

## Chapter 2

# Analysis: Modelling Problem Understanding

The work in this chapter is structured as follows:

- **Section 2.1: Introducing Formal Object Oriented Analysis (FOOA)**  
This section provides a brief review of object orientation and formalisation with respect to the limitations of current analysis methods. It motivates the development of a formal object oriented analysis method.
- **Section 2.2: Analysis: An Overview**  
This section begins with a more complete overview of different analysis methods and models. It proceeds to define a list of criteria for judging analysis techniques. Based on these criteria, we propose formal object oriented analysis as a natural successor to more traditional approaches.
- **Section 2.3: Object Oriented Analysis: An Informal Approach**  
This section examines object oriented analysis from an informal point of view. A number of simple example systems are analysed and these illustrate the types of properties that a customer is likely to want to express in an object oriented framework of understanding. This section concludes by identifying five analysis models which combine in a coherent fashion to give a complete view of object oriented requirements.
- **Section 2.4: Formal Object Oriented Analysis Using Abstract Data Types**  
This section argues that there are advantages in using abstract data types (ADTs) during requirements capture and analysis. It reviews the informal links that already exist between ADTs and the early stages of object oriented development.
- **Section 2.5: Classes and Types**  
This section examines the more practical issues that arise when comparing class with type. In particular, it distinguishes between the different roles that each concept plays within the frameworks in which they are found.
- **Section 2.6: A Formal Object Oriented Requirements Model in ACT ONE: A Preview**

This section previews the work in chapter 3 by reviewing the way in which we propose to model object oriented requirements using ACT ONE. ACT ONE is not used as our object oriented analysis language: it serves only as a semantic model onto which object oriented requirements are mapped.

## 2.1 Introducing Formal Object Oriented Analysis (FOOA)

Requirements capture and analysis (RCA), within software development, is the first step in the long, often arduous, process of satisfying the needs of the customer. In short, it is the process of identifying and recording what is required. Unfortunately, the RCA process must fulfil two very different roles:

- The customer must be convinced that the requirements are completely understood and recorded.
- The designers must be able to use the requirements to produce a structure around which an implementation can be developed and tested.

The requirements act as an interface between problem domain ‘experts’, with potentially very little comprehension of computers, and solution domain professionals, who understand computer systems, languages, models and techniques but may have little knowledge of the problem environment. This dual role makes RCA a not insignificant problem. However, there is much hope in the knowledge that many of the same principles of structuring, organisation and method are found in both domains: the common theme is complexity management.

### 2.1.1 Introducing Traditional (Functional) Approaches

Traditional analysis methods (see [67, 99, 51, 36], for example) define ways in which to control complexity using a functionally oriented divide-and-conquer strategy. These methods, in general, have two major deficiencies:

- **They lack formality.**  
Being informal, they are open to interpretation and inhibit rigorous means of validation and verification. Lack of formality makes the notion of contractual software less appealing. Code re-use, which is dependent on the existence of libraries of well-defined components, is possible only through a formal statement of behaviour and a means of classifying behaviour to facilitate access to appropriate components.
- **The analysis does not lead to one consistent model.**  
Traditionally, the modelling of requirements results in an unnatural division between process and data. This separation of concerns can lead to two models which, at best, are difficult to relate, and, at worst, are contradictory.

### 2.1.2 Object Orientation

Object oriented techniques and concepts have been shown to be applicable in the analysis phase of development (see [25], for example). This should not be surprising since object oriented programming

is often said to be ‘real world modelling’ [31] which, in general, is what analysts are doing. The idea of applying object oriented methods, which initially grew up in the programming domain, to design and analysis corresponds to the way in which structured approaches, in the 1970’s, gradually infiltrated each stage of software development.

The object oriented philosophy does not throw away all the previous work on structured analysis; it re-uses many of the ideas and combines them in a consistent and coherent fashion. We argue that the application of object oriented methods does not make requirements capture easy, but it does make it easier. Object oriented techniques can be applied to different systems with much greater confidence in the underlying principles. In this way, the method becomes second nature and understanding of the systems being analysed is given prominence. With traditional analysis techniques, the balancing of the process and data parts of the problem inhibits understanding. Consequently, the structure of the problem domain is often compromised. The object oriented approach promotes the maintenance of problem domain structure throughout the whole development process. It is the conceptual integrity of the object oriented paradigm which provides the essential bridge between customer requirements and program design.

### 2.1.3 Formalisation

In light of the previous section, one could be forgiven for believing that object orientation does everything you could ever want (and more). However good object oriented methods are at modelling real world requirements, they do not provide a formal framework for mathematical reasoning and manipulation. Like traditional approaches, the diagrams central to object oriented methods are not formally defined. The strength of these diagrams is that the customer finds them easy to understand. This is also the root of their weakness: by ensuring ‘lay-person’ readability, the informal approaches lose much of their potential for reasoning and manipulation.

