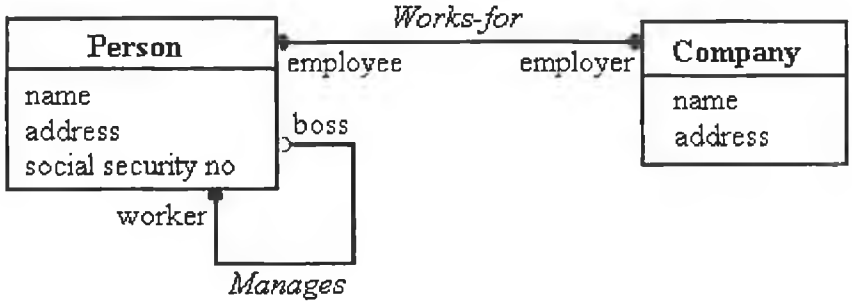


Fig 1.9 Role Names

1.2.3.5 Ordering

When dealing with associations which have one-to-many, or many-to-many multiplicity, there exists a set of objects on the many end of the association. It may be necessary to explicitly order this set of objects, for example, each football team consists of many players, each with their own numbered shirt. The OMT notation for ordering consists of putting "{ordered}" next to the multiplicity dot.

Fig 1.10 Ordering

1.2.3.6 Qualification

One-to-many and many-to-many associations may be qualified by using a *qualifier*, which distinguishes among a set of objects on the many end of the association. For example, a directory has many files, but a file can only belong to one directory. However a directory and a file name can be combined to specify a unique file, thus *file name* is the qualifier. Qualification usually reduces multiplicity form many to one, but not always. The OMT symbol for a qualifier is a small box identifying the qualifier, at the end of the association line beside the class it qualifies.

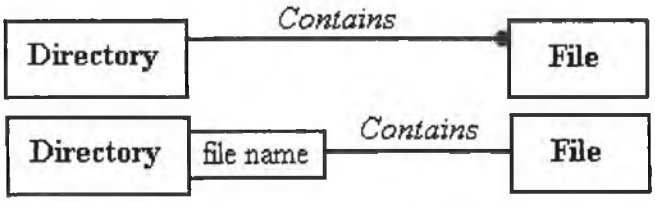


Fig 1.11 Qualification

1.2.3.7 Aggregration

Aggregation is a special form of association, representing the *a-part-of* relationship between objects, and reduces complexity by treating many objects as one object [Odell, '94]. The OMT notation for aggregation consists of an association line, with a small diamond drawn at the end of the line, near the aggregate object. For example, a book consists of many chapters.

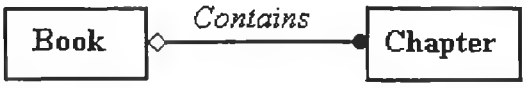


Fig 1.12 Aggregation

Aggregation possesses properties of transitivity (i.e. if A is part of B and B is part of C, then A is part of C) and anti-symmetry (i.e. if A is part of B, then B is not part of A) [Blaha, '93]. Aggregation can have an arbitrary number of levels, in which case it is often easier to draw multilevel aggregation as a tree structure.

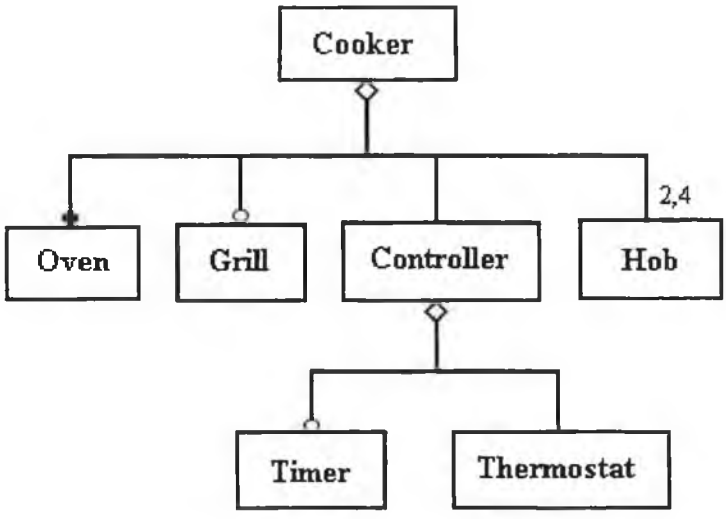


Fig 1.13 Multilevel Aggregation

1.2.4 Generalization and Inheritance

Generalization represents the *is-a* relationship between classes. Classes at the bottom of the generalization tree, are called subclasses and are more specific than classes at the top of the tree, which are called superclasses. Each instance of a subclass is an instance of its superclass as well. A subclass inherits attributes and operations from at least one superclass, and then adds its own specific attributes and operations, or overrides the superclasses' operations with versions of its own. The OMT notation for generalization is a triangle connecting the superclass to its subclasses, optionally the *discriminator*, which is the property of the object which is being abstracted, can be written next to the triangle.

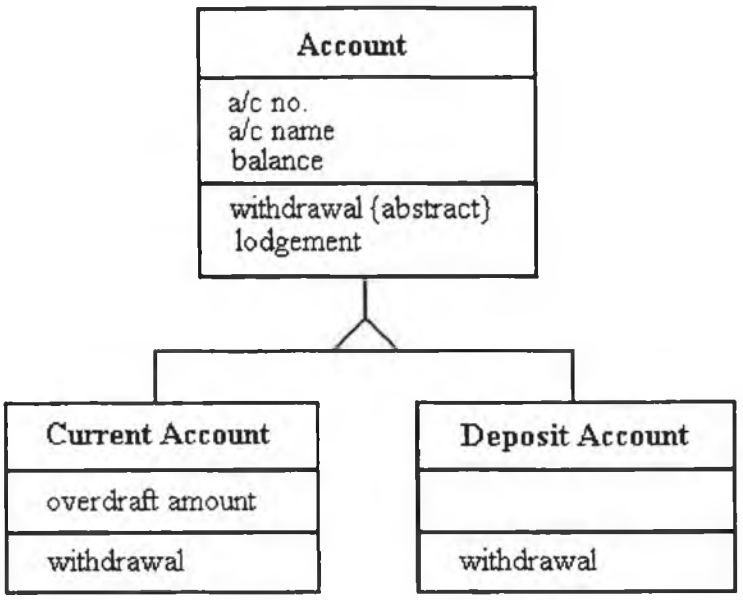


Fig 1.14 Generalisation and Inheritance

Multiple Inheritance permits a class to have more than one superclass, and thus the subclass inherits attributes and operations from all its superclasses. The OMT notation for multiple inheritance is the same as for single inheritance.

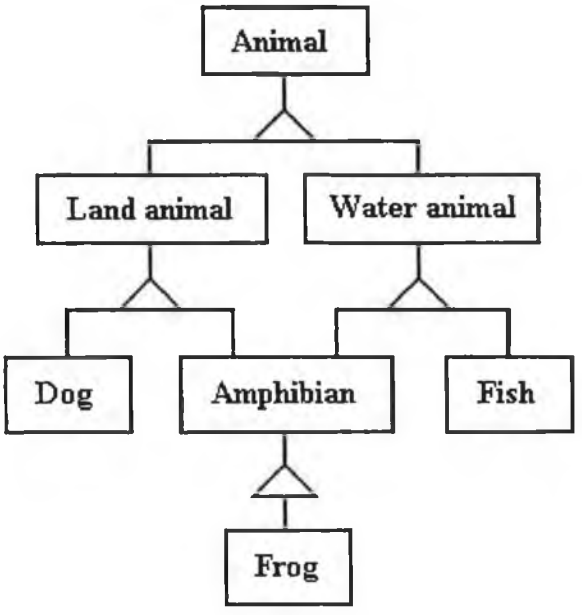


Fig 1.15 Multiple Inheritance

1.2.5 Constructing the Object Model

To construct the object model it is necessary to study the problem statement and to extract the object classes embedded within the statement; then to recognise the associations between the selected classes; then to identify the attributes of an object belonging to each particular class; then to organize the classes using inheritance (if necessary); then to provide operations to access and update each of the attributes; and finally to build the object diagram which illustrates the identified classes, associations, attributes and operations.

Problem Statement : Simple Order Processing System

The company takes orders from customers, and attempts to fill those orders with on-hand stock. Before the order can be filled, an order authorization request is sent to the stores giving details of the current order. If sufficient stock exists to fill the order, authorization to prepare the order is granted, the stock is removed from the stores and used to fill the order, the company's stock levels are updated to reflect this depletion, the order is shipped, and an invoice is issued to the customer. The customer either part-pays the invoice in instalments, or pays the balance in full.

However, if there is insufficient stock to fill the order, authorization to prepare the order is refused, and a standard quantity of the out-of-stock item is reorderd from the supplier. A standard quantity would be a multiple of the safety stock level of that particular item, and it is assumed that it would always be sufficient to fill any outstanding order. In addition, if the stock level of a particular item falls below the safety stock level for that item, then a standard quantity of that item is reordered. When a delivery is received from the supplier, the company's stock levels are updated to reflect this replenishment.

This problem statement is refered to further on in the chapter for both the dynamic model example and the functional model example.

1.2.5.1 Identify Object Classes

Using Rumbaugh's guidelines for choosing object classes from the problem statement and for selectively retaining appropriate classes and discarding unnecessary ones, four distinct classes emerge.

Obviously there is an *order* class and an *invoice* class, however, *stockitem* is also a class, which contains both stock level and safety level data, as well as pricing information for each item of stock. A more subtle class is the *company* itself, whose data is the orders, invoices and stock. It may seem that both *customer* and *supplier* should also be classes, but they merely provide the input and output of the system, and within the context of this example, they do not process any data they provide.

1.2.5.2 Identify Associations

Associations are references from one class to another, or any dependency between two or more classes. Thus, associations exist between the *company* class and the *order* class as the company takes orders and places them in the orders file, between the *company* class and the *invoice* class as the company issues an invoice for each order taken, and between the *company* class and the *stockitem* class as the company manages the stock levels, the safety stock levels and the pricing of each particular item. In addition there is also an association between the *order* class and the *invoice* class, as each order has one and only one invoice associated with it, and similarily, each invoice has one and only one order associated with it. This is due to the fact that the order number is identical to the invoice number on the invoice that relates to the order, and vice versa.

1.2.5.3 Identify Attributes

Each order has an order number *(order_no)*, a customer name *(cust_name)* and address *(cust_addr)*, the item ordered *(item)* and the quantity of that item ordered *(qty)*, and some status variables; *authorized* which indicates whether the order has been authorized for preparation and *status* which reflects the current status of the order (i.e. pending, outstanding or shipping).

Each invoice has an invoice number *(invoice_no)* which is identical to the order number on the order that it relates to, a customer name *(cust_name)* and address *(cust_addr)*, the item *(item)* and quantity *(qty)* on the invoice, the balance due *(balance)*, and an invoice status variable *(status)* which indicates whether the invoice is unpaid, paid or part-paid.

Each stockitem has a stock level *(stock_level)* which indicates the quantity of that item on-hand, a safety stock level *(safety_level)* which indicates the minimum stock level per item and below which the item must be reordered, and a price *(price)*.

Each company has a file of orders *(orders_file)*, a file of invoices *(invoices_file)*, and a file of stockitems *(stock_file)*. As each of these files contain many orders, invoices and stockitems, a company also has a means of accessing each individual order *(curr_order)*, invoice *(curr_invoice)* and stockitem *(curr_stockitem)*.

1.2.5.4 Identify Operations

The task of identifying operations at this early stage of analysis is limited to those obvious operations required to access and update the attributes of each individual object of each given class.

The order class has an operation to access each attribute, and an operation to update the authorization and status of the order. It is assumed that once an order is placed, the order number, customer name and address, item and quantity is not changed.

This order information is copied to the invoice relating to the order, and as there is a one-to-one relationship between the order and the invoice relating to the order, there is no need to replicate these operations for the invoice class. The invoice class hence only has operations to access and update the balance on the invoice and the status of the invoice.

The stock class has an operation to access and update each of its attributes, and two operations to update the stock level (one to decrease the stock level by the quantity of the current order, and one to increase the stock level by the quantity of the current delivery).

The company class has operations to access and update each of its files.

1.2.5.5 Build the Object Diagram

With the object classes, associations between classes, attributes and trivial operations identified thus far, an initial object model can be constructed from this information.

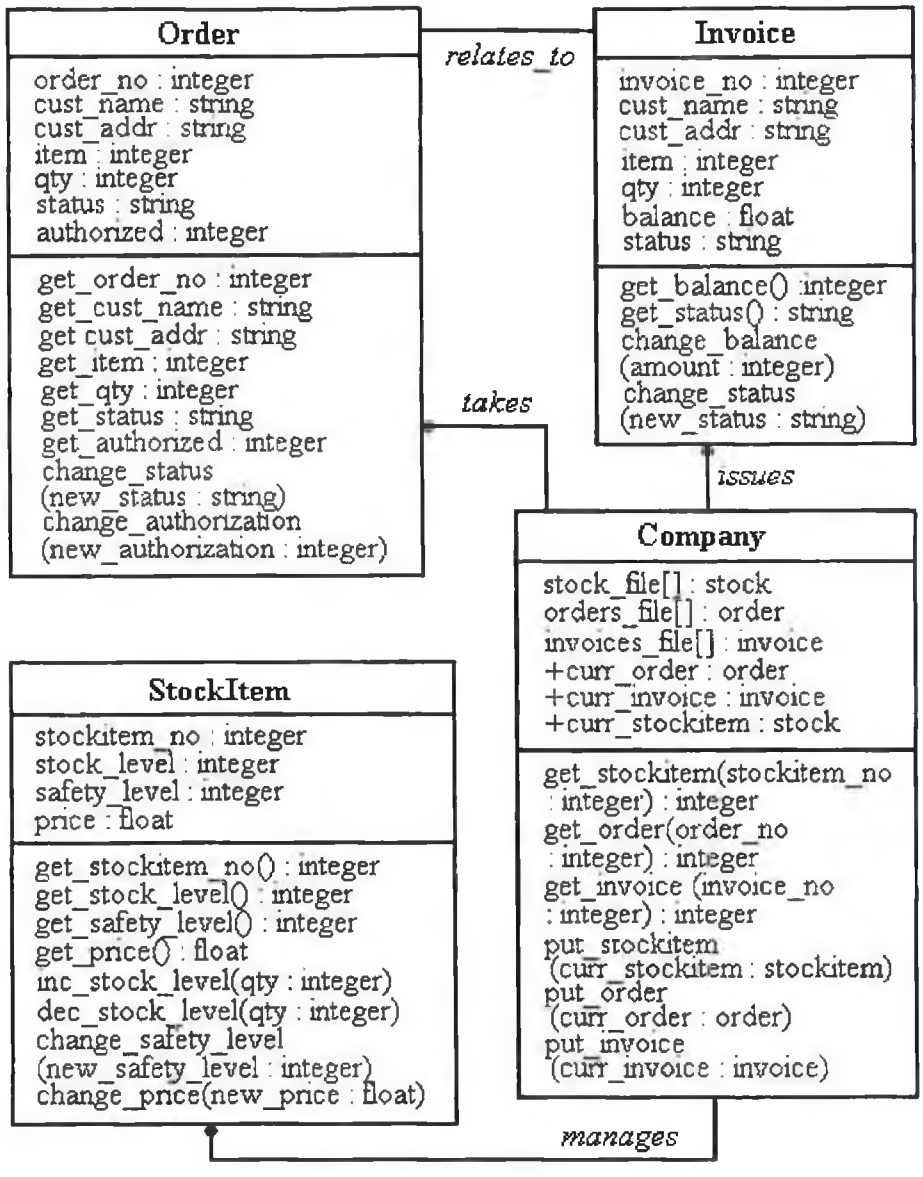


Fig 1.16 Object Model

Note : The eternal visibility of an attribute or operation can be private (-), public (+) or protected (#), according to the second-generation OMT method, briefly outlined by Rumbaugh [Rumbaugh, '95-1]. In this example, all attributes are private, except for the three public attributes of the company class, (identified by the "+" sign), and all operations are public.

1.3 The Dynamic Model

The dynamic model is responsible for representing control information, such as sequences of events, states, and operations that occur within a system of objects. While the object model describes the structure of the objects and the nature of their inter-relationships, the dynamic model is concerned with changes to the objects and their relationships over time.

1.3.1 State Diagrams

The dynamic model is represented graphically using multiple state diagrams, one for each class with non-trivial dynamic behaviour. State diagrams are based on Harel's statecharts [Harel, '87], which extend basic state transition diagrams by the addition of depth *(or-states)*, orthogonality *(and-states)* and broadcast communication *(events)*. Each state diagram shows the pattern of events, states, and state transitions permitted in a system for one class of objects. A state diagram is a template which describes a set of sequences, in the same way that a class is a template which describes a set of objects. The state diagrams for the various classes interact with each other via shared events.

1.3.2 Events and States

Events are stimuli from one object to another while states are abstractions of the attribute values and links of an object. Events represent points in time, they are instantaneous, whereas states represent intervals of time. The time duration which an object spends in a state, depends on the time interval between the event which placed the object in that state and the event which removed the object from the state. The response of an object to an event depends on the state of the receiving object, and may include a change of state, or the sending of another event.

State diagrams relate events and states in a graph whose nodes are states, and whose arcs are transitions between states, caused by events. A state is drawn as a rounded box, and a transition as an arc between two states, labelled by the event causing the transition. Thus if an object is in a state, and the event on one of its transitions occurs, the transition fires, and the object enters the new state. An event can also carry data values called *attributes*, which are shown in parenthesis after the event name.

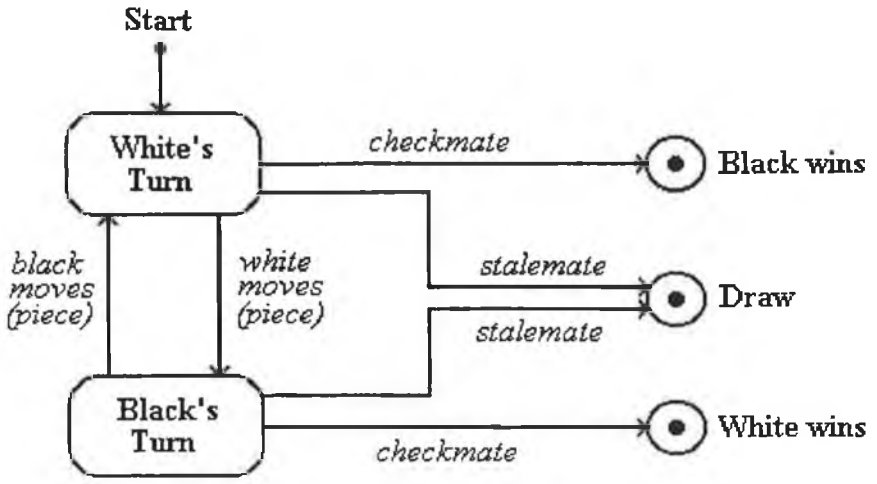


Fig 1.17 State diagram for a Chess Game

1.3.3 Conditions

A condition is a boolean function and can be used as a *guard* on a transition. Thus a guarded transition will fire only if the event on the transition occurs, and the condition is true. Unlike an event, which has no time duration, a condition is valid over an interval of time. For example, in the diagram below, a system can be in one of two states *Power Off* or *Power On*, but it can only enter the *Power On* state provided the system has been off for at least a minute.

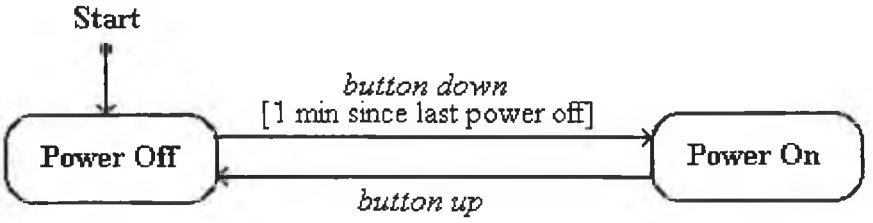


Fig 1.18 Conditions on Transitions

1.3.4 Operations

Operations are triggered by events. There are two types of operation : actions and activities. Actions are attached to transitions and are performed in response to the corresponding event, while activities are attached to states and are performed in response to being in a state.

1.3.4.1 Actions

An action is associated with an event. It is an instantaneous operation, or an operation whose duration is insignificant with respect to the granularity of real events under consideration, and thus it may as well be instantaneous. The OMT notation for an action is a forward slash "/" followed by the action name, placed after the event name on a transition between states.

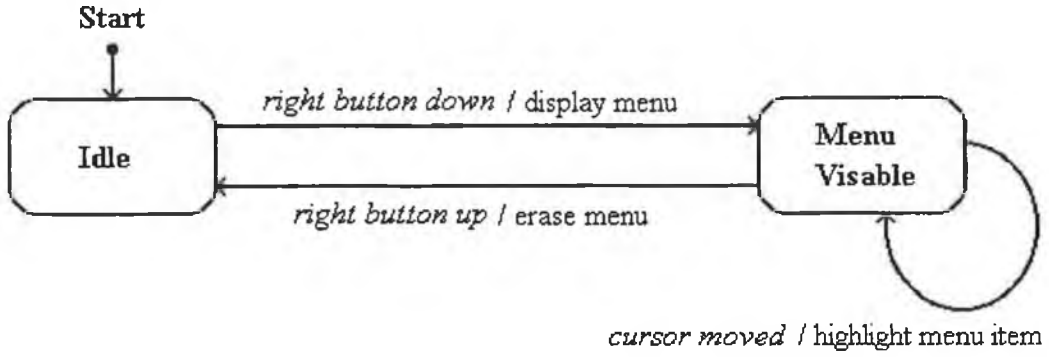


Fig 1.19 Actions on Transitions

1.3.4.1.1 Entry and Exit Actions

Entry actions are the first operation performed after a state is entered, and exit actions are the last operation performed before a state is exited. Associating an action with entry to a state, is equivalent to placing that action on each transition to that state. This form of notation for actions is particularly useful when several transitions enter a state, each one with the same action, thus the action can be removed from each transition and placed within the state as an entry action. The order of execution of actions is as follows : actions on the incoming transition, entry actions, exit actions, actions on the outgoing transition. The OMT notation for entry and exit actions is the word *entry* or *exit* within the state box respectively, followed by a forward slash and the action name.

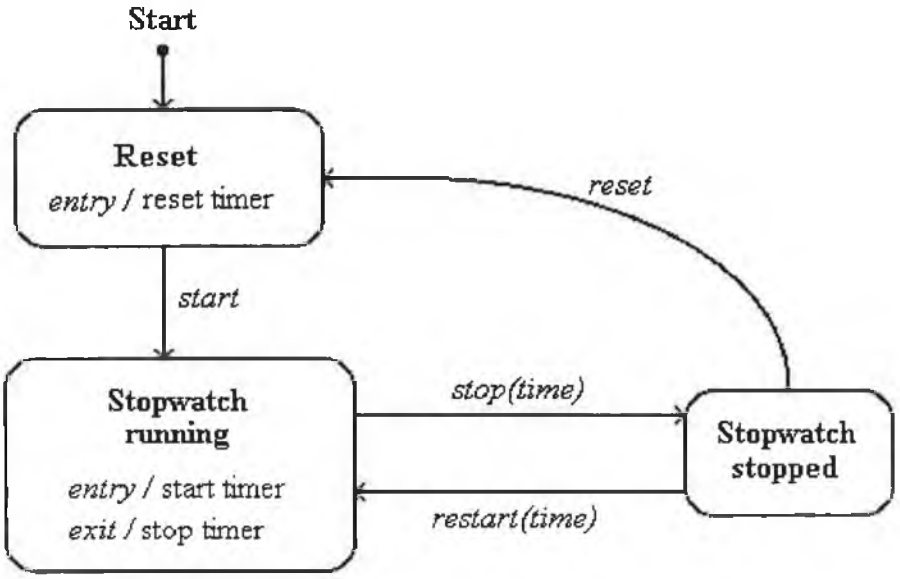


Fig 1.20 Entry and Exit Actions

1.3.4.1.2 Internal Actions

It is also possible for an event to occur which does not cause a transition from the current state. Such an event is listed inside the state box followed by a forward slash and the name of the action to be performed in response to this event. This action is an internal action. When the event associated with this action occurs, the action is executed, but not the entry and exit actions for the state. For example, *pause* and *unpause* are internal actions.

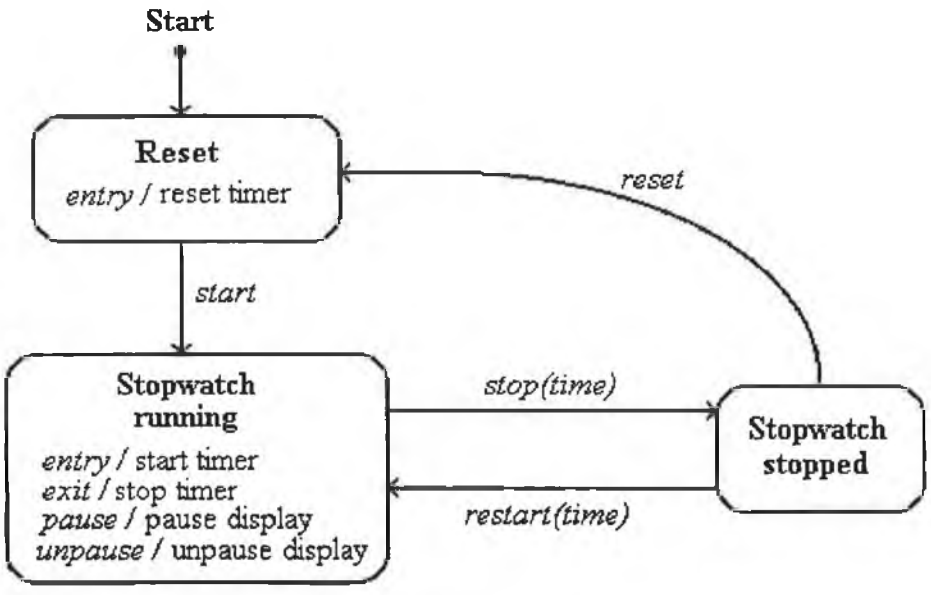


Fig 1.21 Internal Actions

1.3.4.1.3 Actions Sending Events

A system of objects interacts by sending and receiving events. An object can perform the action of sending an event to another object or even a set of objects. The action "*send E(attributes)*" sends the event *E* with given attributes to a set of objects, and thus any object with a transition on the event *E* can accept the event concurrently.

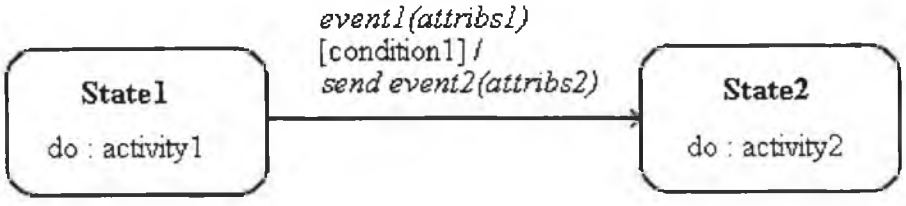


Fig 1.22 Actions Sending Events
(First Notation)

Another notation for sending an event from one object to another is a dotted line from a transition to an object, labelled with the event name. Thus, when the transition fires the event is sent to the object.

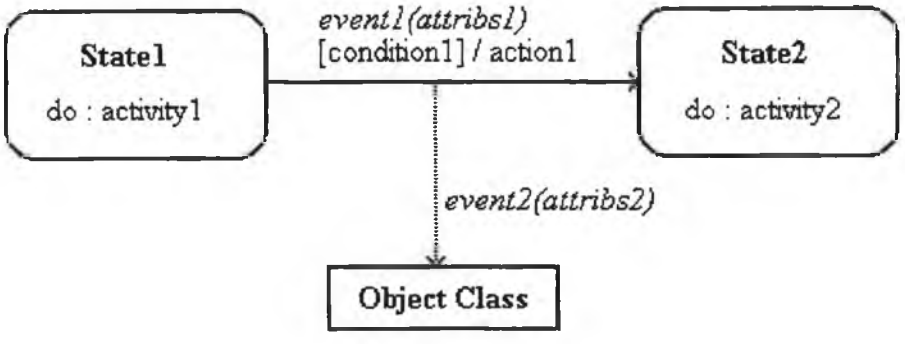


Fig 1.23 Actions Sending Events
(Second Notation)

1.3.4.2 Activities

An activity is associated with a state. Unlike an action it is not instaneous, but it takes time to complete. Activities include both sequential activities and continuous activities. The same state diagram notation is used for both types of activity, *"do: A"* within a state box indicates that the activity *A* starts on entry to the state and stops on exit from the state.

1.3.4.2.1 Sequential Activities

Sequential activities progress until completed, or interrupted and terminated prematurely by an event, thus causing a transition from the state. A sequential activity starts once the state is entered and stops on exit, or when completed if is not interrupted by an event. In the diagram below, which depicts an application program, *run program* is a sequential activity which runs to completion, or until it is terminated prematurely by an interrupt.

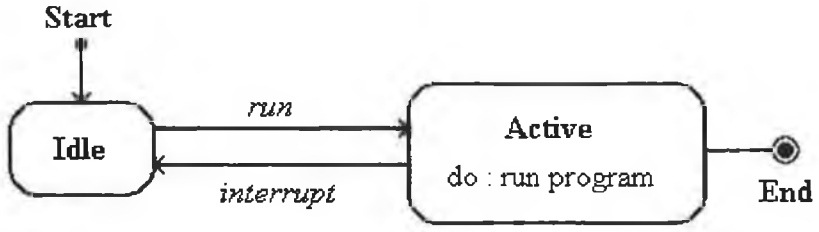


Fig 1.24 Sequential Activities

1.3.4.2.2 Continuous Activities

Unlike sequential activities, continuous activities do not complete, instead they persist until they are terminated by an event, thus causing a transition from the state. A continuous activity starts once the state is entered and stops on exit. In the diagram below, which depicts an alarm clock, both *display time* and *sound alarm* are continuous activities, but *set alarm time* is a sequential activity which completes when the hours and minutes of the alarm time have been set.

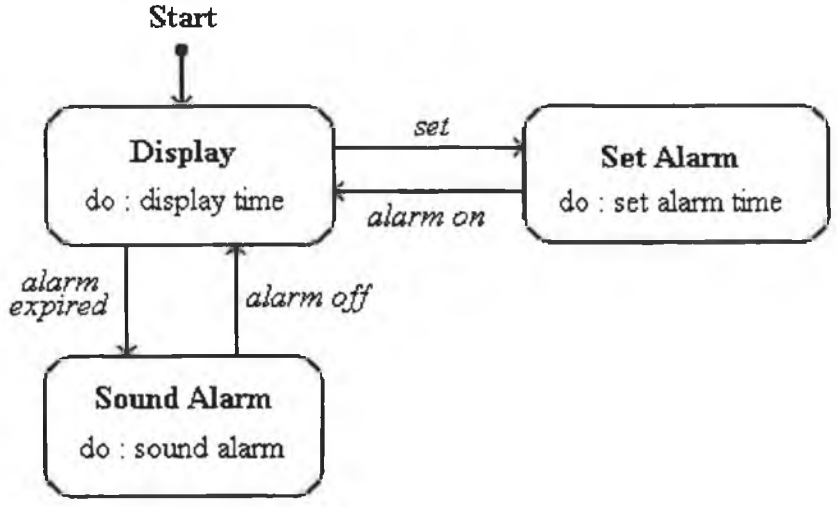


Fig 1.25 Continuous Activities

1.3.4.2.3 Automatic Transitions

A transition without an event name is an automatic transition, which fires as soon as the sequential activity associated with the state has completed. If there is no activity associated with the state, then the transition fires as soon as the state is entered, but the entry and exit actions (if any) are always performed. A state can have more than one automatic transition, but each such transition should have a guard condition, such that only one of the transitions can have a valid guard condition at any one time. In the example below, when the sequential activity *dispense item and change* completes, if the vending machine is not empty, the ***Ready*** state is entered, otherwise the vending machine is empty and the ***Not ready*** state is entered.

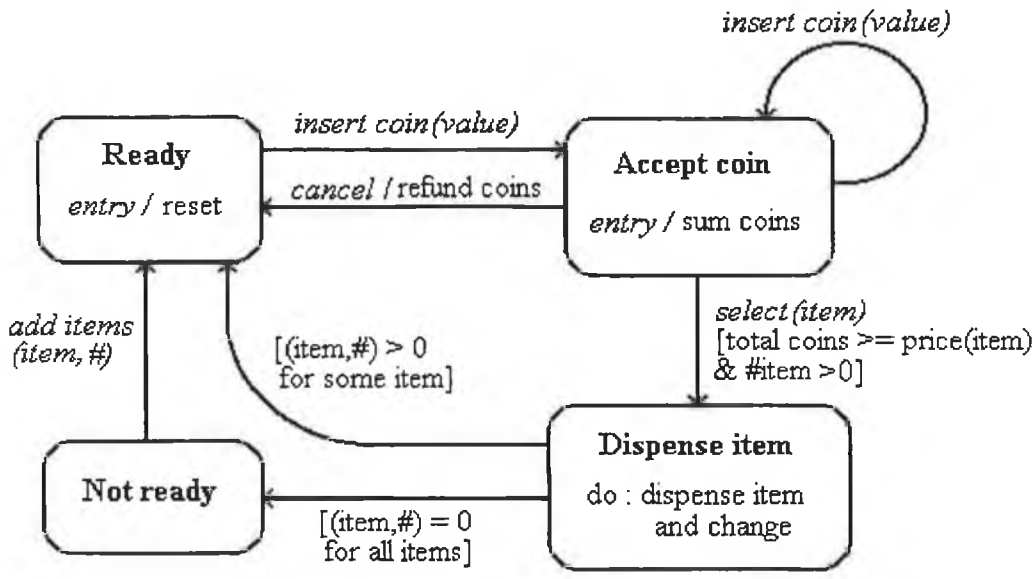


Fig 1.26 Automatic Transitions

1.3.5 Generalization

In the same way that classes can have sub-classes which inherit the attributes and operations of their superclasses, states can have substates which inherit the transitions of their superstates. Generalization represents the *or* relationship, thus within a superstate, the object must be in one and only one of the substates within that superstate. So for example, if the superstate has three substates, the object can be in the first substate *or* the second substate *or* the third substate.

Figure 1.27 is a state diagram representing an automatic transmission for a car. The *transmission* state is a superstate which has three substates, thus the transmission can be in *neutral*, *reverse* or *forward*. The forward state in turn is also a superstate which has three substates, thus when moving forward, the transmission can be in *first*, *second* or *third* gear. The transitions of a superstate are inherited by each of its substates, thus the event *push N* causes a transition from *first*, *second* or *third* gear within the *forward* state, to the *neutral* state. Similarily the event *stop* causes a transition from any forward gear to *first* gear.

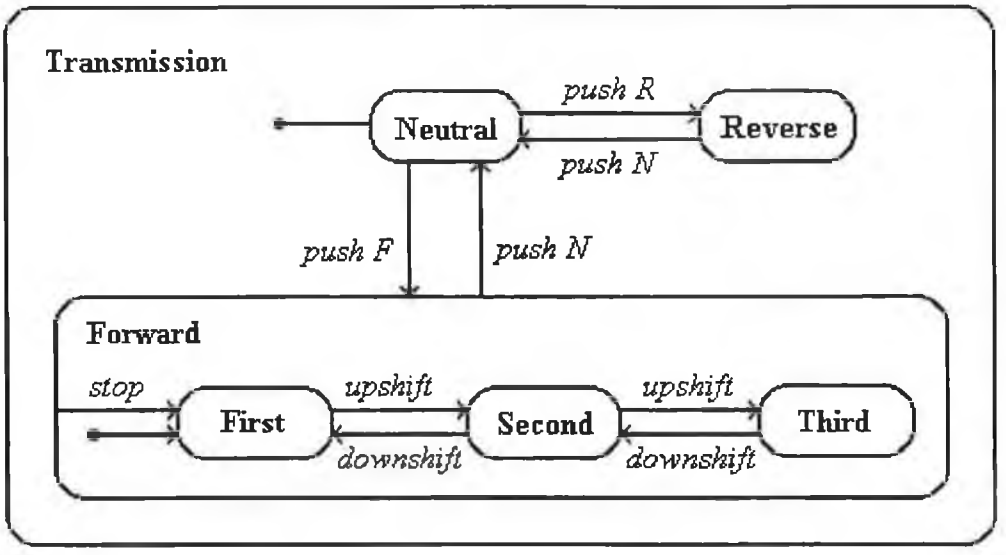


Fig 1.27 Generalization

1.3.6 Aggregation

Aggregation represents the *and* relationship, thus within a superstate, the object can be in more than one of the substates. So for example, if the superstate has two substates, the object can be in the first substate *and* the second substate.

Figure 1.28 is a state diagram representing a course which is taken by a student. The *incomplete* state is a superstate with three concurrent substates relating to *lab 1 & lab 2*, the *term project* and the *final test*. The course is *incomplete* until the two laboratory sessions, the term project and the final test have all been done. Only then can the transition to the *complete* state be made. If the laboratory sessions, term project and final test have all been successfully completed, then the course has been passed, however if the final test is failed, then the course has been failed, regardless of the outcome in the other concurrent states.

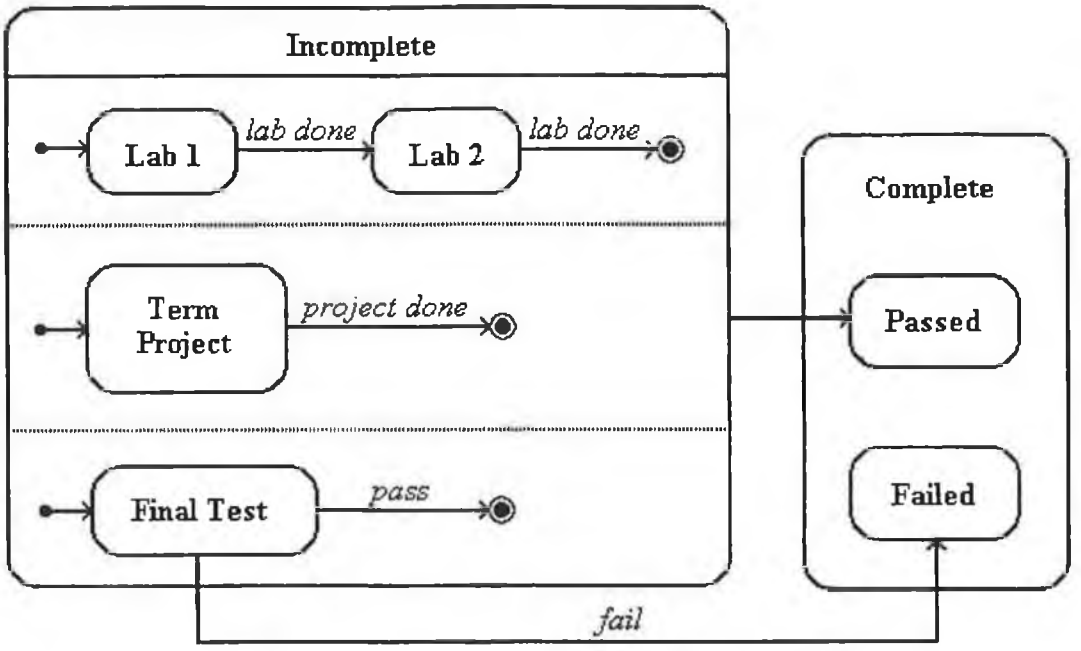


Fig 1.28 Aggregation (Case #1)

In addition, where one object is an aggregation of a number of components. The aggregate state of the object corresponds to the combined states of all the state diagrams of the components. Hence the aggregate state is one state from the first component state diagram *and* one state from the second component state diagram, *and* one state from each other component state diagram.

A cooker is an aggregate object, comprising one of more ovens, a grill, and two or four hobs. [Figure 1.29(a)] The composite state of the cooker cannot be represented by a single state in a single object, hence the state of the cooker includes one state from oven state diagram, one state from the hob state diagram and one state from the grill state diagram [Figures 1.29(b), 1.29(c), 1.29(d)]

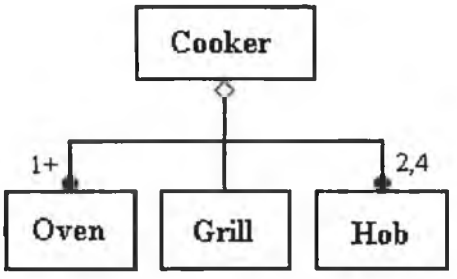


Fig 1.29(a) Aggregation (Case #2) Cooker Object Model

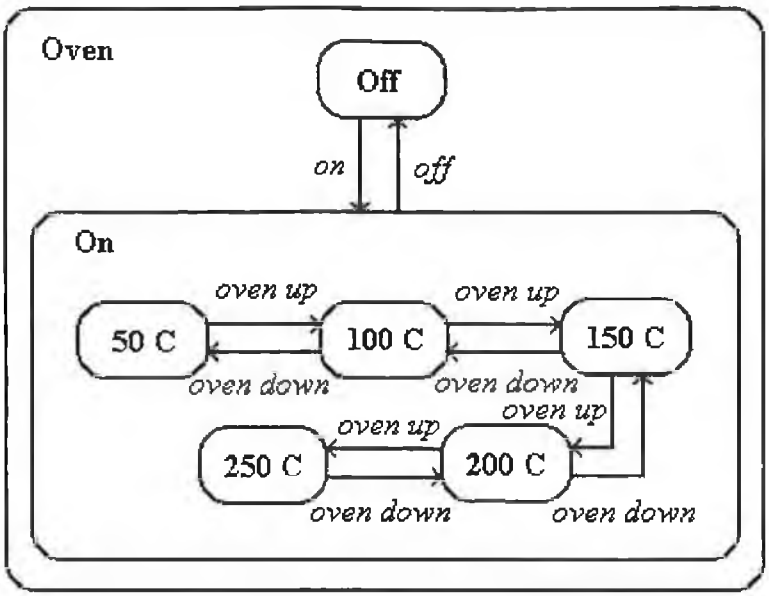


Fig 1.29(b) Aggregation (Case #2) Oven State Diagram

The oven can be turned *on*, or turned *off*. Within the *on* state there are five substates, and the temperature of the oven can be regulated between five temperature settings, by adjusting the heat of the oven either up or down.

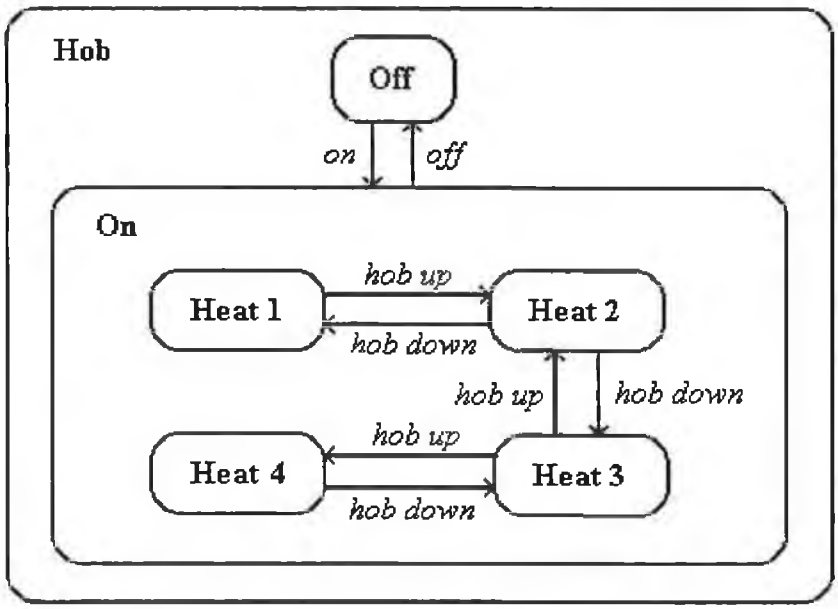


Fig 1.29(c) Aggregation (Case #2) Hob State Diagram

The hob can be turned *on* or turned *off*. Within the *on* state there are four substates, and the heat of the hob can be regulated between four heat settings, by adjusting the heat of the hob either up or down.

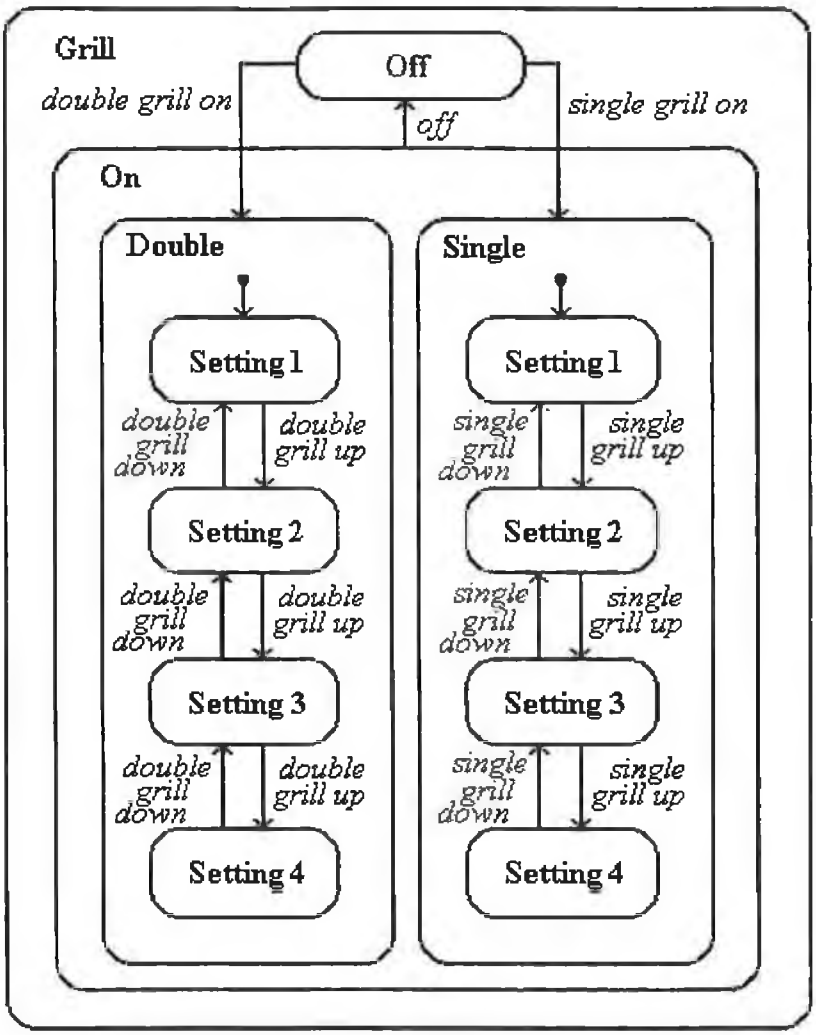


Fig 1.29(d) Aggregation (Case #2) Grill State Diagram

The grill can be turned *on*, and turned *off*. Within the *double* and *single* substates of the *on* state, there are four substates, and the heat of the grill regulated between four heat settings, by adjusting the heat of the grill either up or down.

1.3.7 Constructing the Dynamic Model

The dynamic model consists of a state diagram for each class with non-trivial dynamic behaviour. To construct the dynamic model it is necessary to prepare scenarios of typical interaction sequences, to show expected system behaviour; then to identify the events between the objects in the system; then to build an event trace diagram for each identified scenario; then to build an event flow diagram which summarises events between classes; and finally to construct a state diagram for each non-trivial class.

The dynamic model example is based on the problem statement for the order processing system, as outlined in section 1.2.5. Hence the dynamic model will refer to the classes, associations, attributes and operations which were identified when constructing the object model example.

1.3.7.1 Prepare Scenarios

A scenario is a sequence of events that occurs during one particular execution of a system. Scenarios relate to use cases [Jacobson, '94], in the same way as objects relate to a classes, in that a use case is not a single scenario, but rather a description of a set of potential scenarios. Hence there can be an infinite number of possible scenarios [Rumbaugh, '94], just like there can be an infinite number of possible objects, and a scenario is an instance of a use case, in the same way as an object is an instance of a class [Rumbaugh, '95-2].

Within the order processing system, three distinct scenarios can be identified : firstly, the normal sequence of events where on-hand stock is available to fill the customer's order; secondly, the abnormal sequence of events where on-hand stock is not available to fill the customer's order, and a delivery of stock from the supplier is required before the order can be shipped; and finally special circumstances whereby the customer need not pay the balance on the invoice in full, but may instead choose to pay in installments.

- *Scenario #1*

A customer places an order.
The company creates the order and stores it in the orders file.
The company requests authorization to fill the order from on-hand stock.
Authorization is granted for the stock request.
The authorization on the order is updated accordingly.
The stock level is depleted by the amount on the order.
The status of the order is updated to shipping.
The company creates an invoice and stores it in the invoices file.
The invoice is issued to the customer.
Full payment is received from the customer.
The balance on the invoice is reduced by the amount of the payment.
The status of the invoice is updated to paid.

- ***Scenario #2***

A customer places an order.
The company creates the order and stores it in the orders file.
The company requests authorization to fill the order from on-hand stock.
Authorization is denied for the stock request.
The authorization on the order is updated accordingly.
The outstanding stockitem is reordered.
The status of the order is updated to outstanding.
A delivery of the outstanding stockitem is received from the supplier.
The company requests authorization to fill the order from on-hand stock.
Authorization is granted for the stock request.
The authorization on the order is updated accordingly.
The stock level is depleted by the amount on the order.
The status of the order is updated to shipping.
The company creates an invoice and stores it in the invoices file.
The invoice is issued to the customer.
Payment is received from the customer.
The balance on the invoice is reduced by the amount of the payment.
The status of the invoice is updated to paid.

- ***Scenario #3***

A customer places an order.
The company creates the order and stores it in the orders file.
The company requests authorization to fill the order from on-hand stock.
Authorization is granted for the stock request.
The authorization on the order is updated accordingly.
The stock level is depleted by the amount on the order.
The status of the order is updated to shipping.
The company creates an invoice and stores it in the invoices file.
The invoice is issued to the customer.
Part payment is received from the customer.
The balance on the invoice is reduced by the amount of the payment.
The status of the invoice is updated to part-paid.
Final payment is received from the customer.
The balance on the invoice is reduced by the amount of the payment.
The status of the invoice is updated to paid.

1.3.7.2 Identify Events from Scenarios

The purpose of identifying the events within the system, is to enable each event to be allocated to the object classes that send and receive it. As each event can cause a transition between states, and hence alter the current state of an object, it follows that each event will be associated with an operation on the receiving object.

Within the order processing system there are five external events, which provide the input and output of the system : *order* which is sent from the customer to the company and contains the order details as parameters; *invoice* which is sent from the company to the customer and contains the invoice details pertaining to the order; *payment* which is sent from the customer to the company and contains the invoice number and the amount of the payment as parameters; *reorder* which is sent from the company to the supplier as contains the details of the reorder as parameters; and finally, *delivery* which is sent from the supplier to the company and contains the delivery details as parameters.

The remainder of the events are internal to the system in the sense that they are sent and received by the four object classes within the system : order; invoice; stockitem and company. These events are as follows : *create_order* which is sent from the company to the order, and which causes a new order to be created and stored in the orders file; *order_authorization_request* which is sent from the company to the stockitem, and which requests a particular quantity of a particular stockitem to be made available to fill an order; *order_authorization_response* which is sent from the stockitem to the company, and which either grants or denies the previous request depending on whether or not sufficient on-hand stock exists to fill the order; *change_authorization* and *change_status* which are sent from the company to the order, and which cause the order to be altered accordingly; *create_invoice* which is sent from the company to the invoice, and which causes a new invoice to be created and stored in the invoices file; *change_balance* and *change_status* which are sent from the company to the invoice, and which cause the invoice to be altered accordingly; and finally *increase_stock* and *decrease_stock* which are sent from the company to the stockitem, and which cause the stockitem to be altered accordingly.

These internal events are similar to messages, however their major discriminating factor from other messages in the system is that they cause a change of state within the system, and hence deserve the status of an event.

For example, *get_authorization()* and *change_authorization()* are both operations on an object of type order. There exists a *change_authorization* event which is associated with the *change_authorization()* action, causing a transition to either a state of authorization, or to a state of non-authorization, but there does not exist a *get_authorization* event, because retrieving the current authorization status does not have any effect on the state of the order object.

Section 3.3.3.4, details how these identified events are mapped into operations.

1.3.7.3 Build Event Trace Diagram for each Scenario

An event trace diagram is an ordered list of events between different objects assigned to columns in a table. By scanning a particular column in the trace, it is possible to see the events that directly affect a particular object. An event trace diagram for each identified scenario is illustrated below :

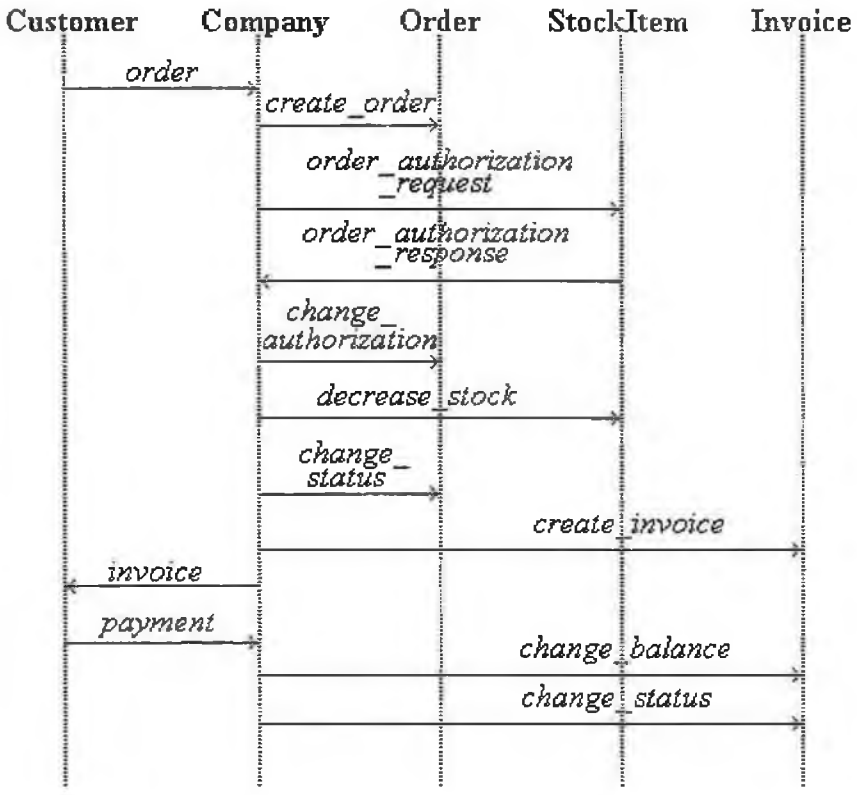


Fig 1.30 Event Trace Diagram - Scenario #1

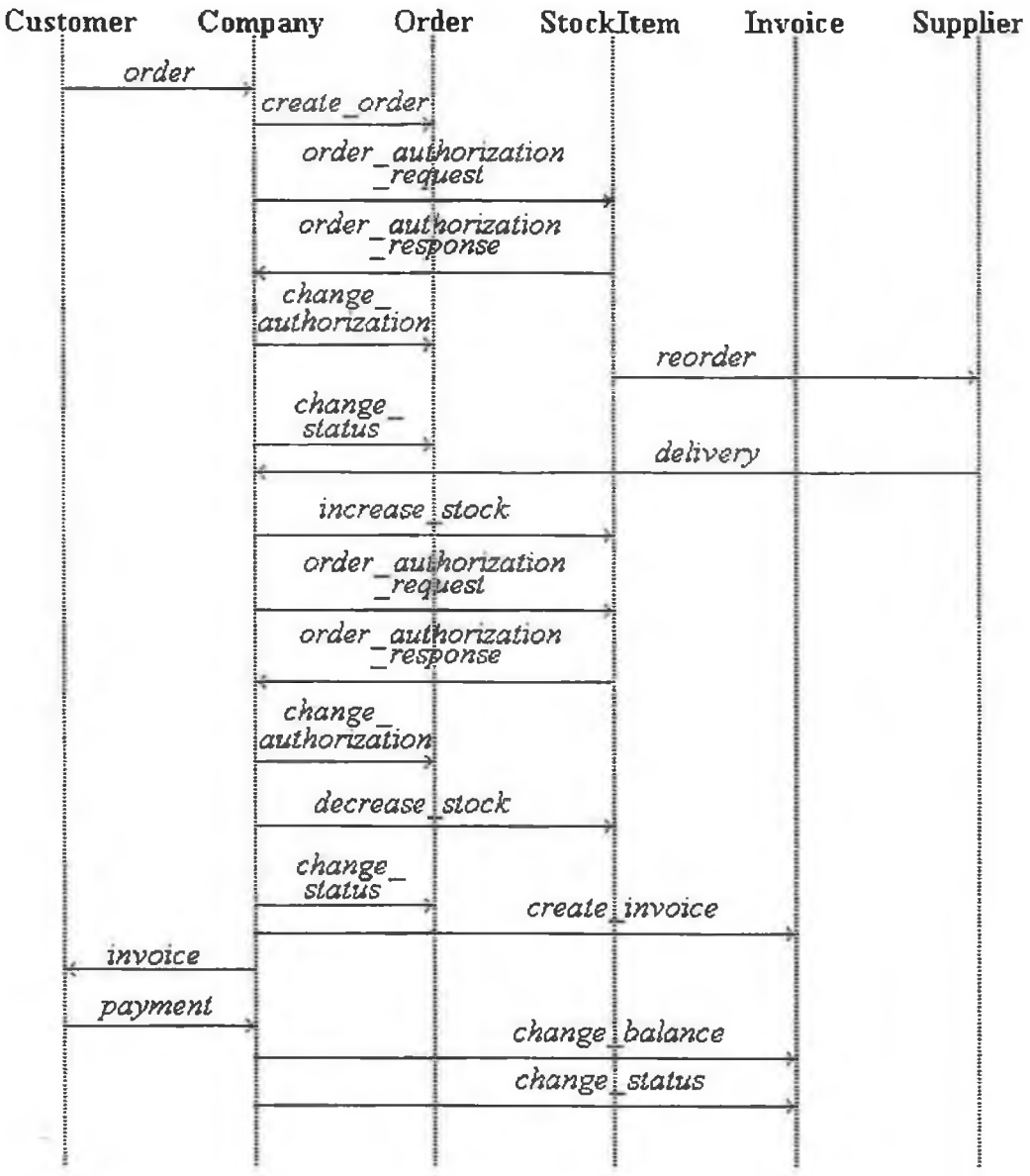


Fig 1.31 Event Trace Diagram - Scenario #2

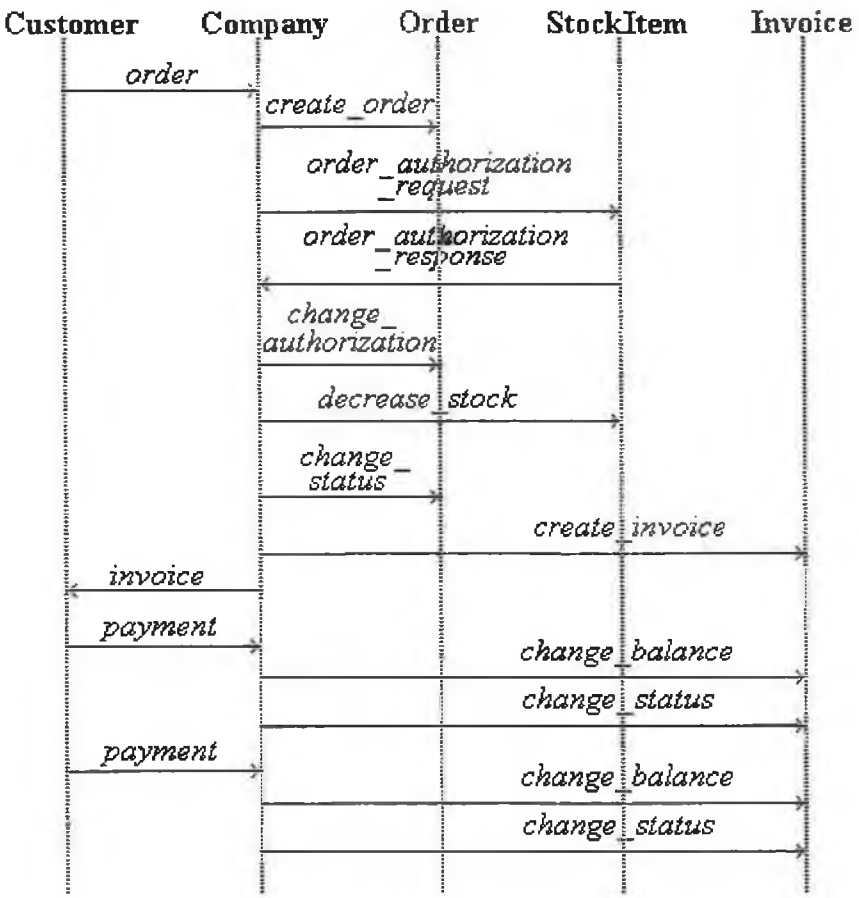


Fig 1.32 Event Trace Diagram - Scenario #3

1.3.7.4 Build Event Flow Diagram

An event flow diagram summarises events between classes, without regard for the sequence in which the events are to be performed. The event flow diagram is a dynamic counterpart to the object diagram, as paths in the object diagram show possible information flows and paths in the event flow diagram show possible control flows.