A formal model of requirements is unambiguous — there is only one correct way for designers to interpret the model. Although the model must still be mapped onto the real world, this mapping is in essence more rigorous than in previous approaches. Building a formal model requires a better understanding of the problem domain and a better understanding of the way in which the problem domain is understood. A formal model can explicitly model nondeterminism — when choice of behaviour is specified<sup>1</sup>. Another important feature of a formal method is that high levels of expressibility allow definition of *what* is required without stating *how* it can be achieved. This, together with nondeterminism, supports a powerful freedom of implementation facility.

### 2.1.4 Formalising the Object Oriented Approach

A formal approach to object oriented analysis requires:

- **A method for gaining understanding** of the problem so that deciding the relevance of any part of the problem domain is straightforward. In other words, a mechanism is needed for

---

<sup>1</sup>Nondeterminism is not the same as ambiguity. An ambiguous statement is one which can be interpreted in more than one way.

formally identifying the scope of the problem in terms of its component parts, i.e. the classes and objects.

- **A means of recording the relevant information** in a structured and coherent fashion. The notation used for capturing the properties of the model is fundamental to the method<sup>2</sup>. It must be able to reflect the structure of the problem domain whilst also specifying requirements in an implementation independent way.
- **A way of validating the model** against user requirements. Validation must check the model for well defined properties, allow analysts to test their understanding of the problem, and facilitate customer accountability.

Rather than promoting one particular formal method of representing object oriented requirements, section 2.3 identifies the properties that such a notation is required to express in an elegant and concise way<sup>3</sup>. We stress that the object oriented aspects of the system being analysed must be prominent in the formal representation. Diagrammatic representations of object oriented properties are encouraged provided they have a formal semantics associated with them.

## 2.2 Analysis: An Overview

### 2.2.1 Analysis is Problem Domain Understanding

Analysis is the process of maximising *problem domain understanding*. Only through complete understanding can an analyst comprehend the responsibilities of a system. The modelling of these responsibilities is a natural way of expressing system requirements. The modelling process increases understanding. Once the model is sufficiently rich to express all that is needed, then the analysis is complete and design can begin.

The simplest way for an analyst to increase understanding is through interaction with the customer. The customer may be one person, in which case the RCA process is much simplified; however, it is more likely that the customer is a group of clients, each with their own particular needs. These clients may be people, machines, or both. One of the main problems in dealing with a set of customers is that the interrelated set of requirements must be incorporated into one coherent framework. Each client must be able to validate his (or her) own needs irrespective of the other clients (unless of course these needs are contradictory). The partitioning of requirements in this way may not be advantageous to designers. An analyst must be able to understand the set of requirements as a whole: the structuring of requirements as a collection of client needs may or may not be recognised in the design, but it is important that such an option is available.

---

<sup>2</sup>Method is often confused with diagrammatic notation — the obvious reason for this is that the diagrams are often the most visible (and accessible) part of a method.

<sup>3</sup>Although we do not promote one particular language for expressing requirements, the examples do require a concrete syntax. Consequently, we chose to define the example specifications in OO ACT ONE (the formal object oriented analysis language defined in chapter 3).

Interaction with the customer is an example of informal communication. It is an important part of analysis and, although it cannot be formalised, it is possible to add rigour to the process. A well-defined analysis method can help the communication process by reducing the amount of information an analyst needs to assimilate. By stating the type of information that is useful, it is possible to structure the communication process. Effective analysis is dependent on knowing the sort of information that is required, extracting it from the customer, and recording it in some coherent fashion. This chapter is concerned with identifying the type of information that needs to be recorded during analysis, with a view to defining a suitable method of representing this information.

### 2.2.2 Traditional Analysis Methods and Models

The past two decades has witnessed the establishment of many different analysis techniques. Each technique places different degrees of emphasis on each of the following:

- **Functional decomposition**, which manages complexity in terms of structured functionality.
- **Information modelling**, which helps to structure understanding by imposing a framework based upon the data in the system.
- **State Transition Diagrams**, which place emphasis on the timing and control aspects of complex behaviour.

The underlying principle of each of these approaches is improving understanding through complexity management. Each approach provides one consistent view of a system and its parts. The problem central to analysis is that some approaches are more useful than others in particular problem domains. Since most systems are complex, to various degrees, in three aspects — function, data, and timing — it is hard to see how three different models can be balanced in one coherent method. An object oriented analysis (OOA) method provides a framework in which all three aspects of system behaviour can be represented, although the data provides the basis for the structure. Before OOA is examined, each of the other approaches is examined in more detail.

#### 2.2.2.1 Functional Decomposition

Whilst functional decomposition is a straightforward application of the divide and conquer maxim, it has one fundamental flaw — there is no explicit statement of problem domain understanding. The mapping between functional requirements and the subject matter is often indicative of the way the analyst sees the problem rather than how the customer views it. By emphasising functionality, the need for mutual understanding between customer and analyst is ignored.

Another difficulty with functional decomposition arises when the system being analysed does not appear to be providing a service which can be characterised by one all encompassing function. A system may be meeting a number of different needs whose inter-relationship can be seen only through a thorough understanding of the whole problem domain.