To build the event flow diagram, simply list the events which can be sent and received by each class, and place these events on the flows between the classes. Only those events listed on the event flow diagram for each respective class, can appear on the state diagram for that particular class.

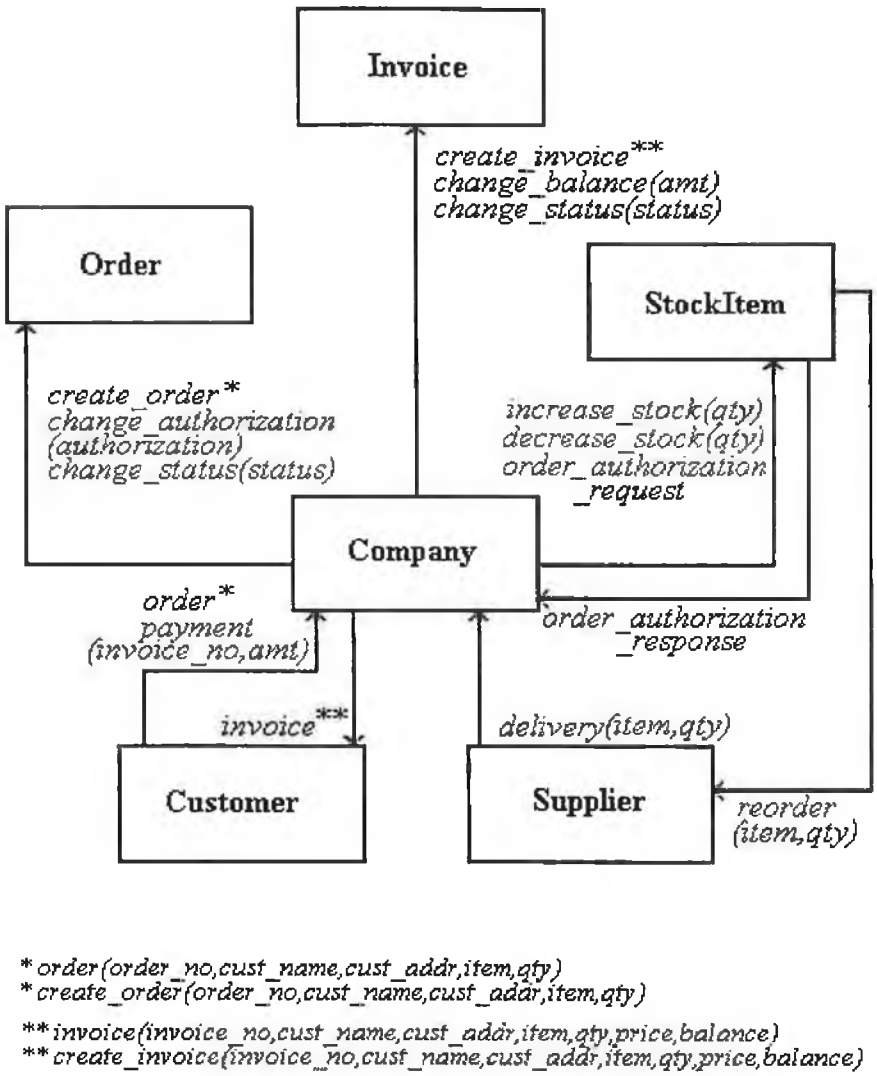


Fig 1.33 Event Flow Diagram

1.3.7.5 Build State Diagram for each Class

By examining the event flow diagram, it is easy to recognise the events pertaining to each class of objects. Each event can cause a transition between states of an object of that particular class, provided the guard on the transition (if any) evaluates to true. In response to a transition firing, an action may be performed, which can change the attribute values and links of the object, and hence the object moves into a new state. Each state with more than one exit transition represents a branch in the flow of control.

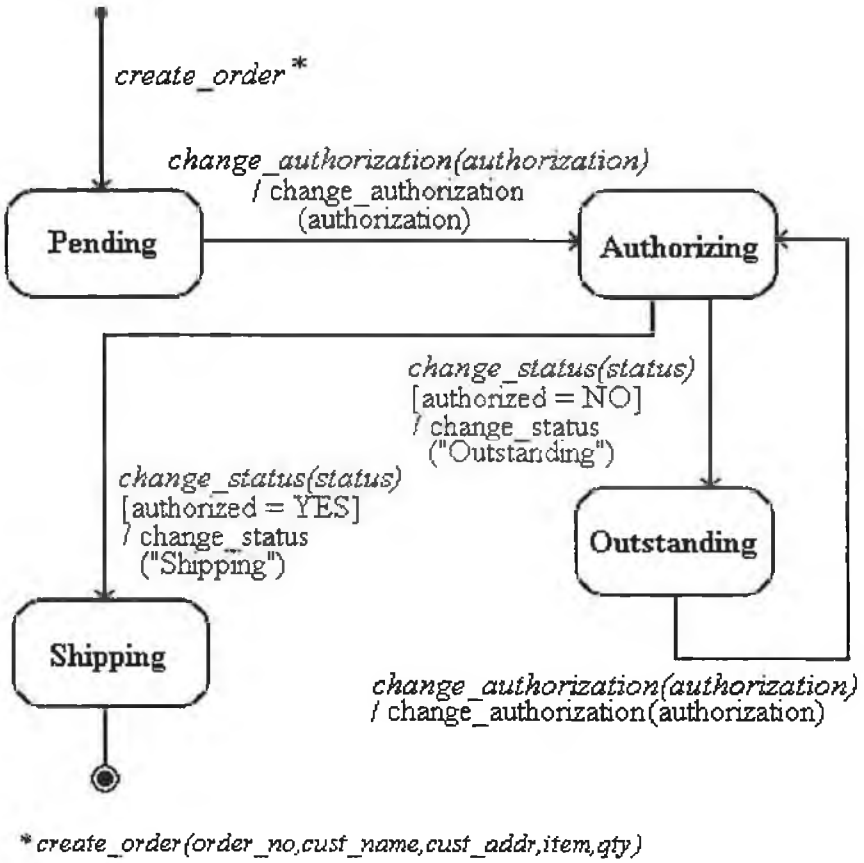


Fig 1.34 Dynamic Model (State Diagram for *Order*)

As a result of the *create_order* event, the order is placed in the ***pending*** state, and remains in this state until a *change_authorization(authorization)* event is received, which then moves the order to the ***authorizing*** state, and depending on whether sufficient stock is on-hand to fill the order or not, updates the authorization of the order to YES or NO respectively. The *change_status(status)* event moves an order which is not authorized, i.e. satisfying the condition [authorization = NO] into the ***outstanding*** state, and an order which is authorized, i.e. satisfying the condition [authorization = YES] into the ***shipping*** state, and updates the status of the order accordingly. Orders can move from being ***outstanding*** to ***authorizing*** as soon as a *change_authorization(authorization)* event is received. Once orders are ***shipping***, the cycle is complete.

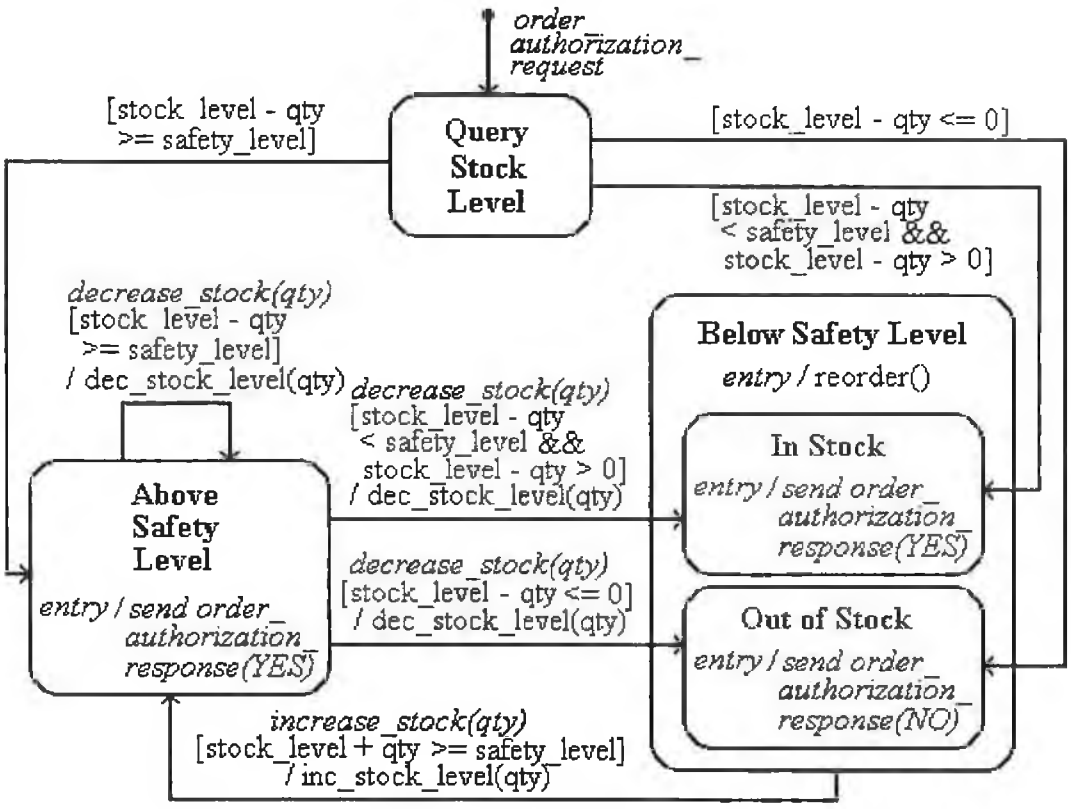


Fig 1.35 Dynamic Model (State Diagram for *StockItem*)

The stockitem object receives an *order_authorization_request* event, which places it into the ***query stock level*** state, and seeks to determine whether or not sufficient stock exists to fill a particular order. A transition is made to either the ***above safety level*** state or the ***below safety level***, depending on the current quantity status of the stockitem in question. If the quantity of the stockitem is sufficient to satisfy the current order and still remain greater than or equal to the safety level, then a transition is made to the ***above safety level*** state, and a favourable *order_authorization_response* will be sent. If the quantity of the stockitem is sufficient to satisfy the current order but it falls below the safety level, then a transition is made to the ***in stock*** state, within the ***below safety level*** state, and a favourable *order_authorization_response* will be sent, since the order can be satisfied. However, if the quantity of the stockitem is insufficient to satisfy the current order, then a transition is made to the ***out of stock*** state, within the ***below safety level*** state, and an unfavourable *order_authorization_response* will be sent, since the order cannot be satisfied.

An *increase_stock(qty)* event will cause a transition between the ***below safety level*** and ***above safety level*** states provided that the quantity delivered is sufficient to push the current quantity status of the stockitem above the safety level.

If the stockitem is ***above safety level***, a *decrease_stock(qty)* event will decrease the stock level by the given quantity, and may remain in the ***above safety level*** state, or move to the ***below safety level*** state, depending on the current quantity status of the stockitem after the decrease of stock. A transition between the ***above safety level*** state and the ***in stock*** state depends on the quantity of the stockitem falling below the safety level but still remaining positive, while a transition between the ***above safety level state*** and the ***out of stock*** state depends on the quantity of the stockitem falling to zero or below.

Note : The *reorder()* action first calculates the quantity of that particular stockitem to be reordered (usually a multiple of the safety level of that particular stockitem), and then sends the event *reorder(item,qty)* to the supplier.

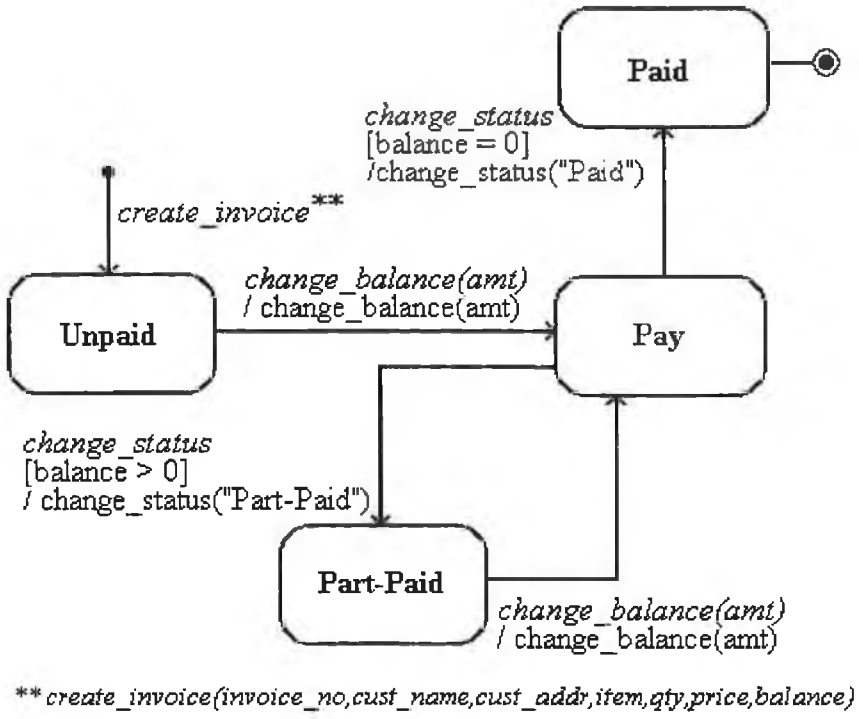
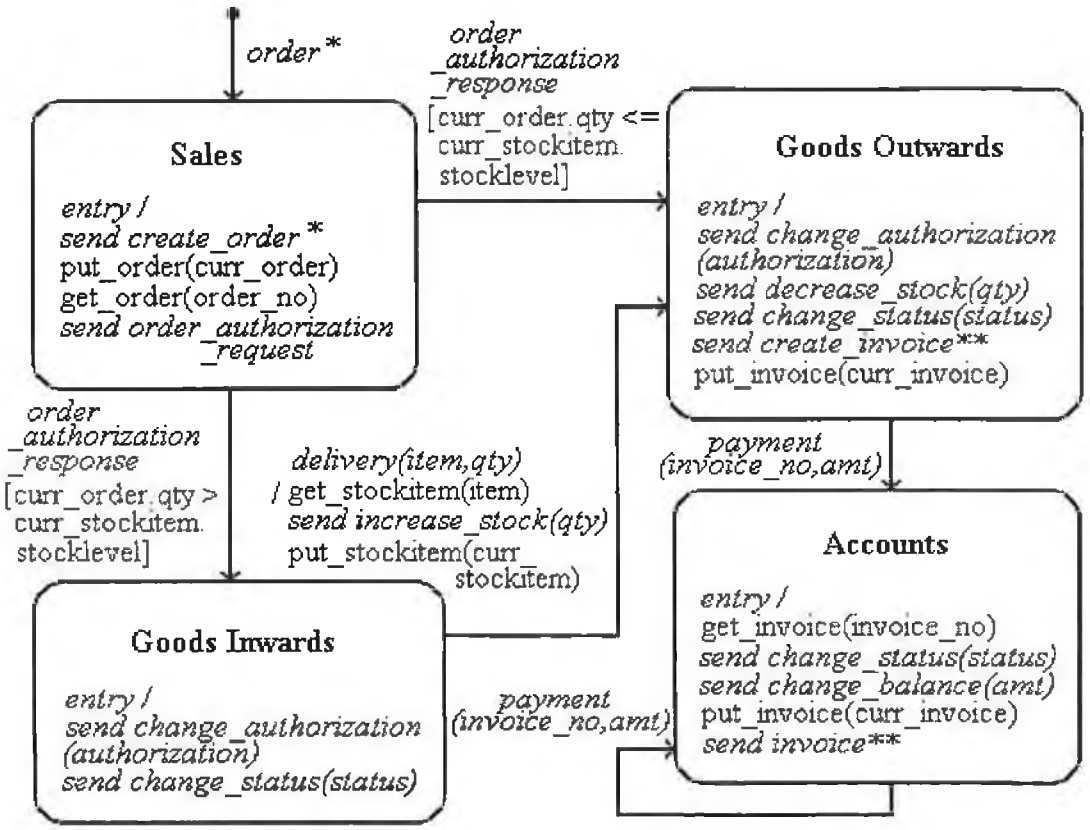


Fig 1.36 Dynamic Model (State Diagram for *Invoice*)

As a result of the *create_invoice* event, the invoice is placed in the ***unpaid*** state, and remains in this state until a *change_balance(amt)* event is received, which causes a transition to the ***pay*** state, and updates the balance of the invoice accordingly. Depending on whether the balance on the invoice was cleared or not, the *change_status(status)* event causes a transition to either the ***paid*** or ***part-paid*** state respectively. Once in the ***part-paid*** state, many *change_balance(amt)* events can be accommodated until the balance has been cleared, and the invoice is ***paid***.



* order(order_no,cust_name,cust_addr,item,qty)
** invoice(invoice_no,cust_name,cust_addr,item,qty,price,balance)
** create_invoice(invoice_no,cust_name,cust_addr,item,qty,price,balance)

Fig 1.37 Dynamic Model (State Diagram for *Company*)

As a result of an *order* event, the company enters the ***sales*** state, and the procedure for processing an order is initiated, which involves executing the following set of actions : sending a *create_order* event to the order class, which creates a new order; updating the orders file with this new order; accessing the orders file to retrieve the next order to be dispatched; and sending an *order_authorization_request* event to the stockitem class, seeking authorization to dispatch the current order.

When an *order_authorization_response* event is received form the stockitem class, and the quantity required to fill the current order is greater than the stock level of the stockitem on the current order, then a transition is made to the ***goods inwards*** state, an a procedure for awaiting delivery of the stock to fill an order is initiated, which involves executing the following set of actions : sending a *change_authorization (authorization)* event to the order class, updating the authorization on the current order to denied; and sending a *change_status(status)* event to the order class, which updates the status of the current order to outstanding.

However, when an *order_authorization_response* event is received form the stockitem class, and the quantity required to fill the current order is less than the stock level of the stockitem on the order, then a transition is made to the ***goods outwards*** state, an a procedure for shipping an order is initiated, which involves executing the following set of actions : sending a *change_authorization (authorization)* event to the order class, updating the authorization on the current order to granted; sending a *decrease_stock (qty)* event to the stockitem class, which decreases the stock level of the stockitem on the current order by the quantity on the current order; sending a *change_status(status)* event to the order class, which updates the status of the current order to shipping; sending a *create_invoice* event to the invoice class, which creates a new invoice; updating the invoices file with this new invoice; and sending an *invoice* event to the customer.

A transition from the goods inwards state to the ***goods outwards*** state can be made as soon as a *delivery(item,qty)* event is received from the supplier, and a procedure for accepting a delivery of stock is initiated, which involves executing the following set of actions : accessing the stock file to retrieve the stockitem relating to the delivery; sending an *increase_stock(qty)* event to the stockitem class, which increase the stock level of the current stockitem by the quantity of the delivery; and updating the stock file with the updated current stockitem.

As a result of a *payment(invoice_no,amt)* event, the company enters the ***accounts*** state, and the procedure for accepting invoice payments is initiated, which involves executing the following set of actions : accessing the invoices file to retrieve the invoice relating to the payment; sending a *change_status(status)* event to the invoice class, which updates the status of the current invoice to paid or part-paid depending on the relationship between the amount of the payment and the balance due on the current invoice; sending a *change_balance(amt)* event to the invoice class, which updates the balance on the current invoice to reflect the amount of the payment; and updating the invoices file with the updated current invoice.

If the balance on the current invoice has not been cleared, then subsequent *payment(invoice_no,amt)* events, can be sent to the company for that particular invoice, until the status of the current invoice is updated to paid, and the balance on the current invoice has been updated to zero.

1.4 The Functional Model

The functional model is responsible for capturing the relationship between input and output values in a system. It is a data oriented view of the system, and shows how external inputs are transformed through operations into external outputs.

1.4.1 Data Flow Diagrams (DFD)

The functional model is represented graphically using multiple data flow diagrams. A data flow diagram is a graphical representation which shows the flow of data values through a system, from their sources in objects, via transformations, to their destinations in other objects. A data flow diagram contains four basic elements : data flows, processes, actors, and data stores.

The diagram below shows the operation of these four basic elements. Actor **A#1** produces *x*'s, which are transformed by process **P#1** into *y*'s (accessing data store **D#1** to do its work), which are subsequently transformed by process **P#2** into *z*'s (updating data store **D#2** to do its work), before being consumed by actor **A#2**.

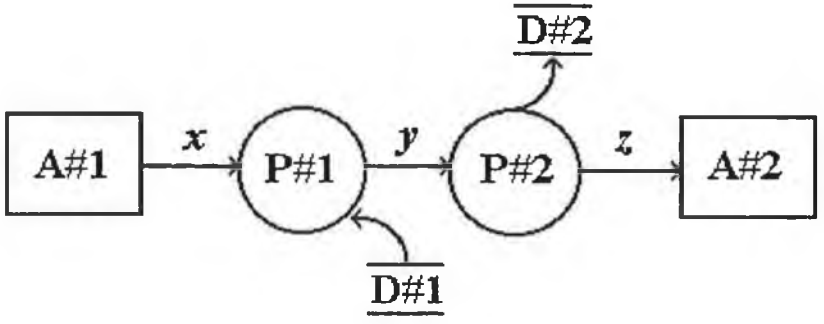


Fig 1.38 Data Flow Diagram Elements

Each of these data flow diagram elements is discussed below :

1.4.1.1 Data Flows

Data flows move data, they are pipelines through which packets of information flow [Demarco, '79]. Data can be moved from processes, actors, or data stores to other processes, actors or data stores. A data flow is drawn as a named vector connecting the source and destination of the data, with the arrowhead showing the direction of flow. The name of the data flow should embody a meaningful description of its contents. Often the data being carried on the data flow is an intermediate value within a computation, except on the boundary of the data flow diagram, where the data flows are inputs and outputs.

1.4.1.2 Processes

Process transform data, they are transformations of incoming data flows into outgoing data flows. [Demarco, '79]. The name of the process indicates the type of work which is being performed on the data. Processes can be nested to an arbitrary depth, depending on the complexity of the system being modeled. For example, a high level process will often be expanded into an entire data flow diagram, and each process in that data flow diagram subsequently expanded also, until each high level process has been decomposed into a number of atomic processes, each of which cannot be further decomposed. A process is drawn as an ellipse, with the name of the process contained within the boundaries of the ellipse.

1.4.1.3 Actors

Actor objects produce and consume data, as they are the net originators and receivers of system data [Demarco, '79]. They lie on the boundary of the data flow diagram and are attached to the inputs and outputs of the system, thus they are responsible for driving the data flow diagram. However, actors lie outside the context of the system, by designating an entity as an actor, it is implicitly external to the system under consideration [Gane, '78], it is merely a source or sink for system data, it does not perform any processing on this data. An actor is drawn as a rectangle as it is an object.

1.4.1.4 Data Stores

Data stores store data passively, they are temporary repositories of data [Demarco, '79]. A data store is drawn as a pair of parallel lines, with the name of the data store contained between the lines. The direction of arrows leading to or from a data store is significant. An input arrow to the data store indicates a modification to the store, e.g. an insertion, deletion or update, while an output arrow from the store indicates a retrieval of information from the store.

1.4.2 Operations

Each atomic process, is implemented as an operation on an object. Each operation can be specified in a number of ways such as natural language or pseudocode, or more formally as pre-conditions and post-conditions or decision tables. Specification of an operation includes a signature and a transformation. The signature defines the interface to the operation, in the form of the arguments it requires and the values it returns, whereas the transformation defines the effect of an operation, in the form of the output values as functions of the input values, and the side effects of the operation on the other objects in the system. There are four categories of operation; access operations; queries; actions; and activities; and each is discussed below.

1.4.2.1 Access Operations

Access operations are operations that read or write attributes or links of an object, and are derived directly from the attributes and associations of a class in the object model. As access operations can be deduced from the mere presence of an attribute or link, it is not necessary to list or specify these operations during analysis since they are trivial. However, during design, it is necessary to note which access operations will be private, protected, or public to the class.

The remaining three categories of operation : queries; actions; and activities are non-trivial, and must be listed in the object model, and fully specified in the functional model.

1.4.2.2 Queries

A query is an operation which is instantaneous, and which has no side effects on other objects in the system. A query with no parameters is a *derived attribute*, and can be grouped with attributes in the object model, but their derived status should be indicated since they do not contribute additional information to the model. For example, if a point is specified in Cartesian co-ordinates, then the radius and angle are derived attributes.

1.4.2.3 Actions

An action is a transformation which is instantaneous, but which has side effects on other objects in the system, and hence can cause a change of state within the system. As the state of a object is an abstraction of its attribute values and links, all actions must be definable in terms of updates to attribute values and links of objects, and thus each action can be defined in terms of the state of the system before and after the action was performed.

1.4.2.4 Activities

Unlike queries and actions which are considered instantaneous, an activity is an operation to an object, or by an object, that has duration in time. Thus an activity will have side effects on other objects in the system due to its time extended nature. Activities only make sense for actor objects, because passive objects are mere data repositories.

1.4.3 Constructing the Functional Model

The functional model transforms input values into output values, hence to construct the functional model it is necessary to identify the input values and output values of the system; then to build the data flow diagram which illustrates the transformations that the input values undergo, until the output values are achieved; and finally to describe the functions which effect the transformation from input values to output values.

The functional model example is based on the problem statement for the order processing system, as outlined in section 1.2.5. Hence the functional model will refer to the classes, associations, attributes and operations which were identified when constructing the object model example, and the events, states, conditions and actions which were identified when constructing the dynamic model example.

1.4.3.1 Identify Input and Output Values

The input and output values of a system can be identified from the problem statement. In addition, since input and output values are parameters of the events between the system and the outside world [Rumbaugh, '91], the input and output values can be also be identified from the external events in the dynamic model of the system.

The list of input values is as follows : *order details* comprising the order number, the customer's name, the customer's address, the item ordered, and the quantity ordered; *payment details* comprising the invoice number, and the amount of the payment; and *delivery details* comprising the item being delivered and the quantity being delivered.

The list of output values is as follows : *invoice details* comprising the invoice number, the customer's name, the customer's address, the item ordered, the quantity ordered, the price of the item, and the balance due; *stock reorder details* comprising the item being reordered, and the quantity being reordered.

1.4.3.2 Build the Data Flow Diagram

The data flow diagram shows how each output value is computed from input values. These input and output values cross the boundary of the data flow diagram, as the input values originate with external actors, and the output values are consumed by external actors. Thus for the simple order processing system, there are two external actors, the customer and the supplier, as they are the net originators and receivers of system data, and are responsible for driving the data flow diagram.

The body of the data flow diagram is filled in by tracing the input values forward from the external sources, and determining the transformations that the data must undergo before it is consumed by external sinks. In the order processing system, six important transformations of data can be identified :

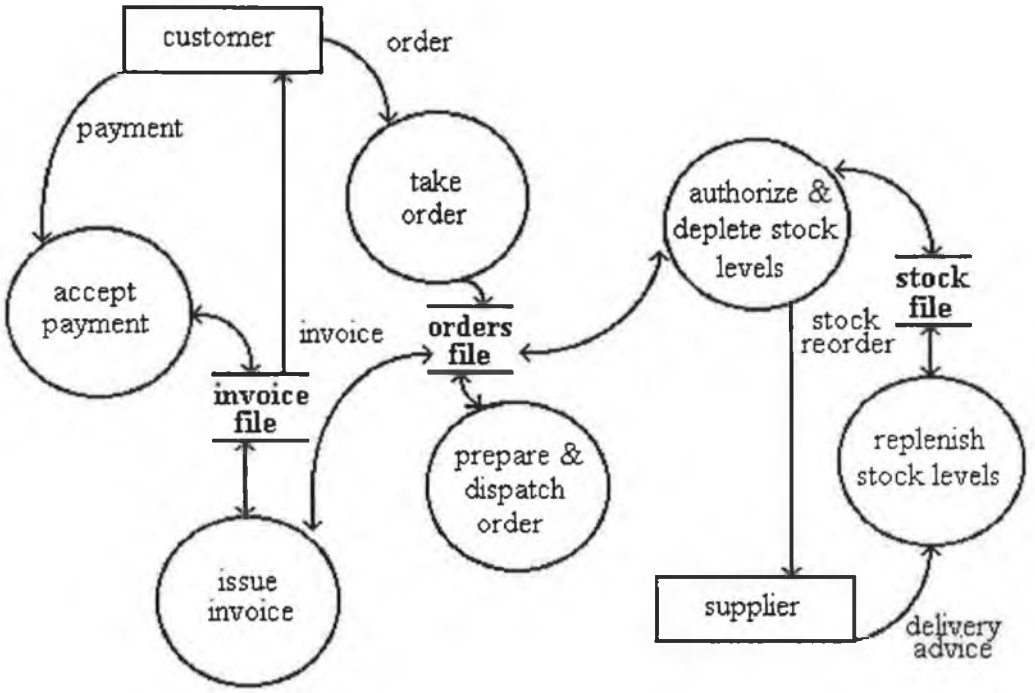


Fig 1.39 Functional Model

Data flow key :

order *- order no, customer's name and address, item and quantity ordered.*
order_authorization_request *- item and quantity ordered.*
order_authorization_response *- yes or no, dependent on stock availability.*
shipment_confirmation *- order no, name, address, item and quantity.*
stock_reorder *- item and quantity required from supplier.*
delivery_advice *- item and quantity received from supplier.*
invoice *- invoice no, name, address, item, quantity, price, balance.*
payment *- invoice no and amount tendered.*

The processes in the (Level 1) data flow diagram are not atomic processes, and hence each of these processes can be further refined as a (Level 2) data flow diagram. However since the processes in this example are not complex, the refinement of each process can be performed as part of the next step in the construction of the functional model.

1.4.3.3 Describe Functions

Six important functions have been identified from the data flow diagram, and each of these is described below. Each of these functions will map into methods of the *company* class in the object model, and each in turn will invoke methods of the *order*, *invoice* and *stockitem* classes.

- The ***take_order*** process reads in the details of the order from the customer, creates a new order using the *order::order()* method, and updates the orders file with the new order using the *company::put_order()* method.

- The ***authorize&deplete_stock_levels*** process reads the ordered stockitem from the stock file using the *company::get_stockitem()* method and then reads the stock level of this stockitem using the *stockitem::get_stock_level()* method. If sufficient stock exists to fill the quantity of the current order, read using the *order::get_qty()* method, it updates the authorization of the order (granted) in the orders file using the *order::change_authorization()* method, and decreases the stock level of that stockitem in the stock file, by the quantity on the order, using the *stockitem::dec_stock_level()* method. If this depletion of stock caused the stock level to fall below the safety stock level, read using the *stockitem::get_safety_level()* method, then the stockitem is reordered using the *stockitem::reorder()* method. If insufficient stock exists to fill the order, it updates the authorization of the order (denied) in the orders file using the *order::change_authorization()* method, and issues a stock reorder to the supplier using the *stockitem::reorder()* method.

- The ***prepare&dispatch_order*** process reads the authorization of the current order using the *order::get_authorization()* method. If the order has been authorized, it updates the status of the order to shipping using the *order::change_status()* method, otherwise it updates the status of the order to outstanding also using the *order::change_status()* method, and finally replaces the updated order in the orders file using the *company::put_order()* method.

- The ***issue_invoice*** process reads the details of the current order using the *order::get_order_no(), order::get_cust_name(), order::get_cust_addr(), order::get_item(), order::get_qty()* methods. It then reads the stockitem on the order from the stock file using the *company::get_stockitem()* method, and reads the price of this stockitem using the *stockitem::get_price()* method.

It then creates an invoice for the order using the *invoice::invoice()* method, stores this new invoice in the invoice file using the *company::put_invoice()* method, and finally the invoice is sent to the customer.

- The ***accept_payment*** process reads in the number of the invoice being paid, and the amount of the payment from the customer, then reads the corresponding invoice from the invoices file using the *company::get_invoice()* method. It then reads the balance due on the invoice using the *invoice::get_balance()* method, determines the new status of the invoice (paid or part-paid), and updates the status of the invoice using the *invoice::change_status()* method, and the balance of the invoice using the *invoice::change_balance()* method. Finally it replaces the updates invoice in the invoices file using the *company::put_invoice()* method.

- The ***replenish_stock_levels*** process reads in the stockitem being delivered and the quantity of that stockitem being delivered, then reads the corresponding stockitem from the stock file using the *company::get_stockitem()* method. It the updates the stock level of the corresponding stockitem using the *stockitem::inc_stock_level()* method, and finally replaces the updated stockitem in the stock file using the *company::put_stockitem()* method.

1.5 Chapter Summary

This chapter has introduced Rumbaugh's Object Modeling Technique (OMT), by describing each of the different models : object; dynamic; and functional; in terms of their purpose, and the notation which is used to construct each of them. A simple illustrated example was used to show how each of these separate models is constructed in a step-by-step manner.

Chapter 2

The Problems Associated with OMT Integration and Consistency

2.1 Overview

The primary strength of the OMT methodology is that it allows a complete specification of a system, covering it's static structure, dynamic behaviour and functionality. Each of the three models employed to abstract a specific view of the system, uses a concise and understandable notation, and thus each model can be appreciated to a large extent on its own merit.

However, the diversity present in the OMT methodology is also paradoxically it's major weakness. Each model is developed more or less independently, and the inter-relationships between the three models are not explicit, resulting in a lack of integration, and subsequent lack of consistency between the object, dynamic and functional models, thus making it difficult to get the overall picture.

The two primary reasons for the unsatisfactory level of integration and consistency are significantly inter-related to each other. To considerably improve to the completeness of the OMT models, both of these problems need to be understood and redressed :

- ***Weak functional model***

  The functional model is ostensibly the weakest of the three models, to the extent that some users of OMT prefer to omit the model entirely [Coleman, '94]. This weakness is a direct result of the mismatch caused by using data flow diagrams to model object-oriented functionality. In addition, this mismatch results in tenuous links from the functional model to the other two models, and hence difficulties arise with the integration of the three OMT models.

- ***Inadequate inter-model relationships***

  The relationship between the three OMT models is not well-defined, and is not supported by concrete steps in the methodology. It is thus difficult to recognise how the separate models integrate together, and furthermore it is not easy to check the models for consistency with one another. There is little doubt that the weakness of the functional model contributes to this problem.

Both of these points are discussed in detail below.

2.2 Weak Functional Model

The functional model is primarily concerned with the transformational aspects of a system. It's purpose is to show how the input values of a system are transformed via operations, into the output values of a system, and to describe these operations which transform the system, in terms of what each operation does and how each operation works.

Published researchers in the area of object-oriented analysis and design, such as D'Souza [D'Souza, '93, '94-1, 94-2, '95], Coleman [Coleman, '94], and Monarchi [Monarchi, '92], whose work is presented in the following sections, agree that the functional model has failed in it's purpose, since a DFD is a poor medium to illustrate the transformations of an object-oriented system. Even Rumbaugh himself is currently working on a second generation OMT methodology, where the approach to the functional model is a major departure from the conventional use of DFDs [Rumbaugh, '95-3].

There is little doubt that the weakness of the functional model lies in the unsuitability of using DFDs for designing systems of interacting objects [D'Souza, '93], since DFDs are organized around processes and not around objects. DFDs primarily detail the behaviour of processes and the flow of information between these processes. As such these diagrams are best suited to the modeling of systems which have strong functional emphasis, and which will subsequently be implemented using functional languages (3GLs) such as C, Pascal etc.

These traditional functional languages were found to be inadequate for implementing complex systems, so a revolutionary object-oriented approach was developed which was very different in implementation from its 3GL predecessor. Since the functional paradigm and the object-oriented paradigm are intrinsically different, should it not follow that the analysis and design tools for these distinct paradigms should also be as different to each other as the programming languages which implement them ? Why use a function-oriented DFD to model an object-oriented system ?

Obviously DFDs are well-suited to modeling function-oriented systems, however a large gap exists between the functional paradigm and the object-oriented paradigm, which means that DFDs cannot adequately model an object-oriented system. This point is discussed below by highlighting the contrast between the functional paradigm and the object-oriented paradigm in terms of decomposition; granularity; data access; interaction and mapping, and the failure of DFDs to adequately represent an object-oriented system in each of these categories.

2.2.1 Decomposition

Decomposition refers to the criteria used to abstract the fundamental aspects from a problem. Object-oriented decomposition breaks the system down into objects, where each object encapsulates both data and methods, whereas functional decomposition breaks the system down into functions and sub functions.

Although the data flow approach and the object-oriented approach both concentrate on the data within a system, the data flow approach has more in common with functional decomposition than object-oriented decomposition, and thus is not suited to the latter methodology.

There are three points which illustrate this unsuitability :

- A DFD is most commonly used as a systems-modeling tool for operational systems, in which the functions of the system are of paramount importance and more complex than the data that the system manipulates [Yourdon, '89]. Within an object-oriented system the data is of prime importance. Abstraction of the fundamental data items reduces the complexity of the system, and hence the functions can often be trivial, merely accessing and updating this fundamental data.

- The DFD gives weak emphasis to the data store. DFDs are not very helpful for systems that primarily update and retrieve data [Coad, '90]. However there is heavy updating and retrieval of data in an object-oriented system, because each object operates like a data store, as it has it's own data, and methods to access that data. Thus if data flow diagrams are inadequate for systems with heavy data access, it follows that they are more inadequate for object-oriented ones.

- DFDs still have strong functional emphasis [Coad, '90]. These diagrams are constructed by first thinking about inputs and outputs, and then putting input data through a sequence of transformations, until output data is achieved, and as such are more applicable to a functional decomposition methodology, than an object-oriented one.

2.2.2 Granularity

Granularity refers to the level of abstraction of the model. The functional paradigm adopts a top-down approach to abstracting the fundamental aspects of a system. It focuses on the overall function of the system, expanding and refining this function step-by-step, thus moving from the general to the specific.

The object-oriented paradigm is a bottom-up approach, where the details of each object class are defined first, and only then are these classes organized into a hierarchy by using inheritance and association, thus moving from the specific to the general.

DFDs use the top-down approach, beginning with a very general context diagram, which is expanded level by level until functional primitives are achieved in each level of the diagram. The main problem with using DFDs is that it makes it difficult to place the functional model in the same granularity context as the existing object and dynamic models, as it is not developed bottom-up on a class-by-class basis. Thus the three models represent different abstraction levels : the object model depicts micro-level structure; the dynamic model portrays micro-level states and transitions; but the functional model describes macro-level functionality. The diverse levels of granularity make it difficult to integrate or synthesize the separate models [Monarchi, '92].

2.2.3 Data Access

Data access refers to what parts of the system can read or write the data, and the restrictions (if any) imposed on the access of that data. The main principle behind functional programming is to declare variables locally within a function, and to pass them either by value or by reference to other functions. As such data access is unrestricted because it can be passed into functions throughout the program, and furthermore it can be somehow transformed by these functions.

However the main principle behind object-oriented programming is to encapsulate data and the functions which operate on that data (methods), within an object. Data access is restricted in an object-oriented environment as only those methods contained within an object, are allowed to access the data contained within the object.

As DFDs are organized around processes and not around objects, it becomes difficult to determine which data belongs to which objects, because what the system does is divorced from what objects it does it to. Therefore DFDs are an acceptable medium to illustrate the unrestricted data access of functional programming, but an unsatisfactory one to portray the restricted data access of object-oriented programming.

2.2.4 Interaction

Interaction refers to the mechanism by which the constituent parts of a system communicate with each other to make the system work. In a functional environment, it is the functions which interact, by means of a function-invoking mechanism whereby functions call other functions and pass data to one another. However in an object-oriented environment, it is the objects and not the methods (functions which operate on data) which interact.

Objects interact by means of a messaging mechanism, the requesting object must send a message to the object whose data it requires, the receiving object invokes a method which retrieves the requested data and passes it to the requesting object. But this is only a copy of the data (similar to call-by-value), to change the value of the data (similar to call-by-reference), the requesting object must send a message to the object whose data it requires to change, and the receiving object invokes a method which changes the value of the requested data [Wirfs-Brock, '90].

Furthermore data does not need to be passed between methods of the same object, rather it moves seamlessly between methods, because all methods have access to their object's data.

As DFDs only show the movement of data within a system, they cannot fully illustrate the interaction which takes place within an object-oriented system, because object interaction by means of the messaging mechanism does not always involve an exchange of data. This point is further illustrated by the variance between the function-oriented DFD in figure 2.2.5.1 and the object-oriented DFD in figure 2.2.5.2

2.2.5 Mapping

Mapping refers to the ease or difficulty of the transition from analysis to design and implementation, it's a measure of how appropriate the model was for the design of the system. Although a model should be free from implementation detail, it should lend itself well to it's implementation. For example, when designing a real-time system, a petri-net would be more appropriate than a structure chart. There is growing evidence in literature and from a recent panel discussion at the OOPSLA/ECOOP '90 conference that structured analysis cannot be used effectively when subsequent design and implementation is to be done in an object-oriented manner [de Champeaux, '92]. This is largely due to the incompatibility between the functional and object-oriented paradigms, as the movement from DFDs to object-oriented representation is not only radical, but abrupt and disjoint [Coad, '91-2], thus causing transitional difficulties from functional analysis to object-oriented design. With this in mind, it seems incongruent that one third of the OMT methodology is modeled using structured analysis techniques. Although it is possible to model an object-oriented system using a DFD, the resulting DFD lacks the expressive power of a DFD modeling the same system but from a function-oriented point of view. This is because DFDs cannot fully represent the object-oriented paradigm, in the same way as they can fully represent the functional paradigm.

Furthermore, when DFDs are applied as tools for functional modeling, they lend themselves well to almost trivial mapping from the model to the code in the case of 3GL's like C. For example, each process in the DFD will map into a function, each data flow will map into a parameter list to a function or a return value from a function, each data store will map into a file, and each actor will map into a source of input and output. Thus, the DFD is a very useful model for a function-oriented

system, since the potential run-time behaviour of the system is easily visible, which is an aid for eventual design and implementation. However when DFDs are applied as tools for object-oriented modeling, there is no such trivial mapping from the model to the code in the case of object-oriented langauges such as C++, primarily because the DFD is not organized around objects. Thus, the DFD is not a useful model for an object-oriented system, since there is a lack of visibility of the potential run-time behaviour of the system, and hence it does not greatly aid eventual design and implementation of the system. An appropriate model is particularly relevant to OMT because there are no design models in OMT. Hence design is essentially a process of coding the analysis models, and the large gap between analysis and code can make this a daunting task [Coleman, '94].

To illustrate these points, outlined below is a simple DFD for an order processing system. Figure 2.1 details the flow of data in a function-oriented environment, and proposes a mapping to it's implementation in the form of sample code for C, while Figure 2.2 details the flow of data in an object-oriented environment, and proposes a mapping to it's implementation in the form of sample code for C++.

2.2.5.1 DFD representation of a function-oriented system

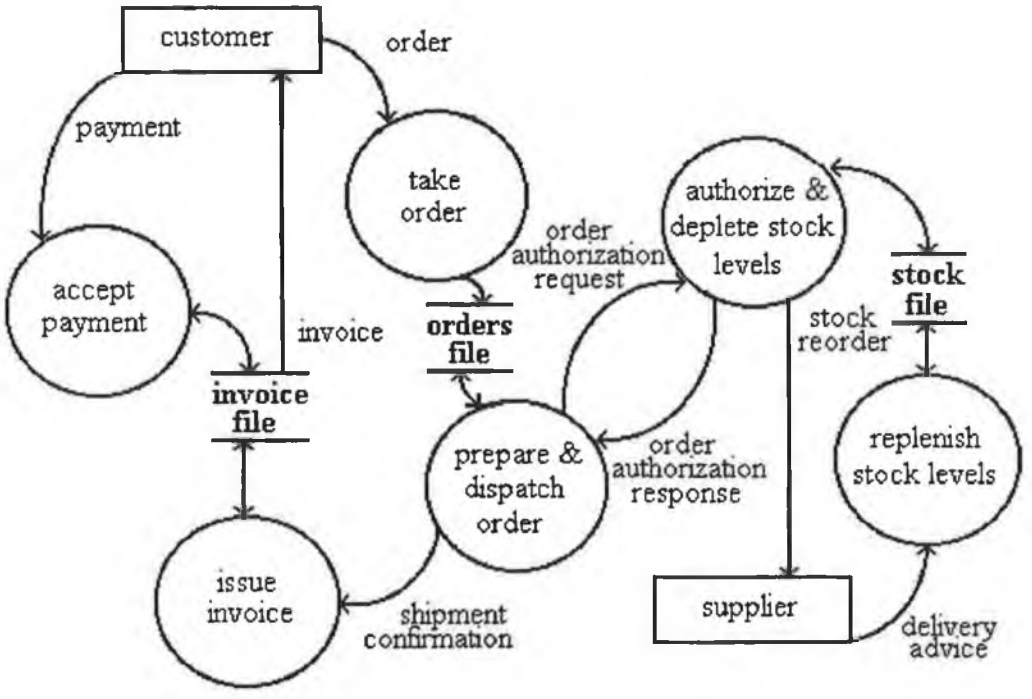


Fig 2.1 Function-Oriented DFD

Data flow key :

order *- order no, customer's name and address, item and quantity ordered.*
order_authorization_request *- item and quantity ordered.*
order_authorization_response *- yes or no, dependent on stock availability.*
shipment_confirmation *- order no, name, address, item and quantity.*
stock_reorder *- item and quantity required from supplier.*
delivery_advice *- item and quantity received from supplier.*
invoice *- invoice no, name, address, item, quantity, price, balance.*
payment *- invoice no and amount tendered.*

2.2.5.1.1 Structures

In order to implement the above DFD in C, two structures need to be declared - an order structure to hold details of the order, and an invoice structure to hold details of the invoice.

```
struct order {
   int order_no;
   char cust_name[20];
   char cust_addr[20];
   char status[15];
   int item;
   int qty;
   }
```

```
struct invoice {
   int invoice_no;
   char cust_name[20];
   char cust_addr[20];
   char status[10];
   int item;
   int qty;
   float balance;
   }
```

In addition there are three arrays - stock_level[20], which holds the on-hand stock quantity of each of the 20 stock items that this company supplies; safety_level[20] which holds the safety stock quantity of each of the 20 stock items, once the stock level of any item falls below its pre-defined safety level, it is re-ordered; and price[20] which holds the price of each of the 20 stock items.

```
int stock_level[20];
int safety_level[20];
float price[20];
```

The orders file and the invoices file are implemented as arrays : orders_file[100] and invoices_file[100], each holding 100 items numbered from 0 to 99 and in direct correspondence with each other (i.e. invoice #50 belongs to order #50).

```
struct order *orders_file[100];
struct invoice *invoices_file[100];
```

2.2.5.1.2 Functions

- The ***take_order*** process reads in the details of the order from the customer, assigns the order an order number, and updates the orders file with the new order.

```
void take_order()
{
   struct order *new_order;

   new_order = new(order);
   scanf("%d",order->order_no);
   gets(order->cust_name);
   gets(order->cust_addr);
   strcpy(order->status,"Pending");
   scanf("%d",order->item);
   scanf("%d",order->qty);

   orders_file[order->order_no] = new_order;
}
```

- The ***prepare&dispatch_order*** process reads the next order which has yet to be shipped from the orders file, and looks for authorization to prepare the order by sending an order authorization request to the stores. This request consists of the item on the order and the quantity of that item required to fill the order. If stock is available to fill the order, authorization is granted, the order status is updated to shipping and an invoice is issued. If stock is not available to fill the order, authorization is refused, and the order status is updated to outstanding, until the required stock is received from the supplier. The new status of the order is rewritten back to the orders file.

(It is assumed that there will always exist at least one order which has yet to be shipped, and that the delivery time for an out-of-stock item to arrive from the supplier is insubstantial, hence if the status of an order is updated to outstanding due to insufficient on-hand stock, then the stock will have arrived from the supplier by the next time this order is designated as the current order.)

```
void prepare&dispatch_order(struct order *curr_order)
{
  int authorized;
  authorized = authorize&deplete_stock_levels(curr_order->item, curr_order->qty);

  if (authorized == YES)
     {
     strcpy(curr_order->status,"Shipping");
     issue_invoice(curr_order);
     }
  else
     strcpy(curr_order->status,"Outstanding");

  orders_file[curr_order->order_no] = curr_order;
}
```

- The ***authorize&deplete_stock_levels*** process checks the stock level of the item requested to see if the required quantity is available. If so, the stock level for that item is reduced by that quantity, and if the resulting stock level is below the safety level then a standard quantity is reordered, and finally authorization is granted. If the required quantity is not available, a standard quantity is reordered and authorization is refused. (It is assumed that the standard quantity reordered is large enough to fill any outstanding order).

```
int authorize&deplete_stock_levels(int item, int qty)
{
  if (stock_level[item] >= qty)
     {
     stock_level[item] = stock_level[item] - qty;
     if (stock_level[item] < safety_level[item])
        reorder(item);
```

```
    return YES;
    }
  else
    {
    reorder(item);
    return NO;
    }
}
```

- The ***issue_invoice*** process takes in the order details in the form of a shipment confirmation, and creates an invoice for the order, and stores this new invoice in the invoice file.

```
void issue_invoice(struct order *curr_order)
{
  struct invoice *new_invoice;

  new_invoice = new(invoice);
  strcpy(new_invoice->invoice_no,curr_order->order_no);
  strcpy(new_invoice->cust_name,curr_order->cust_name);
  strcpy(new_invoice->cust_addr,curr_order->cust_addr);
  strcpy(new_invoice->status,"Unpaid");
  new_invoice->item = curr_order->item;
  new_invoice->qty = curr_order->qty;
  new_invoice->balance = price[new_invoice->item] * new_invoice->qty;

  invoices_file[new_invoice->invoice_no] = new_invoice;
}
```

- The ***accept_payment*** process takes in the order number of the invoice being paid and extracts the appropriate invoice from the invoice file. If the amount being paid is equal to the balance on the invoice then the invoice is marked as paid and the balance is cleared. If the amount being paid is less than the balance on the invoice then the balance is reduced by that amount, and the status of the invoice remains as unpaid. (It is assumed that no customer will pay more than the balance on the invoice).

```
void accept_payment(int invoice_no, float amount)
{
   struct invoice *curr_invoice;
   curr_invoice = invoices_file[invoice_no];

   if (amount == curr_invoice->balance)
      strcpy(curr_invoice->status,"Paid");
   curr_invoice->balance = curr_invoice->balance - amount;

   invoices_file[invoice_no] = curr_invoice;
}
```

- The ***replenish_stock_levels*** process is invoked when a delivery is received from the supplier. The delivery advice consists of the item being delivered and the quantity of that item being delivered. The stock level for the relevant item is increased by the relevant quantity.

```
void replenish_stock_levels(int item, int qty)
{
   stock_level[item] = stock_level[item] + qty;
}
```

2.2.5.2 DFD representation of an object-oriented system

The flow of data on the object-oriented DFD [Figure 2.1] is different from the flow of data on the function-oriented DFD [Figure 2.2], for example, the flow of data to the orders file and the flow of data from the orders file has changed, as well as the flow of data between the four processes which access the orders file.

In addition the mapping from the object-oriented DFD to its implementation, is not as smooth as the mapping from the function-oriented DFD to its implementation. This is because data flow diagrams cannot represent the object-oriented paradigm as completely as they can represent the functional paradigm.

The differences between the object-oriented DFD and the function-oriented DFD are discussed below in terms of the structures and functions required to implement this system using an object-oriented language (C++) as opposed to the previous C implementation, and furthermore, the inadequacy of the DFD is also discussed in terms of its inability to sufficiently represent these structures, functions and the interaction between them beyond a trivial level.

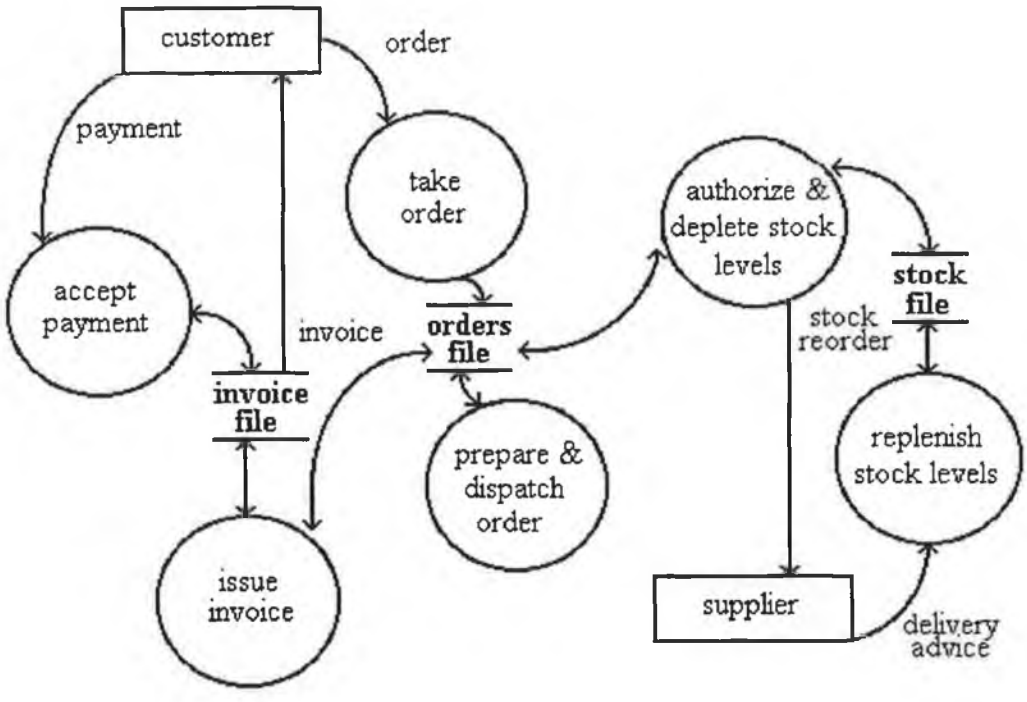


Fig 2.2 Object-Oriented DFD

2.2.5.2.1 Structures

Firstly, the classes of the system are not abundantly clear from the DFD. Obviously there is an *order* class and an *invoice* class, which coincides with the structures in the C implementation. However, *stockitem* is also a class, as it contains the stock level, safety stock level, and price of the stockitem along with methods to access this data. Therefore in this C++ implementation there will be a uniform interface when updating stock levels, in contrast to the C implementation where the *stocklevel*, *safetylevel* and *price* arrays could be updated by many processes.

A more subtle class is the *company* itself, whose data is the orders, invoices and stock, and whose methods are the DFD processes, which in turn call methods of the other sub-ordinate classes, as well as methods for accessing and updating the orders file, invoices file and stock file.

There is also the problem of phantom classes. It would appear from the DFD that both *customer* and *supplier* should also be classes, but they merely provide the input and output of the system, and within the context of this example, they do not process any data they provide.

The classes are as follows :

```
class order {
private :
   int order_no;
   char cust_name[20];
   char cust_addr[20];
   int item;
   int qty;
   char status[15];
   int authorized;
public :
   order(int new_order_no, char *new_cust_name, char *new_cust_addr,
         int item, int qty);
   int get_order_no();
   char *get_cust_name();
   char *get_cust_addr();
   int get_item();
   int get_qty();
   char *get_status();
   int get_authorization();
   void change_status(char *new_status);
   void change_authorization(int new_authorization);
   };
```

```
class invoice {
private :
   int invoice_no;
   char cust_name[20];
   char cust_addr[20];
   int item;
   int qty;
   float balance;
   char status[10];
public :
   invoice(int new_invoice_no, char *new_cust_name, char *new_cust_addr,
           int item, int qty);
  float get_balance();
  char *get_status();
  void change_balance(float amount);
  void change_status(char *new_status);
  };


class stockitem {
private :
   int stockitem_no;
   int stock_level;
   int safety_level;
   float price;
public :
   stockitem(int new_stockitem_no, int new_stock_level, int new_safety_level,
             float new_price);
   int get_stockitem_no();
   int get_stock_level();
   int get_safety_level();
   float get_price();
   void inc_stock_level(int qty);
   void dec_stock_level(int qty);
   void change_safety_level(int new_safety_level);
   void change_price(float new_price);
   void reorder();
   };
```

```
class company {
private :
   order *orders_file[100];
   invoice *invoices_file[100];
   stockitem *stock_file[20];
public :
   order *curr_order;
   invoice *curr_invoice;
   stockitem *curr_stockitem;
   company();
   order *get_order(order_no);
   invoice *get_invoice(invoice_no);
   stockitem *get_stockitem(stockitem_no);
   void put_order(order *curr_order);
   void put_invoice(invoice *curr_invoice);
   void put_stockitem(stockitem *curr_stockitem);
   void take_order();
   void prepare&dispatch_order();
   void authorize&deplete_stock_levels();
   void issue_invoice();
   void accept_payment(int invoice_no, float amount);
   void replenish_stock_levels(int item, int qty);
   };
```

2.2.5.2.2 Functions

Secondly, not only are there more structures identified by using the object-oriented approach, but there are also more functions. This is because each piece of data within an object must have a method to access it. More noticeable than the increased number of functions is that the interfaces to the processes in the DFD have changed dramatically. This is due to the fact that these processes are methods of the *company* object, and a such each of these methods has access to the company's data, and hence this data needs no longer to be passed between the processes. This is particularly noticeable between the four processes : *take_order*, *prepare&dispatch_order*, *authorize&deplete_stock_levels* and *issue_invoice*, which all access the data of the order object.

The resulting sample C++ code is quite different from the sample C code outlined earlier, and these differences are outlined in points underneath each process. Due to the variance between the C and C++ implementations, and in particular the variance between the nature of the data flows in both cases, it is clear that the DFD is more than adequate for a function-oriented C implementation, but is less than ideal for an object-oriented C++ implementation.

Each of the objects described above has a constructor method which creates an instance of the object. The constructors for the above objects are as follows :

```
order::order(int new_order_no, char *new_cust_name, char *new_cust_addr,
             int new_item, int new_qty)
{
   order_no = new_order_no;
   strcpy(cust_name, new_cust_name);
   strcpy(cust_addr, new_cust_addr);
   strcpy(status,"Pending");
   item = new_item;
   qty = new_qty;
   authorized = NO;
}


invoice::invoice(int new_invoice_no, char *new_cust_name,
                 char* new_cust_addr, int new_item, int new_qty,
                 float new_price)
{
   invoice_no = new_invoice_no;
   strcpy(cust_name,new_cust_name);
   strcpy(cust_addr,new_cust_addr);
   strcpy(status,"Unpaid");
   item = new_item;
   qty = new_qty;
   price = new_price;
   balance = price * qty;
}
```

```
stockitem::stockitem(int new_stockitem_no, int new_stock_level,
                     int new_safety_level, float new_price)
{
   stockitem_no = new_stockitem_no;
   stock_level = new_stock_level;
   safety_level = new_safety_level;
   price = new_price;
}


company::company()
{
   for(int i=0; i<100; i++)
      orders_file[i] = (orders *) NULL;

   for(i=0; i<100; i++)
      invoices_file[i] = (invoices *) NULL;

   for(int i =0; i<20; i++)
      stock_file[i] = new stockitem(i, 1000, 100, 10.00);
}
```

Each of the processes still has the same functionality and purpose, but the way in which the process is implemented has changed :

```
void company::take_order()
{
   int new_order_no, new_item, new_qty;
   char new_cust_name[20], new_cust_addr[20];

   cin.get(new_order_no);
   cin.get(new_cust_name);
   cin.get(new_cust_addr);
   cin.get(new_item);
   cin.get(new_qty);
```

```
  curr_order = new order(new_order_no, new_cust_name, new_cust_addr,
                         new_item, new_qty);
  put_order(curr_order);
}
```

The differences between the C and C++ implementations of the *take_order* process are as follows :

- A new object of type order is created instead of a structure of type order.
- The orders file is no longer directly accessible to this process, instead the new order must be inserted into the orders file using the *put_order()* method.

```
void company::prepare&dispatch_order()
{
  authorize&deplete_stock_levels();

  if (curr_order->get_authorization() == YES)
     {
     curr_order->change_status("Shipping");
     issue_invoice();
     }
  else
     curr_order->change_status("Outstanding");

  put_order(curr_order);
}
```

The differences between the implementations of the *prepare&dispatch_order* process are as follows :

- This process has direct access to the current order, thus it is no longer necessary to pass the order as a parameter to this process.
- The order authorization request which consisted of the item and quantity on the current order, no longer needs to be passed to the *authorize&deplete_stock_levels()* process, because the process has direct access the current order.

- Authorization has been implemented as a data item within the order object, thus to check the authorization for a particular order, the *get_authorization()* method must be invoked.

- Also, the status of the order can no longer be updated directly, instead the *change_status()* method must be invoked to alter the status of the order.

- Similarly this process no longer has direct access to the orders file, instead the current order must be updated in the orders file using the *put_order()* method.

```
void company::authorize&deplete_stock_levels()
{
  curr_stockitem = get_stockitem(curr_order->get_item());

  if (curr_stockitem->get_stock_level() >= curr_order->get_qty())
     {
     curr_order->change_authorization(YES);
     curr_stockitem->dec_stock_level(curr_order->get_qty());
     if (curr_stockitem->get_stock_level() < curr_stockitem->get_safety_level())
        curr_stockitem->reorder();
     }
  else
     {
     curr_order->change_authorization(NO);
     curr_stockitem->reorder();
     }

  put_stockitem(curr_stockitem);
}
```

Differences between the implementations of the *authorize&deplete_stock_levels* process are as follows :

- This process has direct access to the current order, thus it is no longer necessary to pass the order authorization request consisting of the item and quantity on the order as parameters to this process.

- The stock levels and safety levels of the various stock items can no longer be accessed or updated directly, instead methods to perform these actions are invoked.

- Similarly the stock file is no longer directly accessible to this process, instead the stockitem on the current order must be retrieved using the *get_stockitem()* method, and updated using the *put_stockitem()* method.

- Due to the fact that the authorization is now a part of the order object, it can be updated within this process by invoking the *change_authorization()* method, and hence an order authorization response is no longer returned from this process.

```
void company::issue_invoice()
{
   int new_order_no, new_item, new_qty, new_price;
   char new_cust_name[20], char new_cust_addr[20];

   new_invoice_no = curr_order->get_order_no();
   new_cust_name = curr_order->get_cust_name();
   new_cust_addr = curr_order->get_cust_addr();
   new_item = curr_order->get_item();
   new_qty = curr_order->get_qty();

   curr_stockitem = get_stockitem(curr_order->get_item());
   new_price = curr_stockitem->get_price();

   curr_invoice = new invoice(new_invoice_no, new_cust_name, new_cust_addr,
                              new_item, new_qty, new_price);
   put_invoice(curr_invoice);
}
```

Differences between the implementations of the *issue_invoice* process are as follows :

- This process has direct access to the current order, and so the details of the order as per shipment confirmation need no longer be passed into this process.

- Invoice details are read directly from the current order as in the previous implementation, but a new invoice object is created instead of a invoice structure.

- The invoices file is no longer directly accessible by this process, instead the new invoice must be inserted into the invoices file using the *put_invoice()* method.

```
void company::accept_payment(int invoice_no, float amount)
{
   curr_invoice = get_invoice(invoice_no);

   if (amount == curr_invoice->get_balance())
      curr_invoice->change_status("Paid");
   else
      curr_invoice->change_status("Part-Paid");

   curr_invoice->change_balance(amount);

   put_invoice(curr_invoice);
}
```

Differences between the implementations of the *accept_payment* process are as follows :

- The invoices file is no longer directly accessible, instead it must be accessed through the *get_invoice()* method and updated through the *put_invoice()* method.

- The balance on the invoice can no longer be accessed or updated directly, instead the *get_balance()* and *change_balance()* methods must be invoked, and passed the appropriate information (if any).

- Similarly, the status of the invoice can't be updated directly. The new status is passed as a parameter to the *change_status()* method of the invoice object.

```
void company::replenish_stock_levels(int item, int qty)
{
   curr_stockitem = get_stockitem(item);
   curr_stockitem->inc_stock_level(qty);
   put_stockitem(curr_stockitem);
}
```

Differences between the implementations of the *replenish_stock_levels* process are as follows :

- Firstly the stock file is no longer directly accessible, instead it must be accessed using the *get_stockitem()* method and updated using the *put_stockitem()* method.

- Similarly the stock levels cannot be increased directly, instead the quantity of the stock item delivered as per delivery advice, is passed to the *inc_stock_level()* method, which then increases the stock level of the item by the appropriate amount.

2.3 Inadequate Inter-Model Relationships

As each of the three models; object, dynamic and functional, is developed more or less independently, explicit well-defined instructions are incorporated into the methodology to guide the developer in this task, and comprehensive examples detailing how to build each of the individual models are available. However, as each of these models represents a different view of the system, they need to be integrated together in order to get the overall picture.

Unfortunately the integration of the models is not as straight-forward as the compilation of the models. A major problem is mapping the data flow processes from the functional model, and the events and activities from the dynamic model into behaviour of particular objects in the object model [Monarchi, '92]. There are a number of reasons for the difficulty of the integration process, each of which stems from the inadequacy of the inter-model relationships as defined by Rumbaugh. In my opinion these relationships are poorly defined, supported, reconciled, and illustrated, and each of these points is discussed below.

2.3.1 Poorly Defined Relationships

The relationship between the three models as published by Rumbaugh in the book "Object-Oriented Modeling and Design", is not well-defined [D'Souza, '93]. A terse description of the inter-model relationships is briefly explained at various points in the book but the relationship is never fully expanded and developed. This results in the relationships between the models being open to various interpretations, particularly in the case of the poorer links from the functional model to the other two models.

The basic relationship as outlined by Rumbaugh consists of three couplings between the object, dynamic and functional models, detailing three bi-directional integrations. The couplings are as follows :

- ***Relationship between the Object & Dynamic Models***

  The object model describes the structure of objects in a system - their attributes, operations and relationships to other objects. Objects with common structure and behaviour are organized into classes, which are then associated with other classes. Each class in the object model which has non-trivial dynamic behaviour, is represented in the dynamic model by a state diagram, which shows the state and event sequences permitted in a system for that class of objects. In this way each state diagram describes the life history of an object.

  While the object model describes the structure of the objects and the nature of their inter-relationships, the dynamic model is concerned with changes to the objects and their relationships over time. States in the state diagram are abstractions of the attribute values and links of an object. Transitions between states, caused by events, involve a change in the state of the object, thus each event received by an object can be associated with an operation on that object. Similarly, an event sent by an object is represented by an operation on another object.

- ***Relationship between the Dynamic & Functional Models***

  In the dynamic model, actions are associated with events and activities are associated with states. Actions and activities in state diagrams correspond to functions in the functional model, where a function can be thought of as logical group of processes, and hence each action or activity in the dynamic model may expand into an entire data flow diagram.

- ***Relationship between the Object & Functional Models***

  In the functional model, the network of processes within the data flow diagram represents the body of an operation. The flows in the diagram are intermediate values in the operation, and the processes in the data flow diagram constitute sub-operations. Often there is a direct correspondence at each level of nesting. A top-level process corresponds to an operation on a complex object, and lower level processes correspond to operations on more basic objects that are part of the complex object or that implement it. [See Figure 2.2] Sometimes one process corresponds to several operations, and sometimes one operation corresponds to several processes.

A fairly strong integration link exists between the object and dynamic models, as there exists a separate state diagram for each class with non-trivial dynamic behaviour, thus both models co-exist on the same level of granularity. However, it is obvious that the integration links between the object and functional models, and the dynamic and functional models, are not as clear. A major reason for the tenuosity of the links to the functional model is the difficulty of mapping concepts between a network of processes, and objects existing in a real world system. There are at least three reasons for this difficulty : firstly, since a processes' main connection to a real-world system is through a process name describing a system operation, it is not obvious how the processes relate to real world objects; secondly, most processes associate with multiple system objects; and thirdly, states of system objects, and relationships among objects, are buried within the process network and scattered among the processing details [Embley '92].

Rumbaugh maintains that operations in the object model and actions in the dynamic model correspond to functions in the functional model. However since a function can embody many processes and data flows, and each process can reference many objects, there is no direct coherent link between the three models at the same level of granularity. There is little doubt that the incongruency of the functional model (as detailed in the first section of this chapter) is a major factor leading to the poor integration between the functional model and the other two models. This point is further illustrated by the example outlined in the diagram of section 2.3.4.

2.3.2 Poorly Supported Relationships

More damaging than badly defined relationships is that the integration of the models, as outlined by the basic description, is not supported by concrete steps in the methodology. Detailed instructions are supplied for constructing each of the individual models, but integrating the models almost appears to be an afterthought, implemented by a hap-hazard process of converting actions from the dynamic model and functions from the functional model into operations in the object model.

This absence of formal procedures for inter-relating the models makes it difficult to recognise how an object, attribute or operation in the object model relates to a data flow or process in the functional model, or to an event or state in the dynamic model. [Monarchi, '92]. Furthermore OMT does not pay much attention to the final reconciliation of these operations, [Iivari, '95] leading to the operations not being very consistently integrated into the static model [Eckert, '94]. This point is only mentioned here for completeness, but is further elaborated in the next section.

2.3.3 Poorly Reconciled Relationships

A major problem with OMT is that it does not provide concrete guidelines for checking the models for consistency with one another [D'Souza, '94-2]. Aspects of functionality are identified in all modeling perspectives, operations in object modeling, actions and activities in dynamic modeling and functions in functional modeling. Even though Rumbaugh outlines how to extract operations from the object models, state diagrams and data flow diagrams, the process pays only minimal attention to possible inconsistencies between the models and to their reconciliation. [Iivari, '95].

Integration and consistency are very closely related. It follows that if the object, dynamic and functional models are not integrated through common structure and behaviour, then there exists no basis by which to test the models for consistency with each other. Obviously similar criteria must be used, both to incorporate integration into the models and to test the models for consistency. Achieving consistency across the models involves testing the models for anomalies. If no such anomalies exist then consistency is achieved, if not, the anomalies need to be redressed.

2.3.4 Poorly Illustrated Relationships

To further compound the lack of support for the inter-model relationships there is no fully documented and illustrated example in the book which explicitly shows how to integrate and reconcile the models to each other. The case studies given seem to adopt a fait-accompli approach, never explaining the intermediary steps on the road to integration. Furthermore, these examples of so-called integrated object, dynamic and functional models neither appear to be complete nor consistently integrated.

To illustrate this point, outlined below is the computer animation case study from the "Object-Oriented Modeling and Design" book by Rumbaugh [Rumbaugh, '91]. This example is typical of the other case studies detailed in the book, in that there exists a close coupling between the object and dynamic models, but the functional model bears little resemblance to either of the other two models.

By examining the three models illustrated below [Figure 2.3 - Object Model, Figure 2.4 - Dynamic Model (Scene), Figure 2.5 - Dynamic Model (Cue), Figure 2.6 - Functional Model], it is clear that the object and dynamic models have some appropriate points of integration. However it is also clear that the inappropriate level of granularity of the functional model, makes it very difficult to see how the processes and data flows of the functional model are reconciled to the operations of the object model, or to the actions and activities of the dynamic model.

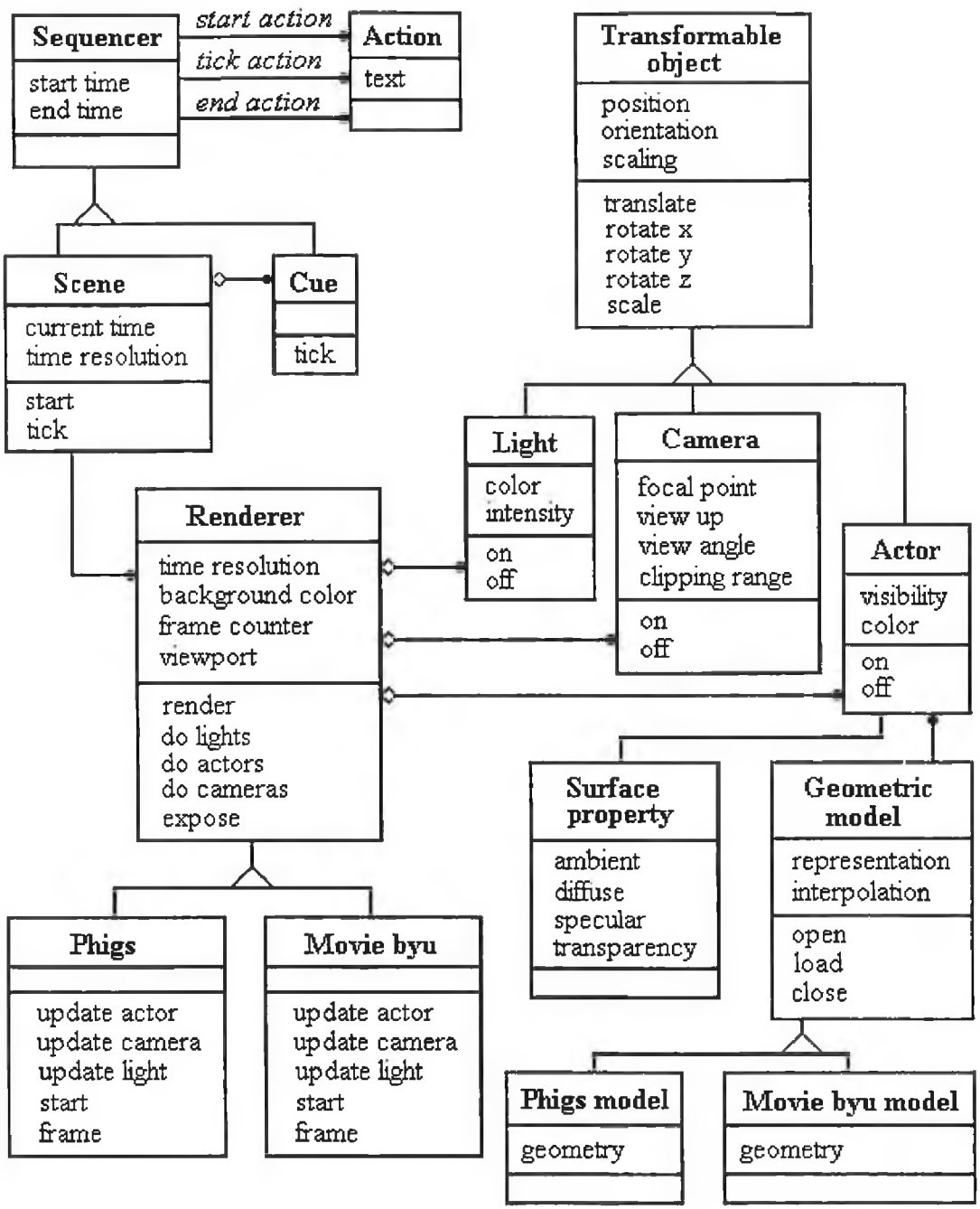


Fig 2.3 Computer Animation Object Model

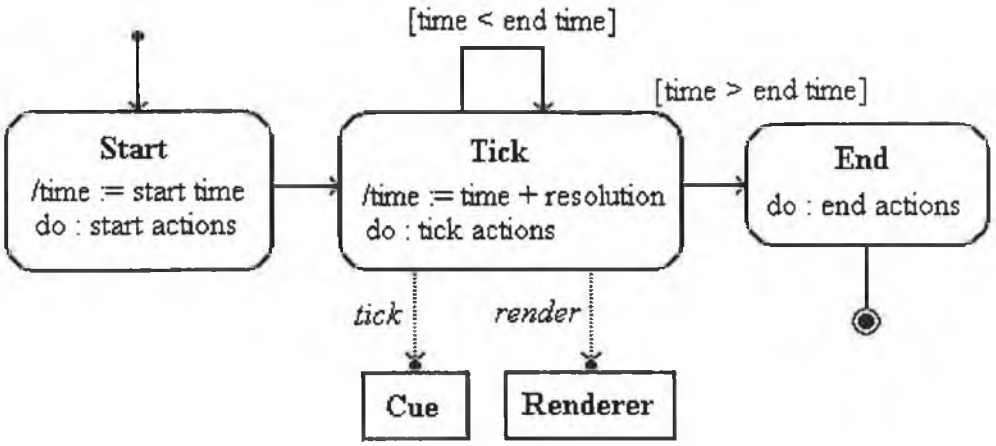


Fig 2.4 Computer Animation Dynamic Model
(State Diagram for *Scene*)

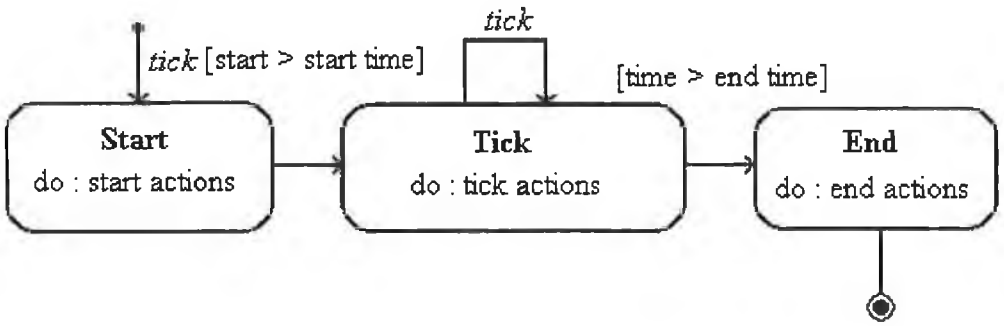


Fig 2.5 Computer Animation Dynamic Model
(State Diagram for *Cue*)

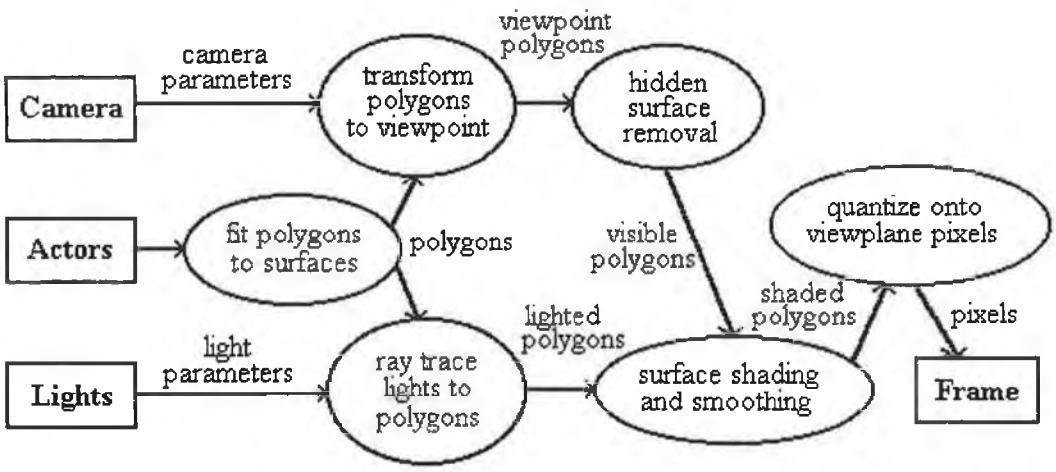


Fig 2.6 Computer Animation Functional Model

2.4 Chapter Summary

This chapter discussed in detail the two major factors which are responsible for the unsatisfactory level of integration and consistency within the OMT methodology, namely the weak functional model and the inadequate inter-model relationships. These two factors are significantly related to one another since the weakness of the functional model contributes to difficulties with the integration of the three OMT models.

Firstly, the weakness of the functional model is a direct result of the inability of data flow diagrams to adequately represent a system of interacting objects. It was stressed that data flow diagrams are excellent tools for modeling systems with a strong functional emphasis, because these diagrams can fully represent the functional paradigm. However, data flow diagrams cannot fully embrace the object-oriented paradigm, and hence difficulties arise when modeling object-oriented systems. The areas in which those difficulties are most noticeable, are the areas where the functional paradigm and the object-oriented paradigm diverge, namely decomposition; granularity; ordering; data access; interaction; control flow; and mapping.

Secondly, the inadequacy of the inter-model relationships is due to these relationships being poorly defined, poorly supported, poorly reconciled and poorly illustrated. These relationships are poorly defined since they are never fully expanded and developed; poorly supported since no procedures for integrating the models are present in the methodology; poorly reconciled since no guidelines exist for checking the models for consistency with each other; and poorly illustrated since there is no fully documented and illustrated example in the book which explicitly shows how to consistently integrate the models.

Chapter 3

Proposed Solutions to Improve OMT Integration and Consistency

3.1 Overview

The root of the unsatisfactory level of integration and consistency within the OMT methodology has been identified as : a weak functional model, caused directly by the unsuitability of using data flow diagrams to model the functionality of an object-oriented system; and inadequate inter-model relationships, which are both badly defined and unsupported by either formal steps within the methodology, or a comprehensive illustrated example. Both of these shortcomings are significantly inter-related. The weakness of the functional model results in tenuous links from the functional model to the other two models, and causes difficulties with the integration of the three OMT models. In addition, because it is difficult to recognise how the separate models integrate together, it is not easy to check the three models for consistency with one another.

Therefore, a suitable solution would be to improve the level of integration and consistency within the OMT methodology, by redressing the weak functional model and inadequate inter-model relationships, and by extending the methodology to incorporate guidelines for constructing an integrated analysis model, as well as guidelines for checking the completed model for consistency.

To this end, in the next two sections I have outlined my proposed functional model [Section 3.2] and my proposed inter-model relationships [Section 3.3]. Since both the weak functional model and inadequate inter-model relationships are significantly inter-related, it thus became necessary during the development of the proposed functional model, to consider exactly how this model would integrate into Rumbaugh's existing object and dynamic models, in order to achieve consistency across the three OMT models. Thus, there exists a trade-off between the total independence of the proposed functional model and a high level of integration and consistency within the methodology.

3.2 Proposed Functional Model

The purpose of the functional model is to show how the input values of a system are transformed via operations, into the output values of a system, and to describe these operations which transform the system, in terms of what each operation does, and how each operation interacts with other operations to make the system work, where an interaction comprises any communication between operations, such as one operation calling another operation.

As demonstrated in the previous chapter, although data flow diagrams work well in a function-oriented environment, they are severely mismatched to an object-oriented one, and thus are poor medium to either describe operations or to illustrate their interaction. DFDs do not adequately describe operations since they deal with processes not objects, and hence operations are often buried within the process network. Furthermore DFDs do not adequately illustrate how operations interact since they are incapable of illustrating the potential run-time behaviour of objects, and thus the interaction between the objects within the system is never easily visible.

Therefore, it appears that a new functional model is required, which can adequately fulfil its purpose, and which can be easily integrated into the existing object and dynamic models. However, in my opinion, it is impossible for the functional model to be encompassed in a single diagram. This is because operations can be viewed on two completely different levels, namely, what they do and how they work. Each viewpoint is equally as important as the other, since both viewpoints are required to adequately describe each operation within a system. What each operation does is a black-box viewpoint which looks at how the execution of an operation affects the values of the objects in the system, and can be represented by describing the state of the system before and after the execution of the operation, whereas how each operation works is a white-box viewpoint which looks at how the operation works internally, and can be represented by describing the flow of control among sub-ordinate operations in various objects effecting the change of state.

For these reasons, I have transformed the existing functional model from a DFD representation, into two sub-ordinate models :

- an operation model, adapted from Coleman [Coleman, '94], which is a black-box viewpoint describing the operations in terms of the state of the system before and after the execution of each operation;

- an interaction model adapted from Booch [Booch, '94], which is a white box viewpoint illustrating how the operations work internally as well as the flow of control between operations.

3.2.1 Operation Model

The operation model introduces the concept of a *system operation*, which is analogous to a process on a Level 1 DFD. System operations are top-level operations, which either correspond to interactions between the system and the outside world, or correspond to complex transformations of input values to output values. Each system operation is a non-atomic operation on a complex object, which in turn invokes other non-atomic and atomic operations on other more basic objects which are part of the complex object, where a complex object is an object which comprises other objects.

The operation model specifies the behaviour of system operations declaratively by defining their effect in terms of the change of state of the system, where the state of the system is an abstraction of the values of the objects in the system. Thus an operation can be specified by giving pre-conditions and post-conditions on its execution. A pre-condition states assumptions on the state of the system at the beginning of the operation, characterizing the conditions under which the operation may be invoked. A post-condition is a description of the state of a system at the completion of operation execution, in terms of the state of the system at the beginning of operation execution. [Rumbaugh, '95-3].

The operation model is represented as a series of schemata, with each system operation detailed on one schema. The syntax of a schema is as follows :

Operation : *ClassName::OperationName*
Description : *Text*

Reads : *Items*
Updates : *Items*
Pre-conditions : *Condition*
Post-conditions : *Condition*

The meaning of each of these clauses is explained below :

- **Operation :** *ClassName::OperationName*

  ClassName is an identifier for the class which owns the specified system operation and *OperationName* is an identifier for the system operation.

- **Description :** *Text*

  Text is an informal and concise description of the operation.

- **Reads :** *Items*

  Items is a list of all the values that the operation may access but does not change. Each *item* is the typed identifier of an object, attribute or relationship of the system state. The keyword *supplied* preceding an item indicates that the identifier is a parameter of the operation.

- **Updates :** *Items*

  Items is a list of all the values that the operation may access and change. Each *item* is the typed identifier of an object, attribute or relationship of the system state. The keyword *new* preceding an object identifier indicates that the system operation creates a new object.

- **Pre-conditions :** *Condition*

  Condition is a predicate defining the pre-conditions which must be satisfied before the operation can be legally invoked. The *Condition* can only reference parts of the system state defined previously in **Reads** and **Updates**. The pre-condition clause may be omitted if the operation can be legally invoked from any state.

- **Post-conditions :** *Condition*

  Condition is a predicate defining the post-conditions which will be satisfied after the operation has completed. The *Condition* can only reference parts of the system state defined previously in **Reads** and **Updates**.

3.2.2 Interaction Model

One object may interact with another object in a number of ways; an object may send information to another object; an object may request information from another object; an object may alter another object, and an object may cause another object to do some action [de Champeaux, '93]. Objects interact with each other via messages, where the name of the message is the name of the operation to be invoked, and like an operation, the message may have parameters and may return a result. One object, the client, sends a message to another object, the server, which takes control when it receives the message, processes it, and then allows control to return to the client [Cook, '94]. A client may send a message to many server objects, and a server may receive a message from many clients. Thus each interaction is a triple comprising the object which generated the message, the name of the message with any input or output arguments, and the object which received the message [Bear, '90]. An interaction diagram is used to illustrate the interactions between objects, which take place within a system operation, hence the interaction model consists of an interaction diagram for each system operation.

An interaction diagram [Booch, '94] is essentially an object diagram that shows the sequence of messages that implement an operation. It is similar in form to an event trace diagram, with the exception that messages are sent between objects of a particular class instead of events. [Note : The major discriminating factor between events and messages is that events always cause a change of state in the object to which they are sent, whereas messages do not always cause a change of state within the object. Hence an interaction diagram is more detailed than an event trace diagram since it lists all messages sent between objects, and not just those messages which cause a change of state.]

The object classes are written horizontally across the top of the diagram, and a dashed vertical line is drawn below each class. Messages, which denote the invocation of operations, are shown horizontally, and contain any parameters of the message. The endpoints of the message connect with the vertical lines, thus showing the client and server of the operation, with the direction of the arrow signifying which class is the client and which is the server. Ordering is indicated by the vertical position, with the first message shown at the top of the diagram, and the last message show at the bottom, thus it is unnecessary to use sequence numbers.

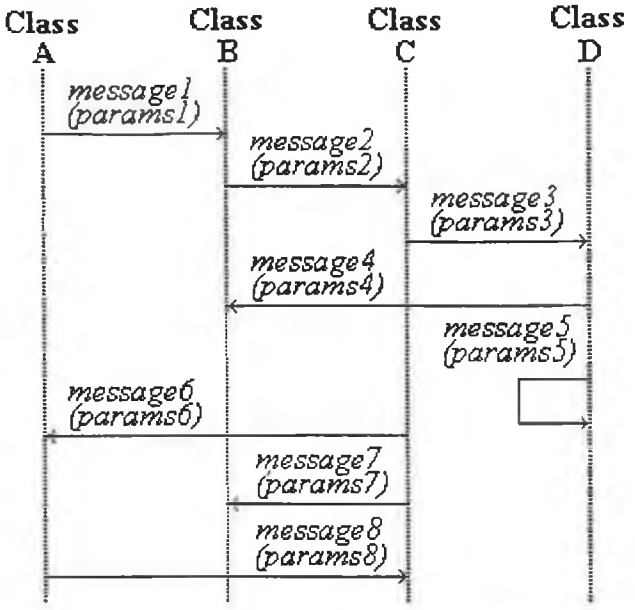


Fig 3.1 Interaction Diagram (Tabular)

Instead of being drawn in tabular form, interaction diagrams can also be drawn as directed graphs with classes as nodes and interaction connections as vertices [de Champeaux, '93]. A connection from class A to class B, labelled with the name of a message, means that instances of class A may communicate in the indicated fashion, with instances of class B. However, as sequencing is no longer by vertical position, it becomes necessary to use sequence numbers to indicate the order in which the various messages are sent and received by the objects.

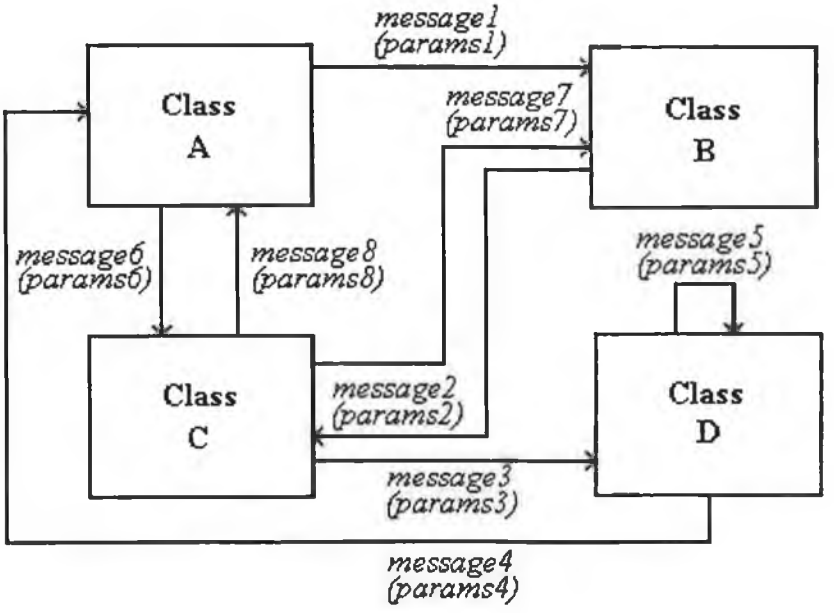


Fig 3.2 Interaction Diagram (Graphical)

Interaction diagrams describe the nature of interaction among instances of different classes, but they do not indicate the precise identities of the partners of any given interaction [de Champeaux, '93], mainly because this knowledge is not available when a class is defined and furthermore it is normal to define a class in a generic fashion so that it can be used in multiple contexts.

3.2.3 Constructing the Proposed Functional Model

The clearest way to document how to construct the proposed functional model for a given system is by means of an example. The example to be used is the order processing system initially outlined in the first chapter. The object and dynamic models as illustrated in the first chapter remain unchanged, and the proposed functional model will be constructed by referring to these existing models.

Problem Statement : Simple Order Processing System

The company takes orders from customers, and attempts to fill those orders with on-hand stock. Before the order can be filled, an order authorization request is sent to the stores giving details of the current order. If sufficient stock exists to fill the order, authorization to prepare the order is granted, the stock is removed from the stores and used to fill the order, the company's stock levels are updated to reflect this depletion, the order is shipped, and an invoice is issued to the customer. The customer either part-pays the invoice in instalments, or pays the balance in full.

However, if there is insufficient stock to fill the order, authorization to prepare the order is refused, and a standard quantity of the out-of-stock item is reorderd from the supplier. A standard quantity would be a multiple of the safety stock level of that particular item, and it is assumed that it would always be sufficient to fill any outstanding order. In addition, if the stock level of a particular item falls below the safety stock level for that item, then a standard quantity of that item is reordered. When a delivery is received from the supplier, the company's stock levels are updated to reflect this replenishment.

The object model [Figure 3.3] and the event flow diagram of the dynamic model [Figure 3.4] as previously illustrated in Chapter 1, are reproduced in this chapter as a reminder of the classes, associations, attributes, operations and events within the system, for which the new functional model will now be compiled.

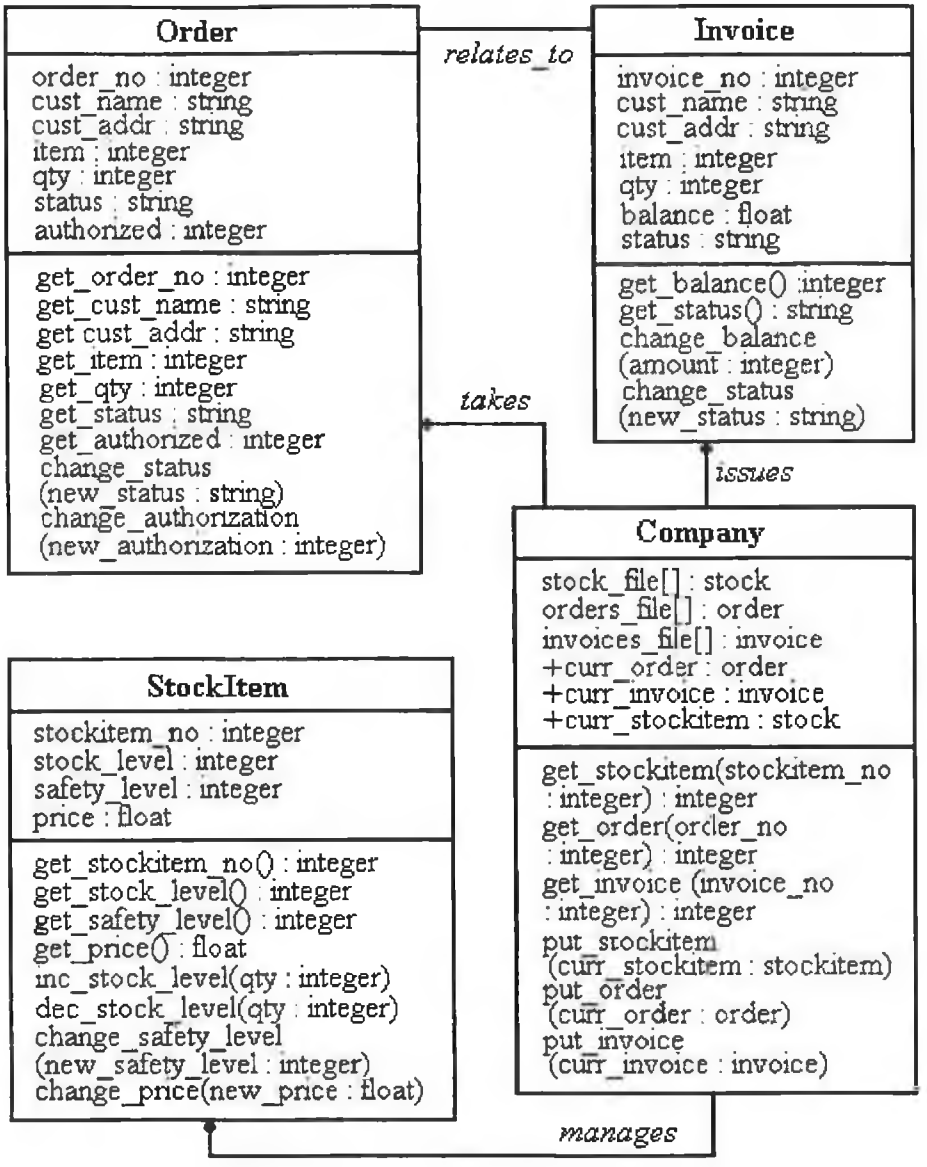


Fig 3.3 Object Model

The entire functional model will consist of an operation model and an interaction model for each system operation. Hence the construction of the new functional model comprises the following steps : identifying the input and output values; identifying the system operations; building an operation model for each identified system operation; and finally building an interaction model for each identified system operation.

3.2.3.1 Identify Input and Output Values

The input and output values of a system can be identified from the problem statement. In addition, since input and output values are parameters of the events between the system and the outside world [Rumbaugh, '91], the input and output values can be also be identified from the external events in the dynamic model of the system.

The list of input values is as follows : *order details* comprising the order number, the customer's name, the customer's address, the item ordered, and the quantity ordered; *payment details* comprising the invoice number, and the amount of the payment; and *delivery details* comprising the item being delivered and the quantity being delivered.

The list of output values is as follows : *invoice details* comprising the invoice number, the customer's name, the customer's address, the item ordered, the quantity ordered, the price of the item, and the balance due; *stock reorder details* comprising the item being reordered, and the quantity being reordered.

3.2.3.2 Identify System Operations

As stated earlier in the chapter system operations are top-level operations (analogous to the processes on a Level 1 DFD), which either correspond to interactions between the system and the outside world, or correspond to complex transformations of input values to output values, where these transformations are non-atomic operations.

Firstly dealing with the interactions between the system and the outside world. The identified list of input and output values, derived from the problem statement and the external events in the event flow diagram, will form the basis for the this first set of system operations. Thus the *order*, *invoice*, *payment*, *delivery* and *reorder* events will each be mapped into a system operation.

Secondly dealing with the complex transformations of input values to output values. The internal events in the event flow diagram which do not correspond to atomic operations will form the basis for this second set of system operations. Thus *order_authorization_request* and *order_authorization_response* will each be mapped into a system operation.

Only two internal events were selected since the *create_order*, *change_ authorization* and *change_status* events, correspond to the ***order*** constructor, ***change_ authorization()*** and ***change_status()*** atomic operations of the order class in the object model. Similarly, the corresponding operations for the *create_invoice*, *change_balance* and *change_status* events, are the ***invoice*** constructor, ***change_ status()*** and ***change_balance()*** atomic operations of the invoice class in the object model. Finally, the corresponding operations for the *decrease_stock* and *increase_stock* events, are the ***dec_stock_level()*** and ***inc_stock_level()*** atomic operations of the stockitem class in the object model. Only the *order_authorization_request* and *order_authorization_response* events have no such corresponding atomic operations in the object model. Hence they are non-atomic operations of the complex *company* object, and each will in turn invoke other non-atomic and atomic operations of other more basic objects (*order, invoice, stockitem*) which are part of the *company* object.

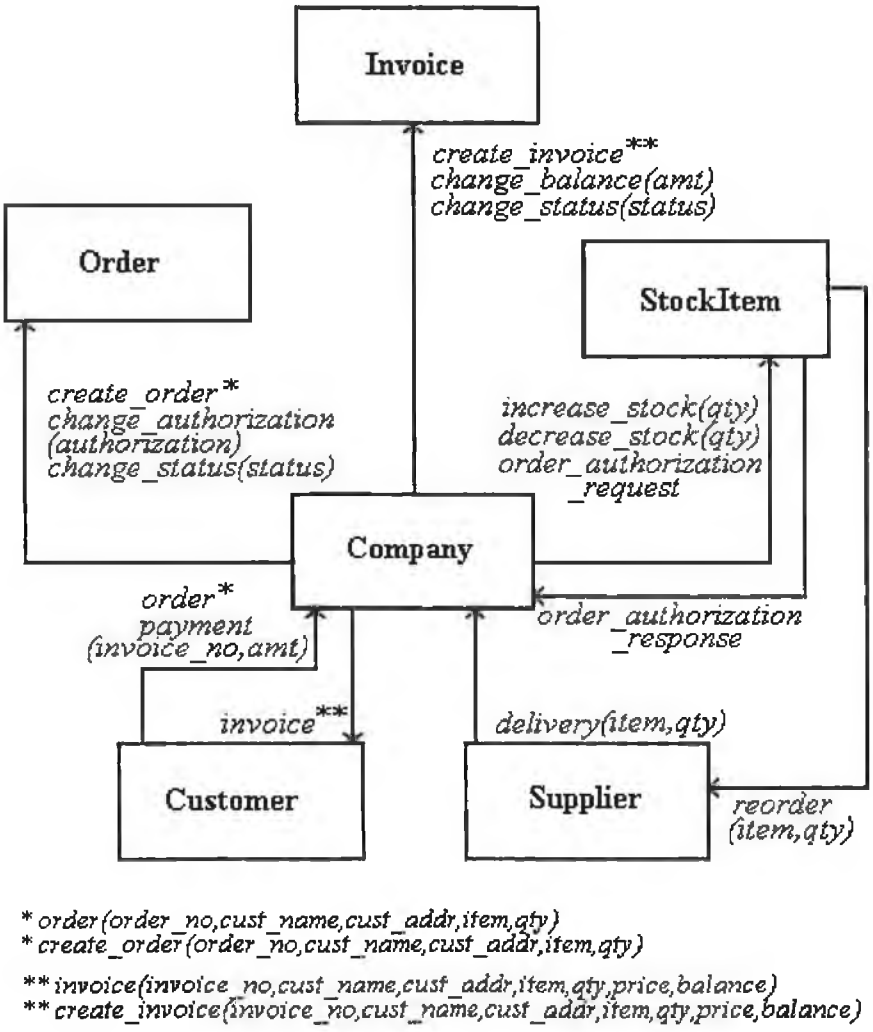


Fig 3.4 Dynamic Model (Event Flow Diagram)

Outlined below are the mappings from the external and internal events in the event flow diagram of the dynamic model, to the system operations of the functional model.

- ***order* –> company::take_order()**

  The *order* event is sent from the customer to the company and contains the details of the order, i.e. order number, customer name, customer address, item and quantity. This event is mapped into an operation on the company class - *take_order()* - which reads in the order details, creates a new order, and stores it in the orders file of the company.

- ***invoice* –> company::issue_invoice()**

  The *invoice* event is sent from the company to the customer and contains the details of the invoice, i.e. invoice number, customer name, customer address, item, quantity, price and balance. This event is mapped into an operation on the company class - *issue_invoice()* - which reads the order details, adds invoice details such as price and balance, creates a new invoice, and stores it in the invoices file of the company.

- ***payment* –> company::accept_payment(int invoice_no, float amount)**

  The *payment* event is sent from the customer to the company and contains the number of the invoice to be paid, and the amount of the payment to be made. This event is mapped into an operation on the company class - *accept_payment(int invoice_no, float amount)* - which updates the balance of the invoice (identified by the invoice number) in the invoices file of the company.

- ***delivery* –> company::replenish_stock_levels(int item, int qty)**

  The *delivery* event is sent from the supplier to the company and contains the identifier and quantity of the stockitem being delivered. This event is mapped into an operation on the company class - *replenish_stock_levels(int item, int qty)* - which updates the quantity of the identified stockitem in the stock file of the company.

- ***reorder*** **--> stockitem::reorder()**

  The *reorder* event is sent from the stockitem to the supplier and contains the identifier and quantity of the stockitem being reordered. This event is mapped into an operation on the stockitem class - *reorder()* - which merely sends the reorder to the supplier.

  Note : Unlike the *invoice* event (the only other event sent to an external actor), which causes an invoice to be created and stored, no reorder is created, and no reorder is stored, hence there is no visible data to be processed. As such, this is a trivial operation, which should be added to the stockitem class, but not awarded the status of a system operation even though it is an external event, due to the triviality of the operation it represents. Thus, this operation will not be included in the operation model or the interaction model.

The internal events which qualify as system operations are as follows :

- ***order_authorization_request*** **--> company::authorize&deplete_stock_levels()**

  The *order_authorization_request* event is sent from the company to the stockitem, requesting authorization to deplete the stock level of the stockitem by the quantity on the current order. This event is mapped into an operation on the company class - *authorize&deplete_stock_levels()* - which updates the authorization on the current order to granted, and updates the stock level of the stockitem to reflect its depletion by the quantity of the current order, when the stock level of the stockitem is sufficient to satisfy the quantity on the current order; or updates the authorization on the current order to denied, when the stock level of the stockitem is insufficient to satisfy the quantity on the current order. This operation also initiates a reorder when the stock level of any stockitem falls below its safety level.

- ***order_authorization_response* –> company::prepare&dispatch_order()**

  The *order_authorization_response* event is sent from the stockitem to the company, responding to the authorization request for the current order. This event is mapped into an operation on the company class - *prepare&dispatch_order()* - which updates the status of the order to shipping and issues an invoice to the customer, when the response is favourable; or updates the status of the order to outstanding, when the response is unfavourable.

3.2.3.3 Build the Operation Model

Construct a schema for each of the identified system operations. Firstly, concisely describe the purpose of the operation; secondly, list the data items which must be read and/or updated, and thirdly, describe the state of the system before the operation is executed and after the operation is executed.

Operation :	**company::take_order**
Description :	Creates a new order for a customer and stores this new order in the orders file of the company.
Reads :	
Updates :	**new** curr_order, orders_file
Pre-conditions :	
Post-conditions :	curr_order.order_no has been assigned a unique value. curr_order.cust_name has been set to new_cust_name. curr_order.cust_addr has been set to new_cust_addr. curr_order.item has been set to new_item. curr_order.qty has been set to new_qty. curr_order.status has been set to pending. curr_order.authorized has been set to NO. curr_order has been added to the orders file.

Operation :	**company::prepare&dispatch_order**
Description :	Requests authorization to prepare the current order, and if granted, dispatches the order and issues an invoice.
Reads :	curr_order.authorized
Updates :	curr_order.status, orders_file
Pre-conditions :	curr_order is not null
Post-conditions :	If (curr_order.authorized == YES) then curr_order.status is set to shipping, and an invoice is issued to the corresponding customer. Otherwise curr_order.status is set to outstanding. The corresponding order in the orders file of the company is updated with the new status of curr_order.

Operation :	**company::authorize&deplete_stock_levels**
Description :	Grants or refuses authorization to prepare the current order. Also depletes the stock level of the ordered item by the ordered quantity (if granted) and reorders stock if necessary.
Reads :	curr_order.item, curr_order.qty, curr_stockitem.safety_level
Updates :	curr_order.authorization, curr_stockitem.stock_level, stock_file
Pre-conditions :	curr_order is not null
Post-conditions :	If (curr_stockitem.stock_level >= curr_order.qty) then curr_order.authorized is set to YES and curr_stockitem.stock_level is depleted by curr_order.qty and curr_stockitem is reordered when this depletion causes curr_stockitem.stock_level < curr_stockitem.safety_level. Otherwise curr_order.authorized is set to NO and curr_stockitem is reordered. The corresponding stockitem in the stock file of the company is updated with the new stock level of curr_stockitem.

Operation :	**company::issue_invoice**
Description :	Creates a new invoice for a customer and stores this new invoice in the invoices file of the company.
Reads :	curr_order.order_no, curr_order.item, curr_order.qty, curr_order.cust_name, curr_order.cust_addr, curr_stockitem.price
Updates :	**new** curr_invoice, invoices_file
Pre-conditions :	curr_order is not null
Post-conditions :	curr_invoice.invoice_no has been set to curr_order.order_no. curr_invoice.cust_name has been set to curr_order.cust_name. curr_invoice.cust_addr has been set to curr_order.cust_addr. curr_invoice.item has been set to curr_order.item. curr_invoice.qty has been set to curr_order.qty. curr_invoice.price has been set to curr_stockitem.price. curr_invoice.balance has been set to curr_invoice.qty * curr_invoice.price. curr_invoice.status has been set to unpaid. curr_invoice has been added to the invoices file.

Operation :	**company::accept_payment**
Description :	Reduces the balance on the invoice by the amount of the payment, and adjusts the payment status of the invoice.
Reads :	**supplied** invoice_no : integer, **supplied** amount : float
Updates :	curr_invoice.balance, curr_invoice.status, invoices_file
Pre-conditions :	invoice_no is a valid invoice number amount > zero and amount <= balance
Post-conditions :	If (amount == curr_invoice.balance) then curr_invoice.status is set to paid. Otherwise curr_invoice.status is set to part-paid. curr_invoice.balance is reduced by the amount. The corresponding invoice in the invoices file of the company is updated with the new status and balance of curr_invoice.

Operation :	**company::replenish_stock_levels**
Description :	Increases the stock level of the given stockitem by the given quantity.
Reads :	**supplied** item : integer, **supplied** qty : integer
Updates :	curr_stockitem.stock_level, stock_file
Pre-conditions :	item is a valid stockitem and qty > zero
Post-conditions :	curr_stockitem.stock_level is increased by qty. The corresponding stockitem in the stock file of the company is updated with the new stock level of curr_stockitem.

3.2.3.4 Build the Interaction Model

Where the operation model specifies what an system operation does, the interaction model describes how the operation works by illustrating the internal workings of the operation in terms of its sub-ordinate operations. Each interaction diagram is constructed essentially from the operation model. By examining the *reads* and *updates* clauses of this model, it becomes obvious which sub-ordinate operations are required, in order to read and update the necessary information relating to this system operation. These sub-ordinate operations are then listed in descending order, between the appropriate classes on the tabular diagram. The order of these operations is determined by the function of the operation, as outlined in the *description* clause and is re-inforced by the order of the *post-conditions* which relate to the information in both the reads and updates clauses.

- **void company::take_order()**

  The *take_order()* operation reads in the order details, constructs a new order object from this information, then stores this order object in the orders file of the company. Hence the sub-ordinate operations are ***order::order()***, which creates the new order, and ***company::put_order()*** which updates the orders file with the new order.

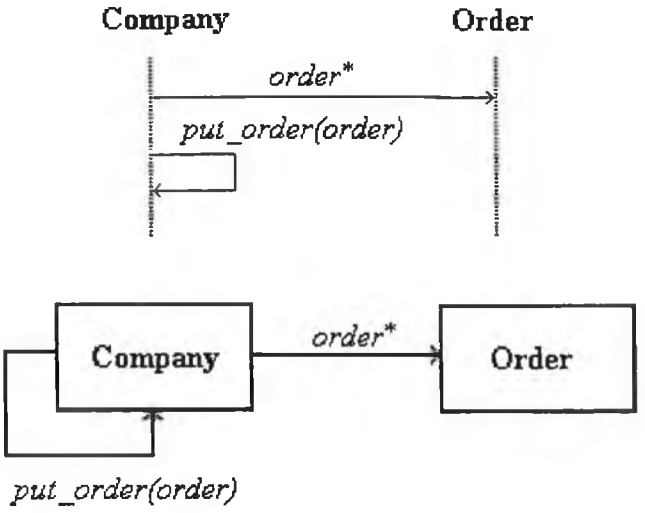
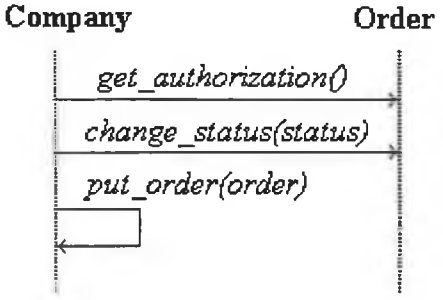


Fig 3.5 Interaction Diagram for *take_order()*

- **void company::prepare&dispatch_order()**

  The *prepare&dispatch_order()* operation requests authorization to prepare the current order by calling the *authorize&deplete_stock_levels()* operation (see below), then it reads the authorization of the current order, [updated by *authorize&deplete_stock_levels()*] updates the status of the order accordingly, and updates the orders file with the updated order. Hence the sub-ordinate operations are ***order::get_authorization()***, which reads the authorization of the current order, ***order::change_ status()*** which updates the status of the current order, and ***company::put_order()***, which updates the orders file.



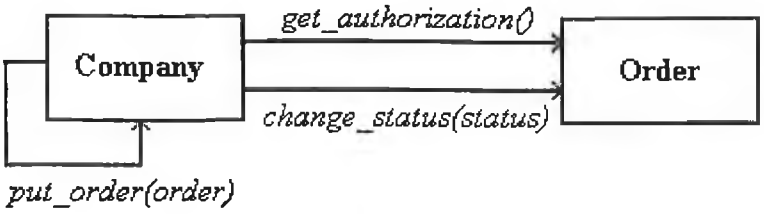


Fig 3.6 Interaction Diagram for *prepare&dispatch_order()*

- **void company::authorize&deplete_stock_levels()**

  The *authorize&deplete_stock_levels()* operation grants or denies authorization for the current order. It reads the item on the current order from the stock file and determines whether the stock level of the item, is above or below the quantity on the current order; if above, it grants authorization for the current order, and decreases the stock level of the item by the quantity; if below, it denies authorization on the current order. Also, it determines if the stock level of the item is below the safety level of the item, and if so, reorders the item. It then updates the stock file with the updated item.