A final problem with a functional approach to decomposition is volatility. It has been argued that the most volatile aspects of system requirements are the functions [84]. Consequently, the structure of

a specification which is based on functionality may not be stable. Stable structure is an important part of the analysis — identifying the most persistent elements of a problem, and basing understanding around them, is fundamental to good requirements capture.

Functional decomposition is not to be totally ignored. It is a useful approach when considering simple systems (or components). When decomposition (not necessarily functional) has produced a component whose behaviour is easily characterised as some function, or group of functions, then any further decomposition should be functional. In an object oriented approach, the services an object provides can be thought of as providing system functionality.

### 2.2.2.2 Data Flow Models

This modelling technique is fundamental to many of the structured analysis methods first proposed in the 1970's and carried through to the present, for example [90, 94, 49, 27]. System requirements are modelled using data flow models as follows:

A customer has a need, or set of needs. Each need is modelled as an interaction with the system. An interaction is represented by input and output flows of data at nodes (which can be thought of as data processors). Data stores (which are also represented by nodes) are connected to other nodes in the system. Grouping nodes into subsystems is the way in which structure is added to improve understanding. However, grouping is informal and often arbitrary. Functional decomposition is applied to nodes at the lowest level.

Data flow models are hard to reason about — especially in large systems where the environment interface is complex. The partitioning of nodes is not well understood, and this can result in a data flow model structure which bears no resemblance to the structure of the problem domain. It is difficult to comprehend the overall behaviour of a reasonably sized data flow model. The difficulties in validating a data flow model against customer requirements are enormous.

### 2.2.2.3 Information Modelling

Modelling the world in data is most closely related to the way in which humans view their environment. Entity relationship diagrams, semantic data models and information modelling all refer to objects or entities in the real world (see [17] for an overview of these techniques). Every object has an associated set of attributes or properties, and there are ways of relating different objects. In its purest form, information modelling shows only the structure of a problem in terms of the data. There is no explicit statement of functional requirements. Therefore, the responsibilities of a system are not explicitly stated.

### 2.2.2.4 State Transition Diagrams

State transition diagrams are most useful for modelling systems whose behaviour progresses through different states over time. For example, a person can be modelled as proceeding through the following

sequence of states — born, child, adolescent, single adult, married adult, widowed adult and dead. This type of behaviour is an important aspect of all systems but it is not clear how other properties can be incorporated in such a simple model. Furthermore, some systems go through a large (potentially infinite) number of states and the structuring of such behaviour is often quite difficult. It is more reasonable to attempt to incorporate the notion of state transitions in a more constructive model. (State transition diagrams, in many different guises, are a useful means of providing an underlying semantics to some other less abstract models. A labelled state transition system plays the semantic role in the definition of OO ACT ONE (the formal object oriented analysis and requirements capture language.)

### 2.2.2.5 Combining Different Models

Before object oriented analysis, the three approaches to handling complexity were ‘thrown together’ in different ways in different methods. This gives rise to confusion when a precise statement of how to carry out the analysis is required. In many cases, there is a great deal of arm waving to connect data, function and structure. An object oriented approach provides a much more meaningful way of incorporating function and state attributes in the same model. The separation of function and state is not an issue during object oriented analysis because the lowest-level building blocks (the objects) are defined as combinations of both parts.

### 2.2.3 Features of Good Analysis

There follows a list of features which should be present in an analysis technique for it to be considered *good*. Each feature is seen to various degrees (or not at all) in each of the afore mentioned modelling approaches. OO ACT ONE, the formal object oriented analysis language proposed in this thesis, is examined in section 4.5.1. as one particular language which facilitates the meeting of these requirements.

A good analysis technique must:

- **Be amenable to changes in the requirements within a stable structure**

It is important that an analysis method is flexible enough to readily incorporate changes in the requirements. Three types of change must be catered for:

- Extension: when new requirements are added.
- Alteration: when old requirements are changed.
- Re-conceptualisation: when the same requirements are expressed differently.

These changes must be kept as localised as possible. Central to controlling change is the development of a stable structure upon which behaviour can be specified as a set of distinct though interrelated parts.

- **Encourage Re-use**

The issue of re-use has been well debated in the programming environment (see [59]). A good analysis method must encourage component and structural re-use. This is one of the areas

in which a formal approach is vital. A more difficult type of re-use to quantify is the notion of experience, when insight is gained into methods of application in different circumstances. Within analysis, the learning of a method should very quickly correspond to gaining experience. Only in the initial learning period should analysts be concerned with notation and concepts: a good analysis method should be based on very simple principles.

- **Act as an interface between customers and designers**

The analysis model must be capable of fulfilling two very different needs. Firstly, it must be *customer oriented*, i.e. there must be a direct correspondence between the model and how the customer views the problem. Secondly, the model must be useful to designers. The system requirements must be easily extracted, and the structure of the problem domain must be visible for (potential) re-use in the solution domain. The easiest way in which a model can play this dual role is if the same underlying notions and principles are present in the problem and solution spaces.