  Hence the sub-ordinate operations are ***company::get_stockitem()***, which reads the item on the current order from the stock file, ***stockitem::get_stock_level()***, which reads the stock level of the item, ***order::get_qty()***, which reads the quantity of the current order, ***order:: change_authorization()***, which updates the authorization of the current order, ***stockitem::dec_stock_level()***, which updates the stock level of the item by decreasing it by the quantity of the order, ***stockitem::get_safety_level()***, which reads the safety level of the item, ***stockitem::reorder()***, which reorders the item, and ***company::put_stockitem()***, which updates the stock file.

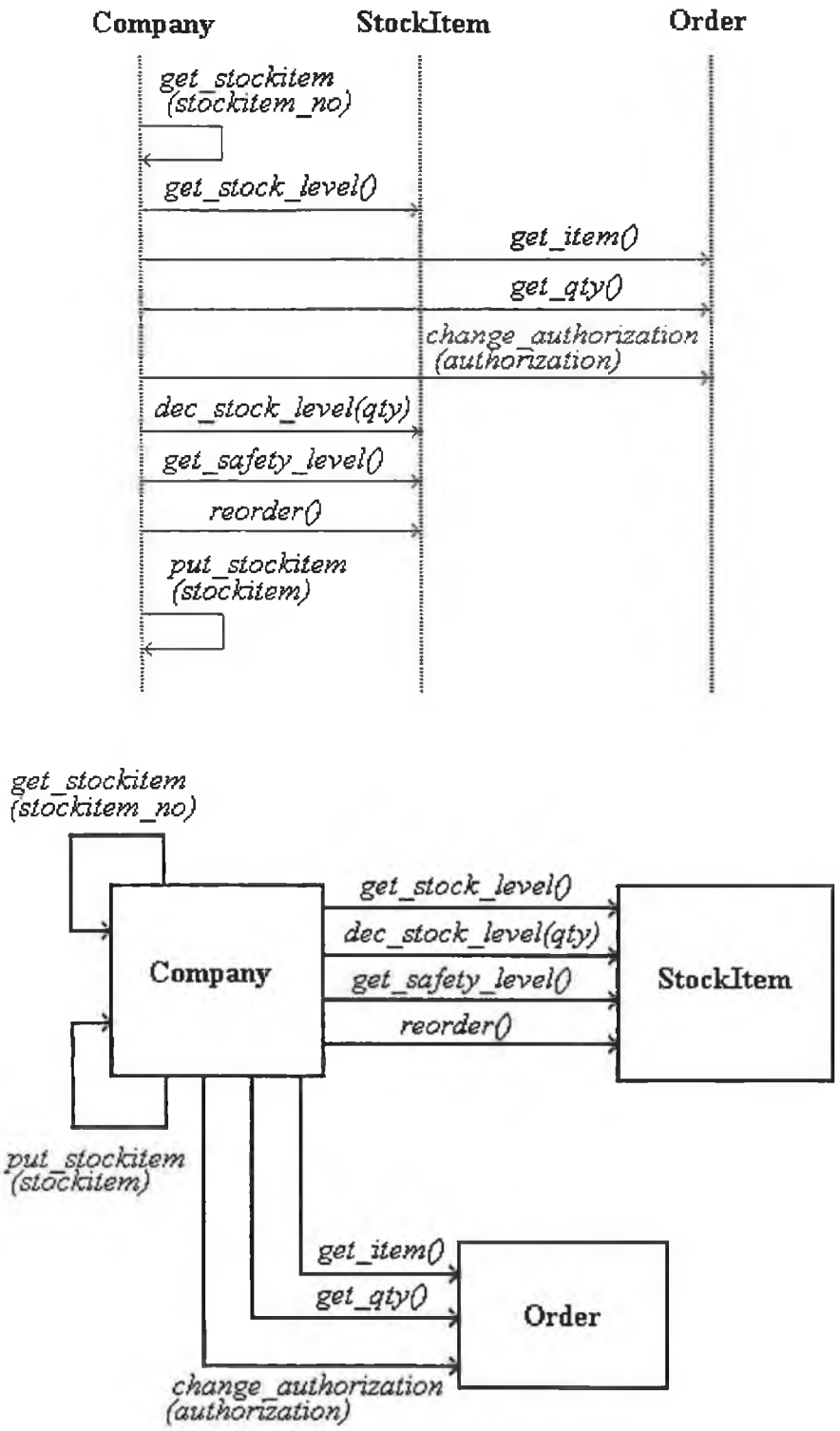
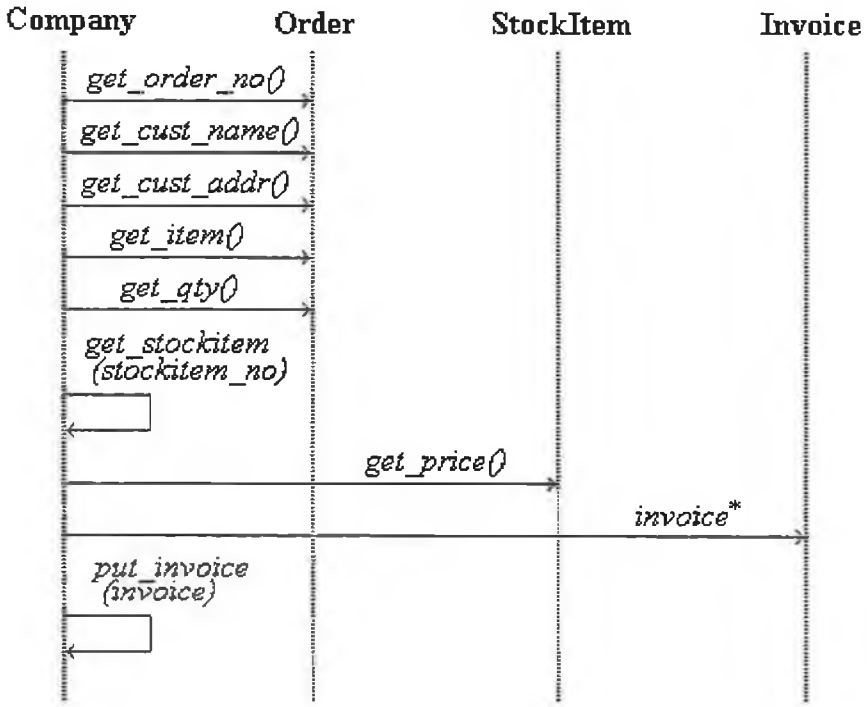


Fig 3.7 Interaction Diagram for *authorize&deplete_stock_levels()*

- **void company::issue_invoice()**

The *issue_invoice()* operation reads the order details from the current order, and the price details from the current stockitem. It then constructs a new invoice object from this information, and stores this invoice object in the invoices file of the company. Hence the sub-ordinate operations are ***order::get_order_no()***, which reads the order number of the current order, ***order::get_cust_name()***, which reads the customer name of the current order, ***order::get_cust_addr()***, which reads the customer address of the current order, ***order::get_item()***, which reads the item of the current order, ***order::get_qty()***, which reads the quantity of the current order, ***company::get_stockitem()***, which reads the item on the current order from the stock file, ***stockitem::get_price()***, which reads the price of the current stockitem, ***invoice::invoice()***, which creates the new invoice, and ***company::put_invoice()*** which updates the invoices file with the new invoice.



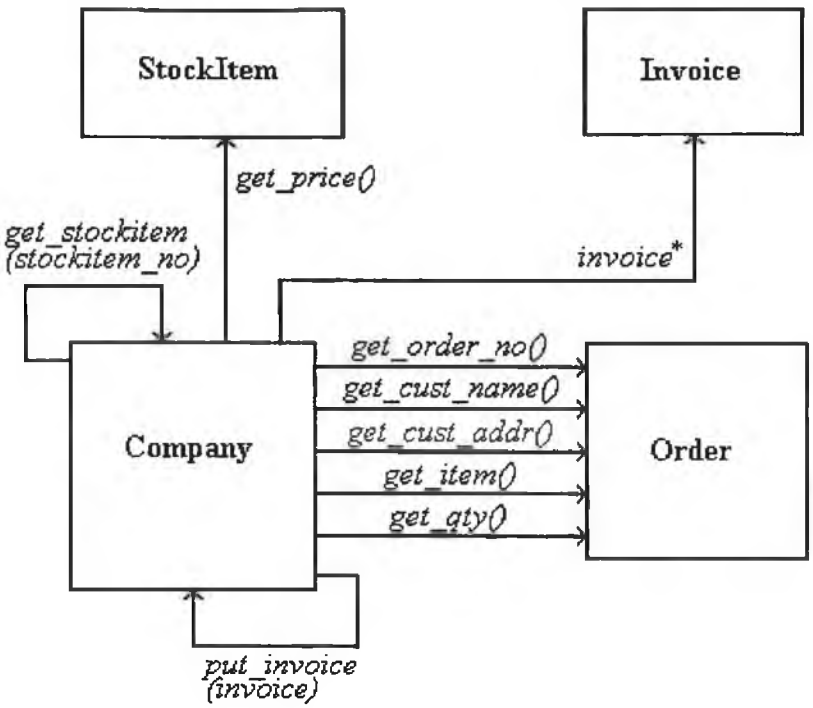


Fig 3.8 Interaction Diagram for *issue_invoice()*

- **void company::accept_payment(int invoice_no, float amount)**

  The *accept_payment()* operation reads the specified invoice from the invoices file, determines whether the specified amount satisfies the balance, updates the status of the invoice to paid or part-paid, updates the balance by the specified amount, and updates the invoices file with the updated invoice. Hence the sub-ordinate operations are ***company::get_invoice()***, which reads the specified invoice, ***invoice::get_balance***, which reads the balance on the invoice, ***invoice::change_status()***, which updates the status of the invoice, ***invoice::change_balance()***, which updates the balance on the invoice, and ***company::put_invoice()***, which updates the invoices file.

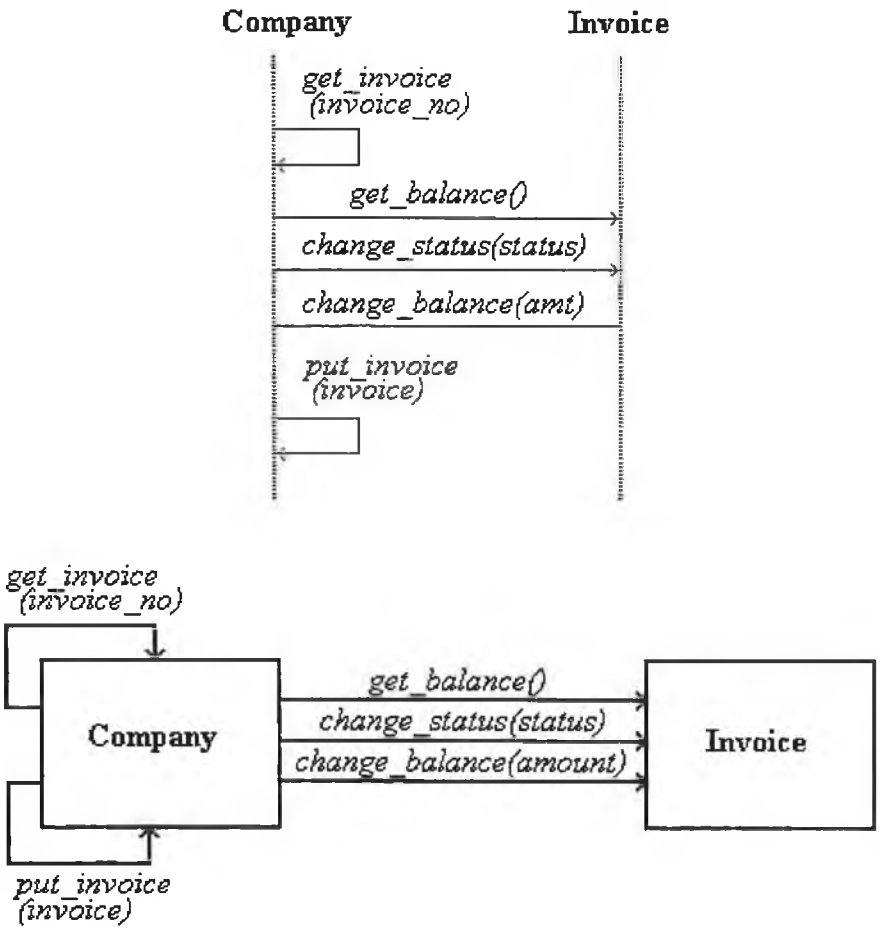


Fig 3.9 Interaction Diagram for *accept_payment()*

- **void company::replenish_stock_levels(int item, int qty)**

  The replenish_stock_levels() operation reads the specified stockitem from the stock file, updates this stockitem with the amount of the delivery, and then updates the stock file with the updated stockitem. Hence the sub-ordinate operations are ***company::get_stockitem()***, which reads the stockitem from the stock file, ***stockitem::inc_stock_level()***, which updates the stock level of the stockitem, and ***company::put_stockitem()*** which updates the stock file.

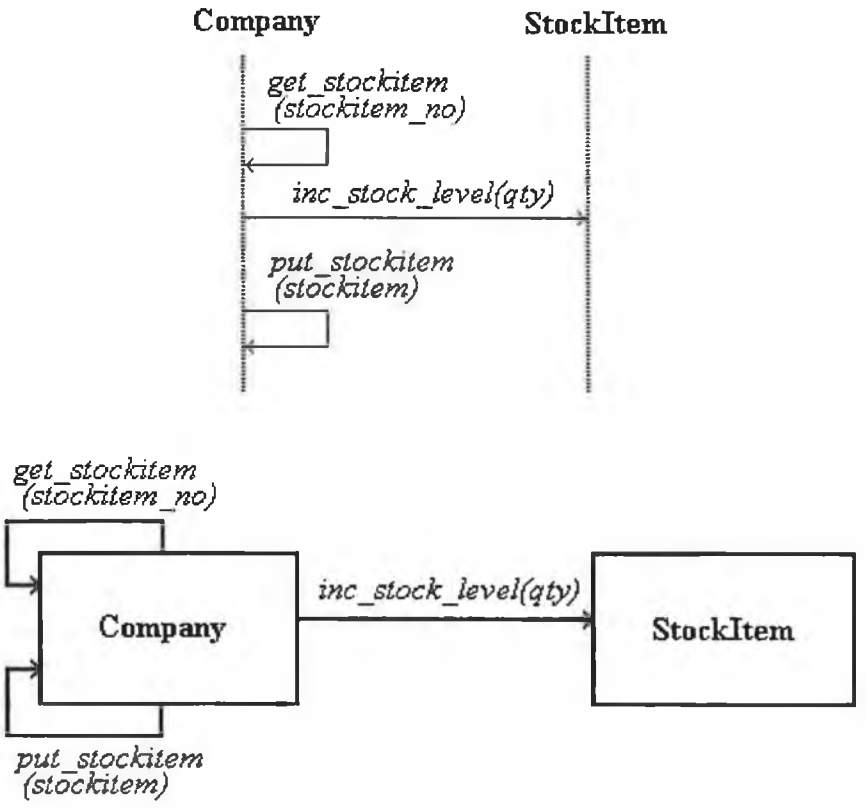


Fig 3.10 Interaction Diagram for *replenish_stock_levels()*

3.2.4 Proposed Functional Model vs. Rumbaugh's Functional Model

The primary problem with Rumbaugh's functional model, is that DFD's cannot represent the functionality of an object-oriented system as completely as they can represent the functionality of a function-oriented system, and hence when compiling the model, difficulties arise in areas where the functional paradigm and the object-oriented paradigm diverge.

These areas of difficulty were identified in the previous chapter as : decomposition; granularity; data access; interaction and mapping. However, the proposed functional model embraces the object-oriented paradigm, and hence major improvements can be seen in all of the previously identified areas of difficulty.

3.2.4.1 Decomposition

Decomposition refers to the criteria used to abstract the fundamental aspects from a problem. The DFD of Rumbaugh's functional model adopts a process of functional decomposition, which breaks the system down into functions and sub functions. This method of decomposition is most suited to systems where the functions are more complex than the data that the system manipulates. Hence this is not suited to object-oriented systems where the data is of prime importance, and the functions are often trivial, merely accessing and updating this fundamental data.

The proposed functional model decomposes the system in terms of objects and not in terms of functions. Each of the identified system operations in the operation model, is a non-atomic operation on a complex object within the system, and each of these system operations is subsequently illustrated in the interaction model, as a compilation of the data and methods of other objects within the system.

3.2.4.2 Granularity

Granularity refers to the level of abstraction of the model. The DFD of Rumbaugh's functional model adopts a top-down functional approach to abstracting the fundamentals aspects of a system, by beginning with a very general context diagram and expanding this diagram level by level until functional primitives are achieved. However, the object model and the dynamic model are both developed using a bottom-up class-by-class approach, hence the old functional model exists on a different level of granularity to the other two models.

The proposed functional model is developed using a bottom-up approach, where what each system operation does is defined in the operation model, and how each system operation works is defined in the interaction model. Each such system operation maps into a method of a complex class within the system, and subsequently invokes methods of the sub-ordinate classes which comprise the complex class. As the new functional model embraces the bottom-up object-oriented paradigm based on classes, instead of the top-down functional paradigm based on functions, there exists a direct coherent link between the three OMT models which was previously lacking, and hence each of the models can now co-exist on the same level of granularity.

3.2.4.3 Data Access

Data access refers to what parts of the system can read or write the data, and the restrictions (if any) imposed on the access of that data. The DFD of the Rumbaugh's functional model is unable to show the restricted data access of an object-oriented system since it is not obvious how processes relate to objects, and furthermore most processes associate with multiple system objects, hence the data and relationships among objects is buried within the process network.

In the operation model of the proposed functional model, the data items of each object which are either read or updated is clearly listed, as too are the changes that each of these items undergoes as a result of the operation being performed. In addition, the interaction model lists the methods of each object that are invoked, in order to read or update the data of the object. In this way the restricted data access of the object-oriented paradigm can be fully illustrated.

3.2.4.4 Interaction

Interaction refers to the mechanism by which the constituent parts of a system communicate with each other to make the system work. The DFD of Rumbaugh's functional model uses a function-invoking mechanism, where a process in the DFD interacts with another process in the DFD, and passes it some data, which is identified by the data flow between the processes. However, a DFD only shows the movement of data within a system, and cannot illustrate the interaction between objects, since the messaging mechanism which objects use to communicate with each other, need not always involve an exchange of data between objects.

The interaction model of the proposed functional model can fully illustrate how the various objects within the system interact. For each system operation it illustrates the objects which need to communicate, and the messages they need to send each other, in order to make the operation work. The messages consist of the name of the method which needs to be invoked and the parameters (if any) of the method, hence the interaction of objects is visible regardless of whether there is an exchange of data or not.

3.2.4.5 Mapping

Mapping refers to the ease or difficulty of the transition from analysis to design and implementation. The DFD of Rumbaugh's functional model lacks the expressive power to adequately model an object-oriented system, hence if the objects cannot be easily modeled, then the transition to design and implementation of these objects will be non-trivial.

The operation model of proposed functional model fully describes what each system operation does, in terms of the data that it reads and updates, as well as the pre-conditions and post-conditions associated with each system operation, indicating the required state of affairs before the operation can be invoked, and the resulting state of affairs after the operation has completed. In addition, the interaction model lists the messages exchanged between the objects, thus showing how the various objects interact with each other, and hence provides a better illustration of the potential run-time behaviour of the objects in the system. The increased detail in the proposed functional model should facilitate the implementation of the analysis model.

3.3 Proposed Inter-Model Relationships

As detailed in the previous chapter, the existing inter-model relationships as defined by Rumbaugh are inadequate in four distinct categories, each of which needs to be redressed individually. Thus the poorly defined inter-model relationships are redressed by new and improved inter-model definitions; the poorly supported relationships are redressed by integration guidelines, detailing how to integrate the three OMT models; the poorly reconciled relationships are redressed by consistency guidelines, detailing how to check the three OMT models for consistency with each other; and the poorly illustrated relationships are redressed by providing a comprehensive illustrated example.

3.3.1 New Inter-Model Definitions

Inter-model definitions define the relationships between the object, dynamic and functional models. Rumbaugh's inter-model definitions are not rigorously defined, and thus are open to various interpretations, particularly the tenuous relationships from the weaker functional model to the other two OMT models.

In particular, it is worth mentioning that CASE tools supporting the OMT methodology (e.g SelectOMT, OMTool) do not provide any form of inter-model integration or inter-model consistency within their tools. One possible theory for this omission is that Rumbaugh's inter-model definitions are not well-defined enough to enable cross-model integration and consitency to be implemented.

The inadequacies of the existing inter-model definitions need to be redressed by providing more rigorous set of inter-model definitions, which are not open to interpretations, and are well-defined enough to potentially enable inter-model cross-checking facilities in OMT CASE tools, which would greatly improve integration and consistency in models developed using these tools.

3.3.1.1 Relationship between the Object and Dynamic Models

Between the object and dynamic models there are three common points of integration, namely : conditions in the dynamic model relate to attributes and operations in the object model; events in the dynamic model relate to operations in the object model; and actions in the dynamic model also relate to operations in the object model. The definition of the relationship between the object and dynamic models in terms of these integration points is explained below.

- Each condition in the state diagram, is defined only in terms of the attributes, operations and associations, listed in the class diagram of the class, which is represented in the state diagram. [Note : Each condition in the state diagram can be defined in terms of the attributes and operations of any class in the object model, provided that the chosen class has an association with the class represented in the state diagram, and the class name of the chosen class preceeds each attribute or operation of the chosen class].

- Each event in the state diagram, is mapped into an operation, and listed in the class diagram of the class, which is represented in the state diagram.

- Each action in the state diagram, is mapped into an operation, and listed in the class diagram of the class, which is represented in the state diagram.

3.3.1.2 Relationship between the Dynamic and Functional Models

Between the dynamic and functional models there are two common points of integration, namely : each operation schema in the operation model relates to external events and "complex" internal events in the dynamic model; and each interaction diagram in the interaction model also relates to external events and "complex" internal events in the dynamic model, where a "complex" internal event is an event which does not have a corresponding atomic operation in the object model. [See Page 84]. The definition of the relationship between the dynamic and functional models in terms of these integration points is explained below.

- Each external event and complex internal event (which relates to a particular class) in the event flow diagram is mapped into a system operation (relating to the same particular class) and listed as an operation schema in the operation model.

- Each external event and complex internal event (which relates to a particular class) in the event flow diagram is mapped into a system operation (relating to the same particular class) and listed as an interaction diagram in the interaction model.

3.3.1.3 Relationship between the Object and Functional Models

Between the object and functional models there are three common points of integration, namely : operations in the operation model relate to attributes in the class diagram; operations in the operation model relate to operations in the class diagram; and individual object interactions in the interaction model also relate to operations in the class diagram. The definition of the relationship between the object and functional models in terms of these integration points is explained below.

- Each system operation in the operation model, is defined only in terms of the attributes and associations listed in the class diagram of the particular class, to which it belongs. [Note : Each system operation in the operation model can be defined in terms of the attributes of any class in the object model, provided that the chosen class has an association with the class the class to which the system operation belongs, and the class name of the chosen class preceeds each attribute of the chosen class].

- Each system operation is represented by an operation schema in the operation model, and is mapped into an operation and listed in the class diagram of the particular class to which it belongs.

- Each system operation is represented by an interaction diagram in the interaction model, which illustrates the individual object interactions within each system operation. Each individual object interaction is mapped into an operation and listed in the class diagram of the particular class to which it belongs.

3.3.1.4 Overall OMT Inter-Model Relationship

Rumbaugh's poorly-defined inter-model relationships consist of non-rigorous informal couplings between the object and dynamic models, dynamic and functional models, and object and functional models. Furthermore, little effort is spent on the integration of the three models other than a hap-hazard attempt to reconcile the operations existing in all three models. In addition, this reconciliation process is not supported by steps in the methodology, and pays only minimal attention to possible inconsistencies between the models [Iivari, 95].

The new inter-model definitions also consist of three (although somewhat different) bi-directional couplings between the object, dynamic and functional models, however the individual OMT models are more closely integrated together through common attributes, common operations, common associations and common events, relating to common classes which exist in each of the three OMT models.

Firstly examining the attributes within the OMT models. In terms of the object model, the class diagram defines all the attributes of that particular class. In terms of the dynamic model, each condition in the state diagram representing a particular class, is defined only in terms of the attributes, operations and associations listed in the class diagram of that particular class. In terms of the functional model, each system operation in the operation model, is defined only in terms of the attributes and associations listed in the class diagram of that particular class, by using the *Reads*, *Updates*, *Pre-conditions*, and *Post-conditions* clauses.

Secondly examining the operations within the OMT models. In terms of the object model, the class diagram defines the signature of all the operations of that particular class. In terms of the dynamic model, each event in the state diagram representing a particular class, is mapped into an operation, and listed in the class diagram of that particular class. In terms of the functional model, each system operation is listed in the class diagram of the particular class to which it belongs.

Thirdly, examining the events within the OMT models. In terms of the dynamic model, the state diagram representing a particular class, defines all the events of that particular class. In terms of the object model, each event in the state diagram representing a particular class is mapped into an operation, and listed in the class diagram of that particular class. In terms of the functional model, each external event and each complex internal event, is mapped into a operation schema in the operation model, and is mapped into an interaction diagram in the interaction model.

3.3.2 Integration Guidelines

Introduced in this section are a set of integration guidelines which relate directly to the new inter-model definitions which were described in the previous section. These integration guidelines should be applied when constructing the three OMT models, because integration is a incremental process which should be practiced throughout the development phase of the models.

Once the object model has been constructed, the integration guidelines can be used when constructing the dynamic model, since the dynamic model will draw on information contained in the object model. Similiarly, the integration guidelines can be used when constructing the functional model, since the functional model will draw on information contained in both the object model and the dynamic model. Finally, the integration guidelines can be used in the iterative process of refining the models, where any new information introduced by the dynamic and functional models is integrated into the object model.

The next three sections summarise the inter-model definitions as short guidelines, and provide an example of the integration process between the three OMT models, by refering to the simple order processing system previously defined, and the object, dynamic and functional models which have previously been illustrated.

3.3.2.1 Integrating the Object and Dynamic Models

There are three important guidelines to note when integrating the object model with the dynamic model, which embody the common points of integration outlined in their inter-model definition.

- ***Each condition in the state diagram, should be defined only in terms of the attributes, operations, and associations listed in the class diagram of the class, which is represented in the state diagram.***

For example, in the state diagram for the *order* class, there are two conditions : [authorized = YES] and [authorized = NO], which guard the transitions from the *authorizing* state, to the *shipping* and *outstanding* states respectively. Both of these conditions are defined only in terms of the attributes of the *order* class, since *authorized* is an attribute of this class.

In the state diagram for the *invoice* class, there are two conditions : [balance = 0] and [balance > 0], which guard the transitions from the *pay* state, to the *paid* and *part-paid* states respectively. Both of these conditions are defined only in terms of the attributes of the *invoice* class, since *balance* is an attribute of this class.

In the state diagram for the *stockitem* class, there are four conditions : [stock_level - qty > safety_level], [stock_level - qty < safety_level && stock_level -qty > 0], [stock_level -qty = 0] and [stock_level + qty > safety_level], which guard the transitions from the *above safety level* state, to itself, and the *below safety level* state respectively. All of these conditions are defined only in terms of the attributes of the *stockitem* class, since *stock_level* and *safety_level* are attributes of this class, and *qty* is a parameter of the *dec_stock_level()* operation of this class.

In the state diagram for the *company* class, there are two conditions : [curr_order.qty > curr_stockitem.stocklevel] and [curr_order.qty <= curr_stockitem.stock_level], which guard the transitions from the *sales* state, to the *goods inwards* and *goods outwards* states. Both of these conditions are defined only in terms of the attributes of the *company* class, since *curr_order* and *curr_stockitem* are attributes of this class.

- ***Each event in the state diagram should be mapped into an operation, and listed in the class diagram of the class, which is represented in the state diagram.***

For example, in the state diagram for the *order* class, there are two events : *change_authorization(authorization)* and *change_status(status)*, which are mapped into the change_authorization(authorization) and change_status(status) operations in the *order* class diagram.

In the state diagram for the *invoice* class, there are two events : *change_status(status)* and *change_balance(amt)*, which are mapped into the change_status(status) and change_balance(balance) operations in the *invoice* class diagram.

In the state diagram for the *stockitem* class, there are two events : *decrease_stock (qty)* and *increase_stock(qty)*, which are mapped into the dec_stock_level(qty) and inc_stock_level(qty) operations in the *stockitem* class diagram.

In the state diagram for the *company* class, all of the events are external events or complex internal events, and hence are mapped into system operations and illustrated as operation schemas in the operation model, and interaction diagrams in the interaction model. [See Section 3.3.2.2]

- ***Each action in the state diagram should be mapped into an operation, and listed in the class diagram of the class, which is represented in the state diagram.***

The events and actions in the state diagrams of the *order, invoice* and *stockitem* classes, have very close relationships to each other, and are mapped into the same operations. Hence, in the state diagram for the *order*, the actions change_authorization(authorization) and change_status(status) are mapped into corresponding operations and listed in the *order* class diagram. Similarly, in the state diagram for the *invoice*, the actions change_status(status) and change_balance(amt) are mapped into corresponding operations and listed in the *invoice* class diagram. And finally, in the state diagram for the *stockitem*, the actions dec_stock_level(qty) and inc_stock_ level(qty) are mapped into related operations and listed in the *stockitem* class diagram.

In the state diagram for the *company* class, (excluding actions which merely send events to sub-ordinate objects, and hence have been already covered), there are six other actions namely, get_order(), get_invoice(), get_stockitem(), put_order(order), put_invoice(invoice), and put_stockitem(stockitem), which are mapped into corresponding operations and listed in the *company* class diagram.

3.3.2.2 Integrating the Dynamic and Functional Models

There are two important guidelines to note when integrating the dynamic model with the functional model, which embody the common points of integration as outlined in their inter-model definition.

- ***Each external event and complex internal event, in the event flow diagram, should be mapped into a system operation and illustrated as an operation schema in the operation model.***

- ***Each external event and complex internal event, in the event flow diagram, should be mapped into a system operation and illustrated as an interaction diagram in the interaction model.***

For example, in the event flow diagram there are five external events : *order*, which is mapped into the system operation company::take_order(); *invoice*, which is mapped into the system operation company::issue_invoice(); *payment*, which is mapped into the system operation company::accept_payment(); *delivery*, which is mapped into the system operation company::replenish_stock_levels(); and *re-order*, which is not mapped into a system operation due to its triviality, but is mapped into stockitem::reorder() instead.

In addition, there are two complex internal events : *order_authorization_request*, which is mapped into the system operation company::authorize&deplete_stock_ levels(); and finally order_authorization_response, which is mapped into the system operation company::prepare& dispatch_order().

Each of these system operations is illustrated as an operation schema in the operation model and illustrated as an interaction diagram in the interaction model.

3.3.2.3 Integrating the Object and Functional Models

There are three important guidelines to note when integrating the object model with the functional model, which embody the common points of integration as outlined in their inter-model definition.

- ***Each system operation in the operation model, should be defined only in terms of the attributes and associations listed in the class diagram of the particular class, to which it belongs.***

Thus, each of the system operations in the operation model should be defined only in terms of the attributes and associations of the *company* class. The definition of an operation involves listing the data items it *reads*, the data items it *updates*, the *pre-conditions* of the operation, and the *post-conditions* of the operation.

As it is a requirement of the operation model that the *pre-conditions* and *post-conditions* be defined only in terms of the items listed in the *reads* and *updates* clauses, it can be assumed that once the *reads* and *updates* clauses are defined in terms of the attributes and associations of a particular class, then the *pre-conditions* and *post-conditions*, are also defined in terms of the attributes and associations of that particular class.

For example, the company::take_order() operation creates a new order, updates *curr_order* with the details of this new order, and then updates the *orders_file* with updated *curr_order*. This operation is defined solely in terms of the attributes of the *company* class since both *curr_order* and *orders_file* are attributes of the *company* class.

The company::prepare&dispatch_order() operation reads *curr_order.authorized* in order to determine the authorization status of the current order, and then updates *curr_order.status* to shipping if the current order is authorized, or updates *curr_order.status* to outstanding if the current order is not authorized. It then updates the *orders_file* with the updated *curr_order*. This operation is defined solely in terms of the attributes of the *company* class since both *curr_order* and *orders_file* are attributes of the *company* class.

The company::authorize&deplete_stock_levels() operation reads *curr_order.item* and *curr_order.qty* in order to determine the item and quantity on the order, and then reads *stockitem.stocklevel* to determine whether the item and quantity on the current order can be satisfied. If sufficient stock is available, it then updates *curr_order .authorization* to YES, and updates *curr_stockitem.stocklevel* by reducing it by the quantity of the order. If insufficient stock is available, it then updates *curr_order .authorization* to NO. It then updates the *stock_file* with the updated *curr_stockitem*. This operation also reads *curr_stockitem.safety_level*, and re-orders a stockitem when the stock level of that stockitem falls below it's safety level. This operation is defined solely in terms of the attributes of the *company* class since *curr_order, curr_stockitem* and *stock_file* are all attributes of the *company* class.

The company::issue_invoice() operation reads *curr_order.order_no*, *curr_order.item*, *curr_order.qty*, *curr_order.cust_name*, *curr_order.cust_addr*, and *curr_stockitem .price*, creates a new invoice from these details, updates *curr_invoice* with the details of this new invoice, and then updates the *invoices_file* with *curr_invoice*. This operation is defined solely in terms of the attributes of the *company* class since *curr_order*, *curr_stockitem* and *invoices_file* are all attributes of the *company* class.

The company::accept_payment() operation accesses the details of the invoice being paid from the *invoices_file*, and updates *curr_invoice* with the details of this invoice. It then updates *curr_invoice.balance* by reducing it by the amount of the payment, and also updates *curr_invoice.status* to either paid or part-paid. Finally it updates the *invoices_file* with the updated *curr_invoice*. This operation is defined solely in terms of the attributes of the *company* class since both *curr_invoice* and *invoices_file* are attributes of the *company* class.

The company::replenish_stock_levels() operation accesses the details of the stockitem being delivered, from the *stock_file*, and updates *curr_stockitem* with the details of this stockitem. It then updates *curr_stockitem.stock_level* by increasing it by the quantity of the delivery, and updates the *stock_file* with the updated *curr_stockitem*. This operation is defined solely in terms of the attributes of the *company* class since both *curr_stockitem* and *stock_file* are attributes of the *company* class.

- ***Each system operation represented by an operation schema in the operation model, should mapped into an operation, and listed in the class diagram of the particular class, to which it belongs.***

Each system operation belongs to the *company* class, and hence, should be listed in the class diagram of the *company* :

company::take_order();
company::prepare&dispatch_order();
company::authorize&deplete_stock_levels();
company::issue_invoice();
company::accept_payment(int invoice_no, float amt);
company::replenish_stock_levels(int item, int qty).

- ***Each individual object interaction, within each system operation represented by an interaction diagram in the interaction model, should be mapped into an operation, and listed in the class diagram of the particular class, to which it belongs.***

For example, the take_order() system operation has 2 individual interactions

Individual Interaction	*Class*	*Operation*
order	*order*	*order::order()*
put_order(order)	*company*	*company::put_order(order)*

The prepare&dispatch_order() system operation has 3 individual interactions

Individual Interaction	*Class*	*Operation*
get_authorization	*order*	*order::get_authorization()*
change_status(status)	*order*	*order::change_status(status)*
put_order(order)	*company*	*company::put_order(order)*

The authorize&deplete_stock_levels() system operation has 8 individual interactions

Individual Interaction	*Class*	*Operation*
get_stockitem(stockitem_no)	*company*	*company::get_stockitem(stockitem_no)*
get_stock_level()	*stockitem*	*stockitem::get_stock_level()*
get_qty()	*order*	*order::get_qty()*
change_authorization(auth)	*order*	*order::change_authorization(auth)*
dec_stock_level(qty)	*stockitem*	*stockitem::dec_stock_level(qty)*
get_safety_level()	*stockitem*	*stockitem::get_safety_level()*
reorder()	*stockitem*	*stockitem::reorder()*
put_stockitem(stockitem)	*company*	*company::put_stockitem(stockitem)*

The issue_invoice() system operation has 9 individual interactions :

Individual Interaction	*Class*	*Operation*
get_order_no()	*order*	*order::get_order_no()*
get_cust_name()	*order*	*order::get_cust_name()*
get_cust_addr()	*order*	*order::get_cust_addr()*
get_item()	*order*	*order::get_item()*
get_qty()	*order*	*order::get_qty()*
get_stockitem()	*company*	*company::get_stockitem(stockitem_no)*
get_price()	*stockitem*	*stockitem::get_price()*
invoice	*invoice*	*invoice::invoice()*
put_invoice(invoice)	*company*	*company::put_invoice(invoice)*

The accept_payment() system operation has 5 individual interactions :

Individual Interaction	*Class*	*Operation*
get_invoice(invoice_no)	*company*	*company::get_invoice(invoice_no)*
get_balance()	*invoice*	*invoice::get_balance()*
change_status(status)	*invoice*	*invoice::change_status(status)*
change_balance(amt)	*invoice*	*invoice::change_balance(amt)*
put_invoice(invoice)	*company*	*company::put_invoice(invoice)*

The replenish_stock_levels() system operation has 3 individual interactions :

Individual Interaction	*Class*	*Operation*
get_stockitem(stockitem_no)	*company*	*company::get_stockitem(stockitem_no)*
inc_stock_level()	*stockitem*	*invoice::get_balance()*
put_stockitem(stockitem)	*company*	*company::put_stockitem(stockitem)*

Note : For the existing object, dynamic and functional models to be fully integrated, it is necessary only to add the following operations to the class diagrams of the respective classes :

company::take_order()
company::prepare&dispatch_order()
company::authorize&deplete_stock_levels()
company::issue_invoice()
company::accept_payment(int invoice_no, float amount)
company::replenish_stock_levels(int item, int qty)
stockitem::reorder()

3.3.3 Consistency Guidelines

Introduced in this section are a set of consistency guidelines which relate directly to the new inter-model definitions and integration guidelines which were described previously. Integration and consistency are very closely related, thus it follows that the criteria used to integrate the three OMT models, must also be used to check the models for consistency with each other. Hence the consistency guidelines act as a checklist when the models are thought to be complete. If any of the consistency checks fail, then an anomaly exists in one of the models which needs to be redressed. If none of the consistency checks fail, then the OMT models are consistent.

The consistency guidelines are as follows :

- Check that each condition in each state diagram of the dynamic model, is defined only in terms of the attributes, operations and associations listed in the class diagram of the class, which is represented in the state diagram.

- Check that each event in each state diagram of the dynamic model, is mapped into an operation, and listed in the class diagram of the class, which is represented in the state diagram.

- Check that each action in each state diagram of the dynamic model, is mapped into an operation, and listed in the class diagram of the class, which is represented in the state diagram.

- Check that each external event and complex internal event, in the event flow diagram of the dynamic model, is mapped into a system operation and illustrated as an operation schema in the operation model of the functional model.

- Check that each external event and complex internal event, in the event flow diagram of the dynamic model, is mapped into a system operation and illustrated as an interaction diagram in the interaction model of the functional model.

- Check that each system operation in the operation model, is defined only in terms of the attributes and associations listed in the class diagram of the particular class, to which it belongs.

- Check that each system operation represented by an operation schema in the operation model, is mapped into an operation, and listed in the class diagram of the particular class, to which it belongs

- Check that each individual object interaction, within each system operation represented by an interaction diagram in the interaction model, is mapped into an operation, and listed in the class diagram of the particular class, to which it belongs.

3.3.4 Comprehensive Illustrated Example

A more challenging example than the simple order processing system is necessary in order to fully illustrate the effect of the new functional model, and the new inter-model relationships, beyond a trivial level. Chapter 4 details a such a comprehensive case study, which documents and illustrates the object, dynamic and functional models for an insurance company which deals with clients, agents, policies, risks and claims.

3.4 Chapter Summary

This chapter proposed a detailed solution to each of the problems of the OMT methodology that were identified and discussed in the previous chapter. The first problem, namely the weakness of the functional model, was redressed by transforming the functional model from a DFD representation, into two separate models : an operation model which describes what each system operation does; and an interaction model which describes how each system operation works. The second problem, namely the inadequacy of the inter-model relationships was redressed by providing new inter-model definitions; integration guidelines; consistency guidelines; and a comprehensive illustrated example in the form of the Chapter 4 case study.

Chapter 4

Case Study

4.1 Overview

The purpose of documenting a case study is to provide a more comprehensive example than the simple order processing system, in which to illustrate the improved integration and consistency in the OMT methodology. In this chapter, the object and dynamic models are constructed according to Rumbaugh's specifications, the functional model is constructed according to my proposed specifications, and finally the three OMT models are inter-related and reconciled using my proposed integration guidelines and consistency guidelines.

Problem Statement : Insurance Company Processing System

A company insures clients by providing two types of policies, a car insurance policy and a house insurance policy. Each policy consists of a number of risks (e.g. fire, theft, etc.), and each of these risks has a risk premium, hence the total premium on each policy is calculated by summing the risk premiums for all the risks on the policy. The premium on each policy is paid in instalments, where the amount due at each instalment is calculated by dividing the total premium by the number of instalments (usually 12 monthly instalments), and where the due date of the instalment is calculated by incrementing the start date of the policy by one month each time a payment is made, until the end date of the policy is reached. Thus if the due date precedes the current date, then the status of the policy is unpaid, whereas if the current date precedes the due date then the status of the policy is paid.

Each policy is sold by an agent (e.g. an insurance broker) who is licensed by the company to sell policies on its behalf. Each agent receives commission for his sales, which is calculated by multiplying the premium on each of the policies that have been sold by him by his commission rate.

Each policy can suffer many claims which are settled by the company. A settlement results in either the claim being awarded or the claim being disallowed. If the claim date is between the start and end dates of the policy suffering the claim, and the status of this policy is paid, then the claim is awarded, if however the claim date is not between the start and end dates of the policy suffering the claim, or the status of the policy is unpaid, then the claim is disallowed.

4.2 Constructing the Object Model

Using Rumbaugh's standard OMT approach, the object model consists of an object diagram which illustrates each object class in terms of its attributes, operations and relationships to other classes. Hence there are 5 steps in the construction of the object model, namely :

- Identify object classes
- Identify associations
- Identify attributes
- Identify operations
- Build the object diagram

4.2.1 Identify Object Classes

By examining the problem statement, 8 distinct classes emerge. Obviously, there is a *client* class, a *policy* class, an *agent* class, and a *claim* class. Furthermore, there is a *car policy* class and a *house policy* class, derived from the *policy* class. Hence the *policy* class will hold the general policy details of each policy (e.g. start date, end date, etc.), while the *car policy* and *house policy* classes will hold details specific to car policies (e.g. manufacturer, model, etc.) and house policies (e.g. house value, contents value, etc.) respectively. Each policy holds a number of risks, thus *risk* is also a class, which is part of the *policy* class. And finally, each company holds a list of clients, policies, agents, and claims, thus *company* is also a class.

4.2.2 Identify Associations

Associations exist between the *company* class and the *client* class because the company insures a list of clients, between the *company* class and the *policy* class because the company manages a list of policies, between the *company* class and the *agent* class because the company licences a list of agents, and between the *company* class and the *claims* class because a company settles a list of claims.

In addition, associations exist between the *client* class and the *policy* class because a client holds policies, between the *agent* class and the *policy* class because an agent sells policies, between the *policy* class and the *risk* class because a policy consists of a number of risks, and between the *policy* class and the *claim* class because a policy suffers claims. Finally, an inheritance association exists between the *policy* class and the *car policy* class because a car policy is a specific type of policy, and an inheritance association also exists between the *policy* class and the *house policy* class because a house policy is also a specific type of policy.

4.2.3 Identify Attributes

Each *client* has a client number *(ClientNo)*, a surname *(Surname)*, a first name *(Firstname)*, an address *(Address)*, a phone number *(Phone)*, an occupation *(Occupation)*, a date of birth *(Birthdate)*, and a sex [e.g. male or female M/F] *(Sex)*. In addition, each client also has a list of the policy numbers on the policies that they hold *(PolicyNoList[])*, and a counter indicating the total number of policies that they hold *(PolicyCount)*.

Each *agent* has an agent number *(AgentNo)*, a surname *(Surname)*, a first name *(Firstname)*, a name of the company for whom they work *(Company)*, an address of the company for whom they work *(Address)*, a phone number of the company for whom they work *(Phone)*, and a commission rate *(CommRate)*. In addition, each agent also has a list of the policy numbers on the policies that they've sold *(PolicyNoList[])*, a counter indicating the total number of policies that they've sold *(PolicyCount)*, and a numerical figure indicating the commission due to them *(TotalAmtDue)*, [which is a derived attribute, since it is obtained by multiplying the commission rate of the agent by the sum of the premiums on all the policies sold by the agent].

Each *policy* has a client number *(ClientNo)*, identifying the client who holds the policy, an agent number *(AgentNo)*, identifying the agent who sold the policy, a policy number *(PolicyNo)*, the start date of the policy *(StartDate)*, the end date of the policy *(EndDate)*, the original date on which the policy was created *(OriginalDate)*, [which is a derived attribute, since it is the same as the start date on the policy when it was created, but unlike the start and end dates of the policy which will be updated if the policy is renewed, the original date will always remain the same]. In addition, each policy also has a list of the risks on the policy *(RiskList[])*, a counter indicating the total number of risks on the policy *(TotalRisks)*, the total premium of the policy *(TotalPremium)*, [which is a derived attribute, since it is obtained by summing the premiums on all the risks on the policy], the number of payment instalments to be made over the period of the policy *(Instalments)* [e.g. 12 monthly instalments per year], the number of payments made so far over the period of the policy *(Payments)*, the amount of the payment due at each instalment *(AmtDue)* [which is a derived attribute, sine it is obtained by dividing the total premium by the number of instalments], the due date of the next payment *(DueDate)* [which is a derived attribute, since it is obtained by incrementing the start date of the policy by one month each time a payment is received], the status of the policy *(Status)* [which is a derived attribute, since it is obtained by comparing the due date of the policy to the current date, if the due date precedes the current date, then the policy is "unpaid", and if the current date precedes the due date, then the policy is "paid"]. Each policy also has a list of the claim numbers on the claims suffered by the policy *(ClaimNoList[])*, and a counter indicting the total number of claims suffered by the policy *(ClaimNoCount)*.

Each *car policy* has the additional attributes of the manufacturer of the car *(Manufacturer)* [e.g. Toyota], the model of the car *(Model)* [e.g. Carina E], the registration of the car *(Registration)*, the engine size of the car *(EngineSize)* [e.g. 1.6 Litres], the value of the car *(CarValue)* [e.g. £14,000], and the full licence status of the driver of the car *(FullLicenceStatus)* [e.g. 0 if the driver does not have a full drivers licence, and 1 if the driver does have a full drivers licence].

Each *house policy* has the additional attributes of the type of the house *(HouseType)* [e.g. semi, detached, etc.], the number of rooms in the house *(Rooms)* [e.g. 3-bedrooms], the area code of the area in which the house is located *(AreaCode)* [e.g. A1 - A9], the value of the house *(HouseValue)* [e.g. £80,000], the value of the contents of the house *(ContentsValue)* [e.g. £25,000], and the house alarm status of the house *(HouseAlarmStatus)* [e.g. 0 if the house does not have an alarm, and 1 if the house does have an alarm].

Each *risk* has a risk number *(RiskNo)* [e.g. fire, theft, etc.], a risk type *(RiskType)* [e.g. car policy risk or house policy risk], a risk value *(RiskValue)* [e.g. the value of the car or the house], a risk status *(RiskStatus)* [e.g. whether the driver of the car has a full licence, or whether the house has an alarm], and a risk premium *(RiskPremium)* [which is a derived attribute, since it is obtained by combining the risk type, risk value and risk status into a formula which yields a numerical figure].

Each *claim* has a claim number *(ClaimNo)*, a claim date *(ClaimDate)*, a claim value *(ClaimValue)*, a description of the details of the claim *(ClaimDetails)*, a description of the status of the claim *(ClaimStatus)*, [which is a derived attribute, since it is obtained by comparing the claim date to the start and end dates of the policy suffering the claim, and by examining the status of the policy suffering the claim, so if the claim date is between the start and end dates of the policy and the status of the policy is "paid" then the claim is "awarded", and if the claim date is not between the start and end dates of the policy or the status of the policy is "unpaid" then the claim is "disallowed"]. Thus each claim has the additional attributes of the number of the policy suffering the claim *(PolicyNo)*, the start date of the policy *(PolicyStart)*, the end date of the policy *(PolicyEnd)*, and the status of the policy *(PolicyStatus)*.

Each *company* has a name *(CompanyName)*, an address *(Address)*, a phone number *(Phone)*, a list of clients *(ClientList[])*, a list of policies *(PolicyList[])*, a list of agents *(AgentList[])*, a list of claims *(ClaimList[])*, a counter for the total number of clients *(TotalClients)*, a counter for the total number of policies *(TotalPolicies)*, a counter for the total number of agents *(TotalAgents)*, a counter for the total number of claims *(TotalClaims)*, and access attributes for accessing each of the respective lists [e.g. *Client* accesses the *ClientList[], Policy* accesses the *PolicyList[], Agent* accesses the *AgentList[],* and *Claim* access the *ClaimList[]*.

4.2.4 Identify Operations

Each class must have an operation to access each attribute, and an operation to update each attribute which is permitted to be updated. The access operation merely returns the value of the attribute, while the update operation sets the value of the attribute to a new value which is passed as a parameter to the operation. In the case of derived attributes, the update operation calculates the new value of the attribute from the values of the other attributes in the class from which it is derived, hence the new value of the attribute is not passed as a parameter.

The *client* class has an operation to access ClientNo *(GetClientNo)*, Surname *(GetSurname)*, Firstname *(GetFirstname)*, Address *(GetAddress)*, Phone *(GetPhone)*, Occupation *(GetOccupation)*, Birthdate *(GetBirthdate)*, Sex *(GetSex)*, PolicyNoList[] *(GetPolicyNo)*, and PolicyCount *(GetPolicyCount)*. The ClientNo is not updateable, however the *client* class has an operation to update Surname *(PutSurname)*, Firstname *(PutFirstname)*, Address *(PutAddress)*, Phone *(PutPhone)*, Occupation *(PutOccupation)*, Birthdate *(PutBirthdate)*, Sex *(PutSex)*, PolicyNoList[] *(PutPolicyNo)* and PolicyCount *(IncPolicyCount* and *DecPolicyCount)*. In addition, the *client* class has an operation to add a policy number to the policy number list *(AddPolicy)* [which invokes *PutPolicyNo* and *IncPolicyCount*], and an operation to delete a policy number from the policy number list *(DeletePolicy)* [which invokes *PutPolicyNo* and *DecPolicy Count*].

The *agent* class has an operation to access AgentNo *(GetAgentNo)*, Surname *(GetSurname)*, Firstname *(GetFirstname)*, Company *(GetCompany)*, Address *(GetAddress)*, Phone *(GetPhone)*, CommRate *(GetCommRate)*, TotalAmtDue *(GetTotalAmtDue)*, PolicyNoList[] *(GetPolicyNo)*, and PolicyCount *(GetPolicy Count)*. The AgentNo is not updateable, however the *agent* class has an operation to update Surname *(PutSurname)*, Firstname *(PutFirstname)*, Company *(PutCompany)*, Address *(PutAddress)*, Phone *(PutPhone)*, CommRate *(PutCommRate)*, TotalAmtDue *(CalcTotalAmtDue)* [since it is a derived attribute], PolicyNoList[] *(PutPolicyNo)* and PolicyCount *(IncPolicyCount* and *DecPolicyCount)*. In addition, the *agent* class has an operation to add a policy number to the policy number list *(AddPolicy)* [which invokes *PutPolicyNo* and *IncPolicyCount*], and an operation to delete a policy number from the policy number list *(DeletePolicy)* [which invokes *PutPolicyNo* and *DecPolicy Count*].

The *policy* class has an operation to access ClientNo *(GetClientNo)*, AgentNo *(GetAgentNo)*, PolicyNo *(GetPolicyNo)*, OriginalDate *(GetOriginalDate)*, StartDate *(GetStartDate)*, EndDate *(GetEndDate)*, TotalPremium *(GetTotalPremium)*, Instalments *(GetInstalments)*, Payments *(GetPayments)*, AmtDue *(GetAmtDue)*, DueDate *(GetDueDate)*, Status *(GetStatus)*, RiskList[] *(GetRisk)*, TotalRisks *(GetTotalRisks)*, ClaimNoList[] *(GetClaimNo)* and ClaimCount *(GetClaimCount)*. The ClientNo, AgentNo or PolicyNo are not updateable, however the *policy* class has an operation to update StartDate *(PutStartDate)*, EndDate *(PutEndDate)*, TotalPremium *(CalcTotalPremium)* [since it is a derived attribute], Instalments *(PutInstalments)*, Payments *(PutPayments)*, AmtDue *(CalcAmtDue)* [since it is a derived attribute], DueDate *(CalcDueDate)* [since it is a derived attribute], Status

(CalcStatus) [since it is a derived attribute], RiskList[] *(PutRisk)*, TotalRisks *(IncTotalRisks* and *DecTotal Risks)*, ClaimNoList[] *(PutClaimNo)* and ClaimCount *(IncClaimCount* and *DecClaim Count)*. In addition, the *policy* class has an operation to add a risk to the risk list *(AddRisk)* [which invokes *PutRisk* and *IncTotalRisks*], and to delete a risk from the risk list *(DeleteRisk)* [which invokes *PutRisk* and *DecTotalRisks*]. Similarly, there is an operation to add a claim number to the claim number list *(AddClaim)* [which invokes *PutClaimNo* and *IncClaimCount*], and an operation to delete a claim number from the claim number list *(DeleteClaim)* [which invokes *PutClaimNo* and *DecClaimCount*].

The *car policy* class has an operation to access Manufacturer *(GetManufacturer)*, Model *(GetModel)*, Registration *(GetRegistration)*, EngineSize *(GetEngineSize)*, CarValue *(GetCarValue)* and FullLicenceStatus *(GetFullLicenceStatus)*. The *car policy* class also has an operation to update Manufacturer *(PutManufacturer)*, Model *(PutModel)*, Registration *(PutRegistration)*, EngineSize *(PutEngineSize)*, CarValue *(PutCarValue)* and FullLicenceStatus *(PutFullLicenceStatus)*.

The *house policy* class has an operation to access HouseType *(GetHouseType)*, Rooms *(GetRooms)*, AreaCode *(GetAreaCode)*, HouseValue *(GetHouseValue)*, ContentsValue *(GetContentsValue)* and HouseAlarmStatus *(GetHouseAlarmStatus)*. The *house policy* class also has an operation to update HouseType *(PutHouseType)*, Rooms *(PutRooms)*, AreaCode *(PutAreaCode)*, HouseValue *(PutHouseValue)*, ContentsValue *(PutContentsValue)* and HouseAlarmStatus *(PutHouseStatus)*.

The *risk* class has an operation to access RiskNo *(GetRiskNo)*, RiskType *(GetRiskType)*, RiskValue *(GetRiskValue)*, RiskStatus *(GetRiskStatus)*, RiskPremium *(GetRiskPremium)*. The RiskNo is not updateable, however the *risk* class has an operation to update RiskType *(PutRiskType)*, RiskValue *(PutRiskValue)*, RiskStatus *(PutRiskStatus)* and RiskPremium *(CalcRiskPremium)* [since it is a derived attribute].

The *claim* class has an operation to access ClaimNo *(GetClaimNo)*, ClaimDate *(GetClaimDate)*, ClaimValue *(GetClaimValue)*, ClaimDetails *(GetClaimDetails)*, ClaimStatus *(GetClaimStatus)*, PolicyNo *(GetPolicyNo)*, PolicyStart *(GetPolicyStart)*, PolicyEnd *(GetPolicyEnd)* and PolicyStatus *(GetPolicyStatus)*. The ClaimNo and PolicyNo are not updateable, however the *claim* class has an operation to update ClaimDate *(PutClaimDate)*, ClaimValue *(PutClaimValue)*, ClaimDetails *(PutClaim Details)*, PolicyStart *(PutPolicyStart)*, PolicyEnd *(PutPolicyEnd)* and PolicyStatus *(PutPolicyStatus)*.

The *company* class has an operation to access ClientList[] *(GetClientNo)*, PolicyList *(GetPolicyNo)*, AgentList *(GetAgentNo)*, ClaimList *(GetClaimNo)*, TotalClients *(Get TotalClients)*, TotalPolicies *(GetTotalPolicies)*, TotalAgents *(GetTotalAgents)* and TotalClaims *(GetTotalClaims)*. The company class also has an operation to update ClientList[] *(PutClient)*, PolicyList[] *(PutPolicy)*, AgentList[] *(PutAgent)*, ClaimList[] *(PutClaim)*, TotalClients *(IncTotalClients* and *DecTotalClients)*, TotalPolicies *(IncTotalPolicies* and *DecTotalPolicies)*, TotalAgents *(IncTotalAgents* and *DecTotal Agents)* and TotalClaims *(IncTotalClaims* and *DecTotalClaims)*. In addition there is an operation to add a client *(AddClient)*, delete a client *(DeleteClient)*, update a client *(UpdateClient)*, retrieve a client *(GetClient)*, add a policy *(AddPolicy)*, delete a policy *(DeletePolicy)*, update a policy *(UpdatePolicy)*, retrieve a policy *(GetPolicy)*, add an agent *(AddAgent)*, delete an agent *(DeleteAgent)*, update an agent *(UpdateAgent)*, retrieve an agent *(GetAgent)*, add a claim *(AddClaim)*, delete a claim *(DeleteClaim)*, update a claim *(UpdateClaim)* and retrieve a claim *(GetClaim)*.

4.2.5 Build the Object Diagram

Due to space restrictions, the attributes and operations of each class cannot be displayed on the object diagram below. However, the class definitions for each class are provided in the appendix of the thesis. [For example, the class definition for the *TClient* class is listed in TClient.H, etc.]

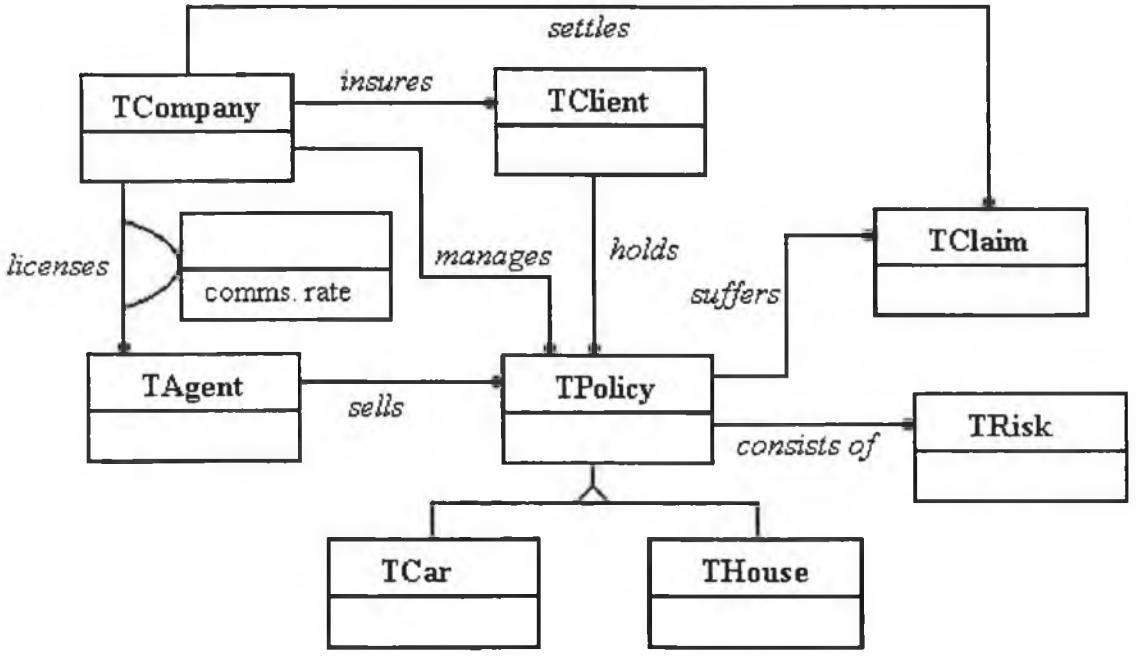


Fig 4.1 Object Model

4.3 Constructing the Dynamic Model

Using Rumbaugh's standard OMT approach, the dynamic model consists of a set of scenarios depicting expected system behaviour, a set of event trace diagrams illustrating each distinct scenario, an event flow diagram summarising the events between the various classes, and finally, a state diagram for each class with non-trivial dynamic behaviour. Hence there are 5 steps in the construction of the dynamic model, namely :

- Prepare scenarios
- Identify events from scenarios
- Build event trace diagram for each scenario
- Build event flow diagram
- Build state diagram for each class

4.3.1 Prepare Scenarios

Within the insurance application, there exists an infinite number of possible scenarios, however 10 important scenarios can be identified. These scenarios are important since each imposes a strict order on the sequence in which a set of events can occur. Hence these scenarios could be described as core scenarios, which can be used to form part of other more complex scenarios, however the sequence of events within each of these scenarios cannot be altered.

- ***Scenario #1***

This scenario illustrates the strict order in which items may be added to the system. A client and an agent must already exist within the system before a policy can be added, since when a policy is created, the policy number is added to the policy number list of the client and the policy number list of the agent. Due to the fact that a risk is part of a policy, a policy must already exist within the system before a risk can be added, since when a risk is created, the risk is added to the risk list of the policy. Similarly, due to the fact that a claim is made against a policy, a policy must already exist within the system before a claim can be added, since when a claim is created, the claim number is added to the claim number list of the policy.

A user adds a client to the system.
The company creates the client and stores it in the client list of the company.
A user adds an agent to the system.
The company creates the agent and stores it in the agent list of the company.
A user adds a policy to the system.
The company creates the policy* and stores it in the policy list of the company.
The company adds the policy number to the policy number list of the client.
The company adds the policy number to the policy number list of the agent.
A user adds a risk to a policy.
The company creates the risk and stores it in the risk list of the policy.
A user adds a claim to the system.
The company creates the claim and stores it in the claim list of the company.
The company adds the claim number to the claim number list of the policy.
* *(indicates car insurance policy or house insurance policy)*

- ***Scenario #2***

This scenario illustrates the strict order in which items may be deleted from the system. All claims made against a policy must be deleted before the policy can be deleted from the system. All risks existing on a policy must be deleted before the policy can be deleted from the system. All policies belonging to a particular agent must be deleted before that agent can be deleted from the system. All policies belonging to a particular client must be deleted before that client can be deleted from the system.

A user deletes a claim from the system.
The company destroys the claim in the claim list of the company.
The company deletes the claim number from the claim number list of the policy.
A user deletes a risk from a policy.
The company destroys the risk in the risk list of the policy.
A user deletes a policy from the system.
The company destroys the policy* in the policy list of the company.
The company deletes the policy number from the policy number list of the client.
The company deletes the policy number from the policy number list of the agent.
A user deletes an agent from the system.
The company destroys the agent in the agent list of the company.
A user deletes a client from the system.
The company destroys the client in the client list of the company.

- ***Scenario #3***

This scenario illustrates the strict order in which clients may be updated in the system. Clients may only be updated after they have been added and before they are deleted. (i.e. a client must exist within the system).

A user adds a client to the system.
The company creates the client and stores it in the client list of the company.
A user updates a client in the system.
A user deletes a client from the system.
The company destroys the client in the client list of the company.

- ***Scenario #4***

This scenario illustrates the strict order in which agents may be updated in the system. Agents may only be updated after they have been added and before they are deleted. (i.e. an agent must exist within the system).

A user adds an agent to the system.
The company creates the agent and stores it in the agent list of the company.
A user updates an agent in the system.
A user deletes an agent from the system.
The company destroys the agent in the agent list of the company.

- ***Scenario #5***

This scenario illustrates the strict order in which policies may be updated in the system. Policies may only be updated after they have been added and before they are deleted. (i.e. a policy must exist within the system).

A user adds a policy to the system.
The company creates the policy* and stores it in the policy list of the company.
The company adds the policy number to the policy number list of the client.
The company adds the policy number to the policy number list of the agent.
A user adds a risk to a policy.
The company creates the risk and stores it in the risk list of the policy.
A user updates a policy in the system.
The company updates the status of the policy.

A user deletes a risk from a policy.
The company destroys the risk in the risk list of the policy.
A user deletes a policy from the system.
The company destroys the policy in the policy list of the company.
The company deletes the policy number from the policy number list of the client.
The company deletes the policy number from the policy number list of the agent.

- ***Scenario #6***

This scenario illustrates the strict order in which claims may be updated in the system. Claims may only be updated after they have been added and before they are deleted. (i.e. a claim must exist within the system).

A user adds a claim to the system.
The company creates the claim and stores it in the claim list of the company.
A user updates a claim in the system.
The company updates the status of the claim.
A user deletes a claim from the system.
The company destroys the claim in the claim list of the company.

- ***Scenario #7***

This scenario illustrates the strict order in which clients may be retrieved from the system. Clients may only be retrieved after they have been added and before they are deleted. (i.e. a client must exist within the system).

A user adds a client to the system.
The company creates the client and stores it in the client list of the company.
A user retrieves a client from the system.
A user deletes a client from the system.
The company destroys the client in the client list of the company.

- ***Scenario #8***

This scenario illustrates the strict order in which agents may be retrieved from the system. Agents may only be retrieved after they have been added and before they are deleted. (i.e. an agent must exist within the system).

A user adds an agent to the system.
The company creates the agent and stores it in the agent list of the company.
A user retrieves an agent from the system.
A user deletes an agent from the system.
The company destroys the agent in the agent list of the company.

- ***Scenario #9***

This scenario illustrates the strict order in which policies may be retrieved from the system. Policies may only be retrieved after they have been added and before they are deleted. (i.e. a policy must exist within the system).

A user adds a policy to the system.
The company creates the policy* and stores it in the policy list of the company.
The company adds the policy number to the policy number list of the client.
The company adds the policy number to the policy number list of the agent.
A user adds a risk to a policy.
The company creates the risk and stores it in the risk list of the policy.
A user retrieves a policy from the system.
A user deletes a risk from a policy.
The company destroys the risk in the risk list of the policy.
A user deletes a policy from the system.
The company destroys the policy in the policy list of the company.
The company deletes the policy number from the policy number list of the client.
The company deletes the policy number from the policy number list of the agent.

- ***Scenario #10***

This scenario illustrates the strict order in which claims may be retrieved from the system. Claims may only be retrieved after they have been added and before they are deleted. (i.e. a claim must exist within the system).

A user adds a claim to the system.
The company creates the claim and stores it in the claim list of the company.
A user retrieves a claim from the system.
A user deletes a claim from the system.
The company destroys the claim in the claim list of the company.

4.3.2 Identify Events from Scenarios

Within the insurance application there are 18 external events, which provide the input and output of the system : *add_client* which is sent from the user to the company and contains the client details as parameters; *delete_client* which is sent from the user to the company and contains the client number as a parameter; *update_client* which is sent from the user to the company and contains the details to be updated on the client as parameters; *find_client* which is sent from the user to the company and contains the client number as a parameter; *add_policy* which is sent from the user to the company and contains the policy details as parameters; *delete_policy* which is sent from the user to the company and contains the policy number as a parameter; *update_policy* which is sent from the user to the company and contains the details to be updated on the policy as parameters; *find_policy* which is sent from the user to the company and contains the policy number as a parameter; *add_agent* which is sent from the user to the company and contains the agent details as parameters; *delete_agent* which is sent from the user to the company and contains the agent number as a parameter; *update_agent* which is sent from the user to the company and contains the details to be updated on the agent as parameters; *find_agent* which is sent from the user to the company and contains the agent number as a parameter; *add_claim* which is sent from the user to the company and contains the claim details as parameters; *delete_claim* which is sent from the user to the company and contains the claim number as a parameter; *update_claim* which is sent from the user to the company and contains the details to be updated on the claim as parameters; *find_claim* which is sent from the user to the company and contains the claim number as a parameter; *add_risk* which is sent from the user to the policy and contains the risk details as parameters; and finally, *delete_risk* which is sent from the user to the policy and contains the risk number as a parameter.

The internal events are as follows : *create_client* which is sent from the company to the client, and which causes a new client to be created and stored in the client list of the company; *create_agent* which is sent from the company to the agent, and which causes a new agent to be created and stored in the agent list of the company; *create_policy* which is sent from the company to the policy, and which causes a *create_car* event to be sent to the car (creating a new car insurance policy) or a *create_house* event to be sent to the house (creating a new house insurance policy), which is then stored in the policy list of the company; *create_risk* which is sent from the policy to the risk, and which causes a new risk to be created and stored in the risk list of the policy; *create_claim* which is sent from the company to the claim, and which causes a new claim to be created and stored in the claim list of the company.

Also there exist the following events : *destroy_client* which is sent from the company to the client, and which causes an existing client to be destroyed and removed from the client list of the company; *destroy_agent* which is sent from the company to the agent, and which causes an existing agent to be destroyed and removed from the agent list of the company; *destroy_policy* which is sent from the company to the policy, and which causes a *destroy_car* event to be sent to the car (destroying an existing car insurance policy) or a *destroy_house* event to be sent to the house (destroying an existing house insurance policy), which is then removed from the policy list of the company; *destroy_risk* which is sent from the policy to the risk, and which causes an existing risk to be destroyed and removed from the risk list of the policy; *destroy_claim* which is sent from the company to the claim, and which causes an existing claim to be destroyed and removed from the claim list of the company;

Finally, there also exists the following events : *add_policy_to_client* which is sent from the company to the client, and which causes the policy number of the newly created policy to be added to the policy number list of the client to which that policy belongs; *add_policy_to_agent* which is sent from the company to the agent, and whicli causes the policy number of the newly created policy to be added to the policy number list of the agent to which that policy belongs; *delete_policy_from_client* which is sent from the company to the client, and which causes the policy number of an already existing policy to be deleted from the policy number list of the client to which that policy belongs; *delete_policy_from_agent* which is sent from the company to the agent, and which causes the policy number of an already existing policy to be deleted from the policy number list of the agent to which that policy belongs; *change_status* which is sent from the company to the policy, and which causes the policy status to be updated to paid or unpaid; and change_status which is sent from the company to the claim, and which causes the claim status to be updated to awarded or disallowed.

These internal events are similar to messages, however there major discriminating factor from other messages in the system is that they cause a change of state within the system, and hence deserve the status of an event. For example, the *change_status* event associated with the claim class, is the only event resulting from an update to a claim. This is because updating the claim details does not have any effect on the state of the claim object, whereas updating the claim status causes the claim to move to an "awarded" state or a "disallowed" state. Similarly, the *change_status* event associated with the policy class, is the only event resulting from an update to a policy, and causes the policy to move to a "paid" or an "unpaid" state.

4.3.3 Build Event Trace Diagram for each Scenario

Each scenario is a sequence of events that occurs during one particular execution of a system, and the event trace diagram illustrates this sequence of events, as well as illustrating the objects in the system which are directly affected by each event. An event trace diagram for each identified scenario is illustrated below :

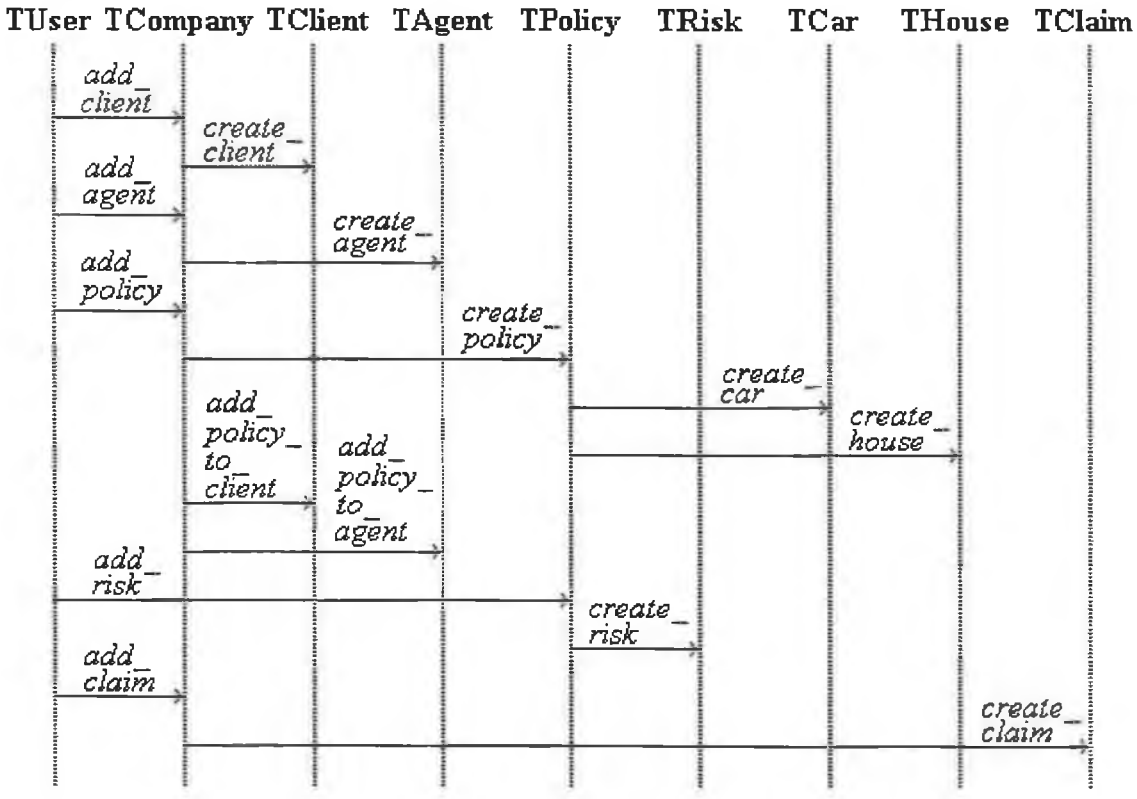


Fig 4.2 Event Trace Diagram - Scenario #1

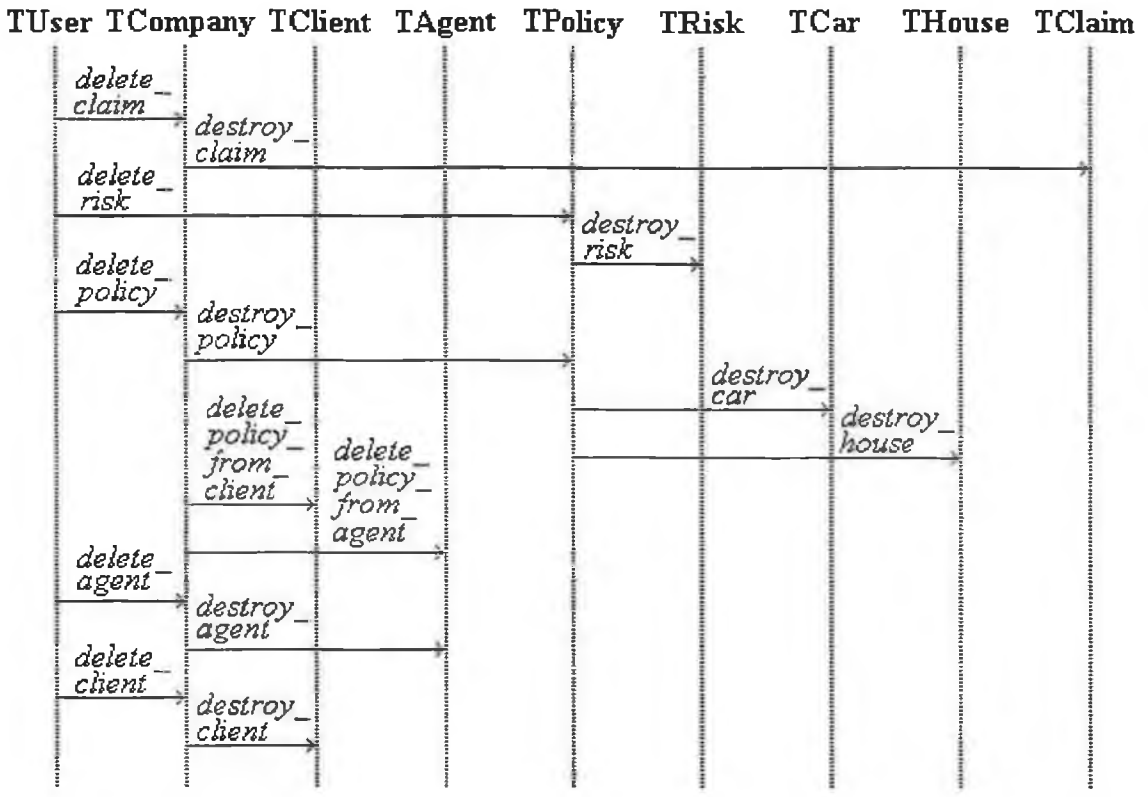


Fig 4.3 Event Trace Diagram - Scenario #2

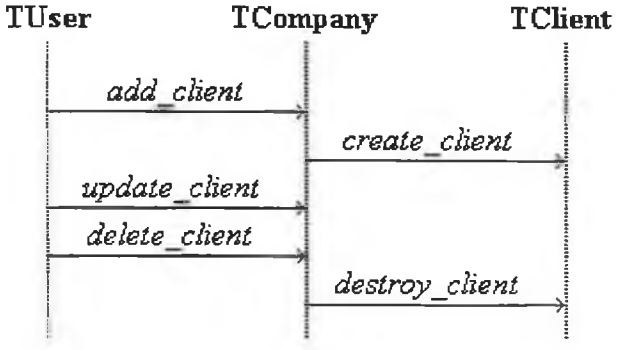


Fig 4.4 Event Trace Diagram - Scenario #3

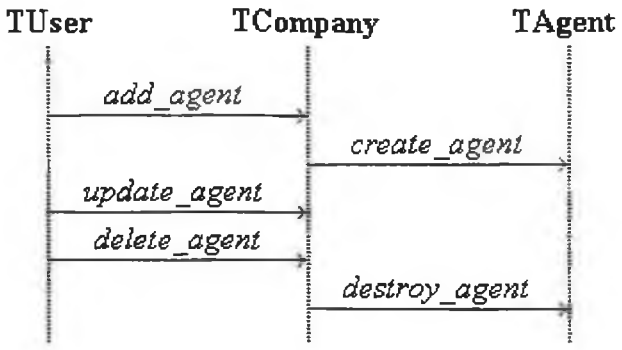


Fig 4.5 Event Trace Diagram - Scenario #4

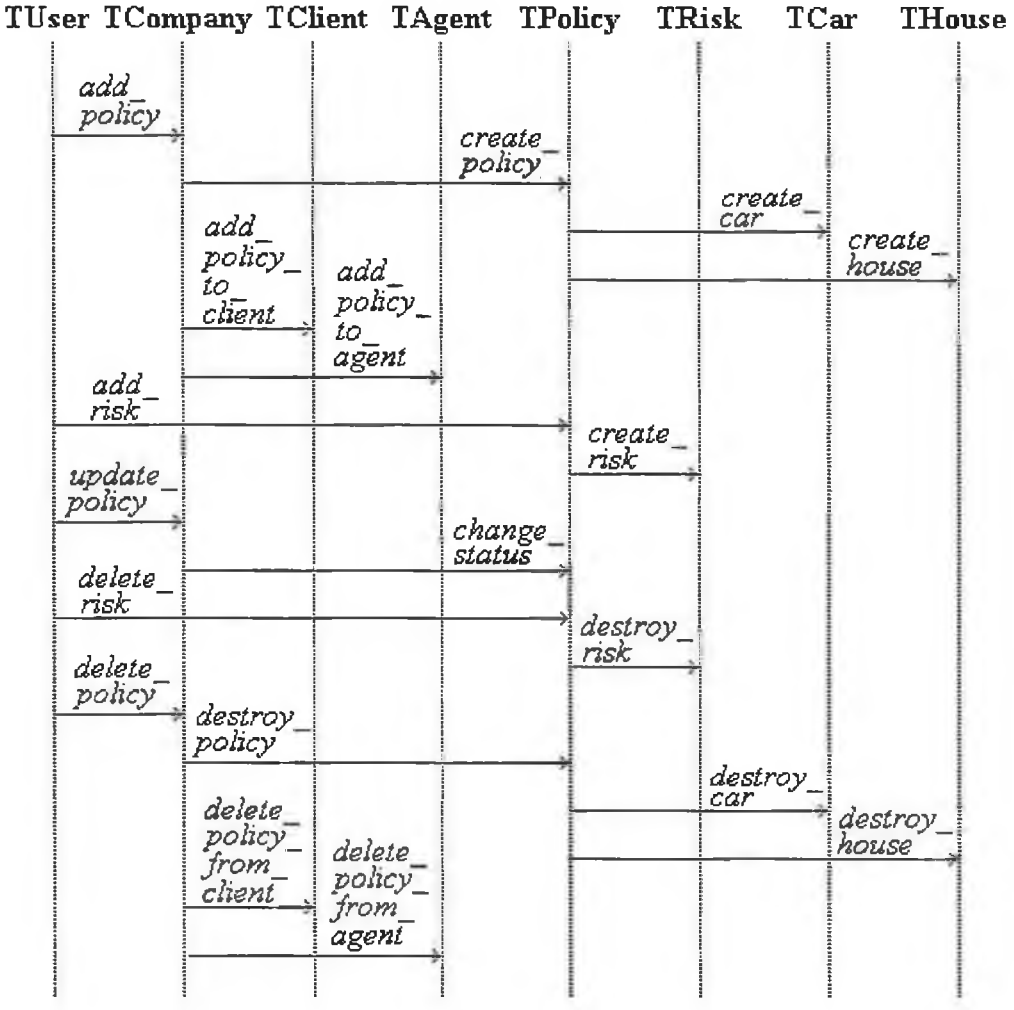


Fig 4.6 Event Trace Diagram - Scenario #5

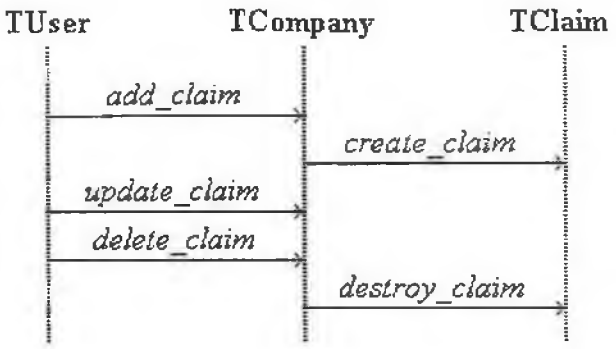


Fig 4.7 Event Trace Diagram - Scenario #6

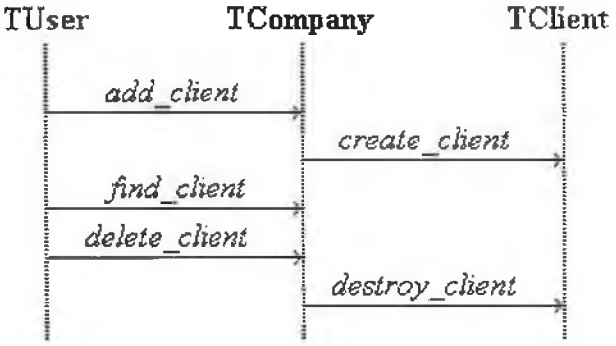


Fig 4.8 Event Trace Diagram - Scenario #7

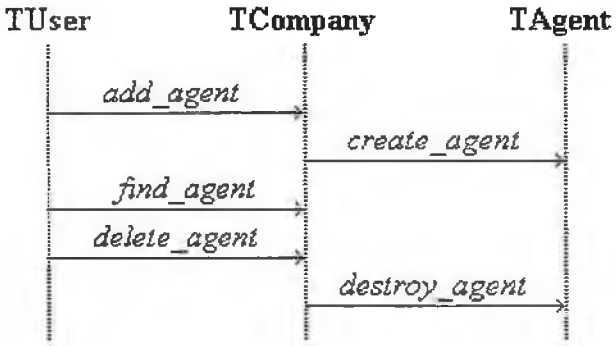


Fig 4.9 Event Trace Diagram - Scenario #8

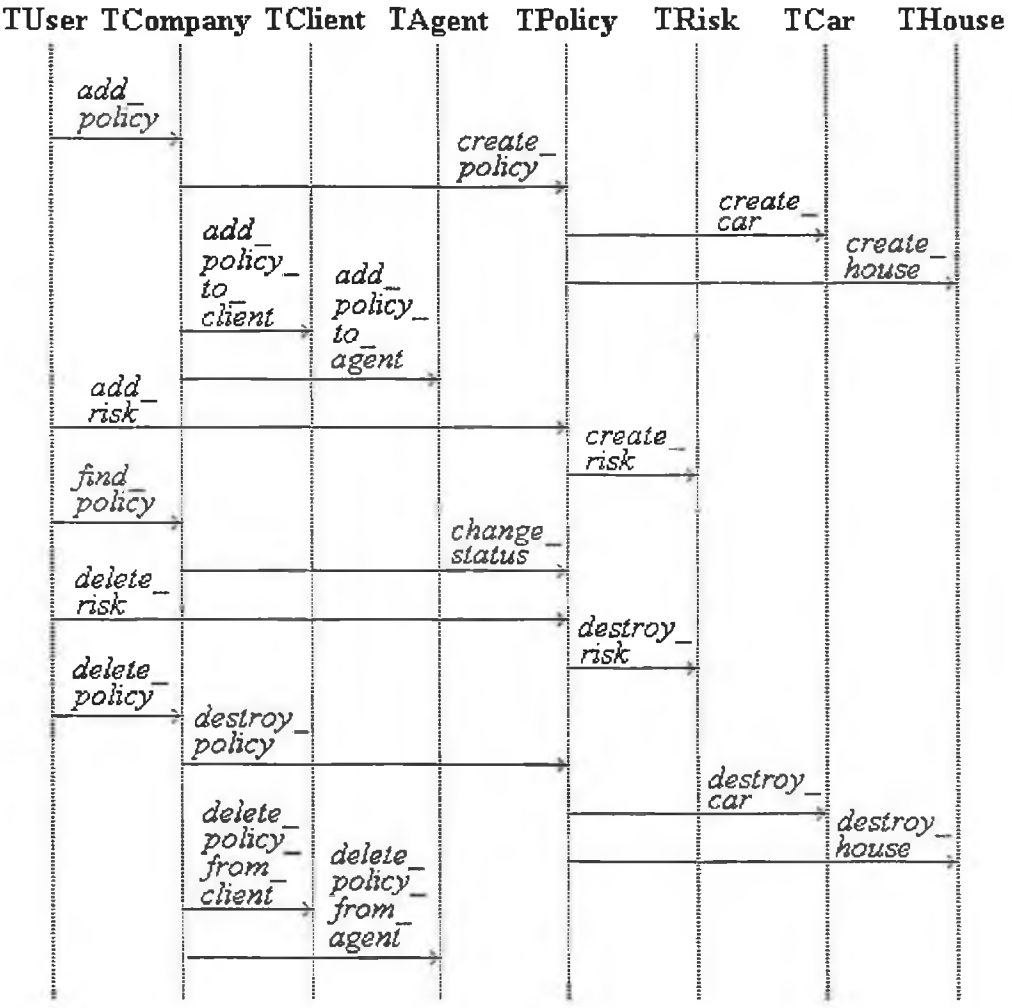


Fig 4.10 Event Trace Diagram - Scenario #9

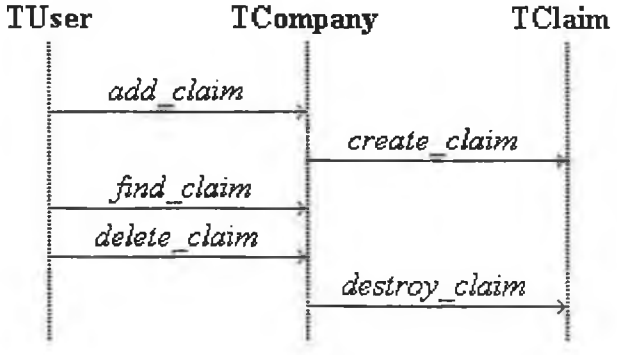
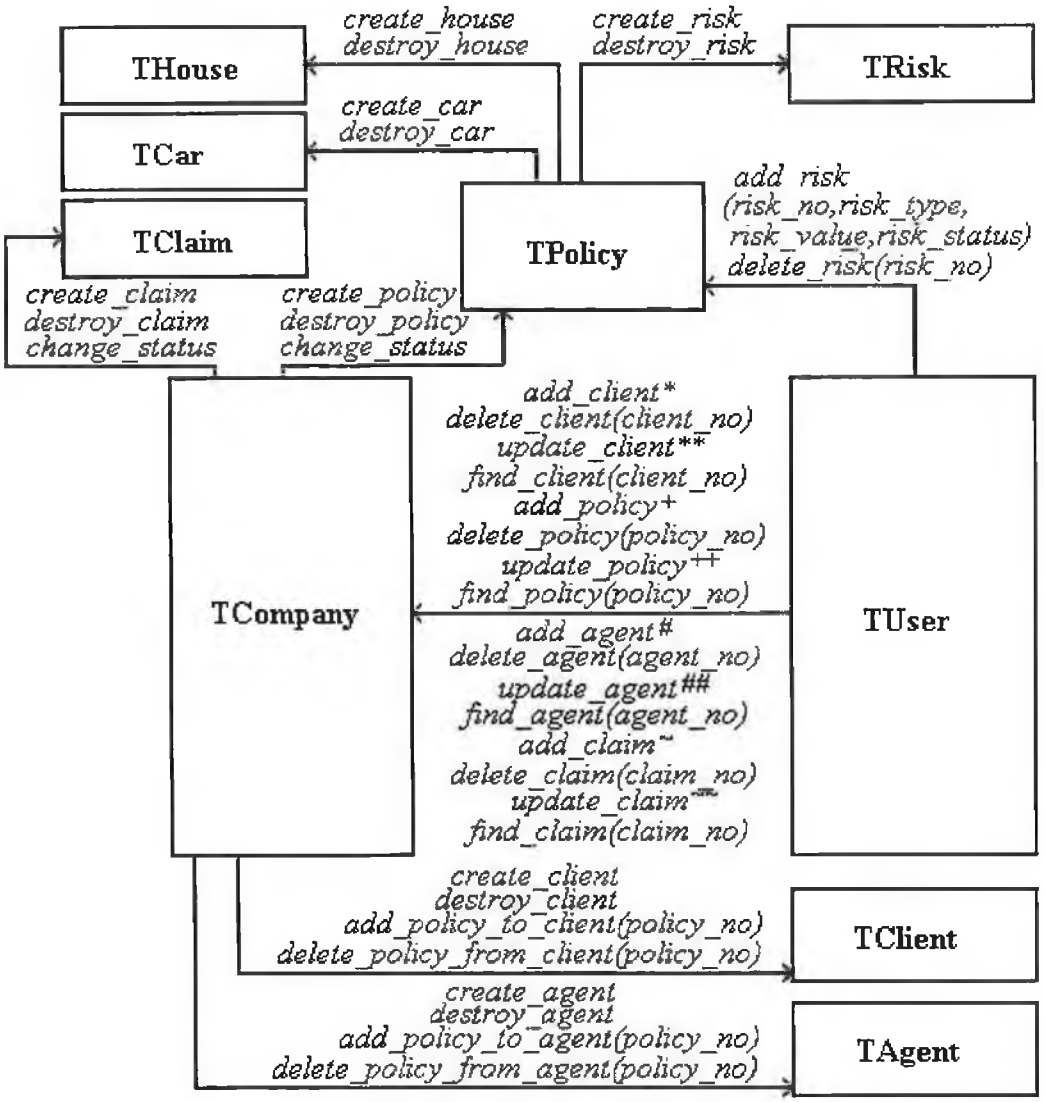


Fig 4.11 Event Trace Diagram - Scenario #10

4.3.4 Build Event Flow Diagram

The event flow diagram summarises events between classes by simply listing the events which can be sent and received by each class, without regard for the sequence in which the events are to be performed. Only those events listed on the event flow diagram for each respective class, can appear on the state diagram for that class.



* add_client(client_no,surname,firstname,address,phone,occupation,birthdate,sex)
** update_client(client_no,surname,firstname,address,phone,occupation,birthdate,sex)
\+ add_policy(client_no,agent_no,policy_no,start_date,end_date,car_details)
\+ add_policy(client_no,agent_no,policy_no,start_date,end_date,house_details)
++ update_policy(policy_no,start_date,end_date,car_details)
++ update_policy(policy_no,start_date,end_date,house_details)
\# add_agent(agent_no,surname,firstname,company,address,phone,comm_rate)
update_agent(agent_no,surname,firstname,company,address,phone,comm_rate)
~ add_claim(claim_no,claim_date,claim_value,claim_details,
policy_no,policy_start,policy_end,policy_status)
~~ update_claim(claim_no,claim_date,claim_value,claim_details)

Fig 4.12 Event Flow Diagram

4.3.5 Build State Diagram for each Class

As a result of the *create_client* event, the client is placed in the ***inactive*** state (a client is "inactive", when he does not hold any policies, and a client is "active" when he holds at least one policy) and remains in this state until an *add_policy_to_client(policy_no)* event is received, which moves the client to the ***active*** state. Once in the ***active*** state, subsequent *add_policy_to_client(policy_no)* events will not alter the state of the client [the *AddPolicy(PolicyNo)* action will add the policy number to the policy number list of the client, and will increment the value of *PolicyCount* by 1]. However, a *delete_policy_from_client(policy_no)* event will only allow the client to remain in the ***active*** state provided he holds more than one policy at the time the event is received [the *DeletePolicy(PolicyNo)* action will delete a policy from the policy number list of the client, and will decrement the value of *PolicyCount* by 1]. If the client holds only one policy at the time the *delete_policy_from_client(policy_no)* event is received, then the *DeletePolicy(PolicyNo)* action will delete this final policy from the policy number list of the client, and will decrement the value of *PolicyCount* down to zero, and hence the client enters the ***inactive*** state. Finally as a result of the *destroy_client* event, the life cycle of the client is over.

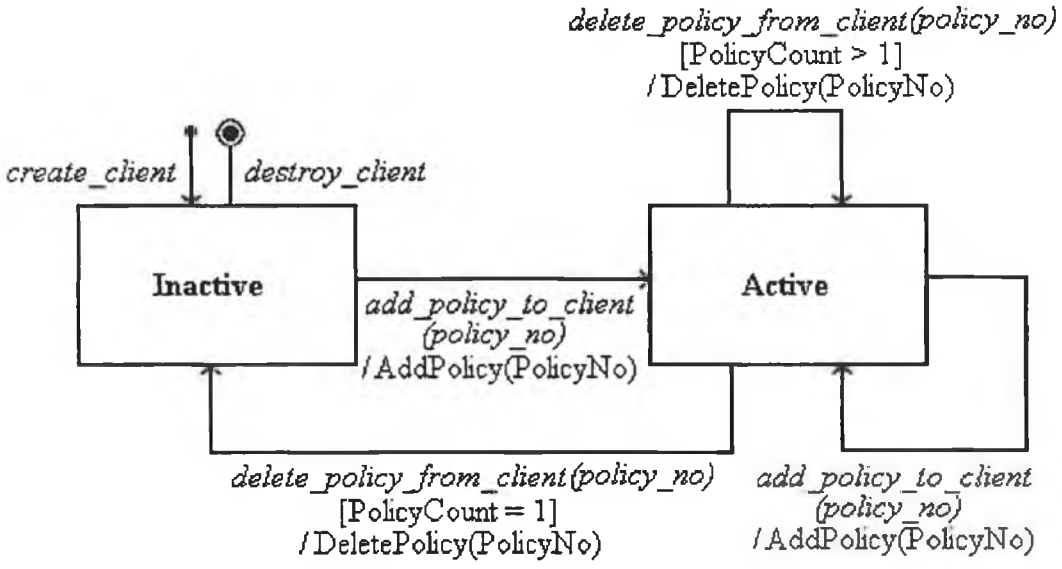


Fig 4.13 Dynamic Model (State Diagram for *TClient*)

As a result of the *create_policy* event, the client is placed in the ***inactive*** state (a policy is "inactive", when it does not hold any risks, and a policy is "active" when it holds at least one risk) and remains in this state until an *add_risk* event is received, which moves the policy to the ***active*** state. Once in the ***active*** state, subsequent *add_risk* events will not alter the state of the client [the *AddRisk()* action will add the risk to the risk list of the policy, and will increment the value of *TotalRisks* by 1].

However, a *delete_risk* event will only allow the policy to remain in the ***active*** state provided it holds more than one risk at the time the event is received [the *DeleteRisk()* action will delete a risk from the risk list of the policy, and will decrement the value of *TotalRisks* by 1]. If the policy holds only one risk at the time the *delete_risk* event is received, then the *DeleteRisk()* action will delete this final risk from the risk list of the policy, and will decrement the value of *TotalRisks* down to zero, and hence the policy enters the ***inactive*** state.

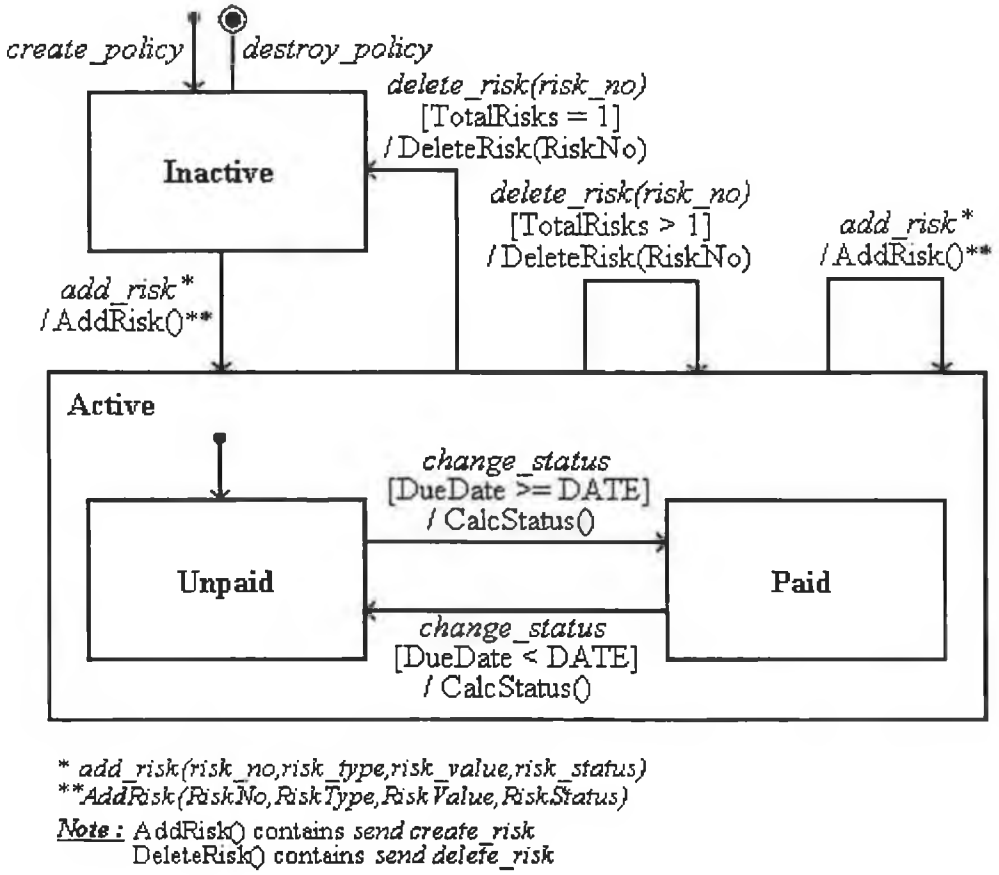


Fig 4.14 Dynamic Model (State Diagram for *TPolicy*)

Within the ***active*** state, there are two substates, ***paid*** and ***unpaid***. The default state is the ***unpaid*** state, but when a *change_status* event is received, a transition can be made to the ***paid*** state provided the current date precedes the *DueDate* for payment on the policy. Similarly, within the ***paid*** state, when a *change_status* event is received, a transition can be made to the ***unpaid*** state provided the *DueDate* for payment on the policy precedes the current date. Finally as a result of the *destroy_policy* event, the life cycle of the policy is over.

As a result of the *create_agent* event, the agent is placed in the ***inactive*** state (an agent is "inactive", when he does not hold any policies, and an agent is "active" when he holds at least one policy) and remains in this state until an *add_policy_to_agent (policy_no)* event is received, which moves the agent to the ***active*** state. Once in the ***active*** state, subsequent *add_policy_to_agent(policy_no)* events will not alter the state of the agent [the *AddPolicy(PolicyNo)* action will add the policy number to the policy number list of the agent, and will increment the value of *PolicyCount* by 1]. However, a *delete_policy_from_agent(policy_no)* event will only allow the agent to remain in the ***active*** state provided he holds more than one policy at the time the event is received [the *DeletePolicy(PolicyNo)* action will delete a policy from the policy number list of the agent, and will decrement the value of *PolicyCount* by 1]. If the agent holds only one policy at the time the *delete_policy_from_agent(policy_no)* event is received, then the *DeletePolicy(PolicyNo)* action will delete this final policy from the policy number list of the agent, and will decrement the value of *PolicyCount* down to zero, and hence the agent enters the ***inactive*** state. Finally as a result of the *destroy_agent* event, the life cycle of the agent is over.

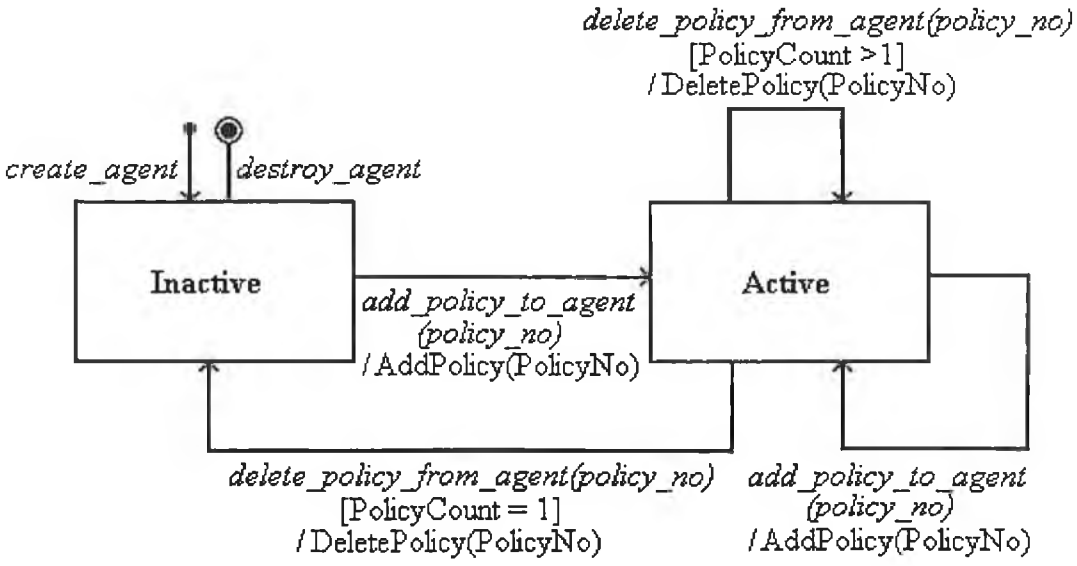


Fig 4.15 Dynamic Model (State Diagram for *TAgent*)

As a result of the *create_claim* event, the claim is placed in the ***pending*** state, and remains in this state until a *change_status* event is received. If the *ClaimDate* is between the *PolicyStart* and the *PolicyEnd* dates, and the *PolicyStatus* is "Paid", then a transition is made to the ***awarded*** state. If however, the *ClaimDate* is not between the *PolicyStart* and *PolicyEnd* dates, or the *PolicyStatus* is "Unpaid", then a transition is made to the ***disallowed*** state. Finally as a result of the *destroy_claim* event, the life cycle of the claim is over.

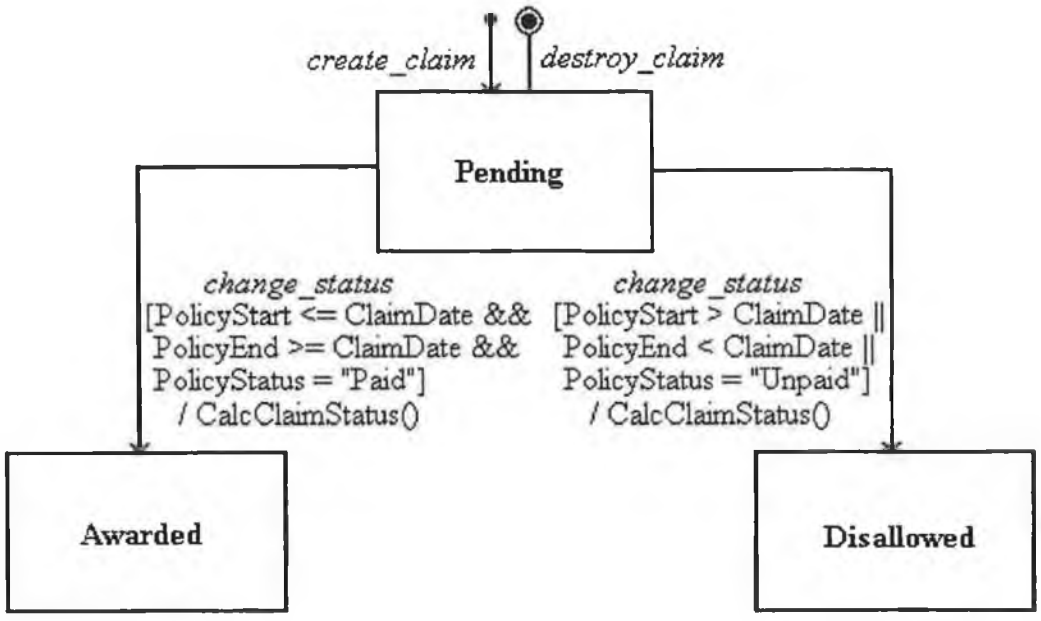
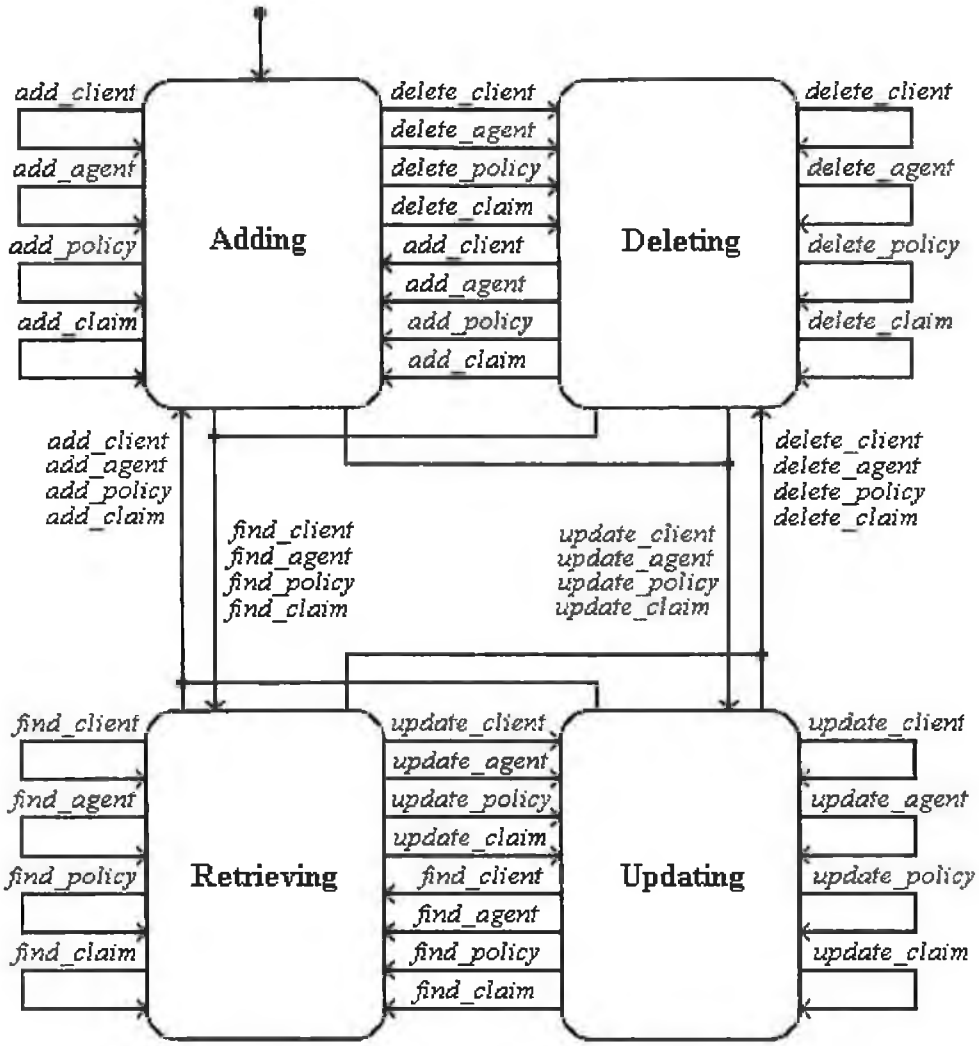


Fig 4.16 Dynamic Model (State Diagram for *TClaim*)

When the company is created, there are no clients, agents, policies or claims, hence the company enters the ***adding*** state where each of these items may be added to the company. As a result of an *add_client* event, the *AddPolicy()* action creates a new client, adds it to the client list and increments *TotalClients* by 1. Similarly, for an *add_agent* event, the *AddAgent()* action creates a new agent, adds it to the agent list and increments *TotalAgents* by 1. If an *add_policy* event is received, there must exist at least one client and one agent in the company, since the policy number must be added to the policy number list of the client who holds the policy, and also added to the policy number list of the agent who sold the policy. The *AddPolicy()* action creates a new policy, adds it to the policy list and increments *TotalPolices* by 1. If an *add_claim* event is received, there must exist at least one policy in the company, since the details of the policy suffering the claim are required to create the claim. The *AddClaim()* action creates the claim, adds it to the claim list and increments *TotalClaims* by 1.



Adding

add_client
/ AddClient(ClientNo,Surname,Firstname,Address, Phone,Occupation,Birthdate,Sex)

add_agent
/ AddAgent(AgentNo,Surname,Firstname, Company,Address,Phone,CommRate)

add_policy [TotalClients > 0 && TotalAgents > 0]
/ AddPolicy(ClientNo,AgentNo,PolicyNo, StartDate,EndDate,CarDetails|HouseDetails)
send add_policy_to_client;
send add_policy_to_agent;

add_claim [TotalPolicies > 0]
/ AddClaim(ClaimNo,ClaimDate,ClaimValue,ClaimDetails, PolicyNo,PolicyStart,PolicyEnd,PolicyStatus)

* AddClient() contains *send create_client*
* AddAgent() contains *send create_agent*
* AddPolicy() contains *send create_policy*
* AddClaim() contains *send create_claim*

Deleting

delete_client [TotalClients > 0]
/ DeleteClient(ClientNo)

delete_agent [TotalAgents > 0]
/ DeleteAgent(AgentNo)

delete_policy [TotalPolicies > 0]
/ DeletePolicy(PolicyNo)
send delete_policy_from_client;
send delete_policy_from_agent;

delete_claim [TotalClaims > 0]
/ DeleteClaim(ClaimNo)

* DeleteClient() contains *send destroy_client*
* DeleteAgent() contains *send destroy_agent*
* DeletePolicy() contains *send destroy_policy*
* DeleteClaim() contains *send destroy_claim*

Retrieving

find_client [TotalClients > 0]
/ GetClient(ClientNo)

find_agent [TotalAgents > 0]
/ GetAgent(AgentNo)

find_policy [TotalPolicies > 0]
/ GetPolicy(PolicyNo)

find_claim [TotalClaims > 0]
/ GetClaim(ClaimNo)

Updating

update_client [TotalClients > 0]
/ UpdateClient(ClientNo,Surname,Firstname,Address,
Phone,Occupation,Birthdate,Sex)

update_agent [TotalAgents > 0]
/ UpdateAgent(AgentNo,Surname,Firstname,
Company,Address,Phone,CommRate)

update_policy [TotalPolicies > 0]
/ UpdatePolicy(PolicyNo,StartDate,EndDate,
CarDetails|HouseDetails)

update_claim [TotalClaims > 0]
/ UpdateClaim(ClaimNo,ClaimDate,ClaimValue,ClaimDetails)

Fig 4.17 Dynamic Model (State Diagram for *TCompany*)

So long as at least one client, agent, policy or claim exists in the company, transitions can be made from any state to any other state : ***adding***; ***deleting***; ***updating***; or ***retrieving***. However, if it arises (through deletion) that these transitions are no longer viable, a transition can always be made from any state back to the ***adding*** state, and more clients, agents, policies and claims can be created.

Note : The state diagrams for *TCar*, *THouse*, and *TRisk* are trivial.

4.4 Constructing the Functional Model

Using my proposed approach which was fully detailed in the previous chapter, the proposed functional model consists of an operation model and an interaction model for each identified system operation.

Hence there are 4 steps in the construction of the functional model :

- Identify input and output values
- Identify system operations
- Build the operation model
- Build the interaction model

4.4.1 Identify Input and Output Values

The input and output values of a system can be identified from the problem statement. In addition, since input and output values are parameters of the events between the system and the outside world, the input and output values can be also be identified from the external events in the dynamic model of the system.

The list of input and output values is as follows : *client details* comprising the client number, the client's surname, the client's firstname, the client's address, the client's phone number, the client's occupation, the client's birthdate and the sex of the client; *agent details* comprising the agent number, the agent's surname, the agent's firstname, the agent's company, the agent's address, the agent's phone number, and the agent's commission rate; *policy details* comprising the client number, the agent number, the policy number, the policy start date, the policy end date, and car policy details or house policy details; *claim details* comprising the claim number, the claim date, the claim value, the claim description, the policy details of the policy on which the claim is being made (i.e. policy number, policy start date, policy end date and policy status); and finally, *risk details* comprising the risk number, the risk type, the risk value and the risk status.

4.4.2 Identify System Operations

As discussed in Chapter 3, system operations are derived from input and output values, and external and internal events in the event flow diagram of the system. From the list of input and output values identified in the previous section, and the external events in the event flow diagram [Figure 4.12] we can identify 18 interactions between the system and the outside world : *add_client*; *delete_client*; *update_client*; *find_client*; *add_policy*; *delete_policy*; *update_policy*; *find_policy*; *add_agent*; *delete_agent*; *update_agent*; *find_agent*; *add_claim*; *delete_claim*; *update_claim*; *find_claim*; *add_risk*; and *delete_risk*;

Each of these will be mapped into a system operation.

[*Note* : No system operations were identified from the internal events of the event flow diagram].

- ***add_client* –> TCompany::AddClient()**

  The *add_client* event is sent from the user to the company and contains the details of the client, i.e. the client number, surname, firstname, address, phone, occupation, birthdate, and sex of the client. This event is mapped into an operation on the TCompany class - *AddClient()*, which creates a new client, stores this new client in the client list of the company, and increments the total number of clients of the company.

- ***delete_client* –> TCompany::DeleteClient()**

  The *delete_client* event is sent from the user to the company and contains the client number of the client which is to de deleted. This event is mapped into an operation on the TCompany class - *DeleteClient()*, which searches through the client list of the company, deletes the specified client, decrements the total number of clients of the company, and re-orders the client list.

- ***update_client* –> TCompany::UpdateClient()**

  The *update_client* event is sent from the user to the company and contains the details to be updated on the client. This event is mapped into an operation on the TCompany class - *UpdateClient()*, which searches through the client list of the company, and updates the specified client with the supplied details.

- ***find_client* –> TCompany::GetClient()**

  The *find_client* event is sent from the user to the company and contains the client number of the client to be retrieved from the client list. This event is mapped into an operation on the TCompany class - *GetClient()*, which searches through the client list of the company and returns the specified client.