- **Incorporate standard modelling techniques**

There are many tried and trusted techniques for complexity management. These are found in many different forms and in many different areas. The five central concepts are:

- **i) Abstraction:** any mechanism by which irrelevant information can be set aside (perhaps for consideration at a later stage). Functional abstraction is a means of saying what something does without saying how it does it. Data abstraction is a way of specifying state as an interface rather than as contents. (In object oriented approaches, the notion of service further abstracts away from the difference between function and state.) Functional and data abstraction hide detail at a lower level. A different kind of abstraction emphasises detail in one part of the system by hiding information in a different part of the system at the same level.
- **ii) (De)Composition:** any mechanism by which a system (or component of a system) can be defined as a number of interacting (sub)components. Composition is the grouping together of behaviours to form a more complex behaviour. Decomposition is the realisation of a complex behaviour through division into simpler components. In other words, they are opposite sides of the same coin.
- **iii) Classification:** a means of classifying components into meaningful hierarchies. The way in which humans think is based on a conceptualisation of complex, often intertwined, classifications. The ability to group together objects according to shared (common) properties is fundamental to real world understanding. The same can be said of real world modelling.
- **iv) Communication:** some means of modelling interaction between components. In a complex system it is important that all interaction between components is well defined and clearly understood by customer, analyst and designers.
- **v) Relationship Co-ordination:** categorisation, composition and communication properties give rise to three different types of relationship, namely *is-a*, *has-a* and *interacts-with*.

It is important that the analyst models these relationships in a consistent and coherent fashion.

- **Have a formal basis**

Mathematical rigour is necessary for formal validation, testing and completeness and consistency checking. The advantages of formal methods in the specification of requirements are well documented (see [20, 50], for example).

### 2.2.4 Introducing Object Oriented Analysis

The principle upon which object oriented analysis (OOA) is based is the direct mapping of problem domain entities and responsibilities into a requirements model. The entities (objects) are described in terms of the interface through which they interact with their environment. The services offered at an interface abstract away from the *how* to the *what*. Encapsulation and abstraction, two of the most important modelling techniques, are implicit in an object oriented approach. OOA incorporates all the desirable features (other than the requirement for formality) within a consistent framework of understanding. The structure of problem understanding is the framework upon which the remaining stages of development, namely design and implementation, are based. OOA terminology has arisen from two very different sources:

- **Object Oriented programming languages**

A programming language is said to be object oriented<sup>4</sup> if it includes the notions of object, encapsulation, message passing between objects, class, inheritance, dynamic binding and polymorphism. The conceptual consistency between the different development stages, which is one of the main advantages of the object oriented approach, is also a disadvantage when the terminology is not clearly defined. Generally, the programming notions of class, object and inheritance are imprecise. It is important that this imprecision is not evident in analysis.

- **Information modelling**

Information modelling has resulted in a more analysis-like view of objects/entities. However, information models do not facilitate the definition of function or behaviour. Also, the modelling diagrams are informal and open to interpretation. Information modelling is good for representing the structure of the data being considered. It is not good at representing the classification relationships between the data containers.

As a preview of section 2.3, the key concepts of OOA are given below. They are not formally defined and as such only introduce the notions. Object oriented terminology is employed differently in different environments. This is one of the main problems with object oriented methods. A major part of the development of a FOOA technique is the provision of well-defined meaning to the concepts. The informal list of terminology, below, illustrates the problems — one person's class is another person's object!

---

<sup>4</sup>[124] defines three categories of 'object language', namely object oriented, class oriented and object based.

### 2.2.5 Objects and Classes: The Problems with Terminology

For each key concept, a number of definitions are given. Each of the definitions is ‘correct’ in its own particular context. This illustrates the confusion that exists in object oriented terminology.

- **Objects:**

- An object is anything which can be uniquely identified.
- An object is an entity which plays some role in the behaviour of the system under consideration.
- An object is some thing which encapsulates state, and the set of operations on that state.
- Objects are instances of abstract data types.
- An object is an element of a particular set (or class).

- **Classes:**

- A class is a collection of objects.
- A class is a set of related behaviours.
- A class is a type.
- A class is a description of properties common to a set of objects.
- A class describes an implementation, or group of implementations, of an abstract data type.

- **Inheritance:**

- Inheritance is a means of representing relationships between classes.
- Inheritance is a subtyping relation.
- Inheritance is an incremental code modification technique.
- Inheritance is a means of defining a class as a modification of one (or more) other classes.
- Inheritance is a code re-use facility.
- Inheritance is a tool for conceptually grouping together sets of behaviours with some properties in common.
- Inheritance is a tool for enforcing properties between instances of different classes.

- **Object interaction:**

- Objects interact by passing messages to each other.
- Object interaction is through a well defined interface.
- A service is provided by one object when it is asked to do something by another object.

- **(De)Composition:**

- A class can be defined as some sort of composition of two (or more) other classes.
- An object can be realised as a composition of instances of two (or more) interacting objects.
- (De)composition is a code re-use facility.
- (De)composition is a structuring mechanism.