- ***add_policy*** **--> TCompany::AddPolicy()**

  The *add_policy* event is sent from the user to the company and contains the details of the policy, i.e. the client number, policy number, agent number, start date, and end date of the policy. Furthermore, if the policy is a car insurance policy, it also contains the manufacturer, model, registration, engine size, and value of the car, and the licence status of the driver of the car (full or provisional). Similarily, if the policy is a house insurance policy, it also contains the house type (semi, bungalow, detatched, etc.), the number of rooms, the area code, the value of the house, the value of the contents in the house, and whether or not the house has an alarm. This event is mapped into an operation on the TCompany class - *AddPolicy()*, which creates a new policy, stores this new policy in the policy list of the company, and increments the total number of policies of the company.

- ***delete_policy*** **--> TCompany::DeletePolicy()**

  The *delete_policy* event is sent from the user to the company and contains the policy number of the policy which is to de deleted. This event is mapped into an operation on the TCompany class - *DeletePolicy()*, which searches through the policy list of the company, deletes the specified policy, decrements the total number of policies of the company, and re-orders the policy list.

- ***update_policy*** **--> TCompany::UpdatePolicy()**

  The *update_policy* event is sent from the user to the company and contains the details to be updated on the policy. This event is mapped into an operation on the TCompany class - *UpdatePolicy()*, which searches through the policy list of the company, and updates the specified policy with the supplied details.

- ***find_policy*** **--> TCompany::GetPolicy()**

  The *find_policy* event is sent from the user to the company and contains the policy number of the policy to be retrieved from the policy list. This event is mapped into an operation on the TCompany class - *GetPolicy()*, which searches through the policy list of the company and returns the specified policy.

- ***add_agent* –> TCompany::AddAgent()**

  The *add_agent* event is sent from the user to the company and contains the details of the agent, i.e. the agent number, surname, firstname, company, address, phone, and commission rate of the agent. This event is mapped into an operation on the TCompany class - *AddAgent()*, which creates a new agent, stores this new agent in the agent list of the company, and increments the total number of agents of the company.

- ***delete_agent* --> TCompany::DeleteAgent()**

  The *delete_agent* event is sent from the user to the company and contains the agent number of the agent which is to de deleted. This event is mapped into an operation on the TCompany class - *DeleteAgent()*, which searches through the agent list of the company, deletes the specified agent, decrements the total number of agents of the company, and re-orders the agent list.

- ***update_agent* –> TCompany::UpdateAgent()**

  The *update_agent* event is sent from the user to the company and contains the details to be updated on the agent. This event is mapped into an operation on the TCompany class - *UpdateAgent()*, which searches through the agent list of the company, and updates the specified agent with the supplied details.

- ***find_agent* --> TCompany::GetAgent()**

  The *find_agent* event is sent from the user to the company and contains the agent number of the agent to be retrieved from the agent list. This event is mapped into an operation on the TCompany class - *GetAgent()*, which searches through the agent list of the company and returns the specified agent.

- ***add_claim* --> TCompany::AddClaim()**

  The *add_claim* event is sent from the user to the company and contains the details of the claim, i.e. the claim number, date, value, and details of the claim, as well as the policy number, start date, end date, and status of the policy suffering the claim. This event is mapped into an operation on the TCompany class - *AddClaim()*, which creates a new claim, stores this new claim in the claim list of the company, and increments the total number of claims of the company.

- ***delete_claim* --> TCompany::DeleteClaim()**

  The *delete_claim* event is sent from the user to the company and contains the claim number of the claim which is to de deleted. This event is mapped into an operation on the TCompany class - *DeleteClaim()*, which searches through the claim list of the company, deletes the specified claim, decrements the total number of claims of the company, and re-orders the claim list.

- ***update_claim* --> TCompany::UpdateClaim()**

  The *update_claim* event is sent from the user to the company and contains the details to be updated on the claim. This event is mapped into an operation on the TCompany class - *UpdateClaim()*, which searches through the claim list of the company, and updates the specified claim with the supplied details.

- ***find_claim* --> TCompany::GetClaim()**

  The *find_claim* event is sent from the user to the company and contains the claim number of the claim to be retrieved from the claim list. This event is mapped into an operation on the TCompany class - *GetClaim()*, which searches through the claim list of the company and returns the specified claim.

- ***add_risk*** **--> TPolicy::AddRisk()**

  The *add_risk* event is sent from the user to the policy and contains the details of the risk, i.e. the risk number (fire, theft, etc.) the risk type (car risk, house risk, etc.), the value of the risk (value of car, value of house, etc.) and the status of the risk (status of car, status of house), to be added to the policy. This event is mapped into an operation on the TPolicy class - *AddRisk()*, which creates a new risk, stores this new risk in the risk list of the policy, and increments the total number of risks on the policy.

- ***delete_risk*** **--> TPolicy::DeleteRisk()**

  The *delete_risk* event is sent from the user to the policy and contains the risk number (fire, theft, etc.) of the risk to be deleted from the policy. This event is mapped into an operation on the TPolicy class - *DeleteRisk()*, which searches through the risk list of the policy, deletes the specified risk, decrements the total number of risks on the policy, and re-orders the risk list.

4.4.3 Build the Operation Model

The operation model specifies the behaviour of system operations declaratively by defining their effect in terms of the change of state of the system, where the state of the system is an abstraction of the values of the objects in the system. Hence, the operation model consists of a set of operation schema, which define the change of state, where there exists one operation schema for each of the identified system operations.

To construct the operation schema for each system operation : firstly, concisely describe the purpose of the operation; secondly, list the data items which must be read and/or updated, and thirdly, describe the state of the system before the operation is executed and after the operation is executed, in terms of pre-conditions and post-conditions respectively.

The operation schemas for each system operation are given below :

Operation :	**TCompany::AddClient**
Description :	Creates a new client and adds this new client to the client list of the company.
Reads :	**supplied** ClientNo : string; **supplied** Surname : string; **supplied** Firstname : string; **supplied** Address : string; **supplied** Phone : string; **supplied** Occupation : string; **supplied** Birthdate : string; **supplied** Sex : string;
Updates :	**new** Client, ClientList, TotalClients
Pre-conditions :	TotalClients < MAX_CLIENTS. ClientNo is a unique value not existing in the ClientList.
Post-conditions :	Client.ClientNo has been set to ClientNo. Client.Surname has been set to Surname. Client.Firstname has been set to Firstname. Client.Address has been set to Address. Client.Phone has been set to Phone. Client.Occupation has been set to Occupation. Client.Birthdate has been set to Birthdate. Client.Sex has been set to Sex. Client has been added to the ClientList. TotalClients has been incremented by 1.

Operation :	**TCompany::DeleteClient**
Description :	Deletes an existing client from the client list of the company.
Reads :	**supplied** ClientNo : string;
Updates :	ClientList, TotalClients
Pre-conditions :	ClientNo is a unique value existing in the ClientList.
Post-conditions :	The client with Client.ClientNo = ClientNo has been deleted from the ClientList. TotalClients has been decremented by 1.

Operation :	**TCompany::UpdateClient**
Description :	Updates an existing client in the client list of the company.

Reads :	**supplied** ClientNo : string; **supplied** Surname : string; **supplied** Firstname : string; **supplied** Address : string; **supplied** Phone : string; **supplied** Occupation : string; **supplied** Birthdate : string; **supplied** Sex : string;
Updates :	Client, ClientList
Pre-conditions :	ClientNo is a unique value existing in the ClientList.
Post-conditions :	The client with Client.ClientNo = ClientNo has been updated as follows : Client.Surname has been set to Surname. Client.Firstname has been set to Firstname. Client.Address lias been set to Address. Client.Phone has been set to Phone. Client.Occupation has been set to Occupation. Client.Birthdate has been set to Birthdate. Client.Sex has been set to Sex.

Operation :	**TCompany::GetClient**
Description :	Retrieves an existing client from the client list of the company.

Reads :	**supplied** ClientNo : string;
Updates :	
Pre-conditions :	ClientNo is a unique value existing in the ClientList.
Post-conditions :	The client with Client.ClientNo = ClientNo has been retrieved from the ClientList.

Operation :	**TCompany::AddPolicy**
Description :	Creates a new car policy and adds this new car policy to the policy list of the company.
Reads :	**supplied** ClientNo : string; **supplied** PolicyNo : string; **supplied** AgentNo : string; **supplied** StartDate : string; **supplied** EndDate : string; **supplied** Manufacturer : string; **supplied** Model : string; **supplied** Registration : string **supplied** EngineSize : string; **supplied** CarValue : long **supplied** FullLicenceStatus : integer;
Updates :	**new** Policy, PolicyList, TotalPolicies
Pre-conditions :	TotalPolicies < MAX_POLICIES. PolicyNo is a unique value not existing in the PolicyList.
Post-conditions :	Policy.ClientNo has been set to ClientNo. Policy.PolicyNo has been set to PolicyNo. Policy.AgentNo has been set to AgentNo. Policy.StartDate has been set to StartDate. Policy.EndDate has been set to EndDate. Policy.Manufacturer has been set to Manufacturer. Policy.Model has been set to Model. Policy.Registration has been set to Registration. Policy.EngineSize has been set to EngineSize. Policy.CarValue has been set to CarValue. Policy.FullLicenceStatus has been set to FullLicenceStatus. TotalPolicies has been incremented by 1.

Operation :	**TCompany::AddPolicy**
Description :	Creates a new house policy and adds this new house policy to the policy list of the company.
Reads :	**supplied** ClientNo : string; **supplied** PolicyNo : string; **supplied** AgentNo : string; **supplied** StartDate : string; **supplied** EndDate : string; **supplied** HouseType : string; **supplied** Rooms : integer; **supplied** AreaCode : string **supplied** HouseValue : long; **supplied** ContentsValue : long; **supplied** HouseAlarmStatus : integer;
Updates :	**new** Policy, PolicyList, TotalPolicies

Pre-conditions :	TotalPolicies < MAX_POLICIES.
	PolicyNo is a unique value not existing in the PolicyList.
Post-conditions :	Policy.ClientNo has been set to ClientNo.
	Policy.PolicyNo has been set to PolicyNo.
	Policy.AgentNo has been set to AgentNo.
	Policy.StartDate has been set to StartDate.
	Policy.EndDate has been set to EndDate.
	Policy.HouseType has been set to HouseType.
	Policy.Rooms has been set to Rooms.
	Policy.AreaCode has been set to AreaCode.
	Policy.HouseValue has been set to HouseValue.
	Policy.ContentsValue has been set to ContentsValue.
	Policy.HouseAlarmStatus has been set to HouseAlarmStatus.
	TotalPolicies has been incremented by 1.

Operation :	**TCompany::DeletePolicy**
Description :	Deletes an existing policy from the policy list of the company.

Reads :	**supplied** PolicyNo : string;
Updates :	PolicyList, TotalPolicies
Pre-conditions :	PolicyNo is a unique value existing in the PolicyList.
Post-conditions :	The policy with Policy.PolicyNo = PolicyNo has been deleted from the PolicyList.
	TotalPolicies has been decremented by 1.

Operation :	**TCompany::UpdatePolicy**
Description :	Updates an existing car policy in the policy list of the company.

Reads :	**supplied** PolicyNo : string; **supplied** StartDate : string;
	supplied EndDate : string; **supplied** Manufacturer : string;
	supplied Model : string; **supplied** Registration : string;
	supplied EngineSize : string; **supplied** CarValue : long;
	supplied FulllLicenceStatus : integer;
Updates :	CarPolicy, PolicyList

Pre-conditions :	PolicyNo is a unique value existing in the PolicyList.
Post-conditions :	The policy with CarPolicy.PolicyNo = PolicyNo has been updated as follows : CarPolicy.StartDate has been set to StartDate. CarPolicy.EndDate has been set to EndDate. CarPolicy.Manufacturer has been set to Manufacturer. CarPolicy.Model has been set to Model. CarPolicy.Registration has been set to Registration. CarPolicy.EngineSize has been set to EngineSize. CarPolicy.CarValue has been set to CarValue. CarPolicy.FullLicenceStatus has been set to FullLicenceStatus.

Operation :	**TCompany::UpdatePolicy**
Description :	Updates an existing house policy in the policy list of the company.

Reads :	**supplied** PolicyNo : string; **supplied** StartDate : string; **supplied** EndDate : string; **supplied** HouseType: string; **supplied** Rooms : integer; **supplied** AreaCode : string; **supplied** HouseValue : long; **supplied** ContentsValue : long; **supplied** HouseAlarmStatus : integer;
Updates :	HousePolicy, PolicyList
Pre-conditions :	PolicyNo is a unique value existing in the PolicyList.
Post-conditions :	The policy with HousePolicy.PolicyNo = PolicyNo has been updated as follows : HousePolicy.StartDate has been set to StartDate. HousePolicy.EndDate has been set to EndDate. HousePolicy.HouseType has been set to HouseType. HousePolicy.Rooms has been set to Rooms. HousePolicy.AreaCode has been set to AreaCode. HousePolicy.HouseValue has been set to HouseValue. HousePolicy.ContentsValue has been set to ContentsValue. HousePolicy.HouseAlarmStatus has been set to HouseAlarmStatus.

Operation :	**TCompany::GetPolicy**
Description :	Retrieves an existing policy from the policy list of the company
Reads :	**supplied** PolicyNo : string;
Updates :	
Pre-conditions :	PolicyNo is a unique value existing in the PolicyList.
Post-conditions :	The policy with Policy.PolicyNo = PolicyNo has been retrieved from the PolicyList.

Operation :	**TCompany::AddAgent**
Description :	Creates a new agent and adds this new agent to the agent list of the company.
Reads :	**supplied** AgentNo : string; **supplied** Surname : string; **supplied** Firstname : string; **supplied** Company : string; **supplied** Address : string; **supplied** Phone : string;
Updates :	**new** Agent, AgentList, TotalAgents
Pre-conditions :	TotalAgents < MAX_AGENTS. AgentNo is a unique value not existing in the AgentList.
Post-conditions :	Agent.AgentNo has been set to AgentNo. Agent.Surname has been set to Surname. Agent.Firstname has been set to Firstname. Agent.Company has been set to Company. Agent.Address has been set to Address. Agent.Phone has been set to Phone. Agent has been added to the AgentList. TotalAgents has been incremented by 1.

Operation :	**TCompany::DeleteAgent**
Description :	Deletes an existing agent from the agent list of the company.
Reads :	**supplied** AgentNo : string;
Updates :	AgentList, TotalAgents
Pre-conditions :	AgentNo is a unique value existing in the AgentList.
Post-conditions :	The agent with Agent.AgentNo = AgentNo has been deleted from the AgentList. TotalAgents has been decremented by 1.

Operation :	**TCompany::UpdateAgent**
Description :	Updates an existing agent in the agent list of the company.
Reads :	**supplied** AgentNo : string; **supplied** Surname : string; **supplied** Firstname : string; **supplied** Company : string; **supplied** Address : string; **supplied** Phone : string;
Updates :	Agent, AgentList
Pre-conditions :	AgentNo is a unique value existing in the AgentList.
Post-conditions :	The agent with Agent.AgentNo = AgentNo has been updated as follows : Agent.Surname has been set to Surname. Agent.Firstname has been set to Firstname. Agent.Company has been set to Company. Agent.Address has been set to Address. Agent.Phone has been set to Phone.

Operation :	**TCompany::GetAgent**
Description :	Retrieves an existing agent from the agent list of the company.
Reads :	**supplied** AgentNo : string;
Updates :	
Pre-conditions :	AgentNo is a unique value existing in the AgentList.
Post-conditions :	The agent with Agent.AgentNo = AgentNo has been retrieved from the AgentList.

Operation :	**TCompany::AddClaim**
Description :	Creates a new claim and adds this new claim to the claim list of the company.

Reads :	**supplied** ClaimNo : string; **supplied** ClaimDate : string; **supplied** ClaimValue : long; **supplied** ClaimDetails : string; **supplied** PolicyNo : string; **supplied** PolicyStart : string; **supplied** PolicyEnd : string; **supplied** PolicyStatus : string;
Updates :	**new** Claim, ClaimList, TotalClaims
Pre-conditions :	TotalClaims < MAX_CLAIMS. ClaimNo is a unique value not existing in the ClaimList.
Post-conditions :	Claim.ClaimNo has been set to ClaimNo. Claim.ClaimDate has been set to ClaimDate. Claim.ClaimValue has been set to ClaimValue. Claim.ClaimDetails has been set to ClaimDetails. Claim.PolicyNo has been set to PolicyNo. Claim.PolicyStart has been set to PolicyStart. Claim.PolicyEnd has been set to PolicyEnd. Claim.PolicyStatus has been set to PolicyStatus. Claim has been added to the ClaimList. TotalClaims has been incremented by 1.

Operation :	**TCompany::DeleteClaim**
Description :	Deletes an existing claim from the claim list of the company.

Reads :	**supplied** ClaimNo : string;
Updates :	ClaimList, TotalClaims
Pre-conditions :	ClaimNo is a unique value existing in the ClaimList.
Post-conditions :	The claim with Claim.ClaimNo = ClaimNo has been deleted from the ClaimList. TotalClaims has been decremented by 1.

Operation :	**TCompany::UpdateClaim**
Description :	Updates an existing claim in the client list of the company.

Reads :	**supplied** ClaimNo : string; **supplied** ClaimDate : string; **supplied** ClaimValue : string; **supplied** ClaimDetails : string;
Updates :	Claim, ClaimList
Pre-conditions :	ClaimNo is a unique value existing in the ClaimList.
Post-conditions :	The claim with Claim.ClaimNo = ClaimNo has been updated as follows : Claim.ClaimDate has been set to ClaimDate. Claim.ClaimValue has been set to ClaimValue. Claim.ClaimDetails has been set to ClaimDetails.

Operation :	**TCompany::GetClaim**
Description :	Retrieves an existing claim from the claim list of the company.

Reads :	**supplied** ClaimNo : string;
Updates :	
Pre-conditions :	ClaimNo is a unique value existing in the ClaimList.
Post-conditions :	The claim with Claim.ClaimNo = ClaimNo has been retrieved from the ClaimList.

Operation :	**TPolicy::AddRisk**
Description :	Creates a new risk and adds this new risk to the risk list of the policy.

Reads :	**supplied** RiskNo : string; **supplied** RiskType : long; **supplied** RiskValue : string; **supplied** RiskStatus : long;
Updates :	**new** Risk, RiskList, TotalRisks
Pre-conditions :	TotalRisks < MAX_RISKS. RiskNo is a unique value not existing in the RiskList.
Post-conditions :	Risk.RiskNo has been set to RiskNo. Risk.RiskType has been set to RiskType. Risk.RiskValue has been set to RiskValue. Risk.RiskStatus has been set to RiskStatus. TotalRisks has been incremented by 1.

Operation :	**TPolicy::DeleteRisk**
Description :	Deletes an existing risk from the risk list of the policy.
Reads :	**supplied** RiskNo : string;
Updates :	RiskList, TotalRisks
Pre-conditions :	RiskNo is a unique value existing in the RiskList.
Post-conditions :	The risk with Risk.RiskNo = RiskNo has been deleted from the RiskList. TotalRisks has been decremented by 1.

4.4.4 Build the Interaction Model

Where the operation model specifies what an operation does, the interaction model specifies how the operation works by illustrating the internal workings of the operation in terms of its sub-ordinate operations. Hence, the interaction model consists of a set of interaction diagrams, which illustrate the sub-ordinate operations comprising each of the identified system operations, where there exists one interaction diagram for each system operation.

To construct the interaction diagram for each system operation : firstly, examine the *reads* and *updates* clauses of the operation model, to yield the sub-ordinate operations required in order to read and update the necessary information relating to this system operation; secondly, list these sub-ordinate operations in descending order between the appropriate classes on the tabular diagram, using the same ordering for these sub-ordinate operations as the ordering in the *post-conditions* clause (which relates to the information in both the *reads* and *updates* clauses), to achieve the functionality outlined in the *description* clause.

The interaction diagrams for each system operation are given below :

- **void TCompany::AddClient(char *ClientNo, char *Surname,**
  char *Firstname, char *Address, char *Phone,
  char *Occupation, char *Birthdate, char *Sex)

The *AddClient()* operation checks whether the total number of clients exceeds the maximum number of clients, and if not, it creates a new client, stores this client in the client list of the company, and increments the total number of clients. Hence the sub-ordinate operations are ***TCompany::GetTotalClients()***, which reads the total number of clients; ***TClient::TClient()***, which creates the new client; ***TCompany::PutClient()***, which updates the client list with the new client; and ***TCompany::IncTotalClients()***, which increments the total number of clients.

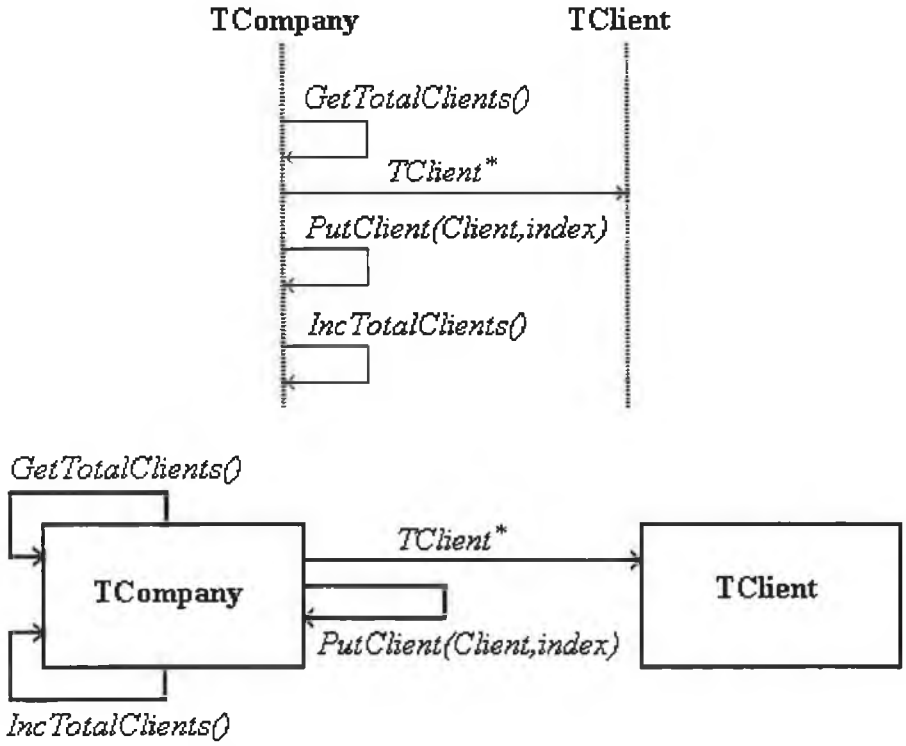


Fig 4.18 Interaction Diagram for *AddClient()*

- **void TCompany::DeleteClient(char *ClientNo)**

The *DeleteClient()* operation searches through the client list until the specified client is found, the client is then deleted, the total number of clients is decremented, and the client list is re-ordered. Hence the sub-ordinate operations are ***TCompany::GetTotalClients()***, which reads the total number of clients (required to search the client list); ***TCompany::GetClientNo()***, which reads the client number at the current position in the client list, to determine if the current client is the specified client; ***TClient::~TClient()***, which deletes the specified client from the client list; ***TCompany::DecTotalClients()***, which decrements the total number of clients; and ***TCompany::PutClient()***, which is called for each client in the list after the deleted client, thus re-ordering the list, and filling the empty space caused by the deletion.

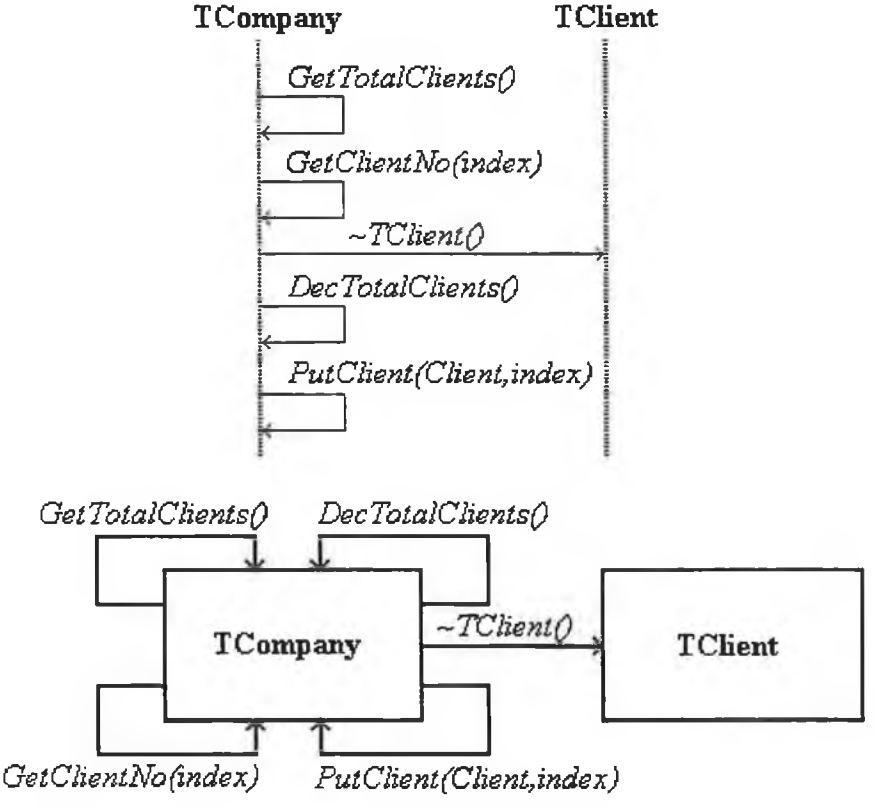


Fig 4.19 Interaction Diagram for *DeleteClient()*

- **void TCompany::UpdateClient(char *ClientNo, char *Surname,**
  char *Firstname, char *Address,
  char *Phone, char *Occupation,
  char *Birthdate, char *Sex)

The *UpdateClient()* operation retrieves the specified client from the client list and updates the details of this client. Hence the sub-ordinate operations are ***TCompany::GetClient()***, which retrieves the specified client from the client list; ***TClient::PutSurname()***, which updates the surname of the client; ***TClient:: PutFirstname()***, which updates the first name of the client; ***TClient:: PutAddress()***, which updates the address of the client; ***TClient::PutPhone()***, which updates the phone number of the client; ***TClient::PutOccupation()***, which updates the occupation of the client; ***TClient::PutBirthdate()***, which updates the birthdate of the client; and ***TClient::PutSex()***, which updates the sex of the client.

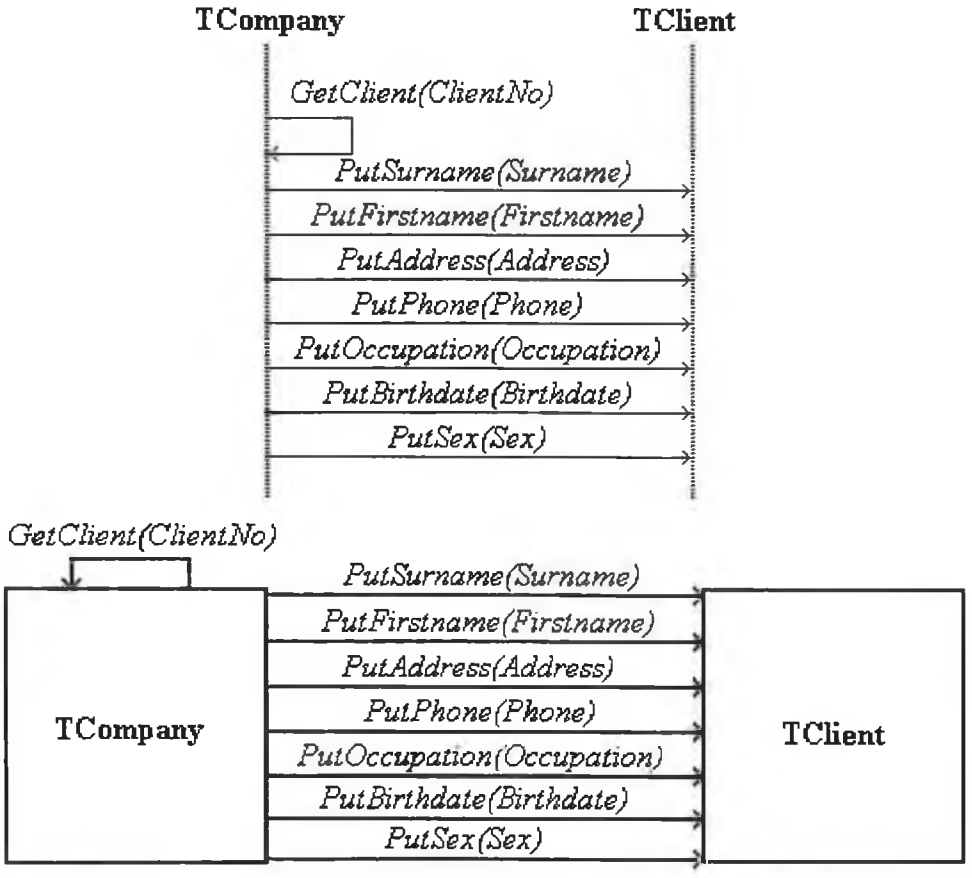


Fig 4.20 Interaction Diagram for *UpdateClient()*

- **TClient *TCompany::GetClient(char *ClientNo)**

The *GetClient()* operation searches the client list and returns the specified client. Hence the sub-ordinate operations are ***TCompany::GetTotalClients()***, which reads the total number of clients (required to search the client list); and ***TCompany::GetClientNo()***, which reads the client number at the current position in the client list, to determine if the current client is the specified client.

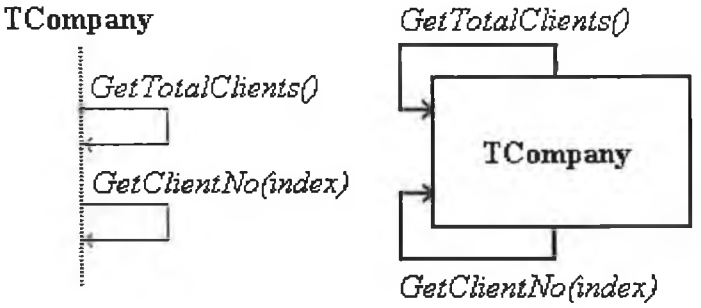


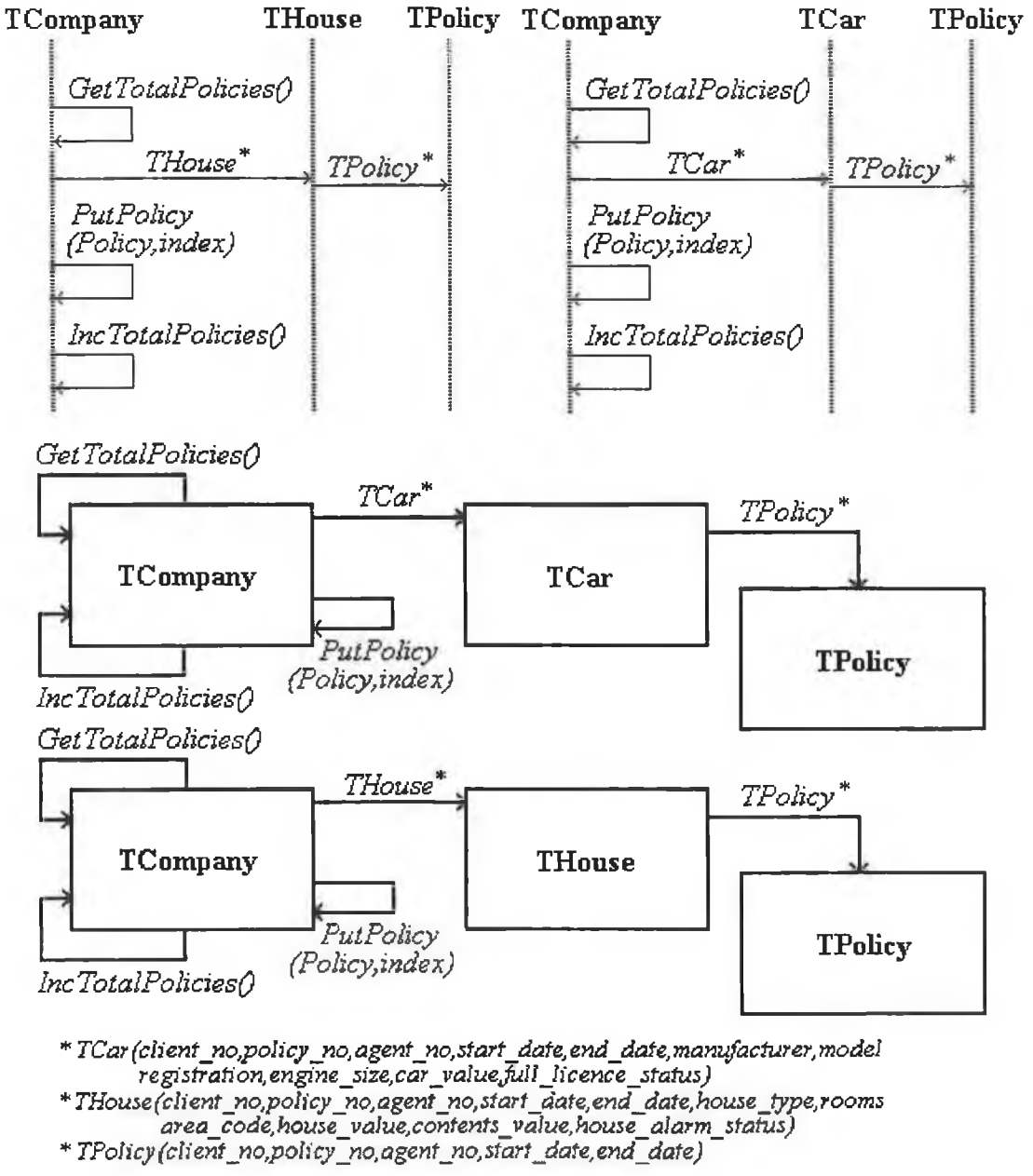
Fig 4.21 Interaction Diagram for *GetClient()*

- **void TCompany::AddPolicy(char *ClientNo, char *PolicyNo,**
  char *AgentNo, char *StartDate,
  char *EndDate, char *Manufacturer,
  char *Model, char *Registration,
  char *EngineSize, long CarValue,
  int FullLicenceStatus)

also :

void TCompany::AddPolicy(char *ClientNo, char *PolicyNo,
char *AgentNo, char *StartDate,
char *EndDate, char *HouseType,
int Rooms, char *AreaCode,
long HouseValue, long ContentsValue,
int FullLicenceStatus)

The *AddPolicy()* operation checks whether the total number of policies exceeds the maximum number of policies, and if not, it creates a new policy, stores this policy in the policy list of the company, and increments the total number of policies. Hence the sub-ordinate operations are ***TCompany::GetTotalPolicies()***, which reads the total number of policies; ***TCar::TCar()***, which creates the new car insurance policy; ***THouse::THouse()***, which creates the new house insurance policy; (both the ***TCar*** and ***THouse*** constructors call the constructor of the abstract ***TPolicy*** class - ***TPolicy::TPolicy()***, from which they are both inherited); ***TCompany::PutPolicy()***, which updates the policy list with the new policy; and ***TCompany::IncTotalPolicies()***, which increments the total number of policies.



* *TCar(client_no,policy_no,agent_no,start_date,end_date,manufacturer,model registration,engine_size,car_value,full_licence_status)*
* *THouse(client_no,policy_no,agent_no,start_date,end_date,house_type,rooms area_code,house_value,contents_value,house_alarm_status)*
* *TPolicy(client_no,policy_no,agent_no,start_date,end_date)*

Fig 4.22 Interaction Diagram for *AddPolicy()*

- **void TCompany::DeletePolicy(char *PolicyNo)**

The *DeletePolicy()* operation searches through the policy list until the specified policy is found, the policy is then deleted, the total number of policies is decremented, and the policy list is re-ordered. Hence the sub-ordinate operations are ***TCompany::GetTotalPolicies()***, which reads the total number of policies (required to search the policy list); ***TCompany::GetPolicyNo()***, which reads the policy number at the current position in the policy list, to determine if the current policy is the specified policy; ***TPolicy::~TPolicy()***, which deletes the specified policy from the policy list; ***TCompany::DecTotalPolicies()***, which decrements the total number of policies; and ***TCompany::PutPolicy()***, which is called for each policy in the list after the deleted policy, thus re-ordering the list, and filling the empty space caused by the deletion.

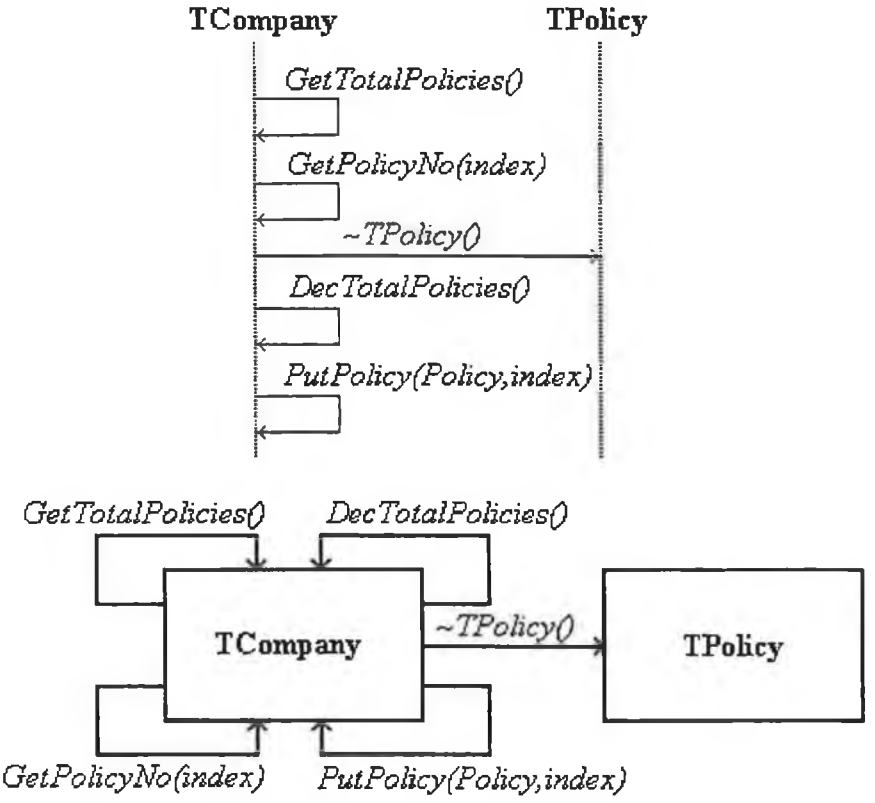


Fig 4.23 Interaction Diagram for *DeletePolicy()*

- **void TCompany::UpdatePolicy(char *PolicyNo, char *StartDate,**
  char *EndDate, char *Manufacturer,
  char *Model, char *Registration,
  char *EngineSize, long CarValue,
  int FullLicenceStatus)

  also :

  void TCompany::UpdatePolicy(char *PolicyNo, char *StartDate,
  char *EndDate, char *HouseType,
  int Rooms, char *AreaCode,
  long HouseValue, long ContentsValue,
  int HouseAlarmStatus)

  The *UpdatePolicy()* operation retrieves the specified policy from the policy list and updates the details of this car insurance policy or house insurance policy. Hence the sub-ordinate operations are ***TCompany::GetPolicy()***, which retrieves the specified policy from the policy list; ***TCar::PutStartDate()***, which updates the start date of the car policy; ***TCar::PutEndDate()***, which updates the end date of the car policy; ***TCar::PutManufacturer()***, which updates the manufactuer of the car on the car policy; ***TCar::PutModel()***, which updates the model of the car on the car policy; ***TCar::PutRegistration()***, which updates the registration number of the car on the car policy; ***TCar::PutEngineSize()***, which updates the engine size of the car on the car policy; ***TCar::PutCarValue()***, which updates the value of the car on the car policy; and ***TCar::PutFullLicenceStatus()***, which updates the licence status of the driver on the car policy. Also, ***THouse::PutStartDate()***, which updates the start date of the house policy; ***THouse::PutEndDate()***, which updates the end date of the house policy; ***THouse::PutHouseType()***, which updates the house type of the house on the house policy; ***THouse::PutRooms()***, which updates the number of rooms in the house on the house policy; ***THouse::PutAreaCode()***, which updates the area code of the house on the house policy; ***THouse::PutHouseValue()***, which updates the value of the house on the house policy; ***THouse::PutContentsValue()***, which updates the value of the contents of the house on the house policy; and ***THouse::PutHouseAlarm Status()***, which updates the house alarm status of the house on the house policy.

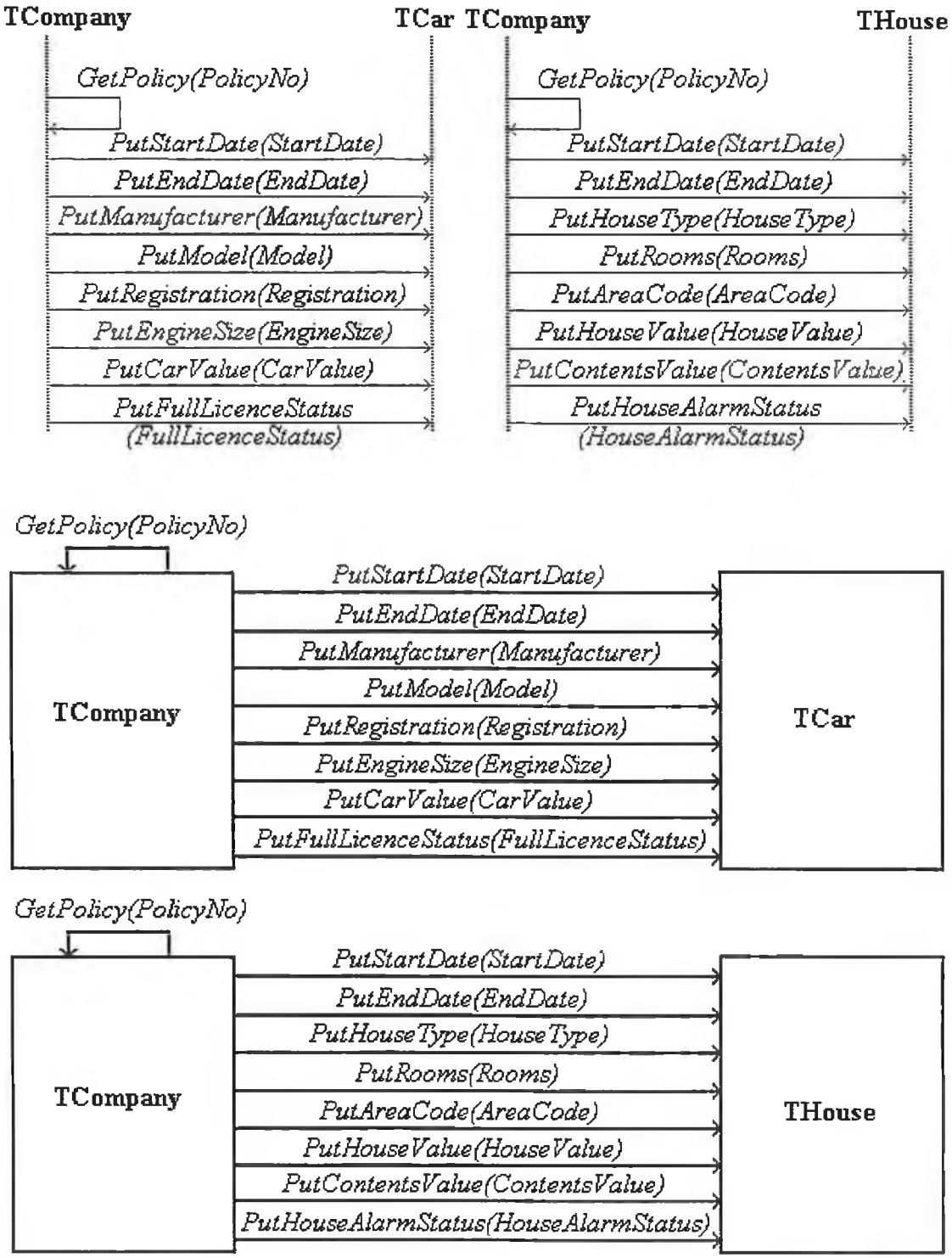


Fig 4.24 Interaction Diagram for *UpdatePolicy()*

- **TPolicy *TCompany::GetPolicy(char *PolicyNo)**

The *GetPolicy()* operation searches the policy list and returns the specified policy. Hence the sub-ordinate operations are ***TCompany::GetTotalPolicies()***, which reads the total number of policies (required to search the policy list); and ***TCompany::GetPolicyNo()***, which reads the policy number at the current position in the policy list, to determine if the current client is the specified policy.

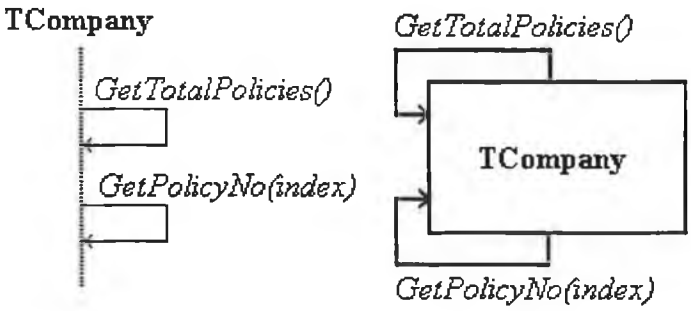
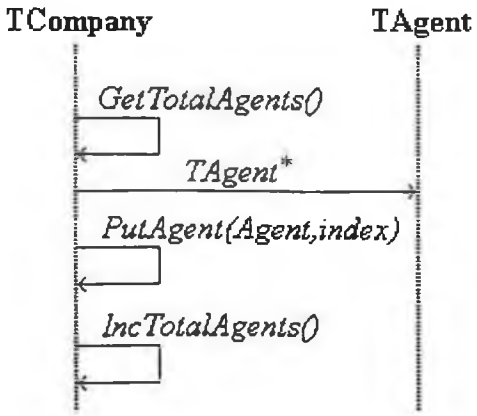
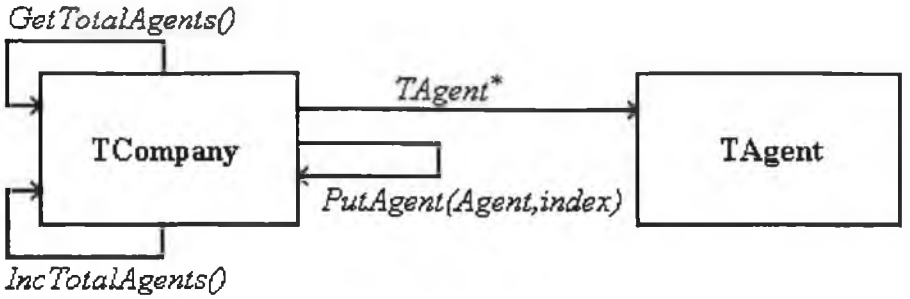


Fig 4.25 Interaction Diagram for *GetPolicy()*

- **void TCompany::AddAgent(char *AgentNo, char *Surname,**
  char *Firstname, char *Company,
  char *Address, char *Phone, int CommRate)

The *AddAgent()* operation checks whether the total number of agents exceeds the maximum number of agents, and if not, it creates a new agent, stores this agent in the agent list of the company, and increments the total number of agents. Hence the sub-ordinate operations are ***TCompany::GetTotalAgents()***, which reads the total number of agents; ***TAgent::TAgent()***, which creates the new agent; ***TCompany::PutAgent()***, which updates the agent list with the new agent; and ***TCompany::IncTotalAgents()***, which increments the total number of agents.



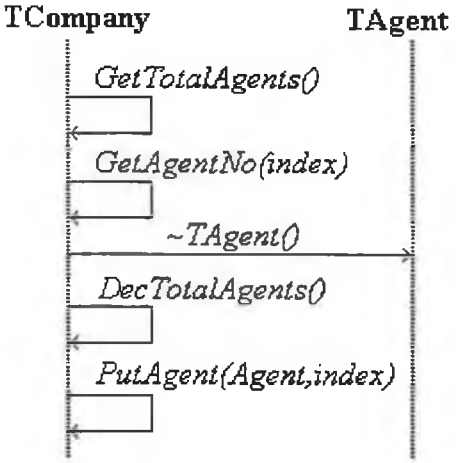


* *TAgent(agent_no,surname,firstname,company,address,phone,comm_rate)*

Fig 4.26 Interaction Diagram for *AddAgent()*

- **void TCompany::DeleteAgent(char *AgentNo)**

The *DeleteAgent()* operation searches through the agent list until the specified agent is found, the agent is then deleted, the total number of agents is decremented, and the agent list is re-ordered. Hence the sub-ordinate operations are ***TCompany::GetTotalAgents()***, which reads the total number of agents (required to search the agent list); ***TCompany::GetAgentNo()***, which reads the agent number at the current position in the agent list, to determine if the current agent is the specified agent; ***TAgent::~TAgent()***, which deletes the specified agent from the agent list; ***TCompany::DecTotalAgents()***, which decrements the total number of agents; and ***TCompany::PutAgent()***, which is called for each agent in the list after the deleted agent, thus re-ordering the list, and filling the empty space caused by the deletion.



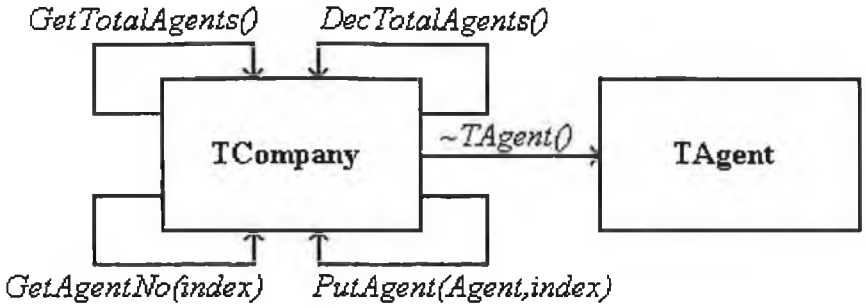
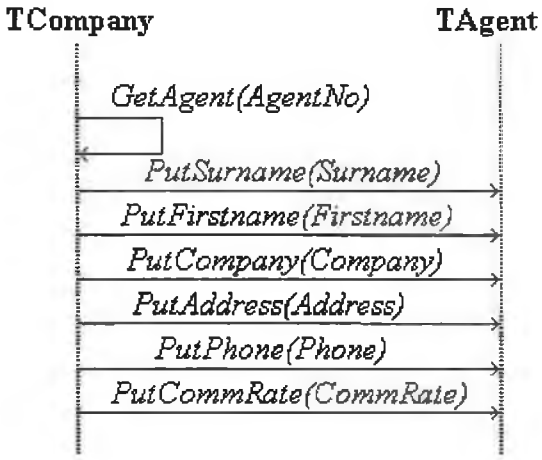


Fig 4.27 Interaction Diagram for *DeleteAgent()*

- **void TCompany::UpdateAgent(char *AgentNo, char *Surname,**
  char *Firstname, char *Company,
  char *Address, char *Phone, int CommRate)

The *UpdateAgent()* operation retrieves the specified agent from the agent list and updates the details of this agent. Hence the sub-ordinate operations are ***TCompany::GetAgent()***, which retrieves the specified agent from the agent list; ***TAgent::PutSurname()***, which updates the surname of the agent; ***TAgent::PutFirstname()***, which updates the first name of the agent; ***TAgent::PutCompany()***, which updates the company name of the agent; ***TAgent::PutAddress()***, which updates the company address of the agent; ***TAgent::PutPhone()***, which updates the company phone number of the agent; and ***TAgent::PutCommRate()***, which updates the commission rate of the agent.



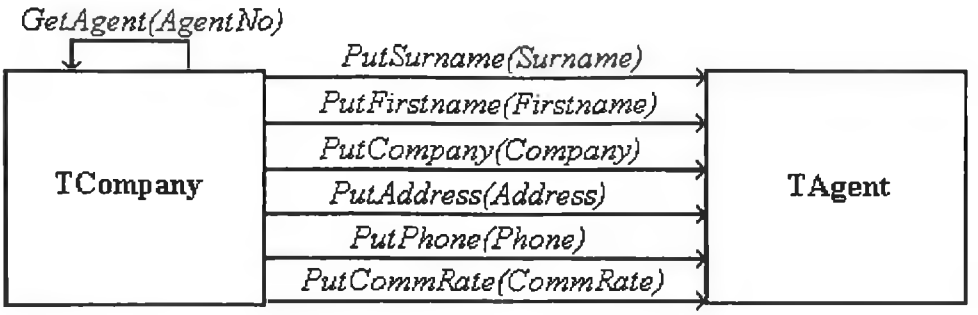


Fig 4.28 Interaction Diagram for *UpdateAgent()*

- **TAgent *TCompany::GetAgent(char *AgentNo)**

The *GetAgent()* operation searches the agent list and returns the specified agent. Hence the sub-ordinate operations are ***TCompany::GetTotalAgents()***, which reads the total number of agents (required to search the agent list); and ***TCompany::GetAgentNo()***, which reads the agent number at the current position in the agent list, to determine if the current client is the specified agent.

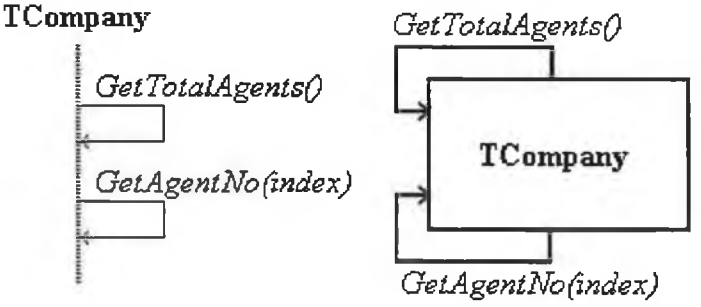


Fig 4.29 Interaction Diagram for *GetAgent()*

- **void TCompany::AddClaim(char *ClaimNo, char *ClaimDate,**
  long ClaimValue, char *ClaimDetails,
  char *PolicyNo, char *PolicyStart,
  char *PolicyEnd, char *PolicyStatus)

The *AddClaim()* operation checks whether the total number of claims exceeds the maximum number of claims, and if not, it creates a new claim, stores this claim in the claim list of the company, and increments the total number of claims. Hence the sub-ordinate operations are ***TCompany::GetTotalClaims()***, which reads the total number of claims; ***TClaim::TClaim()***, which creates the new claim; ***TCompany::PutClaim()***, which updates the claim list with the new claim; and ***TCompany::IncTotalClaims()***, which increments the total number of claims.

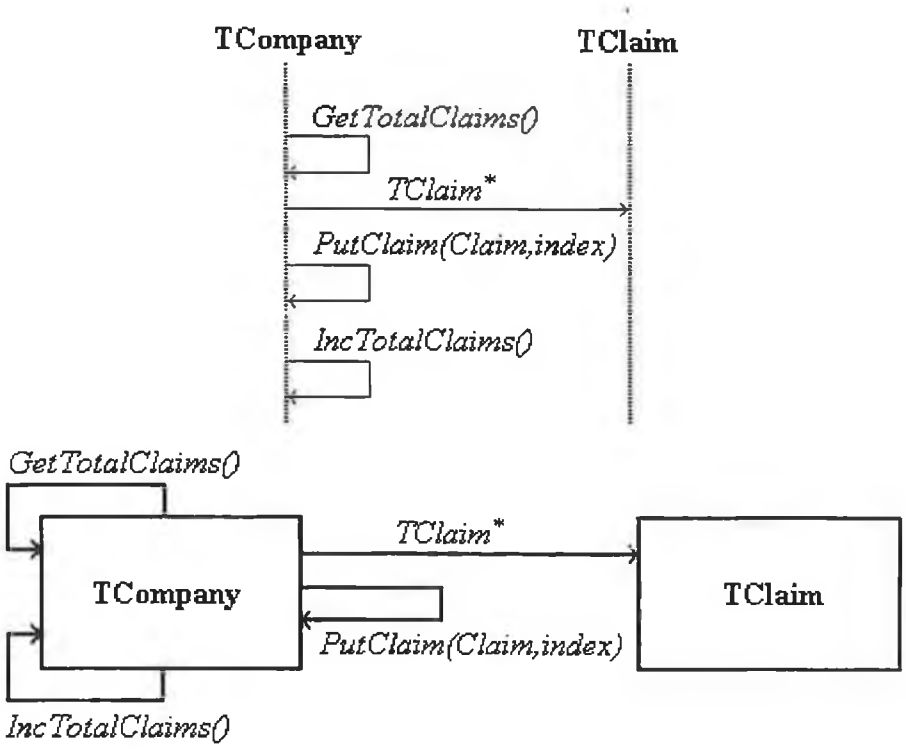


Fig 4.30 Interaction Diagram for *AddClaim()*

- **void TCompany::DeleteClaim(char *ClaimNo)**

The *DeleteClaim()* operation searches through the claim list until the specified claim is found, the claim is then deleted, the total number of claims is decremented, and the claim list is re-ordered. Hence the sub-ordinate operations are ***TCompany::GetTotalClaims()***, which reads the total number of claims (required to search the claim list); ***TCompany::GetClaimNo()***, which reads the claim number at the current position in the claim list, to determine if the current claim is the specified claim; ***TClaim::~TClaim()***, which deletes the specified claim from the claim list; ***TCompany::DecTotalClaims()***, which decrements the total number of claims; and ***TCompany::PutClaim()***, which is called for each claim in the list after the deleted claim, thus re-ordering the list, and filling the empty space caused by the deletion.