The statements above emphasise the informal (and sometimes inconsistent) use of object oriented terminology. A FOQA method must remove this informality by defining each of the terms in a clear, concise and unambiguous way.

## 2.3 Object Oriented Analysis: An Informal Approach

This section does not continue the promotion of OOA through an extension of the list of well documented object oriented ‘blessings’. The three main undisputed features of the object oriented paradigm are:

- Consistency of method and notation throughout development.
- Modelling of the problem as it is viewed in the real world.
- Inherent abstraction and encapsulation.

These are the foundations upon which claims for extensibility, re-usability, improved understanding, and maintainability are built. Rather than elaborating on the object oriented claims, this section makes the assumption that object orientation provides the basis of a good approach to requirements capture and analysis. In this way, the crux of OOA can finally be considered: *how?* rather than *why?*

Sections 2.3.1, 2.3.2 and 2.3.3 give an informal introduction to the notions of objects and class, and the relationships between them. This gives rise to a number of other important issues which are best considered by adopting a particular language for the expression of object oriented requirements.

In section 2.3.4, many examples which are used in which a concrete syntax for recording object oriented properties of a system is introduced. This serves two purposes. Firstly, it introduces the concepts and relationships that are important in object oriented analysis. Secondly, it provides a means of examining the requirements that an analyst is likely to place on a formal object oriented language.

### 2.3.1 Identifying Objects

At first glance, the notion of object seems to be the key to the object oriented approach. It is important that our intuitive feel for what makes an object an object is reflected in a formal definition. To stimulate thought, we list a wide variety of *things* that could be considered to be valid objects. These *things* may have been identified during the analysis of a number of systems. We attempt to identify their ‘objectness’, i.e. the features that they share in common.

- **People:** You, me, John Major, the British Prime Minister, my mother, etc.
- **Structures:** The Eiffel Tower, Edinburgh Castle, the M25, your bank, the house you last slept in, etc.
- **Places:** Europe, Australia, Stirling University, Paris, the Eiffel Tower, your bank, the bank manager’s office in your bank, etc.
- **Machines:** The watch on your wrist<sup>5</sup>, the computer on which this was written, my car, etc.
- **Systems:** The computer on which this was written, the M25, the BT telephone network, your bank, etc.
- **Events:** The second world war, your birth, the last world chess championships, the 1992 Olympics, etc.

---

<sup>5</sup>If there is one — a watch that is!

- **Concepts (Abstractions):** Chess, the English language, the number 6, the Greek letter  $\Pi$ , the mathematical constant  $\Pi$ , etc.
- **Classifications:** people, structures, cars, trains, beds, houses, games, trees, songs, tunes, computers, planets, number systems, etc.

The first thing to notice is that there is always an informal link between the label (in this case a string of characters) and the object which the label identifies. In fact, several different labels can be applied to the same object. Worse still is the fact that one label can be applied to two different objects — John Major is not always a reference to the British Prime Minister. A major<sup>6</sup> difficulty is that the context in which the label is used, and the assumptions made by the reader, are fundamental in the mapping between label and object. It is clear that an analyst cannot work with the actual objects and so the labels act as abstractions for the objects. A **label** is a form of identification. This leads to the first property which must be fulfilled by an object: it must be uniquely identifiable by its label within the context of the problem domain.

An **attribute** is some property of an object which plays a part in it being uniquely identified. For example, an attribute of Paris is that it is the capital of France. A different attribute is that it is the city in which the Eiffel Tower is found. A rather different set of attributes may include the latitude and longitude of the city centre, a list of all the street names, or even a list of all the people in the Paris telephone directory. Each of these attributes is sufficient to identify ‘Paris’; but, it is not clear if this ‘Paris’ is a city, a part of a map, or a tourist centre. **Abstraction** is the means by which only the relevant attributes of some entity are considered. In different systems, and different problem domains, one object may have different model abstractions. To continue with Paris as an example: in a model of the globe, the latitude and longitude are important but, in a model of the telephone network they do not play an obvious role. This leads to the second property of an object: it must have some well defined set of **attributes**.

An object must be **encapsulated** so that its internal mechanisms are used only through some well defined **interface**. We define the **external attributes** of an object as precisely those feature which can be accessed in this way. A class of objects is said to provide a set of **services**. Each **service** corresponds to an **external attribute**. A **service request** to an object (at its external interface) is a means of invoking some response. The object providing the service (the **service provider**) may change its internal state, or output some ‘result’, or a combination of both.

It is evident that an object should be characterised by its external attributes, i.e. the operations which are serviced by its interface, if its representation is to be *implementation independent*. The internal structure, reflecting function and state, is the means by which the external attributes are defined. It is often necessary to structure the statement of requirements to aid understanding. Further, structure is necessary to define behaviour of objects which can attain infinitely many different states. Structure influences the implementation process. However, constructive specifications do not necessarily impose implementation decisions on designers. The structure of problem domain understanding is recorded during object oriented analysis. Implementers (and designers) are not obliged to

---

<sup>6</sup>No pun intended.

use the structure of the analysis in the solution domain but, in some cases, this re-use of structure is beneficial.

In conclusion, the definition of an object must incorporate:

- A means of identification.
- An interface which encapsulates the object by forcing access to the object to be through a well defined set of external attributes (services).
- The ‘meaning’ of the external behaviour, i.e. a statement of how an object responds to service requests.
- An internal state, i.e. a mechanism such that an object, as a dynamic entity, is able to progress through a sequence of states depending on its interaction with its external environment.

Two objects belong to the same **class**(ification) when they exhibit ‘the same behaviour’ through their interfaces<sup>7</sup>. Informally, they must offer the same set of external attributes, and the way in which these attributes are fulfilled must be the same when two objects in the same class have the same internal state. In object oriented terminology, a **class** embodies the concept of a set of objects together with some common behaviour characterised by a set of external attributes.

### 2.3.2 Identifying Classes

A class is a set of member objects offering common behaviour. A class definition must contain the following:

- A list of external attributes which all member objects must provide.
- A means of identifying member objects.
- A semantics defining the behaviour resulting from the servicing of external attributes for every member object of a class.

For example, `Range1to9` can be defined as follows:

- The external attributes are addition, subtraction and equality.
- The members are the integers 1, 2, 3, ..., 9.
- The semantics of the service methods associated with the three external attributes are those normally associated with integers.

We chose to think of a class as a parameterised set of behaviour. Each member of a class is identified by one particular realisation of the behaviour parameters. An object refers to one particular class member at any instance in its lifetime. Thus, a dynamic object references a sequence of class members as it progresses through a sequence of states prompted by interactions with its environment.

Identifying classes in the problem domain is fundamental to OOA. Classification provides us with a basis on which a framework of understanding and representation can be built. Moreover, human understanding has evolved through our ability to classify and categorise to various levels of abstraction. Therefore, it is reasonable to assume that classification must play a leading role in analysis.

---

<sup>7</sup>This notion of class membership is formally defined in chapter 3

### 2.3.3 Classification Relationships

By studying relationships between objects, between classes, and between classes and objects, it is possible to gain a better understanding of these concepts with relation to analysis and requirements capture. The two most fundamental relationships are class membership and subclassing:

**Class Membership:** When an object  $a$  is said to be a member of a class  $A$  we write  $a \in A$ .

**Subclassing:** When a class  $A$  is said to be a subclass of a class  $B$  we write  $A \sqsubseteq B$ .

It is possible for a class to be a subclass of more than one superclass<sup>8</sup>. For example, a cat class is a subclass of the class of mammals, and a cat class is also a subclass of the class of pets<sup>9</sup>. Note that a cat is classified by two different types of attributes: the physical attribute of being a mammal, and the functional attribute of being a pet. Analysis of most systems gives rise to the identification of objects whose relevant set of external attributes is different in different contexts. Distinguishing between different categories of attribute increases understanding of the problem domain, and this understanding can be represented diagrammatically in a **class hierarchy**. Class hierarchies provide a fundamental way of structuring object oriented requirements.

A simple way to structure the analysis of a problem is to first identify the class hierarchies. These structures show one type of relationship between entities in the problem domain, namely subclassing. Identifying classes and putting them into a coherent framework is fundamental to formal object oriented analysis. A simple example (a hall of residence) is examined below. It illustrates the power of an approach in which classification is the main form of analysis. It also focuses attention on the limitations imposed by restricting analysis to the identification of subclassing relationships.

#### Classification Example: A Hall of Residence

A decision has been made to computerise the records for the halls of residence in a university. In particular, one part of the system is concerned with the residents. Analysis of the residents has led to the following classifications:

- Residents are either students or non-students.
- Residents are either staff or non-staff.
- Staff are either wardens or porters.
- Wardens are either subwardens or hallwardens.
- Subwardens are students.
- Hallwardens are non-students.
- Ordinary residents are non-staff and students.

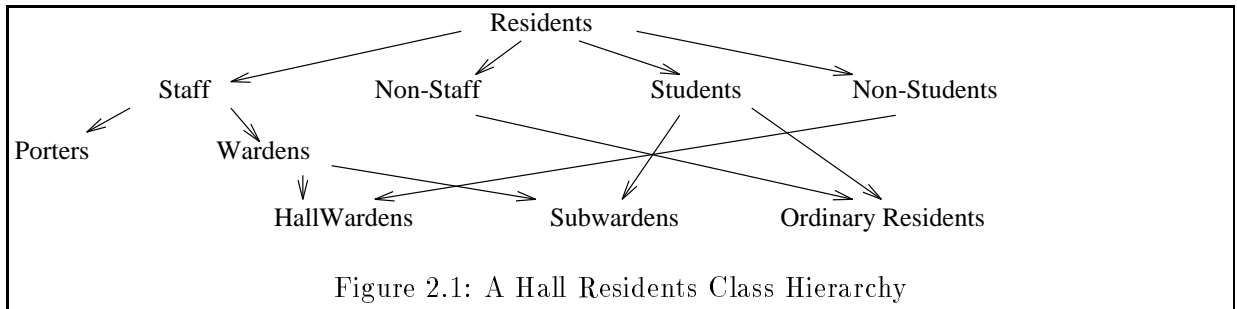
These relationships are shown in the class hierarchy in figure 2.1.