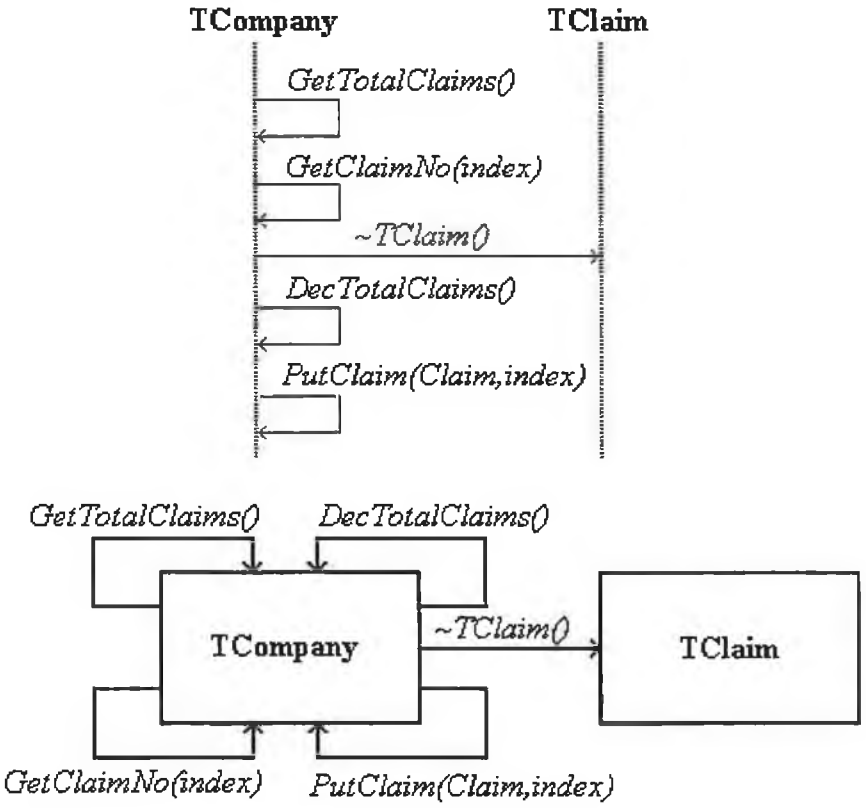


Fig 4.31 Interaction Diagram for *DeleteClaim()*

- **void TCompany::UpdateClaim(char *ClaimNo, char *ClaimDate, long ClaimValue, char *ClaimDetails)**

The *UpdateClaim()* operation retrieves the specified claim from the claim list and updates the details of this claim. Hence the sub-ordinate operations are ***TCompany::GetClaim()***, which retrieves the specified claim from the claim list; ***TClaim::PutClaimDate()***, which updates the claim date of the claim; ***TClaim::PutClaimValue()***, which updates the claim value of the claim; and ***TClaim::PutClaimDetails()***, which updates the claim details of the claim.

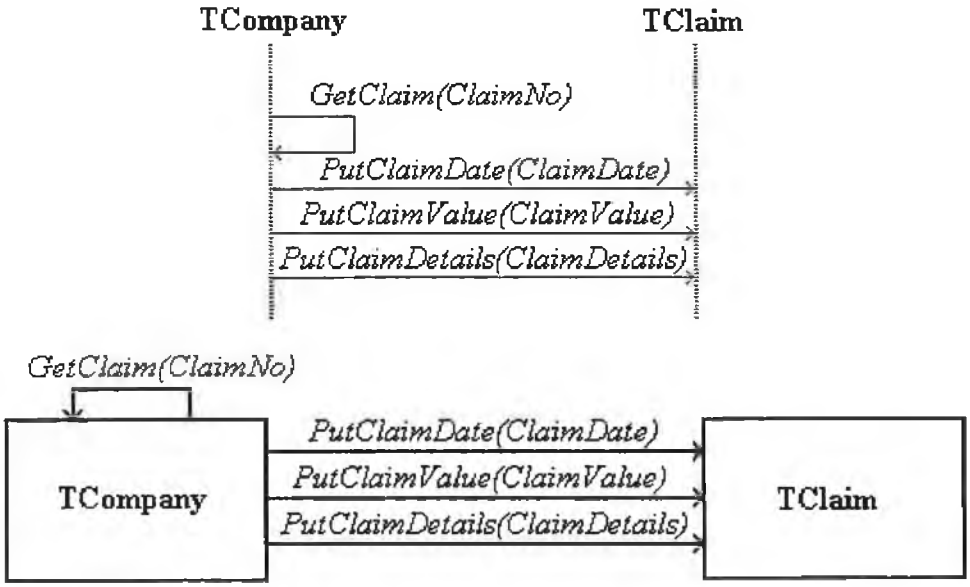


Fig 4.32 Interaction Diagram for *UpdateClaim()*

- **TClaim *TCompany::GetClaim(char *ClaimNo)**

The *GetClaim()* operation searches the claim list and returns the specified claim. Hence the sub-ordinate operations are ***TCompany::GetTotalClaims()***, which reads the total number of claims (required to search the claim list); and ***TCompany::GetClaimNo()***, which reads the claim number at the current position in the claim list, to determine if the current client is the specified claim.

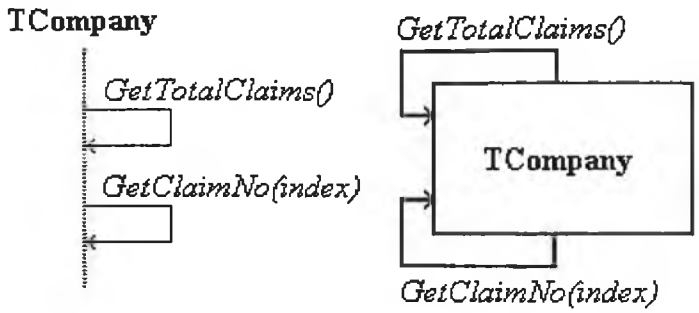


Fig 4.33 Interaction Diagram for *GetClaim()*

- **void TPolicy::AddRisk(char *RiskNo, char *RiskType, long RiskValue, int RiskStatus)**

The *AddRisk()* operation checks whether the total number of risks exceeds the maximum number of risks, and if not, it creates a new risk, stores this risk in the risk list of the policy, and increments the total number of risks. Hence the sub-ordinate operations are ***TPolicy::GetTotalRisks()***, which reads the total number of riskss; ***TRisk::TRisk()***, which creates the new risk; ***TPolicy::PutRisk()***, which updates the risk list with the new risk; and ***TPolicy::IncTotalRisks()***, which increments the total number of risks.

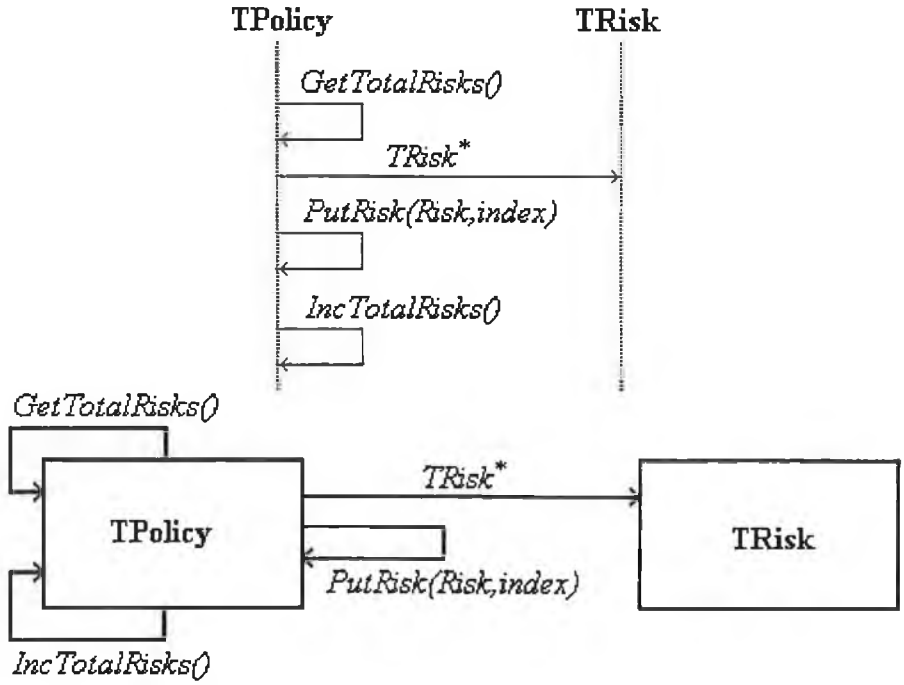


Fig 4.34 Interaction Diagram for *AddRisk()*

- **void TPolicy::DeleteRisk(char *RiskNo)**

The *DeleteRisk()* operation searches through the risk list of the policy until the specified risk is found, the risk is then deleted, the total number of risks on the policy is decremented, and the risk list is re-ordered. Hence the sub-ordinate operations are ***TPolicy::GetTotalRisks()***, which reads the total number of risks (required to search the risk list); ***TPolicy::GetRiskType()***, which reads the risk type at the current position in the risk list, to determine if the current risk is the specified risk; ***TRisk::~TRisk()***, which deletes the specified risk from the risk list of the policy; ***TPolicy::DecTotalRisks()***, which decrements the total number of risks on the policy; and ***TPolicy::PutRisk()***, which is called for each risk in the list after the deleted risk, thus re-ordering the list, and filling the empty space caused by the deletion.

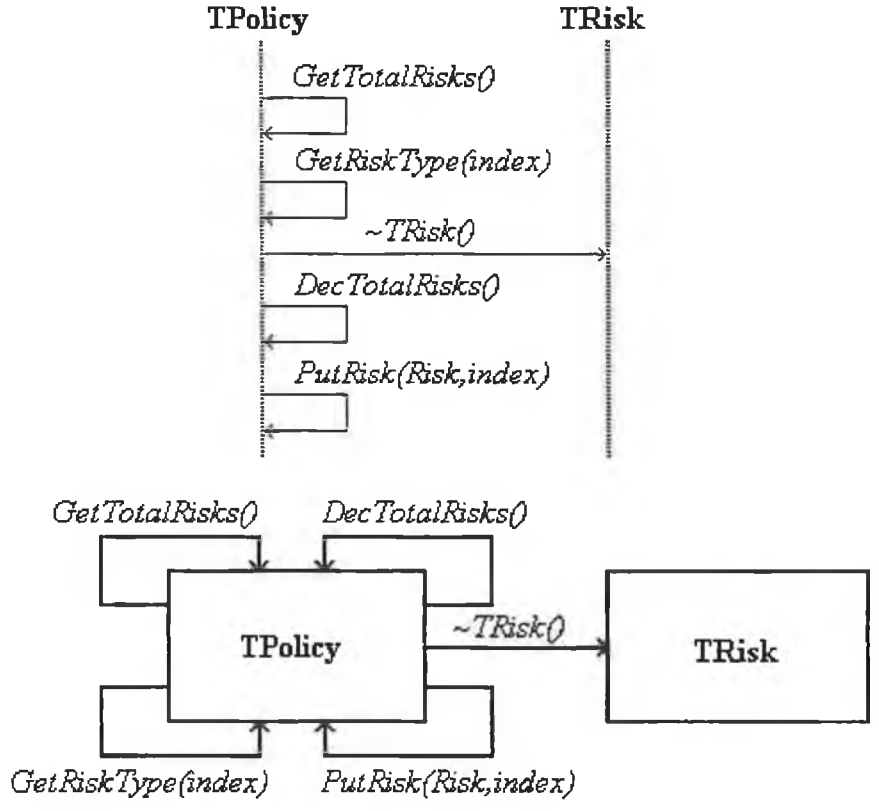


Fig 4.35 Interaction Diagram for *DeleteRisk()*

4.5 Ensuring Integration in the OMT Models

Integration in the OMT models can be ensured by adhering to the set of integration guidelines. These integration guidelines should be applied when constructing the three OMT models, because integration is a incremental process which should be practiced throughout the development phase of the models.

Hence once the object model has been constructed, the integration guidelines can be used when constructing the dynamic model, since the dynamic model will draw on information contained in the object model. Similiarly, the integration guidelines can be used when constructing the functional model, since the functional model will draw on information contained in both the object model and the dynamic model. Finally, the integration guidelines can be used in the iterative process of refining the models, where any new information introduced by the dynamic and functional models is integrated into the object model.

The next section provides an example of the integration process between the three OMT models, by refering to the points of integration between the object, dynamic and functional models which have previously been illustrated.

4.5.1 Integrating the Object and Dynamic Models

There are three important guidelines to note when integrating the object model with the dynamic model, which embody the common points of integration outlined in their inter-model definition.

- ***Each condition in the state diagram, should be defined only in terms of the attributes operations, and associations listed in the class diagram of the class, which is represented in the state diagram.***

For Example, in the state diagram for the *TClient* class, both of the conditions [PolicyCount > 1] and [PolicyCount = 1], are defined only in terms of the attributes of the *TClient* class, since *PolicyCount* is an attribute of this class.

In the state diagram for the *TPolicy* class, the conditions [TotalRisks > 1], [TotalRisks = 1], [DueDate >= DATE] and [DueDate < DATE] are defined only in terms of the attributes of the *TPolicy* class, since *TotalRisks* and *DueDate* are attributes of this class.

In the state diagram for the *TAgent* class, both of the conditions [PolicyCount > 1] and [PolicyCount = 1], are defined only in terms of the attributes of the *TAgent* class, since *PolicyCount* is an attribute of this class.

In the state diagram for the *TClaim* class, the conditions [PolicyStart <= ClaimDate && PolicyEnd >= ClaimDate && PolicyStatus = "Paid"] and [PolicyStart > ClaimDate || PolicyEnd < ClaimDate || PolicyStatus = "Unpaid"], are defined only in terms of the attributes of the *TClaim* class, since *PolicyStart*, *PolicyEnd*, *PolicyStatus* and *ClaimDate* are attributes of this class.

In the state diagram for the *TCompany* class, the conditions [TotalClients > 0 && TotalAgents > 0], [TotalClients > 0], [TotalAgents > 0], [TotalPolicies > 0], and [TotalClaims > 0] are defined only in terms of the attributes of the *TCompany* class, since *TotalClients*, *TotalAgents*, *TotalPolicies* and *TotalClaims* are attributes of this class.

- ***Each event in the state diagram should be mapped into an operation, and listed in the class diagram of the class, which is represented in the state diagram.***

For example, in the state diagram for the *TClient* class, the events *create_client*, *destroy_client*, *add_policy_to_client* and *delete_policy_from_client* are mapped into the *TClient constructor*, the *TClient destructor*, *AddPolicy()* and *DeletePolicy()* operations in the *TClient* class diagram.

In the state diagram for the *TPolicy* class, the events *create_policy*, *destroy_policy*, *add_risk*, *delete_risk* and *change_status* are mapped into the *TPolicy constructor*, the *TPolicy destructor*, *AddRisk()*, *DeleteRisk()* and *ChangeStatus()* operations in the *TPolicy* class diagram.

In the state diagram for the *TAgent* class, the events *create_agent*, *destroy_agent*, *add_policy_to_agent* and *delete_policy_from_agent* are mapped into the *TAgent constructor*, the *TAgent destructor*, *AddPolicy()* and *DeletePolicy()* operations in the *TAgent* class diagram.

In the state diagram for the *TClaim* class, the events *create_claim*, *destroy_claim*, and *change_status* are mapped into the *TClaim constructor*, the *TClaim destructor*, and *ChangeStatus()* operations in the *TClaim* class diagram.

In the state diagram for the *TCompany* class, the events *add_client*, *delete_client*, *update_client*, *find_client*, add_agent, delete_agent, update_agent, *find_agent*, *add_policy*, *delete_policy*, *update_policy*, *find_policy*, *add_claim*, *delete_claim*, *update_claim* and *find_claim* are mapped into the *AddClient()*, *DeleteClient()*, *UpdateClient()*, *GetClient()*, *AddAgent()*, *DeleteAgent()*, *UpdateAgent()*, GetAgent(), *AddPolicy()*, *DeletePolicy()*, *UpdatePolicy()*, *GetPolicy()*, *AddClaim()*, *Delete Claim()*, *UpdateClaim()* and *GetClaim()* operations in the *TCompany* class diagram.

- ***Each action in the state diagram should be mapped into an operation, and listed in the class diagram of the class, which is represented in the state diagram.***

The events and actions in the state diagrams of the *TClient, TPolicy*, *TAgent*, *TClaim* and *TCompany* classes, have very close relationships to each other, and are mapped into the same operations. Hence, in the state diagram for the *TClient* class, the actions AddPolicy() and DeletePolicy() are mapped into corresponding operations and listed in the *TClient* class diagram. Similarly, in the state diagram for the *TPolicy* class, the actions AddRisk(), DeleteRisk() and CalcStatus() are mapped into corresponding operations and listed in the *TPolicy* class diagram. In the state diagram for the *TAgent* class, the actions AddPolicy() and DeletePolicy() are mapped into corresponding operations and listed in the *TAgent* class diagram. In the state diagram for the *TClaim* class, the action CalcClaimStatus() is mapped into the corresponding operation and listed in the *TClaim* class diagram. And finally, in the state diagram for the *TCompany* class, the actions AddClient(), DeleteClient(), UpdateClient(), GetClient(), Add Agent(), DeleteAgent(), UpdateAgent(), GetAgent(), AddPolicy(), DeletePolicy(), UpdatePolicy(), GetPolicy(), AddClaim(), DeleteClaim(), UpdateClaim() and GetClaim() are mapped into corresponding operations and listed in the *TCompany* class diagram.

4.5.2 Integrating the Dynamic and Functional Models

There are two important guidelines to note when integrating the dynamic model with the functional model, which embody the common points of integration as outlined in their inter-model definition.

- ***Each external event and complex internal event, in the event flow diagram, should be mapped into a system operation and illustrated as an operation schema in the operation model.***

- ***Each external event and complex internal event, in the event flow diagram, should be mapped into a system operation and illustrated as an interaction diagram in the interaction model.***

For example, in the event flow diagram there are 18 external events :

add_client which is mapped into the system operation TCompany::AddClient()
delete_client which is mapped into the system operation TCompany::DeleteClient()
update_client which is mapped into the system operation TCompany::UpdateClient()
find_client which is mapped into the system operation TCompany::GetClient()
add_agent which is mapped into the system operation TCompany::AddAgent()
delete_agent which is mapped into the system operation TCompany::DeleteAgent()
update_agent which is mapped into the system operation TCompany::UpdateAgent()
find_agent which is mapped into the system operation TCompany::GetAgent()
add_policy which is mapped into the system operation TCompany::AddPolicy()
delete_policy which is mapped into the system operation TCompany::DeletePolicy()
update_policy which is mapped into the system operation TCompany::UpdatePolicy()
find_policy which is mapped into the system operation TCompany::GetPolicy()
add_claim which is mapped into the system operation TCompany::AddClaim()
delete_claim which is mapped into the system operation TCompany::DeleteClaim()
update_claim which is mapped into the system operation TCompany::UpdateClaim()
find_claim which is mapped into the system operation TCompany::GetClaim()
add_risk which is mapped into the system operation TPolicy::AddRisk()
delete_risk which is mapped into the system operation TPolicy::DeleteRisk()

Each of these system operations is illustrated as an operation schema in the operation model and illustrated as an interaction diagram in the interaction model.

4.5.3 Integrating the Object and Functional Models

There are three important guidelines to note when integrating the object model with the functional model, which embody the common points of integration as outlined in their inter-model definition.

- ***Each system operation in the operation model, should be defined only in terms of the attributes and associations listed in the class diagram of the particular class, to which it belongs.***

For example, the TCompany::AddClient(), TCompany::DeleteClient(), TCompany:: UpdateClient(), and TCompany::GetClient() system operations are defined only in terms of the attributes of the *TCompany* class since *Client*, *ClientList* and *TotalClients* are attributes of this class.

Similarily, the TCompany::AddPolicy(), TCompany::DeletePolicy(), TCompany:: UpdatePolicy(), and TCompany::GetPolicy() system operations are defined only in terms of the attributes of the *TCompany* class since *Policy*, *PolicyList* and *TotalPolicies* are attributes of this class.

Similarily, the TCompany::AddAgent(), TCompany::DeleteAgent(), TCompany:: UpdateAgent(), and TCompany::GetAgent() system operations are defined only in terms of the attributes of the *TCompany* class since *Agent*, *AgentList* and *TotalAgents* are attributes of this class.

Similarily, the TCompany::AddClaim(), TCompany::DeleteClaim(), TCompany:: UpdateClaim(), and TCompany::GetClaim() system operations are defined only in terms of the attributes of the *TCompany* class since *Claim*, *ClaimList* and *TotalClaims* are attributes of this class.

Similarily, the TPolicy::AddRisk() and TPolicy::DeleteRisk() system operations are defined only in terms of the attributes of the *TPolicy* class since *Risk*, *RiskList* and *TotalRisks* are attributes of this class.

- ***Each system operation represented by an operation schema in the operation model, should mapped into an operation, and listed in the class diagram of the particular class, to which it belongs.***

Each of the following system operations belong to the *TCompany* class, and hence, should be listed in the class diagram of the *TCompany* :

TCompany::AddClient();
TCompany::DeleteClient();
TCompany::UpdateClient();
TCompany::GetClient();
TCompany::AddAgent();
TCompany::DeleteAgent();
TCompany::UpdateAgent();
TCompany::GetAgent();
TCompany::AddPolicy();
TCompany::DeletePolicy();
TCompany::UpdatePolicy();
TCompany::GetPolicy();
TCompany::AddClaim();
TCompany::DeleteClaim();
TCompany::UpdateClaim();
TCompany::GetClaim();

Each of the following system operations belong to the *TPolicy* class, and hence, should be listed in the class diagram of the *TPolicy* :

TPolicy::AddRisk();
TPolicy::DeleteRisk();

- ***Each individual object interaction, within each system operation represented by an interaction diagram in the interaction model, should be mapped into an operation, and listed in the class diagram of the particular class, to which it belongs.***

For example, the AddClient() system operation has 4 individual interactions

Individual Interaction	*Class*	*Operation*
GetTotalClients()	*TCompany*	*TCompany::GetTotalClients()*
TClient()	*TClient*	*TClient::TClient()*
PutClient(Client,index)	*TCompany*	*TCompany::PutClient(Client,index)*
IncTotalClients()	*TCompany*	*TCompany::IncTotalClients()*

The DeleteClient() system operation has 5 individual interactions

Individual Interaction	*Class*	*Operation*
GetTotalClients()	*TCompany*	*TCompany::GetTotalClients()*
GetClientNo(index)	*TCompany*	*TCompany::GetClientNo(index)*
~TClient()	*TClient*	*TClient::~TClient()*
DecTotalClients()	*TCompany*	*TCompany::DecTotalClients()*
PutClient(Client,index)	*TCompany*	*TCompany::PutClient(Client,index)*

The UpdateClient() system operation has 8 individual interactions

Individual Interaction	*Class*	*Operation*
GetClient(ClientNo)	*TCompany*	*TCompany::GetClient(ClientNo)*
PutSurname(Surname)	*TClient*	*TClient::PutSurname(Surname)*
PutFirstname(Firstname)	*TClient*	*TClient::PutFirstname(Firstname)*
PutAddress(Address)	*TClient*	*TClient::PutAddress(Address)*
PutPhone(Phone)	*TClient*	*TClient::PutPhone(Phone)*
PutOccupation(Occup)	*TClient*	*TClient::PutOccupation(Occup)*
PutBirthdate(Birthdate)	*TClient*	*TClient::PutBirthdate(Birthdate)*
PutSex(Sex)	*TClient*	*TClient::PutSex(Sex)*

The GetClient() system operation has 2 individual interactions

Individual Interaction	*Class*	*Operation*
GetTotalClients()	*TCompany*	*TCompany::GetTotalClients()*
GetClientNo(index)	*TCompany*	*TCompany::GetClientNo(index)*

[Note : The individual object interactions for each of the system operations relating to *Clients* (as detailed above), are very similar to the individual object interactions for each of the other system operations relating to *Agents*, *Policies*, *Claims* and *Risks*, and thus since they can be easily deduced from the above example, they are not illustrated in detail.]

4.6 Ensuring Consistency in the OMT Models

Consistency in the OMT models can be ensured by adhering to the set of consistency guidelines. Since no consistency check fails for this particular case study example (proved by the previous section), the three OMT models are deemed to be consistent. If any one consistency check, or a number of consistency checks had failed, then an anomaly would exist in one or more of the OMT models, and such an anomaly would need to be redressed before consistency could be achieved.

- Check that each condition in each state diagram of the dynamic model, is defined only in terms of the attributes, operations and associations listed in the class diagram of the class, which is represented in the state diagram.

- Check that each event in each state diagram of the dynamic model, is mapped into an operation, and listed in the class diagram of the class, which is represented in the state diagram.

- Check that each action in each state diagram of the dynamic model, is mapped into an operation, and listed in the class diagram of the class, which is represented in the state diagram.

- Check that each external event and complex internal event, in the event flow diagram of the dynamic model, is mapped into a system operation and illustrated as an operation schema in the operation model of the functional model.

- Check that each external event and complex internal event, in the event flow diagram of the dynamic model, is mapped into a system operation and illustrated as an interaction diagram in the interaction model of the functional model.

- Check that each system operation in the operation model, is defined only in terms of the attributes and associations listed in the class diagram of the particular class, to which it belongs.

- Check that each system operation represented by an operation schema in the operation model, is mapped into an operation, and listed in the class diagram of the particular class, to which it belongs

- Check that each individual object interaction, within each system operation represented by an interaction diagram in the interaction model, is mapped into an operation, and listed in the class diagram of the particular class, to which it belongs.

4.7 Chapter Summary

This chapter documented and illustrated a comprehensive case study which involved constructing the object, dynamic and functional models for an insurance company which dealt with clients, agents, policies, and claims. Furthermore, this chapter also documented and illustrated the improved integration and consistency that can be achieved across the OMT models by adhering to the set of integration guidelines and the set of consistency guidelines.

Note : The full class listings of this case study are available in the appendix.

Chapter 5

Conclusions

5.1 Overview

OMT by Rumbaugh et al. is a methodology for the analysis and design of object-oriented systems. It comprises three individual models : an *object model* which specifies the structural aspects of a system; a *dynamic model* which specifies the behavioural aspects of a system; and a *functional model* which specifies the transformational aspects of a system. These three distinct models are not completely independent, since each model describes one aspect of the system but contains references to the other two models, and hence all three OMT models need to be consistently integrated together in order to get the overall picture of the system.

The problem inherent in the OMT methodology is a lack of integration and consistency between the individual models. This problem has two significantly inter-connected sources : firstly, each of these diverse models is developed more or less independently, and the inter-relationships between the models are not well-defined, and are not supported by explicit steps in the methodology or comprehensive illustrated examples. Thus it is difficult to recognise how the separate models integrate together, and furthermore it is not easy to check the models for consistency with one another. Secondly, the functional model is ostensibly the weakest of the three models, primarily due to the unsuitability of using function-oriented DFDs to model the transformational aspects of an object-oriented system. This unsuitability results in tenuous links from the functional model to the other two models, and hence difficulties arise with the integration of the three OMT models, which further compounds the problem.

The aim of my research was to significantly improve the level of integration and consistency in the OMT methodology. To this end, it was necessary to address both the weakness of the functional model and the inadequacy of the inter-model relationships, since they were significantly inter-connected.

5.2 My Research in the OMT Methodology

The weakness of the functional model was identified as being directly caused by the incapability of function-oriented DFDs to adequately represent the transformational aspects of an object-oriented system, in terms of what the operations do and how the operations work. Firstly, DFDs cannot adequately describe what the operations do since they are organized around processes and not around objects, and hence the operations on the objects of the system are buried within the network of processes. Secondly, DFDs cannot adequately describe how the operations work, since they exclusively detail data flow, and interaction between the operations in an object-oriented system need not always involve an exchange of data, (in the same way that interaction between the processes in a function-oriented system nearly always involves an exchange of data). Furthermore, since the functional model is organized around processes, and the object and dynamic models are organized around objects, attempting to integrate the functional model with either of these two models is non-trivial. To remedy these problems, a proposed functional model was compiled which fully embraced the object-oriented paradigm, and which comprised two sub-ordinate models : an operation model which described what each system operation does; and an interaction model which described how each system operation works.

The inadequacy of the inter-model relationships was identified as being directly caused by these relationships being poorly defined, poorly supported, poorly reconciled and poorly illustrated. Firstly, the relationships were poorly defined since they consisted of a very informal definition of how the models should inter-relate, which was open to various interpretations, particularly the more tenuous relationships from the functional model to the other two models. Secondly, the relationships were poorly supported since there were no exphcit steps in the methodology detailing how to relate an object, attribute or operation in the object model to a data flow or process in the functional model, or to an event or state in the dynamic model. Thirdly, the relationships were poorly reconciled since there were no guidelines for checking the models for consistency with each other, and only minimal attention was given to possible inconsistencies and their reconciliation. Finally, the relationships were poorly illustrated since there was no fully documented example illustrating how to integrate and reconcile the OMT models. To remedy these problems a more rigorous set of proposed inter-model relationships was compiled, which was supported by a set of integration guidelines detailing how to integrate the three OMT models, and a set of consistency guidelines detailing how to reconcile the three OMT models, and these proposed inter-model relationships were illustrated using a comprehensive example.

5.2.1 Strengths of the Revised OMT Approach

Firstly, the transformation of Rumbaugh's functional model from the function-oriented DFD, to the proposed functional model, consisting of an operation model and an interaction model, results in a significant improvement in the ability of the functional model to fully achieve its purpose (of illustrating the transformational aspects of the system), since the operation model describes what each system operation does, in terms of its effect on the state of the system, and the interaction model describes how each system operation works, in terms of how it interacts with other operations in the system. In addition, this transformation also results in improved links between the functional model and the other two OMT models. These improvements are attributed to the fact that the proposed functional model fully embraces the object-oriented paradigm, and hence is more successful at specifying the functionality of an object-oriented system, and more successful at integrating an object-oriented functional model with the object-oriented object model and the object-oriented dynamic model. The proposed functional model achieved significant improvements over Rumbaugh's functional model, in each of the following areas :

- In terms of decomposition, Rumbaugh's functional model decomposes the system in terms of processes and sub processes, whereas the proposed functional model decomposes the system in terms of objects and operations. The former method of decomposition is suited to systems where the functions are more complex than the data they manipulate, which is not true of object-oriented systems, but the latter method of decomposition is suited to systems where the data is more complex than the functions, which is true of object-oriented systems, since the functions are often trivial, merely accessing and updating data attributes of objects.

- In terms of granularity, Rumbaugh's functional model adopts a top-down approach beginning with a context diagram and expanding this diagram level by level until atomic processes are achieved. However both the object and dynamic models are developed using a bottom-up class-by-class approach, hence Rumbaugh's functional model exists on a different level of granularity to the other two models, and this leads to problems with integrating the three models. Since the proposed functional model is developed using a bottom-up approach based on classes and operations, there exists a direct coherent link between the three models on the same level of granularity. As all three OMT models are now organized around classes and objects, this greatly improves the integration between the models.

- In terms of data access, Rumbaugh's functional model is organized around processes, and since the objects of the system are buried within the process network, it is not always possible to determine which data belongs to which objects, and hence the restricted data access of the object-oriented paradigm cannot always be illustrated. However in the proposed functional model, the operation model explicitly lists the data which is either read or updated by each system operation, and furthermore the interaction model lists the methods of each object that are invoked, in order to read or update the data of the object. In this way the restricted data access of the object-oriented paradigm can be illustrated.

- In terms of interaction, Rumbaugh's functional model shows communication within the system in terms of the flow of information between processes. Since communication is shown exclusively in terms of data flow, the interaction between objects cannot be completely illustrated, since the messaging mechanism which objects use to communicate with each other, need not always involve an exchange of data. However, in the proposed functional model, the interaction model documents the name and parameters (if any) of the messages sent between the various objects, hence the interaction of objects can be illustrated regardless of whether there is an exchange of data or not.

- In terms of mapping, Rumbaugh's functional model lacks the expressive power to adequately model an object-oriented system, hence if the objects cannot be easily modeled, then the transition to design and eventual implementation will be non-trivial. The proposed functional model, comprising the operation model, which describes what each system operation does, in terms of the data it reads and updates, as well as the pre-conditions and post-conditions on its execution; and the interaction model, which describes how each system operation works, in terms of the sub-ordinate operations it invokes; provides a more detailed model of the object-oriented system, which should facilitate the design and eventual implementation of the system.

Secondly, the compilation of a proposed set of inter-model relationships results in a significant improvement in the overall level of integration and consistency in the OMT methodology. There are several reasons for this improvement :

- Rumbaugh's original relationships were poorly defined, very informal and open to various interpretations, particularly the more tenuous relationships from the functional model to the other two OMT models. However, the proposed inter-model relationships are more rigorously defined, and are fully illustrated in terms of a comprehensive example, documenting how the various OMT models relate to each other.

- Rumbaugh's original relationships were not supported by concrete steps in the methodology detailing how to integrate the three distinct OMT models, whereas the proposed inter-model relationships contain a set of integration guidelines, which are based on the definitions of the inter-model relationships, and document how to integrate the object, dynamic and functional models together.

- Rumbaugh's original relationships were not supported by concrete steps in the methodology detailing how to check the three distinct OMT models for consistency with each other, whereas the proposed inter-model relationships contain a set of consistency guidelines, which are based on the definitions of the inter-model relationships, and document how to reconcile the object, dynamic and functional models with each other.

- Furthermore, Rumbaugh's original relationships attempted to achieve a somewhat dubious level of integration and consistency, by hap-hazardly trying to integrate the models together through common operations existing in all three models, without paying any attention to possible inconsistencies and their reconciliation. However, the proposed inter-model relationships achieve a much higher level of integration and consistency by integrating the various models together through common attributes, common operations, common associations, and common events, relating to common classes existing in each of the three OMT models, and paying close attention to possible inconsistencies and their reconciliation.

5.2.2 Weaknesses of the Revised OMT Approach

Firstly, the weakness of the proposed functional model is caused by an initial dependency on the dynamic model. This dependency contradicts the notion that all three OMT models are independent orthogonal views of a system, which are subsequently integrated and reconciled. The purpose of my research was to improve the level of integration and consistency in the OMT methodology. Since both sources of this problem, namely the weak functional model and the inadequate inter-model relationships, were significantly inter-related, it was necessary to consider how the proposed functional model would integrate into Rumbaugh's existing object and dynamic models, in order to achieve consistency across the three OMT models. Hence, in trying to ensure integrated and consistent models, it is possible that the proposed functional model and the existing dynamic model are too tightly coupled. Thus, trade-off exists between the total independence of the proposed functional model, and a high level of integration and consistency within the OMT methodology.

In particular, the proposed functional model introduces the concept of a system operation, but the identification of the system operations draws on information already existing in the event flow diagram of the dynamic model. These system operations are top-level operations, which either correspond to interactions between the system and the outside world (identified by external events in the event flow diagram), or correspond to complex transformations of input values to output values (identified by complex internal events in the event flow diagram). Although this initial identification depends on the dynamic model, the proposed functional model enhances the overall OMT analysis model with new information, which is not provided by, or derivable from, any other model in the methodology.

Secondly, the weakness of the proposed inter-model relationships is caused by consistency being implemented as a set of heuristics, rather than as a very formal set of equations defined in terms of mathematics or some formal specification language. Although it is acceptable for the integration guidelines to be described as rules of thumb, the notion of consistency is a more formal concept than that of integration, and it implies the verification of more stringent rules and conditions than can be achieved merely by adhering to heuristics described in natural language, however unambiguous the definition of such heuristics may be. All the same, the proposed inter-model relationships significantly improve on Rumbaugh's original relationships, since they provide a set of consistency guidelines which were not included in Rumbaugh's original model.

5.3 Current Research in the OMT Methodology

Rumbaugh is currently actively involved in the development of a second generation OMT methodology, and has published some proposed alterations to the existing methodology in a set of articles in the Journal of Object-Oriented Programming [Rumbaugh, 95-1, 95-2, 95-3]. In these articles, Rumbaugh proposes minor enhancements to both the object and dynamic models, but addresses the vast majority of his modifications to the functional model.

There are a number of points of comparison and contrast between Rumbaugh's proposed alterations to the OMT methodology, and my proposed alterations detailed within the text of this thesis.

- Firstly, Rumbaugh has conceded that the functional model is weak, and that this weakness is caused by the unsuitability of using DFDs to model the transformational aspects of an object-oriented system. In addition, Rumbaugh proposes an outline for a transformed functional model, which is a major departure from the conventional use of DFDs.

- Secondly, Rumbaugh's transformed functional model comprises operation descriptions, object-oriented data flow diagrams, object interaction diagrams, pseudocode designs and actual method code. The operation descriptions are similar though not the same as my proposed operation model, and furthermore Rumbaugh intends only to describe the top-level operations which are invoked by interactions with external actors, and makes no reference to the complex interactions within the system which I deemed important enough to be included in the operation model. The object interaction diagrams are also similar to my proposed interaction model, since they also illustrate the sequence of messages that implement an operation, however Rumbaugh's diagrams are more detailed, extending into areas of control flow, which is not addressed in my proposed interaction model. Furthermore, my interaction diagrams are illustrated in tabular format (preferred) but also in graphical format, whereas Rumbaugh's object interaction diagrams are illustrated only in graphical format.

Although Rumbaugh's proposals are merely outlines and not full specifications, it is still possible to identify potential strengths and weaknesses to his approach. My opinion of these potential strengths and weaknesses is given below :

In terms of potential strengths...

- Rumbaugh's transformed functional model has dispensed with function-oriented DFDs, and has been organized around objects and operations, which should improve the ability of the functional model to specify the transformational aspects of a system, and should also simplify the process of integrating and reconciling the functional model to the object and dynamic models.

In terms of potential weaknesses...

- Firstly a question hangs over the impetus behind the transformation of the functional model, in terms of the political implications of the Rumbaugh-Booch merger of their respective methodologies. Since Booch's methodology includes an object and dynamic model but not a functional model, it may seem that Rumbaugh is trying to carve out a new role for his functional model as a link between analysis and design, instead of an independent view of the system. This point is supported by the inclusion of control flow information in the object interaction diagram, and the development of actual method code as part of his transformed functional model.

- Secondly, is it necessary to decompose the functional model into five separate models, and will the overhead of constructing these additional models clarify or confuse the functional model? There is not enough information currently available to address this issue.

- Thirdly, although Rumbaugh has dealt with the weakness of the functional model, he has yet to concentrate on improving the tenuous relationships from the functional model to the other two OMT models.

5.4 Future Research in the OMT Methodology

Two important areas of future research which would be of benefit to the OMT methodology, have been identified as : a need to address the area of formal consistency checking between the three OMT models; and a need to address the area of improved tool support for the OMT methodology. Each of these points is discussed below :

5.4.1 Formal Consistency Checking

As stated earlier in this chapter, the proposed inter-model relationships attempt to implement consistency between the various OMT models, as a set of guidelines, instead of the more favourable option of implementing consistency as a set of formally verifiable equations. However, this inadequacy is not easily remedied, since a guarantee of consistency requires that it be possible to semantically check the OMT models fully, which is not possible without the use of a formal specification language, such as Vienna Development Method (VDM) or Z. Unfortunately, these techniques are only practical in safety critical systems where defects must be avoided at all costs. Hence, there is a need to explore ways of making consistency checking more formal than a set of heuristics, without the overhead of a full formal specification language.

5.4.2 Improved Tool Support

Currently OMT is supported by several CASE tools (e.g. OMTool, SelectOMT). Unfortunately, these tools are mainly limited to diagrammatical support and basic code generation. In addition, none of the CASE tools enforce inter-model integration, or support inter-model consistency, perhaps because Rumbaugh's inter-model relationships were not well-defined enough to enable cross-model integration and consistency to be implemented. This points to the need for improved OMT CASE tools, which provide more complete support for the development of consistently integrated models.

5.5 Chapter Summary

This chapter summarised the purpose of my research in terms of the problems with the OMT methodology, and my proposed solutions to these problems. In addition the strengths and weaknesses of my solutions were discussed, and comparisons were made between my research and Rumbaugh's current research into OMT, which led to the potential strengths and weaknesses of his approach also being discussed. Finally areas of future research which would be beneficial to the OMT methodology were also identified.

Bibliography

[Bear, '90] Stephen Bear, Phillip Allen, Derek Coleman, and Fiona Hayes, *"Graphical Specification of Object Oriented Systems"*, ECOOP/OOPSLA Proceedings '90, October 21-25 1990, pg 28-37.

[Blaha, '93] Michael Blaha, *"Aggregation of parts of parts of parts"*, Journal of Object-Oriented Programming, September 1993, Vol. 6 No. 5, pg 14-20.

[Booch, '86] Grady Booch, *"Object-Oriented Development"*, IEEE Transactions on Software Engineering, February 1986, Vol. SE-12 No. 2, pg 211-221.

[Booch, '94] Grady Booch, *Object-Oriented Analysis and Design with Applications (Second Edition)*, Benjamin/Cummings, Redwood City, CA, 1994.

[Coad, '90] Peter Coad, and Edward Yourdon, *"Object-Oriented Analysis"*, System and Software Requirements Engineering, IEEE Computer Society Press, 1990, pg 272-288.

[Coad, '91-1] Peter Coad and Edward Yourdon, *Object-Oriented Analysis*, Yourdon Press/Prentice Hall, Englewood Cliffs, NJ, 1991.

[Coad, '91-2] Peter Coad and Edward Yourdon, *Object-Oriented Design*, Yourdon Press/Prentice Hall, Englewood Cliffs, NJ, 1991.

[Coleman '94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes, *Object-Oriented Development - The Fusion Method*, Prentice Hall, Englewood Cliffs, NJ, 1994.

[Cook, '94] Stephen Cook and John Daniels, *"Object Communication"*, Journal of Object-Oriented Programming, September 1994, Vol. 7 No. 5, pg 14-22.

[de Champeaux, '90] Dennis de Champeaux, Larry Constantine, Ivar Jacobson, Stephen Mellor, Paul Ward, and Edward Yourdon, *"PANEL : Structured Analysis and Object-Oriented Analysis"*, ECOOP/ OOPSLA '90 Proceedings, October 21-25 1990, pg 135-139.

[de Champeaux, '92] Dennis de Champeaux and Penelope Faure, *"A Comparative Study of Object-Oriented Analysis Methods"*, Journal of Object-Oriented Programming, March/April 1992, Vol. 5 No. 1, pg 21-33.

[de Champeaux, '93] Dennis de Champeaux, Douglas Lea, and Penelope Faure, *Object-Oriented Systems Development*, Addison Wesley, Reading, MA, 1993.

[Demarco, '79] Tom Demarco, *Structured Analysis and System Specification*, Chapters 4-10 : Data Flow Diagrams, pg 47-122, Yourdon Press/Prentice Hall, Englewood Cliffs, NJ, 1979.

[D'Souza, '93] Desmond D'Souza, *"Working with OMT"*, Journal of Object-Oriented Programming, October 1993, Vol. 6 No. 6, pg 63-68.

[D'Souza, '94-1] Desmond D'Souza, *"Working with OMT, Part 2"*, Journal of Object-Oriented Programming, February 1994, Vol. 6 No. 9, pg 68-72.

[D'Souza, '94-2] Desmond D'Souza, *"Working with OMT in the construction of large systems"*, Journal of Object-Oriented Programming, March/April 1994, Vol. 7 No. 1, pg 54-58.

[D'Souza, '95] Desmond D'Souza, *"Working with OMT : Model Integration"*, Journal of Object-Oriented Programming, February 1995, Vol. 7 No. 9, pg 22-29.

[Embley, '92] David W. Embley, B. Kurtz, and S. N. Woodfield, *Object-Oriented Systems Analysis*, Yourdon Press/ Prentice Hall, Englewood Cliffs, NJ, 1992.

[Eckert, '94] Gabriel Eckert and Paul Golder, *"Improving Object-Oriented Analysis"*, Information and Software Technology, 1994, Vol. 36 No. 2, pg 67-86.

[Frost, '95] Stuart Frost, *"The Select Perspective - Extending Rumbaugh's OMT for Client Server Systems Development"*, Select Software Tools, 1995, pg 1-13.

[Gane, '78] Chris Gane and Trish Sarson, *Structured Systems Analysis : Tools and Techniques*, Chapter 3 : Drawing Data Flow Diagrams, pg 25-47, Prentice Hall, Englewood Cliffs, NJ, 1978.

[Harel, '87] David Harel, *"Statecharts : a visual formalism for complex systems"*, Science of Computer Programming 8, 1987, pg 231-274.

[Harel, '88] David Harel, "On visual formalisms", Communications of the ACM, May 1988, Vol. 31 No. 5, pg 514-530.

[Iivari '95] Juhani Iivari, *"Object-orientation as structural, functional and behavioural modelling : a comparison of six methods for object- oriented analysis"*, Information and Software Technology, 1995, Vol 37 No. 3, pg 155-163.

[Jacobson, '94] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard, *Object-Oriented Software Engineering - A Use Case Driven Approach (Fourth Edition)*, Addison Wesley, Reading, MA, 1994.

[Jacobson, '95] Ivar Jacobson and Magnus Christenson, *"A growing consensus on use cases"*, Journal of Object-Oriented Programming, March/April 1995, Vol. 8 No. 1, pg 15-19.

[Martin, '92] James Martin and James J. Odell, *Object-Oriented Analysis and Design*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[Meyer, '88] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[Monarchi, '92] David E. Monarchi and Gretchen I. Puhr, *"A Research Typology for Object-Oriented Analysis and Design"*, Communications of the ACM, September 1992, Vol. 35 No. 9, pg 35-47.

[Nigro, '95] Libero Nigro, *"A real-time architecture based on Shlaer-Mellor object lifecycles"*, Journal of Object-Oriented Programming, March/April 1995, Vol. 8 No. 1, pg 20-31.

[Odell, '94] James J. Odell, *"Six different kinds of composition"*, Journal of Object-Oriented Programming, January 1994, Vol. 6 No. 8, pg 10-15.

[Rumbaugh, '91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Rumbaugh, '94] James Rumbaugh, *"Getting Started - Using use cases to capture requirements"*, Journal of Object-Oriented Programming, September 1994, Vol. 7 No. 5, pg 8-23.

[Rumbaugh, '95-1] James Rumbaugh, *"OMT : The Object Model"*, Journal of Object-Oriented Programming, January 1995, Vol. 7 No. 8, pg 21-27.

[Rumbaugh, '95-2] James Rumbaugh, *"OMT : The Dynamic Model"*, Journal of Object-Oriented Programming, February 1995, Vol. 7 No. 9, pg 7-12.

[Rumbaugh, '95-3] James Rumbaugh, *"OMT : The Functional Model"*, Journal of Object-Oriented Programming, March/April 1995, Vol. 8 No. 1, pg 10-14.

[Shlaer, '88] Sally Shlaer and Stephen J. Mellor, *Object-Oriented Systems Analysis : Modeling the World in Data*, Yourdon Press/ Prentice Hall, Englewood Cliffs, NJ, 1988.

[Shlaer, '92] Sally Shlaer and Stephen J. Mellor, *Object Lifecycles : Modeling the World in States*, Yourdon Press/Prentice Hall, Englewood Cliffs, NJ, 1992.

[Tanzer, '95] Christian Tanzer, *"Remarks on object-oriented modeling of associations"*, Journal of Object-Oriented Programming, February 1995, Vol.7 No. 9, pg 43-46.

[Wirfs-Brock, '89] Rebecca Wirfs-Brock and Brian Wilkerson, *"Object-Oriented Design : A Responsibility-Driven Approach"*, OOPSLA '89 Proceedings, October 1-6 1989, pg 71-75.

[Wirfs-Brock, '90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Weiner, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[Yourdon, '89] Yourdon, Edward, *Modern Structured Analysis*, Chapter 9 : Data Flow Diagrams, pg 139-175, Chapter 13 : State-Transition Diagrams, pg 259-274. Prentice Hall, Englewood Cliffs, NJ, 1989.

Glossary

Action	*an instantaneous operation, associated with an event, in the state diagram of the dynamic model.*
Activity	*a time-consuming operation, associated with a state, in the state diagram of the dynamic model.*
Aggregation	*a special form of association, representing the a-part-of relationship between objects in the object model.*
Association	*a relationship between two or more classes in the object model.*
Atomic Operation	*an operation which cannot be further decomposed into sub-ordinate operations.*
Attribute	*a named property belonging a class, describing a data value held by each object of the class.*
Class	*a schema of attributes and operations, documenting a group of objects with similar properties, behaviour and relationships.*
Condition	*a boolean function, describing a guard on a transition, in the state diagram of the dynamic model.*
Data Flow	*a connection between the output of one process, data store or actor and the input to another process, data store or actor, in the data flow diagram of the functional model.*
Data Store	*a temporary repository of data in the data flow diagram of the functional model.*

Data Flow Diagram	*a graphical representation of the functional model, illustrating the flow of data values through a system from their sources in actors, via transformations in processes, to their destinations in other actors.*
Dynamic Model	*a description of the temporal, behavioural aspects of a system.*
Event	*an instantaneous occurrence at a point in time.*
Event Flow Diagram	*a diagram that illustrates the sender and receiver of events, without regard for the sequence of events, in the dynamic model.*
Event Trace Diagram	*a diagram that illustrates the sender an receiver of events, and the sequence of events in the dynamic model.*
Functional Model	*a description of the transformational aspects of a system.*
Generalization	*a special form of association, representing the a-kind-of relationship between objects in the object model.*
Link	*an instance of an association.*
Object	*an instance of a class.*
Object Diagram	*a graphical representation of the object model, illustrating attributes, operations and associations.*
Object Model	*a description of the static structural aspects of a system.*
Operation	*a named function belonging to a class, describing a transformation that may be applied to objects of the class.*

Process	*a transformation of incoming data flows into outgoing data flows, in the data flow diagram of the functional model.*
Scenario	*a sequence of events which occurs during one particular execution of a system.*
State	*the values of the attributes and links of an object at a particular time.*
State Diagram	*a directed graph where nodes represent states and whose arcs represent transitions between states caused by events.*
System Operation	*a top level operation, which either corresponds to an interaction between the system and the outside world, or to a complex transformation of input values to output values (analogous to a Level 1 process on a data flow diagram).*