In this example, the class of **Subwardens** is a subclass of **Wardens**, and a subclass of **Students**. This multiple classification is very powerful within analysis. A subwarden can be regarded as a member of staff in one context, and a student in a different context.

---

<sup>8</sup> $A \sqsubseteq B \Leftrightarrow A$  is a subclass of  $B \Leftrightarrow B$  is a superclass of  $A$ .

<sup>9</sup>In this case it is questionable whether such a simple classification is appropriate — petting tigers is not recommended!



### Subclassing : The Limitations

Restricting analysis to the identification of class relationships has a number of limitations. In the analysis of the hall residents system, classification does not:

- Provide a means of recording the number of class members from each class in the residences.
- Consider the functional aspects of the system. (Although none of responsibilities of the system are shown, the class hierarchy does provide a structure upon which the functional aspects can be decomposed.)
- Represent the communication, synchronisation, or timing aspects of the system.

The first limitation is overcome through the introduction of a different type of relationship, namely composition. The functional aspects of a system are represented when classes are defined in terms of their external attributes. Timing and synchronisation aspects of analysis are more difficult to map directly onto the object oriented framework: they are considered in sections 4.2. and 4.3.

### 2.3.4 Defining Classes of Behaviour

This section introduces the concepts central to recording object oriented requirements during analysis. A number of simple object oriented behaviours are considered. A concrete syntax for the specification of object oriented requirements, namely OO ACT ONE, is introduced. OO ACT ONE is formally defined in chapter 3: its use at this stage of the thesis is intuitive and requires no knowledge of the underlying formality.

#### 2.3.4.1 LITERALS: an explicit identification of class members

The simplest object oriented property to identify and specify must be class membership. In OO ACT ONE, a literal is a label (defined as a sequence of characters) which uniquely identifies one member of a class. Consider the class `ComparisonResult` defined in example 1.

```

(* Example 1: LITERALS *)
CLASS ComparisonResult OPNS
LITERALS: before, after, same
ENDCLASS (* ComparisonResult *)
  
```

The specification defines the `ComparisonResult` class to have three members: `before`, `after` and `same`. There are no external attributes offered by this class (it can be used only as a passive carrier of data).

#### 2.3.4.2 STRUCTURES: parameterising the specification of class membership

The members of a class represent the set of states that an object can attain. It is necessary to extend the LITERAL concept to enable the specification of a set of class members in a parameterised fashion since:

- A class with a large number of literal members is unwieldy and inconcise.
- Parameterisation of class members adds structure to the specification and improves understanding.
- Parameterisation is the only means of defining classes of behaviour with an infinite number of members.

These points are reinforced by examples 2 and 3, which follow.

```
(* Example 2: STRUCTURES for ease of expression *)
CLASS C-RPair USING ComparisonResult OPNS
STRUCTURES: pair<ComparisonResult, ComparisonResult>
ENDCLASS (* CR-Pair *)
```

The set of `C-RPair` class members is not defined explicitly as a list of literal values. Rather, a STRUCTURE operation is used to parameterise the specification of class members. The set of class members can be realised through instantiation of the STRUCTURE parameters:

```
{pair(before,before), pair(before,after), pair(before,same), pair(after,before),
pair(after,after), pair(after,same), pair(same,before), pair(same,after), pair(same,same)}.
```

In this example, the set of nine class members can be defined using nine LITERALS. However, the STRUCTURE definition is more concise and the labelling of the STRUCTURE operation as a `pair` improves the specification. In example 3, the `Number` class has an infinite set of members:  $\{0, \text{succ}(0), \text{succ}(\text{succ}(0)), \dots\}$ . Classes with infinite behaviours arise in two different ways:

- Recursive STRUCTURE definitions define a STRUCTURE operation to have a parameter whose class is the same as the class in which the STRUCTURE is defined (see example 3).
- Non-recursive STRUCTURE definitions define an infinite class when one of the component classes is infinite.

```
(* Example 3: STRUCTURES for specifying infinite classes*)
CLASS Number OPNS
LITERALS: 0
STRUCTURES: succ<Number>
ENDCLASS (* Number *)
```