Appendix

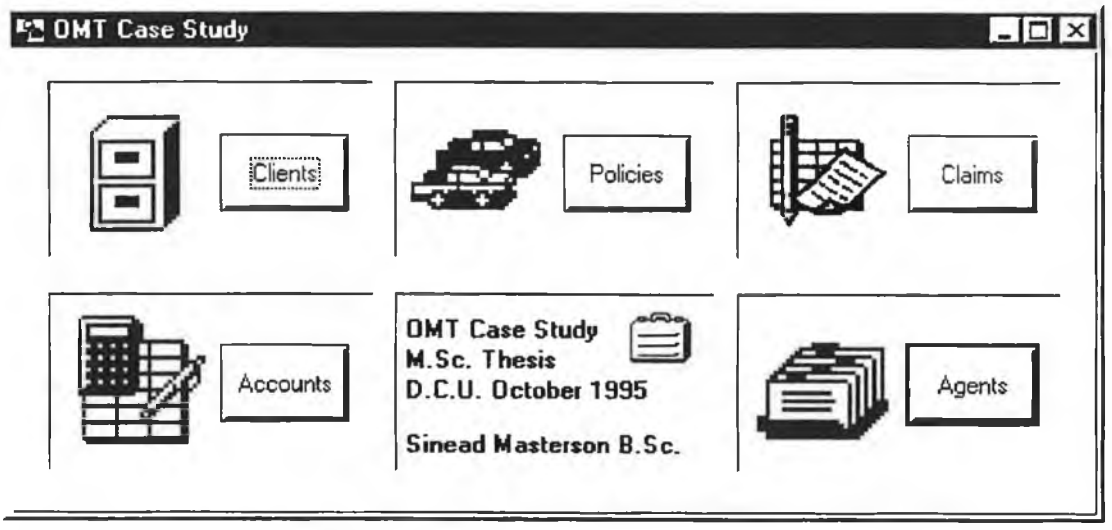


Fig A.1 Main Screen

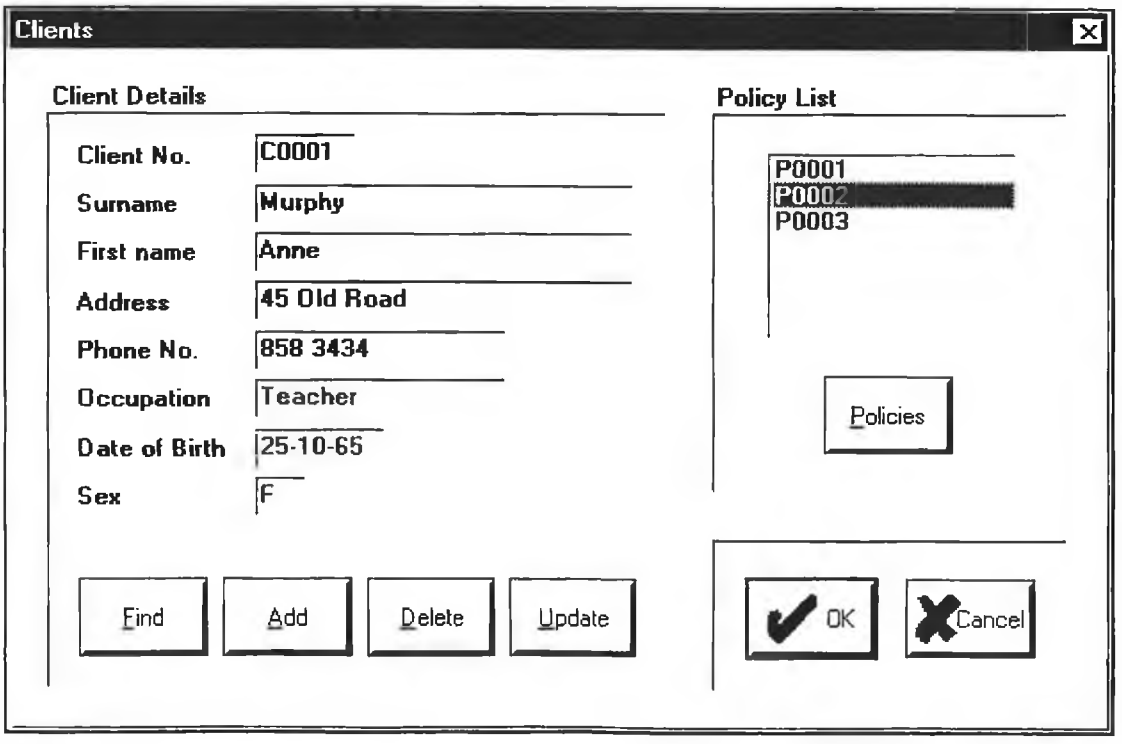


Fig A.2 Clients Screen

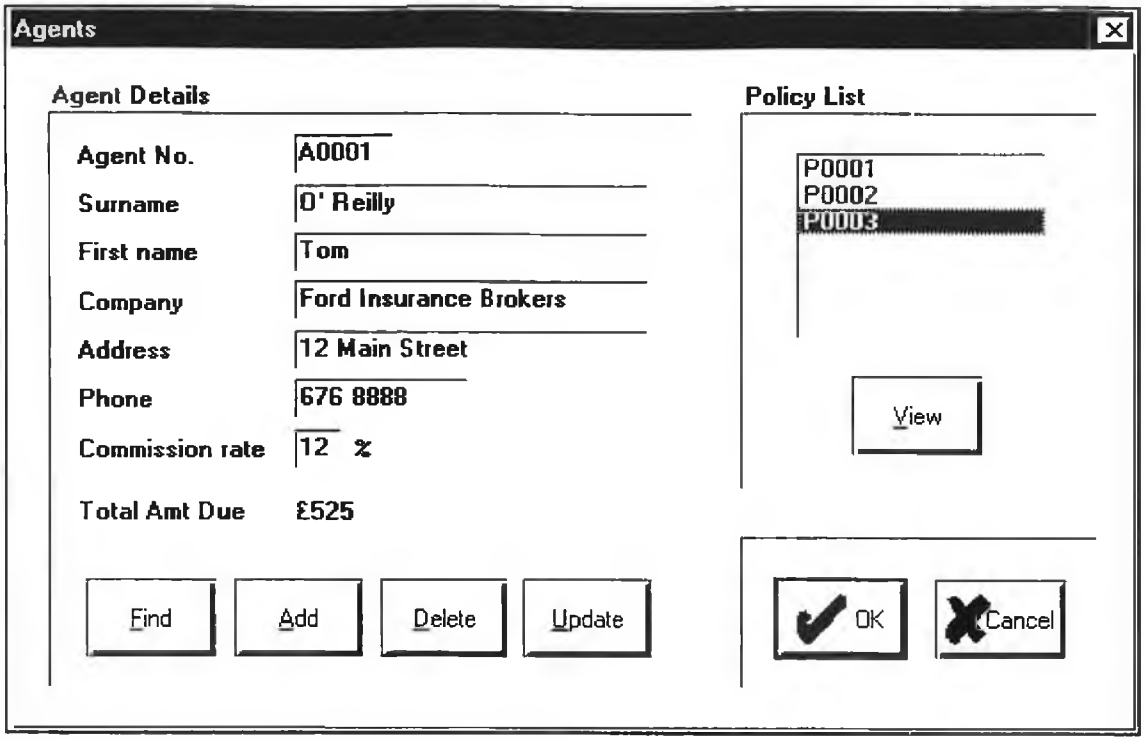


Fig A.3 Agents Screen

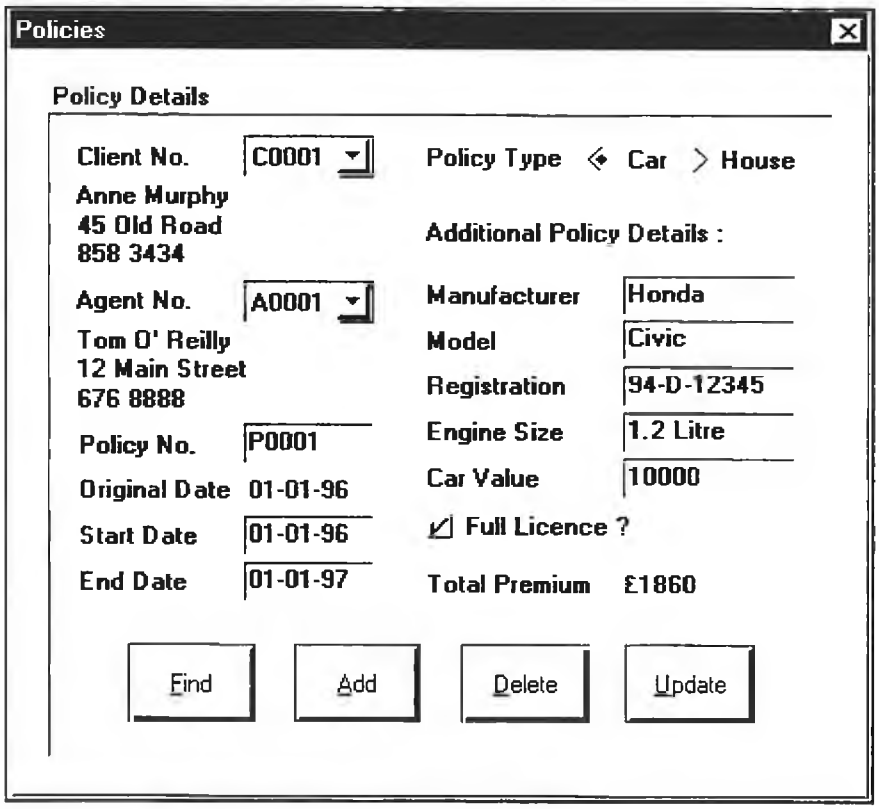


Fig A.4 Policies Screen

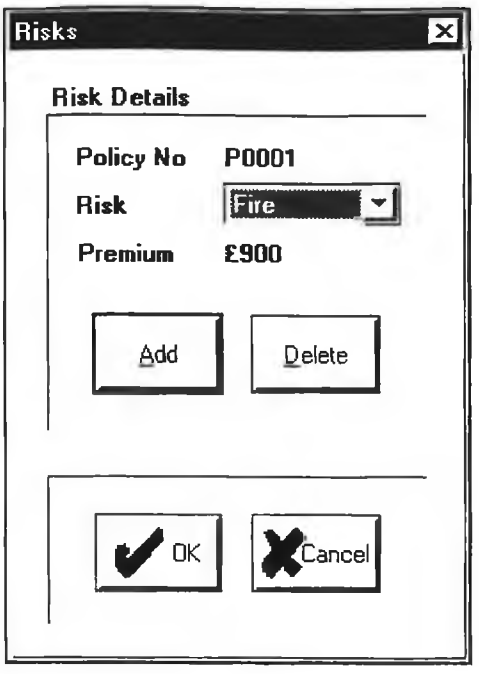


Fig A.5 Risks Screen

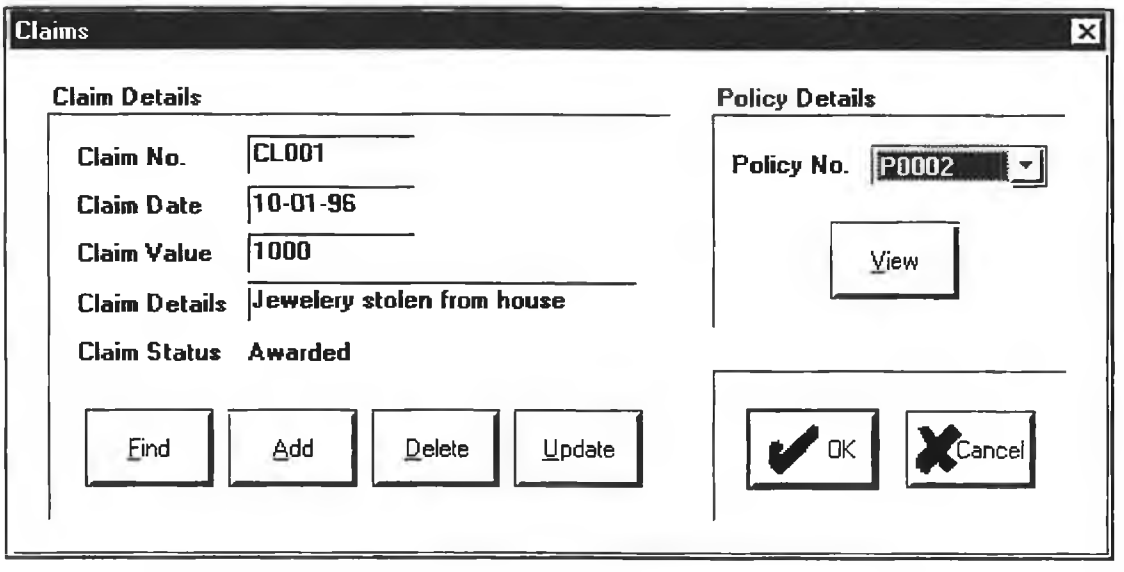


Fig A.6 Claims Screen

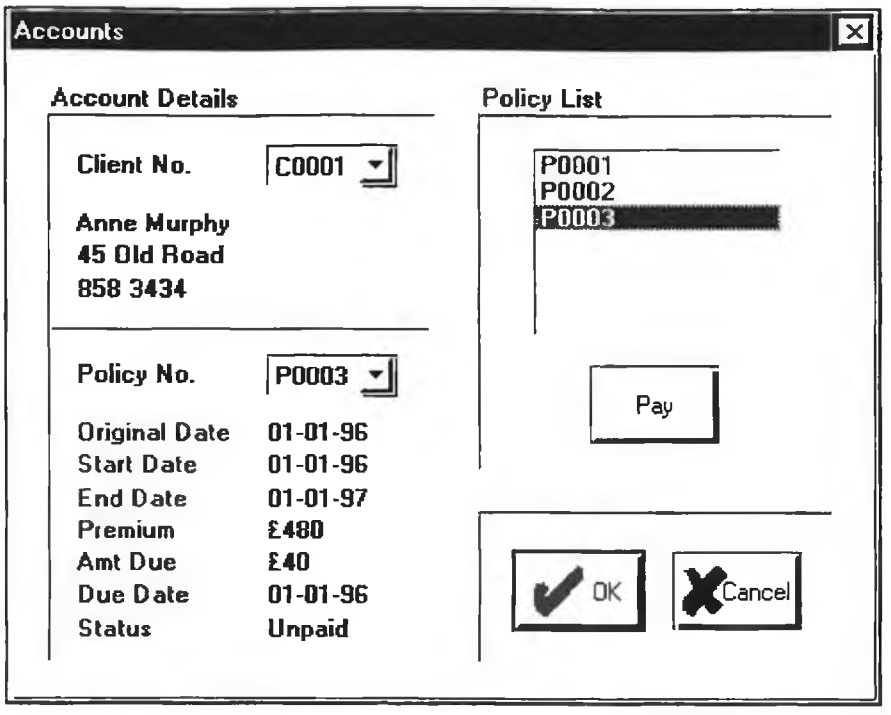


Fig A.7 Accounts Screen

```
// File : TCOMPANY.H

#ifndef __COMPANY_H
#define __COMPANY_H

#include "defines.h"
#include "tclient.h"
#include "tpolicy.h"
#include "tcar.h"
#include "thouse.h"
#include "tagent.h"
#include "tclaim.h"

class TCompany {
private:
  char CompanyName[COMPANY];
  char Address[ADDRESS];
  char Phone[PHONE];
  TClient *ClientList[MAX_CLIENTS];
  TPolicy *PolicyList[MAX_POLICIES];
  TAgent *AgentList[MAX_AGENTS];
  TClaim *ClaimList[MAX_CLAIMS];
  TClient *Client;
  TPolicy *Policy;
  TAgent *Agent;
  TClaim *Claim;
  TCar *CarPolicy;
  THouse *HousePolicy;
  int TotalClients;
  int TotalPolicies;
  int TotalAgents;
  int TotalClaims;

public:
  TCompany(char *NewCompanyName, char *NewAddress, char *NewPhone);
  ~TCompany();

  void AddClient(char *ClientNo, char *Surname, char *Firstname,
       char *Address, char *Phone, char *Occupation,
       char *Birthdate, char *Sex);

  void AddPolicy(char *ClientNo, char *PolicyNo, char *AgentNo,
       char *StartDate, char *EndDate, char *Manufacturer,
       char *Model, char *Registration, char *EngineSize,
       long CarValue, int FullLicenceStatus);

  void AddPolicy(char *ClientNo, char *PolicyNo, char *AgentNo,
       char *StartDate, char *EndDate, char *HouseType, int Rooms,
       char *AreaCode, long HouseValue, long ContentsValue,
       int HouseAlarmStatus);

  void AddAgent(char *AgentNo, char *Surname, char *Firstname,
       char *Company, char *Address, char *Phone, int CommRate);

  void AddClaim(char *ClaimNo, char *ClaimDate, long ClaimValue,
       char *ClaimDetails, char *PolicyNo, char *PolicyStart,
       char *PolicyEnd, char *PolicyStatus);
```

```
    void DeleteClient(char *ClientNo);
    void DeletePolicy(char *PolicyNo);
    void DeleteAgent(char *AgentNo);
    void DeleteClaim(char *ClaimNo);

    void UpdateClient(char *ClientNo, char *Surname, char *Firstname,
         char *Address, char *Phone, char *Occupation,
         char *Birthdate, char *Sex);

    void UpdatePolicy(char *PolicyNo, char *StartDate, char *EndDate,
         char *Manufacturer, char *Model, char *Registration,
         char *EngineSize, long CarValue, int FullLicenceStatus);

    void UpdatePolicy(char *PolicyNo, char *StartDate, char *EndDate,
         char *HouseType, int Rooms, char *AreaCode, long HouseValue,
         long ContentsValue, int HouseAlarmStatus);

    void UpdateAgent(char *AgentNo, char *Surname, char *Firstname,
         char *Company, char *Address, char *Phone, int CommRate);

    void UpdateClaim(char *ClaimNo, char *ClaimDate, long ClaimValue,
         char *ClaimDetails);

    TClient *GetClient(char *ClientNo);
    TPolicy *GetPolicy(char *PolicyNo);
    TAgent *GetAgent(char *AgentNo);
    TClaim *GetClaim(char *ClaimNo);

    void PutClient(TClient *Client, int index);
    void PutPolicy(TPolicy *Policy, int index);
    void PutAgent(TAgent *Agent, int index);
    void PutClaim(TClaim *Claim, int index);

    int GetTotalClients();
    int GetTotalPolicies();
    int GetTotalAgents();
    int GetTotalClaims();

    void IncTotalClients();
    void IncTotalPolicies();
    void IncTotalAgents();
    void IncTotalClaims();

    void DecTotalClients();
    void DecTotalPolicies();
    void DecTotalAgents();
    void DecTotalClaims();

    char *GetClientNo(int index);
    char *GetPolicyNo(int index);
    char *GetAgentNo(int index);
    char *GetClaimNo(int index);
    };

#endif
```

```
// File : TCOMPANY.CPP

#include <iostream.h>
#include <string.h>
#include "tcompany.h"

TCompany::TCompany(char *NewCompanyName, char *NewAddress,
                   char *NewPhone)
{
  strcpy(CompanyName,NewCompanyName);
  strcpy(Address,NewAddress);
  strcpy(Phone,NewPhone);
  TotalClients = 0;
  TotalPolicies = 0;
  TotalAgents = 0;
  TotalClaims = 0;
}

TCompany::~TCompany()
{
}

void TCompany::AddClient(char *ClientNo, char *Surname,
                         char *Firstname, char *Address,
                         char *Phone, char *Occupation,
                         char *Birthdate, char *Sex)
{
  if (GetTotalClients() < MAX_CLIENTS)
     {
     // Create the new client
     Client = new TClient(ClientNo,Surname,Firstname,Address,Phone,
                          Occupation,Birthdate,Sex);

     // Add the new client to the client list
     PutClient(Client,GetTotalClients());

     // Increment client counter
     IncTotalClients();
     }
}

void TCompany::AddPolicy(char *ClientNo, char *PolicyNo,
                         char *AgentNo, char *StartDate,
                         char *EndDate, char *Manufacturer,
                         char *Model, char *Registration,
                         char *EngineSize, long CarValue,
                         int FullLicenceStatus)
{
   if (GetTotalPolicies() < MAX_POLICIES)
      {
      // Create the new car policy
       Policy = (TPolicy *) new TCar
                (ClientNo,PolicyNo,AgentNo,StartDate,EndDate,
                 Manufacturer,Model,Registration,EngineSize,
                 CarValue,FullLicenceStatus);

      // Add the new policy to the policy list
      PutPolicy(Policy,GetTotalPolicies());
```

```
      // Increment policy counter
       IncTotalPolicies();
       }
}

void TCompany::AddPolicy(char *ClientNo, char *PolicyNo,
                         char *AgentNo, char *StartDate,
                         char *EndDate, char *HouseType,
                         int Rooms, char *AreaCode, long HouseValue,
                         long ContentsValue, int HouseAlarmStatus)
{
   if (GetTotalPolicies() < MAX_POLICIES)
       {
       // Create the new house policy
        Policy = (TPolicy *) new THouse
                 (ClientNo,PolicyNo,AgentNo,StartDate,EndDate,
                  HouseType,Rooms,AreaCode,HouseValue,
                  ContentsValue,HouseAlarmStatus);

       // Add the new policy to the policy list
       PutPolicy(Policy,GetTotalPolicies());
       IncTotalPolicies();
       }
}

void TCompany::AddAgent(char *AgentNo, char *Surname,
                        char *Firstname, char *Company,
                        char *Address, char *Phone,
                        int CommRate)
{
  if (GetTotalAgents() < MAX_AGENTS)
     {
     // Create the new agent
     Agent = new TAgent(AgentNo,Surname,Firstname,Company,Address,
                        Phone,CommRate);

     // Add the new agent to the agent list
     PutAgent(Agent,GetTotalAgents());

     // Increment agent counter
     IncTotalAgents();
     }
}

void TCompany::AddClaim(char *ClaimNo, char *ClaimDate,
                        long ClaimValue, char *ClaimDetails,
                        char *PolicyNo, char *PolicyStart,
                        char *PolicyEnd, char *PolicyStatus)
{
  if (GetTotalClaims() < MAX_CLAIMS)
     {
     // Create the new claim
     Claim = new TClaim(ClaimNo,ClaimDate,ClaimValue,ClaimDetails,
                        PolicyNo,PolicyStart,PolicyEnd,PolicyStatus);

     // Add the new claim to the claim list
     PutClaim(Claim,GetTotalClaims());
```

```
        // Increment claim counter
        IncTotalClaims();
        }
}

void TCompany::DeleteClient(char *ClientNo)
{
  // Search all clients until client selected for deletion is found
  for (int i=0; i<GetTotalClients(); i++)
      if (strcmp(GetClientNo(i),ClientNo)==0)
         {
         // Delete selected client from the client list
         delete (TClient *)ClientList[i];

         // Decrement client counter
         DecTotalClients();

         // Reorder the client list
         for (int j=i; j<GetTotalClients(); j++)
             PutClient(ClientList[j+1],j);
         }
}

void TCompany::DeletePolicy(char *PolicyNo)
{
  // Search all policies until policy selected for deletion is found
  for (int i=0; i<GetTotalPolicies(); i++)
      if (strcmp(GetPolicyNo(i),PolicyNo)==0)
         {
         // Delete selected policy from the policy list
         delete (TPolicy *)PolicyList[i];

         // Decrement policy counter
         DecTotalPolicies();

         // Reorder the policy list
         for (int j=i; j<GetTotalPolicies(); j++)
             PutPolicy(PolicyList[j+1],j);
         }
}

void TCompany::DeleteAgent(char *AgentNo)
{
  // Search all agents until agent selected for deletion is found
  for (int i=0; i<GetTotalAgents(); i++)
      if (strcmp(GetAgentNo(i),AgentNo)==0)
         {
         // Delete selected agent from the agent list
         delete (TAgent *)AgentList[i];

         // Decrement agent counter
         DecTotalAgents();

         // Reorder the agent list
         for (int j=i; j<GetTotalAgents(); j++)
             PutAgent(AgentList[j+1],j);
         }
}
```

```
void TCompany::DeleteClaim(char *ClaimNo)
{
  // Search all claims until claim selected for deletion is found
  for (int i=0; i<GetTotalClaims(); i++)
      if (strcmp(GetClaimNo(i),ClaimNo)==0)
         {
         // Delete selected claim from the claim list
         delete (TClaim *)ClaimList[i];

         // Decrement claim counter
         DecTotalClaims();

         // Reorder the claim list
         for (int j=i; j<GetTotalClaims(); j++)
             PutClaim(ClaimList[j+1],j);
         }
}

void TCompany::UpdateClient(char *ClientNo, char *Surname,
                            char *Firstname, char *Address,
                            char *Phone, char *Occupation,
                            char *Birthdate, char *Sex)
{
  // Retrieve specified client
  Client = GetClient(ClientNo);

  // Update client details
  Client->PutSurname(Surname);
  Client->PutFirstname(Firstname);
  Client->PutAddress(Address);
  Client->PutPhone(Phone);
  Client->PutOccupation(Occupation);
  Client->PutBirthdate(Birthdate);
  Client->PutSex(Sex);
}

void TCompany::UpdatePolicy(char *PolicyNo, char *StartDate,
                            char *EndDate, char *Manufacturer,
                            char *Model, char *Registration,
                            char *EngineSize, long CarValue,
                            int FullLicenceStatus)
{
  // Retrieve specified car insurance policy
  CarPolicy = (TCar *)GetPolicy(PolicyNo);

  // Update car insurance policy details
  CarPolicy->PutStartDate(StartDate);
  CarPolicy->PutEndDate(EndDate);
  CarPolicy->PutManufacturer(Manufacturer);
  CarPolicy->PutModel(Model);
  CarPolicy->PutRegistration(Registration);
  CarPolicy->PutEngineSize(EngineSize);
  CarPolicy->PutCarValue(CarValue);
  CarPolicy->PutFullLicenceStatus(FullLicenceStatus);
}
```

```
void TCompany::UpdatePolicy(char *PolicyNo, char *StartDate,
                            char *EndDate, char *HouseType,
                            int Rooms, char *AreaCode,
                            long HouseValue, long ContentsValue,
                            int HouseAlarmStatus)
{
  // Retrieve specified house insurance policy
  HousePolicy = (THouse *)GetPolicy(PolicyNo);

  // Update house insurance policy details
  HousePolicy->PutStartDate(StartDate);
  HousePolicy->PutEndDate(EndDate);
  HousePolicy->PutHouseType(HouseType);
  HousePolicy->PutRooms(Rooms);
  HousePolicy->PutAreaCode(AreaCode);
  HousePolicy->PutHouseValue(HouseValue);
  HousePolicy->PutContentsValue(ContentsValue);
  HousePolicy->PutHouseAlarmStatus(HouseAlarmStatus);
}

void TCompany::UpdateAgent(char *AgentNo, char *Surname,
                           char *Firstname, char *Company,
                           char *Address, char *Phone,
                           int CommRate)
{
  // Retrieve specified agent
  Agent = GetAgent(AgentNo);

  // Update agent details
  Agent->PutSurname(Surname);
  Agent->PutFirstname(Firstname);
  Agent->PutCompany(Company);
  Agent->PutAddress(Address);
  Agent->PutPhone(Phone);
  Agent->PutCommRate(CommRate);
}

void TCompany::UpdateClaim(char *ClaimNo, char *ClaimDate,
                           long ClaimValue, char *ClaimDetails)
{
  // Retrieve specified claim
  Claim = GetClaim(ClaimNo);

  // Update claim details
  Claim->PutClaimDate(ClaimDate);
  Claim->PutClaimValue(ClaimValue);
  Claim->PutClaimDetails(ClaimDetails);
}

TClient *TCompany::GetClient(char *ClientNo)
{
  // Search all clients
  for (int i=0; i<GetTotalClients(); i++)
      if (strcmp(GetClientNo(i),ClientNo)==0)
        // Client is found
        return ClientList[i];
```

```
  // Client is not found
  return (TClient *)NULL;
}

TPolicy *TCompany::GetPolicy(char *PolicyNo)
{
  // Search all policies
  for (int i=0; i<GetTotalPolicies(); i++)
      if (strcmp(GetPolicyNo(i),PolicyNo)==0)
        // Policy is found
        return PolicyList[i];

  // Policy is not found
  return (TPolicy *)NULL;
}

TAgent *TCompany::GetAgent(char *AgentNo)
{
  // Search all agents
  for (int i=0; i<GetTotalAgents(); i++)
      if (strcmp(GetAgentNo(i),AgentNo)==0)
        // Agent is found
        return AgentList[i];

  // Agent is not found
  return (TAgent *)NULL;
}

TClaim *TCompany::GetClaim(char *ClaimNo)
{
  // Search all claims
  for (int i=0; i<GetTotalClaims(); i++)
      if (strcmp(GetClaimNo(i),ClaimNo)==0)
        // Claim is found
        return ClaimList[i];

  // Claim is not found
  return (TClaim *)NULL;
}

void TCompany::PutClient(TClient *Client, int index)
{
  ClientList[index] = Client;
}

void TCompany::PutPolicy(TPolicy *Policy, int index)
{
  PolicyList[index] = Policy;
}

void TCompany::PutAgent(TAgent *Agent, int index)
{
  AgentList[index] = Agent;
}
```

```
void TCompany::PutClaim(TClaim *Claim, int index)
{
  ClaimList[index] = Claim;
}

int TCompany::GetTotalClients()
{
  return TotalClients;
}

int TCompany::GetTotalPolicies()
{
  return TotalPolicies;
}

int TCompany::GetTotalAgents()
{
  return TotalAgents;
}

int TCompany::GetTotalClaims()
{
  return TotalClaims;
}

void TCompany::IncTotalClients()
{
  TotalClients ++;
}

void TCompany::IncTotalPolicies()
{
  TotalPolicies ++;
}

void TCompany::IncTotalAgents()
{
  TotalAgents ++;
}

void TCompany::IncTotalClaims()
{
  TotalClaims ++;
}

void TCompany::DecTotalClients()
{
  TotalClients --;
}

void TCompany::DecTotalPolicies()
{
  TotalPolicies --;
}
```

```
void TCompany::DecTotalAgents()
{
  TotalAgents --;
}

void TCompany::DecTotalClaims()
{
  TotalClaims --;
}

char *TCompany::GetClientNo(int index)
{
  return ClientList[index]->GetClientNo();
}

char *TCompany::GetPolicyNo(int index)
{
  return PolicyList[index]->GetPolicyNo();
}

char *TCompany::GetAgentNo(int index)
{
  return AgentList[index]->GetAgentNo();
}

char *TCompany::GetClaimNo(int index)
{
  return ClaimList[index]->GetClaimNo();
}
```

```
// File : TCLIENT.H

#ifndef __CLIENT_H
#define __CLIENT_H

#include "defines.h"

class TClient {
private:
  char ClientNo[CLIENT];
  char Surname[SURNAME];
  char Firstname[FIRSTNAME];
  char Address[ADDRESS];
  char Phone[PHONE];
  char Occupation[OCCUPATION];
  char Birthdate[DATE];
  char Sex[SEX];
  int PolicyCount;
  char *PolicyNoList[MAX_POLICIES_PER_CLIENT];

public:
  TClient(char *NewClientNo, char *NewSurname, char *NewFirstname,
          char *NewAddress, char *NewPhone, char *NewOccupation,
          char *NewBirthdate, char *NewSex);
  ~TClient();
  void AddPolicy(char *PolicyNo);
  void DeletePolicy(char *PolicyNo);
  char *GetClientNo();
  char *GetFirstname();
  char *GetSurname();
  char *GetAddress();
  char *GetPhone();
  char *GetOccupation();
  char *GetBirthdate();
  char *GetSex();
  int GetPolicyCount();
  char *GetPolicyNo(int index);
  void PutSurname(char *NewSurname);
  void PutFirstname(char *NewFirstname);
  void PutAddress(char *NewAddress);
  void PutPhone(char *NewPhone);
  void PutOccupation(char *NewOccupation);
  void PutBirthdate(char *NewBirthdate);
  void PutSex(char *NewSex);
  void IncPolicyCount();
  void DecPolicyCount();
  void PutPolicyNo(char *PolicyNo, int index);
  };

#endif __CLIENT_H
```

```
// File : TCLIENT.CPP

#include <iostream.h>
#include <string.h>
#include "tclient.h"

TClient::TClient(char *NewClientNo, char *NewSurname,
                 char *NewFirstname, char *NewAddress,
                 char *NewPhone, char *NewOccupation,
                 char *NewBirthdate, char *NewSex)
{
  strcpy(ClientNo,NewClientNo);
  strcpy(Surname,NewSurname);
  strcpy(Firstname,NewFirstname);
  strcpy(Address,NewAddress);
  strcpy(Phone,NewPhone);
  strcpy(Occupation,NewOccupation);
  strcpy(Birthdate,NewBirthdate);
  strcpy(Sex,NewSex);
  PolicyCount = 0;
}

TClient::~TClient()
{
}

void TClient::AddPolicy(char *PolicyNo)
{
  if (GetPolicyCount() < MAX_POLICIES_PER_CLIENT)
     {
     // Add the policy to the policy list of the client
     PutPolicyNo(PolicyNo,GetPolicyCount());

     // Increment the policy count of the client
     IncPolicyCount();
     }
}

void TClient::DeletePolicy(char *PolicyNo)
{
  for (int i=0; i<GetPolicyCount(); i++)
      if (strcmp(GetPolicyNo(i),PolicyNo)==0)
         {
         // Delete the selected policy from the
         // policy list of the client
         delete [] PolicyNoList[i];

         // Decrement the policy count of the client
         DecPolicyCount();

         // Reorder the policy list of the client
         for (int j=i; j<GetPolicyCount(); j++)
             PutPolicyNo(PolicyNoList[j+1],j);
         }
}
```

```
char *TClient::GetClientNo()
{
  return ClientNo;
}

char *TClient::GetSurname()
{
  return Surname;
}

char *TClient::GetFirstname()
{
  return Firstname;
}

char *TClient::GetAddress()
{
  return Address;
}

char *TClient::GetPhone()
{
  return Phone;
}

char *TClient::GetOccupation()
{
  return Occupation;
}

char *TClient::GetBirthdate()
{
  return Birthdate;
}

char *TClient::GetSex()
{
  return Sex;
}

int TClient::GetPolicyCount ()
{
  return PolicyCount;
}

char *TClient::GetPolicyNo (int index)
{
  return PolicyNoList[index];
}

void TClient::PutSurname(char *NewSurname)
{
  strcpy(Surname,NewSurname);
}
```

```
void TClient::PutFirstname(char *NewFirstname)
{
  strcpy(Firstname,NewFirstname);
}

void TClient::PutAddress(char *NewAddress)
{
  strcpy(Address,NewAddress);
}

void TClient::PutPhone(char *NewPhone)
{
  strcpy(Phone,NewPhone);
}

void TClient::PutOccupation(char *NewOccupation)
{
  strcpy(Occupation,NewOccupation);
}

void TClient::PutBirthdate(char *NewBirthdate)
{
  strcpy(Birthdate,NewBirthdate);
}

void TClient::PutSex(char *NewSex)
{
  strcpy(Sex,NewSex);
}

void TClient::IncPolicyCount()
{
  PolicyCount ++;
}

void TClient::DecPolicyCount()
{
  PolicyCount --;
}

void TClient::PutPolicyNo(char *PolicyNo, int index)
{
  PolicyNoList[index] = new char[POLICY];
  strcpy(PolicyNoList[index],PolicyNo);
}
```

```
// File : TAGENT.H

#ifndef __AGENT_H
#define __AGENT_H

#include "defines.h"

class TAgent {
private:
  char AgentNo[AGENT];
  char Surname[SURNAME];
  char Firstname[FIRSTNAME];
  char Company[COMPANY];
  char Address[ADDRESS];
  char Phone[PHONE];
  int CommRate;
  long TotalAmtDue;
  int PolicyCount;
  char *PolicyNoList[MAX_POLICIES_PER_AGENT];

public:
  TAgent(char *NewAgentNo, char *NewSurname, char *NewFirstname,
         char *NewCompany, char *NewAddress, char *NewPhone,
         int NewCommRate);
  ~TAgent();
  void AddPolicy(char *PolicyNo);
  void DeletePolicy(char *PolicyNo);
  char *GetAgentNo();
  char *GetFirstname();
  char *GetSurname();
  char *GetCompany();
  char *GetAddress();
  char *GetPhone();
  int GetCommRate();
  long GetTotalAmtDue();
  int GetPolicyCount();
  char *GetPolicyNo(int index);
  void PutSurname(char *NewSurname);
  void PutFirstname(char *NewFirstname);
  void PutCompany(char *NewCompany);
  void PutAddress(char *NewAddress);
  void PutPhone(char *NewPhone);
  void PutCommRate(int NewCommRate);
  void PutTotalAmtDue(long NewTotalAmtDue);
  void IncPolicyCount();
  void DecPolicyCount();
  void PutPolicyNo(char *PolicyNo, int index);
  };

#endif __AGENT_H
```

```
// File : TAGENT.CPP

#include <iostream.h>
#include <string.h>
#include "tagent.h"

TAgent::TAgent(char *NewAgentNo, char *NewSurname,
               char *NewFirstname, char *NewCompany,
               char *NewAddress, char *NewPhone,
               int NewCommRate)
{
  strcpy(AgentNo,NewAgentNo);
  strcpy(Surname,NewSurname);
  strcpy(Firstname,NewFirstname);
  strcpy(Company,NewCompany);
  strcpy(Address,NewAddress);
  strcpy(Phone,NewPhone);
  CommRate = NewCommRate;
  PolicyCount = 0;
  TotalAmtDue = 0;
}

TAgent::~TAgent()
{
}

void TAgent::AddPolicy(char *PolicyNo)
{
  if (GetPolicyCount() < MAX_POLICIES_PER_AGENT)
     {
     // Add the policy to the policy list of the agent
     PutPolicyNo(PolicyNo,GetPolicyCount());

     // Increment the policy count of the agent
         IncPolicyCount();
     }
}

void TAgent::DeletePolicy(char *PolicyNo)
{
  for (int i=0; i<GetPolicyCount(); i++)
      if (strcmp(GetPolicyNo(i),PolicyNo)==0)
         {
         // Delete the selected policy from the
         // policy list of the agent
         delete [] PolicyNoList[i];

         // Decrement the policy count of the agent
         DecPolicyCount();

         // Reorder the policy list of the agent
         for (int j=i; j<GetPolicyCount(); j++)
             PutPolicyNo(PolicyNoList[j+1],j);
         }
}
```

```
char *TAgent::GetAgentNo()
{
  return AgentNo;
}

char *TAgent::GetSurname()
{
  return Surname;
}

char *TAgent::GetFirstname()
{
  return Firstname;
}

char *TAgent::GetCompany()
{
  return Company;
}

char *TAgent::GetAddress()
{
  return Address;
}

char *TAgent::GetPhone()
{
  return Phone;
}

int TAgent::GetCommRate()
{
  return CommRate;
}

long TAgent::GetTotalAmtDue()
{
  return TotalAmtDue;
}

int TAgent::GetPolicyCount ()
{
  return PolicyCount;
}

char *TAgent::GetPolicyNo (int index)
{
  return PolicyNoList[index];
}

void TAgent::PutSurname(char *NewSurname)
{
  strcpy(Surname,NewSurname);
}
```

```
void TAgent::PutFirstname(char *NewFirstname)
{
  strcpy(Firstname,NewFirstname);
}

void TAgent::PutCompany(char *NewCompany)
{
  strcpy(Company,NewCompany);
}

void TAgent::PutAddress(char *NewAddress)
{
  strcpy(Address,NewAddress);
}

void TAgent::PutPhone(char *NewPhone)
{
  strcpy(Phone,NewPhone);
}

void TAgent::PutCommRate(int NewCommRate)
{
  CommRate = NewCommRate;
}

void TAgent::PutTotalAmtDue(long NewTotalAmtDue)
{
  TotalAmtDue = NewTotalAmtDue;
}

void TAgent::IncPolicyCount()
{
  PolicyCount ++;
}

void TAgent::DecPolicyCount()
{
  PolicyCount --;
}

void TAgent::PutPolicyNo(char *PolicyNo, int index)
{
  PolicyNoList[index] = new char[POLICY];
  strcpy(PolicyNoList[index],PolicyNo);
}
```

```
// File : TPOLICY.H

#ifndef __POLICY_H
#define __POLICY_H

#include "defines.h"
#include "trisk.h"

class TPolicy {
private:
  char ClientNo[CLIENT];
  char AgentNo[AGENT];
  char PolicyNo[POLICY];
  char OriginalDate[DATE];
  char StartDate[DATE];
  char EndDate[DATE];
  int Instalments;
  int Payments;
  int TotalPremium;
  int AmtDue;
  char DueDate[DATE];
  char Status[STATUS];
  int TotalRisks;
  TRisk *RiskList[MAX_RISKS_PER_POLICY];
  TRisk *Risk;
  int ClaimCount;
  char *ClaimNoList[MAX_CLAIMS_PER_POLICY];

public:
  TPolicy(char *NewClientNo, char *NewPolicyNo, char *NewAgentNo,
          char *NewStartDate, char *NewEndDate);
  ~TPolicy();
  void AddRisk(char *RiskNo, int RiskType,
       long RiskValue, int RiskStatus);
  void DeleteRisk(char *RiskNo);
  void AddClaim(char *ClaimNo);
  void DeleteClaim(char *ClaimNo);
  char *GetClientNo();
  char *GetAgentNo();
  char *GetPolicyNo();
  char *GetOriginalDate();
  char *GetStartDate();
  char *GetEndDate();
  int GetInstalments();
  int GetPayments();
  int GetTotalPremium();
  int GetAmtDue();
  char *GetDueDate();
  char *GetStatus();
  int GetTotalRisks();
  TRisk *GetRisk(char *RiskNo);
  char *GetRiskNo(int index);
  int GetClaimCount();
  char *GetClaimNo(int index);
  virtual int GetPolicyType() = 0;
```

```
    void PutStartDate(char *NewStartDate);
    void PutEndDate(char *NewEndDate);
    void PutInstalments(int NewInstallments);
    void PutPayments(int NewPayments);
    void CalcTotalPremium();
    void CalcAmtDue();
    void CalcDueDate();
    void CalcStatus();
    void IncTotalRisks();
    void DecTotalRisks();
    void PutRisk(TRisk *Risk, int index);
    void IncClaimCount();
    void DecClaimCount();
    void PutClaimNo(char *ClaimNo, int index);
    };

#endif __POLICY_H
```

```
// File : TPOLICY.CPP

#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "tpolicy.h"

TPolicy::TPolicy(char *NewClientNo, char *NewPolicyNo,
                 char *NewAgentNo, char *NewStartDate,
                 char *NewEndDate)
{
  strcpy(ClientNo,NewClientNo);
  strcpy(PolicyNo,NewPolicyNo);
  strcpy(AgentNo,NewAgentNo);
  strcpy(OriginalDate,NewStartDate);
  strcpy(StartDate,NewStartDate);
  strcpy(EndDate,NewEndDate);
  strcpy(DueDate,NewStartDate);
  Installments = INSTALLMENTS;
  TotalRisks = 0;
  TotalPremium = 0;
  Payments = 0;
  AmtDue = 0;
  ClaimCount = 0;
}

TPolicy::~TPolicy()
{
}

void TPolicy::AddRisk(char *RiskNo, int RiskType,
                      long RiskValue, int RiskStatus)
{
  if (GetTotalRisks() < MAX_RISKS_PER_POLICY)
     {
     // Create the new risk
     Risk = new TRisk(RiskNo,RiskType, RiskValue, RiskStatus);

     // Add the risk to the risk list of the policy
     PutRisk(Risk,GetTotalRisks());

     // Increment risk counter
     IncTotalRisks();
     }
}

void TPolicy::DeleteRisk(char *RiskNo)
{
  for (int i=0; i<GetTotalRisks(); i++)
      if (strcmp(GetRiskNo(i),RiskNo)==0)
         {
         // Delete the selected risk from the
         // risk list of the policy
         delete (TRisk *) RiskList[i];

         // Decrement risk counter
         DecTotalRisks();
```

```
        // Reorder the risk list of the policy
        for (int j=i; j<GetTotalRisks(); j++)
            PutRisk(RiskList[j+1],j);
        }
}

void TPolicy::AddClaim(char *ClaimNo)
{
  if (GetClaimCount() < MAX_CLAIMS_PER_POLICY)
     {
     // Add the claim to the claim list of the client
     PutClaimNo(ClaimNo,GetClaimCount());

     // Increment the claim count of the policy
     IncClaimCount();
     }
}

void TPolicy::DeleteClaim(char *ClaimNo)
{
  for (int i=0; i<GetClaimCount(); i++)
      if (strcmp(GetClaimNo(i),ClaimNo)==0)
         {
         // Delete the selected claim from the
         // claim list of the policy
         delete [] ClaimNoList[i];

         // Decrement the claim count of the policy
         DecClaimCount();

         // Reorder the claim list of the policy
         for (int j=i; j<GetClaimCount(); j++)
             PutClaimNo(ClaimNoList[j+1],j);
         }
}

char *TPolicy::GetClientNo()
{
  return ClientNo;
}

char *TPolicy::GetAgentNo()
{
  return AgentNo;
}

char *TPolicy::GetPolicyNo()
{
  return PolicyNo;
}

char *TPolicy::GetOriginalDate()
{
  return OriginalDate;
}
```

```
char *TPolicy::GetStartDate()
{
  return StartDate;
}

char *TPolicy::GetEndDate()
{
  return EndDate;
}

int TPolicy::GetInstalments()
{
  return Instalments;
}

int TPolicy::GetPayments()
{
  return Payments;
}

int TPolicy::GetTotalPremium()
{
  return TotalPremium;
}

int TPolicy::GetAmtDue()
{
  return AmtDue;
}

char *TPolicy::GetDueDate()
{
  return DueDate;
}

char *TPolicy::GetStatus()
{
  return Status;
}

int TPolicy::GetTotalRisks()
{
  return TotalRisks;
}

TRisk *TPolicy::GetRisk(char *RiskNo)
{
  // Search all risks
  for (int i=0; i<GetTotalRisks(); i++)
      if (strcmp(GetRiskNo(i),RiskNo) == 0)
         // Risk is found
         return RiskList[i];

  // Risk is not found
  return (TRisk *) NULL;
}
```

```
char *TPolicy::GetRiskNo(int index)
{
  return RiskList[index]->GetRiskNo();
}

int TPolicy::GetClaimCount()
{
  return ClaimCount;
}

char *TPolicy::GetClaimNo(int index)
{
  return ClaimNoList[index];
}

void TPolicy::PutStartDate(char *NewStartDate)
{
  strcpy(StartDate,NewStartDate);
}

void TPolicy::PutEndDate(char *NewEndDate)
{
  strcpy(EndDate,NewEndDate);
}

void TPolicy::PutInstalments(int NewInstalments)
{
  Instalments = NewInstalments;
}

void TPolicy::PutPayments(int NewPayments)
{
  Payments = NewPayments;
}

void TPolicy::CalcTotalPremium()
{
  // Reset the total premium
  TotalPremium = 0;

  // Sum all the risk premiums
  for (int i=0; i<GetTotalRisks(); i++)
      {
      Risk = GetRisk(GetRiskNo(i));
      TotalPremium += Risk->GetRiskPremium();
      }
}

void TPolicy::CalcAmtDue()
{
  AmtDue = GetTotalPremium() / GetInstallments();
}
```

```
void TPolicy::CalcDueDate()
{
  char Start[DATE];
  char Day[3], Month[3], Year[3];

  // Get the start date of the policy
  strcpy(Start,GetStartDate());

  Day[0] = Start[0];
  Day[1] = Start[1];
  Day[2] = '\0';

  Month[0] = Start[3];
  Month[1] = Start[4];
  Month[2] = '\0';

  Year[0] = Start[6];
  Year[1] = Start[7];
  Year[2] = '\0';

  // Calculate the next unpaid month
  int CurrentMonth = atoi(Month);
  int NextMonth = CurrentMonth + GetPayments();
  itoa(NextMonth,Month,10);

  // Fill in the zero
  if (NextMonth < 10)
     {
     Month[2] = '\0';
     Month[1] = Month [0];
     Month[0] = '0';
     }

  // Calculate the due date
  DueDate[0] = Day[0];
  DueDate[1] = Day[1];
  DueDate[2] = '-';
  DueDate[3] = Month[0];
  DueDate[4] = Month[1];
  DueDate[5] = '-';
  DueDate[6] = Year[0];
  DueDate[7] = Year[1];
  DueDate[8] = '\0';
}

void TPolicy::CalcStatus()
{
  char CurrentDate[DATE], CurrentDueDate[DATE], RevDueDate[DATE];
  char CurrentDay[3], CurrentMonth[3], CurrentYear[3];
  time_t t;
  struct tm *gmt;
  t = time(NULL);
  gmt = gmtime(&t);

  // Get the current date
  itoa(gmt->tm_mday,CurrentDay,10);
  itoa(gmt->tm_mon+1,CurrentMonth,10);
  itoa(gmt->tm_year,CurrentYear,10);
```

```
  // Fill in the zero
  if (strcmp(CurrentMonth,"10") < 0)
     {
     CurrentMonth[2] = '\0';
     CurrentMonth[1] = CurrentMonth[0];
     CurrentMonth[0] = '0';
     }

  // Fill in the zero
  if (strcmp(CurrentDay,"10") < 0)
     {
     CurrentDay[2] = '\0';
     CurrentDay[1] = CurrentDay[0];
     CurrentDay[0] = '0';
     }

  // Reverse the current date
  CurrentDate[0] = CurrentYear[0];
  CurrentDate[1] = CurrentYear[1];
  CurrentDate[2] = '-';
  CurrentDate[3] = CurrentMonth[0];
  CurrentDate[4] = CurrentMonth[1];
  CurrentDate[5] = '-';
  CurrentDate[6] = CurrentDay[0];
  CurrentDate[7] = CurrentDay[1];
  CurrentDate[8] = '\0';

  // Reverse the due date
  strcpy(CurrentDueDate,GetDueDate());
  RevDueDate[0] = CurrentDueDate[6];
  RevDueDate[1] = CurrentDueDate[7];
  RevDueDate[2] = CurrentDueDate[5];
  RevDueDate[3] = CurrentDueDate[3];
  RevDueDate[4] = CurrentDueDate[4];
  RevDueDate[5] = CurrentDueDate[2];
  RevDueDate[6] = CurrentDueDate[0];
  RevDueDate[7] = CurrentDueDate[1];
  RevDueDate[8] = '\0';

  // Set the status of the policy
  if (strcmp(CurrentDate,RevDueDate) < 0)
      strcpy(Status,"Paid");
  else
      strcpy(Status,"Unpaid");
}

void TPolicy::IncTotalRisks()
{
  TotalRisks ++;
}

void TPolicy::DecTotalRisks()
{
  TotalRisks --;
}
```

```
void TPolicy::PutRisk(TRisk *Risk, int index)
{
  RiskList[index] = Risk;
}

void TPolicy::IncClaimCount()
{
  ClaimCount ++;
}

void TPolicy::DecClaimCount()
{
  ClaimCount --;
}

void TPolicy::PutClaimNo(char *ClaimNo, int index)
{
  ClaimNoList[index] = new char[CLAIM];
  strcpy(ClaimNoList[index],ClaimNo);
}
```

```
// File : TCAR.H

#ifndef __CAR_H
#define __CAR_H

#include "defines.h"
#include "tpolicy.h"

class TCar : public TPolicy {
private:
  char Manufacturer[MANUFACTURER];
  char Model[MODEL];
  char Registration[REGISTRATION];
  char EngineSize[ENGINE];
  float CarValue;
  int FullLicenceStatus;

public:
  TCar(char *NewClientNo, char *NewPolicyNo, char *NewAgentNo,
       char *NewStartDate, char *NewEndDate, char *NewManufacturer,
       char *NewModel, char *NewRegistration, char *NewEngineSize,
       long NewCarValue, int NewFullLicenceStatus);
  ~TCar();
  virtual int GetPolicyType();
  char *GetManufacturer();
  char *GetModel();
  char *GetRegistration();
  char *GetEngineSize();
  long GetCarValue();
  int GetFullLicenceStatus();
  void PutManufacturer(char *NewManufacturer);
  void PutModel(char *NewModel);
  void PutRegistration(char *NewRegistration);
  void PutEngineSize(char *NewEngineSize);
  void PutCarValue(long NewCarValue);
  void PutFullLicenceStatus(int NewFullLicenceStatus);
  };

#endif __CAR_H
```

```
// File : TCAR.CPP

#include <iostream.h>
#include <string.h>
#include "tcar.h"

TCar::TCar(char *NewClientNo, char *NewPolicyNo, char *NewAgentNo,
           char *NewStartDate, char *NewEndDate,
           char *NewManufacturer, char *NewModel,
           char *NewRegistration, char *NewEngineSize,
           long NewCarValue, int NewFullLicenceStatus)
           : TPolicy(NewClientNo, NewPolicyNo, NewAgentNo,
                     NewStartDate, NewEndDate)
{
  strcpy(Manufacturer,NewManufacturer);
  strcpy(Model,NewModel);
  strcpy(Registration,NewRegistration);
  strcpy(EngineSize,NewEngineSize);
  CarValue = NewCarValue;
  FullLicenceStatus = NewFullLicenceStatus;
}

TCar::~TCar()
{
}

int TCar::GetPolicyType()
{
  return CAR_POLICY;
}

char *TCar::GetManufacturer()
{
  return Manufacturer;
}

char *TCar::GetModel()
{
  return Model;
}

char *TCar::GetRegistration()
{
  return Registration;
}

char *TCar::GetEngineSize()
{
  return EngineSize;
}

long TCar::GetCarValue()
{
  return CarValue;
}
```

```
int TCar::GetFullLicenceStatus()
{
  return FullLicenceStatus;
}

void TCar::PutManufacturer(char *NewManufacturer)
{
  strcpy(Manufacturer,NewManufacturer);
}

void TCar::PutModel(char *NewModel)
{
  strcpy(Model,NewModel);
}

void TCar::PutRegistration(char *NewRegistration)
{
  strcpy(Registration,NewRegistration);
}

void TCar::PutEngineSize(char *NewEngineSize)
{
  strcpy(EngineSize,NewEngineSize);
}

void TCar::PutCarValue(long NewCarValue)
{
  CarValue = NewCarValue;
}

void TCar::PutFullLicenceStatus(int NewFullLicenceStatus)
{
  FullLicenceStatus = NewFullLicenceStatus;
}
```

```
// File : THOUSE.H

#ifndef __HOUSE_H
#define __HOUSE_H

#include "defines.h"
#include "tpolicy.h"

class THouse : public TPolicy {
private:
  char HouseType[HOUSETYPE];
  int Rooms;
  char AreaCode[AREACODE];
  long HouseValue;
  long ContentsValue;
  int HouseAlarmStatus;

public:
  THouse(char *NewClientNo, char *NewPolicyNo, char *NewAgentNo,
         char *NewStartDate, char *NewEndDate, char *NewHouseType,
         int NewRooms, char *NewAreaCode, long NewHouseValue,
         long NewContentsValue, int HouseAlarmStatus);
  ~THouse();
  virtual int GetPolicyType();
  char *GetHouseType();
  int GetRooms();
  char *GetAreaCode();
  long GetHouseValue();
  long GetContentsValue();
  int GetHouseAlarmStatus();
  void PutHouseType(char *NewHouseType);
  void PutRooms(int NewRooms);
  void PutAreaCode(char *NewAreaCode);
  void PutHouseValue(long NewHouseValue);
  void PutContentsValue(long NewContentsValue);
  void PutHouseAlarmStatus(int NewHouseAlarmStatus);
  };

#endif __HOUSE_H
```

```
// File : THOUSE.CPP

#include <iostream.h>
#include <string.h>
#include "thouse.h"

THouse::THouse(char *NewClientNo, char *NewPolicyNo,
               char *NewAgentNo,  char *NewStartDate,
               char *NewEndDate, char *NewHouseType,
               int NewRooms, char *NewAreaCode, long NewHouseValue,
               long NewContentsValue, int NewHouseAlarmStatus)
               : TPolicy(NewClientNo, NewPolicyNo, NewAgentNo,
                         NewStartDate, NewEndDate)
{
  strcpy(HouseType,NewHouseType);
  strcpy(AreaCode,NewAreaCode);
  Rooms = NewRooms;
  HouseValue = NewHouseValue;
  ContentsValue = NewContentsValue;
  HouseAlarmStatus = NewHouseAlarmStatus;
}

THouse::~THouse()
{
}

int THouse::GetPolicyType()
{
  return HOUSE_POLICY;
}

char *THouse::GetHouseType()
{
  return HouseType;
}

int THouse::GetRooms()
{
  return Rooms;
}

char *THouse::GetAreaCode()
{
  return AreaCode;
}

long THouse::GetHouseValue()
{
  return HouseValue;
}

long THouse::GetContentsValue()
{
  return ContentsValue;
}
```

```
int THouse::GetHouseAlarmStatus()
{
  return HouseAlarmStatus;
}

void THouse::PutHouseType(char *NewHouseType)
{
  strcpy(HouseType,NewHouseType);
}

void THouse::PutRooms(int NewRooms)
{
  Rooms = NewRooms;
}

void THouse::PutAreaCode(char *NewAreaCode)
{
  strcpy(AreaCode,NewAreaCode);
}

void THouse::PutHouseValue(long NewHouseValue)
{
  HouseValue = NewHouseValue;
}

void THouse::PutContentsValue(long NewContentsValue)
{
  ContentsValue = NewContentsValue;
}

void THouse::PutHouseAlarmStatus(int NewHouseAlarmStatus)
{
  HouseAlarmStatus = NewHouseAlarmStatus;
}
```

```
// File : TRISK.H

#ifndef __RISK_H
#define __RISK_H

#include "defines.h"

class TRisk {
private:
  char RiskNo[RISK];
  int RiskType;
  long RiskValue;
  int RiskStatus;
  long RiskPremium;

public:
  TRisk(char *NewRiskNo, int NewRiskType,
        long NewRiskValue, int NewRiskStatus);
  ~TRisk();
  char *GetRiskNo();
  int GetRiskType();
  long GetRiskValue();
  int GetRiskStatus();
  long GetRiskPremium();
  void PutRiskType(int NewRiskType);
  void PutRiskValue(long NewRiskValue);
  void PutRiskStatus(int NewStatus);
  void CalcRiskPremium();
  };

#endif __RISK_H
```

```
// File : TRISK.CPP

#include <iostream.h>
#include <string.h>
#include "trisk.h"

TRisk::TRisk(char *NewRiskNo, int NewRiskType,
             long NewRiskValue, int NewRiskStatus)
{
  strcpy(RiskNo,NewRiskNo);
  RiskType = NewRiskType;
  RiskValue = NewRiskValue;
  RiskStatus = NewRiskStatus;
}

TRisk::~TRisk()
{
}

char *TRisk::GetRiskNo()
{
  return RiskNo;
}

int TRisk::GetRiskType()
{
  return RiskType;
}

long TRisk::GetRiskValue()
{
  return RiskValue;
}

int TRisk::GetRiskStatus()
{
  return RiskStatus;
}

long TRisk::GetRiskPremium()
{
  return RiskPremium;
}

void TRisk::PutRiskType(int NewRiskType)
{
  RiskType = NewRiskType;
}

void TRisk::PutRiskValue(long NewRiskValue)
{
  RiskValue = NewRiskValue;
}

void TRisk::PutRiskStatus(int NewRiskStatus)
{
  RiskStatus = NewRiskStatus;
}
```

```
void TRisk::CalcRiskPremium()
{
  if (GetRiskType() == CAR_RISK)
     {
     if (strcmp(GetRiskNo(),"Fire") == 0)
         RiskPremium = (FIRE_PREMIUM + (GetRiskValue() * 0.01) -
                       (GetRiskStatus() * CAR_DISCOUNT)) *
                        INSTALLMENTS;
     if (strcmp(GetRiskNo(),"Theft") == 0)
         RiskPremium = (THEFT_PREMIUM + (GetRiskValue() * 0.01) -
                       (GetRiskStatus() * CAR_DISCOUNT)) *
                        INSTALLMENTS;
     }

  else
     {
     if (strcmp(GetRiskNo(),"Fire") == 0)
         RiskPremium = (FIRE_PREMIUM + (GetRiskValue() * 0.001) -
                       (GetRiskStatus() * HOUSE_DISCOUNT)) *
                        INSTALLMENTS;
     if (strcmp(GetRiskNo(),"Theft") == 0)
         RiskPremium = (THEFT_PREMIUM + (GetRiskValue() * 0.001) -
                       (GetRiskStatus() * HOUSE_DISCOUNT)) *
                        INSTALLMENTS;
     }
}
```

```
// File : TCLAIM.H

#ifndef __CLAIM_H
#define __CLAIM_H

#include "defines.h"
#include "tpolicy.h"

class TClaim {
private:
  char ClaimNo[CLAIM];
  char ClaimDate[DATE];
  long ClaimValue;
  char ClaimDetails[DETAILS];
  char ClaimStatus[STATUS];
  char PolicyNo[POLICY];
  char PolicyStart[DATE];
  char PolicyEnd[DATE];
  char PolicyStatus[STATUS];

public:
  TClaim(char *NewClaimNo, char *NewClaimDate,
         long NewClaimValue, char *NewClaimDetails,
         char *NewPolicyNo, char *NewPolicyStart,
         char *NewPolicyEnd, char *NewPolicyStatus);
  ~TClaim();
  char *GetClaimNo();
  char *GetClaimDate();
  long GetClaimValue();
  char *GetClaimDetails();
  char *GetClaimStatus();
  char *GetPolicyNo();
  char *GetPolicyStart();
  char *GetPolicyEnd();
  char *GetPolicyStatus();
  void PutClaimDate(char *NewClaimDate);
  void PutClaimValue(long NewClaimValue);
  void PutClaimDetails(char *NewClaimDetails);
  void CalcClaimStatus();
  void PutPolicyStart(char *NewPolicyStart);
  void PutPolicyEnd(char *NewPolicyEnd);
  void PutPolicyStatus(char *NewPolicyStatus);
  };

#endif __CLAIM_H
```

```
// File : TCLAIM.CPP

#include <iostream.h>
#include <string.h>
#include "tclaim.h"

TClaim::TClaim(char *NewClaimNo, char *NewClaimDate,
               long NewClaimValue, char *NewClaimDetails,
               char *NewPolicyNo, char *NewPolicyStart,
               char *NewPolicyEnd, char *NewPolicyStatus)
{
  strcpy(ClaimNo,NewClaimNo);
  strcpy(ClaimDate,NewClaimDate);
  ClaimValue = NewClaimValue;
  strcpy(ClaimDetails,NewClaimDetails);
  strcpy(ClaimStatus,"Pending");
  strcpy(PolicyNo,NewPolicyNo);
  strcpy(PolicyStart,NewPolicyStart);
  strcpy(PolicyEnd,NewPolicyEnd);
  strcpy(PolicyStatus,NewPolicyStatus);
}

TClaim::~TClaim()
{
}

char *TClaim::GetClaimNo()
{
  return ClaimNo;
}

char *TClaim::GetClaimDate()
{
  return ClaimDate;
}

long TClaim::GetClaimValue()
{
  return ClaimValue;
}

char *TClaim::GetClaimDetails()
{
  return ClaimDetails;
}

char *TClaim::GetClaimStatus()
{
  return ClaimStatus;
}

char *TClaim::GetPolicyNo()
{
  return PolicyNo;
}
```

```
char *TClaim::GetPolicyStart()
{
  return PolicyStart;
}

char *TClaim::GetPolicyEnd()
{
  return PolicyEnd;
}

char *TClaim::GetPolicyStatus()
{
  return PolicyStatus;
}

void TClaim::PutClaimDate(char *NewClaimDate)
{
  strcpy(ClaimDate,NewClaimDate);
}

void TClaim::PutClaimValue(long NewClaimValue)
{
  ClaimValue = NewClaimValue;
}

void TClaim::PutClaimDetails(char *NewClaimDetails)
{
  strcpy(ClaimDetails,NewClaimDetails);
}

void TClaim::CalcClaimStatus()
{
  char Start[DATE], End[DATE], Claim[DATE];
  char RevStart[DATE], RevEnd[DATE], RevClaim[DATE];

  // Get the start date
  strcpy(Start,GetPolicyStart());

  // Get the end date
  strcpy(End,GetPolicyEnd());

  // Get the claim date
  strcpy(Claim,GetClaimDate());

  // Reverse the start date
  RevStart[0] = Start[6];
  RevStart[1] = Start[7];
  RevStart[2] = Start[5];
  RevStart[3] = Start[3];
  RevStart[4] = Start[4];
  RevStart[5] = Start[2];
  RevStart[6] = Start[0];
  RevStart[7] = Start[1];
  RevStart[8] = '\0';
```

```
  // Reverse the end date
  RevEnd[0] = End[6];
  RevEnd[1] = End[7];
  RevEnd[2] = End[5];
  RevEnd[3] = End[3];
  RevEnd[4] = End[4];
  RevEnd[5] = End[2];
  RevEnd[6] = End[0];
  RevEnd[7] = End[1];
  RevEnd[8] = '\0';

  // Reverse the claim date
  RevClaim[0] = Claim[6];
  RevClaim[1] = Claim[7];
  RevClaim[2] = Claim[5];
  RevClaim[3] = Claim[3];
  RevClaim[4] = Claim[4];
  RevClaim[5] = Claim[2];
  RevClaim[6] = Claim[0];
  RevClaim[7] = Claim[1];
  RevClaim[8] = '\0';

  if ((strcmp(RevStart,RevClaim) <= 0) &&
      (strcmp(RevEnd,RevClaim) >= 0) &&
      (strcmp(GetPolicyStatus(),"Paid")==0))
       strcpy(ClaimStatus,"Awarded");
  else
       strcpy(ClaimStatus,"Disallowed");
}

void TClaim::PutPolicyStart(char *NewPolicyStart)
{
  strcpy(PolicyStart,NewPolicyStart);
}

void TClaim::PutPolicyEnd(char *NewPolicyEnd)
{
  strcpy(PolicyEnd,NewPolicyEnd);
}

void TClaim::PutPolicyStatus(char *NewPolicyStatus)
{
  strcpy(PolicyStatus,NewPolicyStatus);
}
```