### 2.3.4.3 ACCESSORS: external attributes for getting responses from objects

Examples 1 to 3 define only classes as sets of passive members. The objects in each of these classes do not offer external attributes. We require a means of defining objects which can be asked to perform a service through their external interface. One such service an object can provide is to return some information about itself. Example 4 illustrates the specification of three ACCESSOR attributes, i.e. attributes which give *access to* some internal details of the object servicing the request.

```
(* Example 4: Unparameterised ACCESSOR operations *)
CLASS ThreeOrderA USING ComparisonResult OPNS
LITERALS:1,2,3
ACCESSORS: compare1 -> ComparisonResult, compare2 -> ComparisonResult,
compare3 -> ComparisonResult
EQNS
1..compare1 = same; 1..compare2 = before; 1..compare3 = before;
2..compare1 = after; 2..compare2 = same; 2..compare3 = before;
3..compare1 = after; 3..compare2 = after; 3..compare3 = same
ENDCLASS (* ThreeOrderA *)
```

An object of the class `ThreeOrderA` has state corresponding to one of the class members 1, 2 or 3. Such an object offers three external ACCESSOR attributes, namely `compare1`, `compare2` and `compare3`. Servicing an ACCESSOR results in the object servicing the request returning some value (a member of the class specified after the right arrow in the ACCESSOR operation definition). The object does not change its internal state.

The way in which each object of a class responds to an ACCESSOR request must be defined as part of the OO ACT ONE specification. ACCESSOR equations are defined in an expression of the form: `obj..accessor = ...`, where `obj` is a class member, `accessor` is the name of an ACCESSOR operation and the right hand side of the equation represents the result returned when an object with state `obj` services `accessor`.

```
(* Example 5: Parameterised ACCESSORS *)
CLASS ThreeOrderB USING ComparisonResult OPNS
LITERALS:1,2,3
ACCESSORS: compare<ThreeOrderB> -> ComparisonResult
EQNS
1..compare(1) = same; 1..compare(2) = before; 1..compare(3) = before;
2..compare(1) = after; 2..compare(2) = same; 2..compare(3) = before;
3..compare(1) = after; 3..compare(2) = after; 3..compare(3) = same
ENDCLASS (* ThreeOrderB *)
```

The `ThreeOrderB` class, in example 5, shows how ACCESSOR operations can be parameterised. In this case, `compare` is parameterised by class `ThreeOrderB`. The parameterisation of the external attributes of a class is necessary if we wish to model the service requester providing ‘input parameter values’ to the service provider.

#### 2.3.4.4 TRANSFORMERS: defining external attributes for history dependent behaviour

The behaviours of the `ThreeOrder` classes are history independent: they have no external attributes which change the state of the object servicing the request. Such classes are similar to types in imperative programming languages. The class members are analogous to constant values.

The behaviour of most objects depends on the previous services which they have ‘carried out’. One of the most common attributes of an object is the ability to accept some new information, remember it, and use it in response to a later request. In an object oriented analysis language it is necessary that history dependent behaviour can be defined. In OO ACT ONE, the simplest example of this type of behaviour corresponds to the imperative notion of a variable (given below in example 6).

```
(* Example 6: TRANSFORMERS: specifying history dependent behaviour *)
CLASS int-var USING integer OPNS
STRUCTURES: an-integer<integer>
TRANSFORMERS: update<integer>
ACCESSORS: recall -> integer
EQNS
an-integer(integer1)..recall = integer1; an-integer(integer1).update(integer2) =
an-integer(integer2)
ENDCLASS (* int-var *).
```

The `update` TRANSFORMER operation of the `int-var` class changes the internal state of the object servicing the request. The object does not return any value to the service requester. Like ACCESSORS, TRANSFORMER operations can be parameterised. The new state of an object after servicing a TRANSFORMER operation is defined by an expression of the following form: `obj.transformer =`  
`....`

#### 2.3.4.5 Parameterised Structure Equation Definitions

Example 6 illustrates the need to be able to define infinite behaviour in a parameterised form. Given a class `integer` defined to have an infinite number of members, class `int-var` also has an infinite number of members. The behaviour associated with each of these members must be defined in the equation body of the class. This is possible only through some form of equation parameterisation.

For example, `an-integer(integer1)..recall = integer1;` specifies that for every `integer1` which is a member of class `integer`, an object of class `int-var` with state `an-integer(integer1)` returns `integer1` in response to a `recall` service request. Similarly, the TRANSFORMER behaviour is also defined in a parameterised fashion.

Note that it is the variable parameters `integer1` and `integer2` are used to define the behaviour of `int-var` in a parameterised fashion. In our notation, the class of a variable parameter is identified by the string of characters which precede the last numeric character(s) of the variable identifier. All variable parameters in equation definitions must be represented in this way<sup>10</sup>. Equations which are

<sup>10</sup>An advantage of this approach is that the variable identifier also identifies the class to which the variable belongs. The disadvantage of variable names not describing their function is negated by using comments when it is necessary to

```

(* Example 7: Dynamic Structure *)
CLASS list USING integer, Bool OPNS
LITERALS: empty
STRUCTURES: S-list<list, integer>
ACCESSORS: check<integer> -> Bool
TRANSFORMERS: store<integer>
EQNS
empty..check(integer1) = false;
S-list(list1, integer1)..check(integer2) = (integer1..eq(integer2))..or(list1..check(integer2));
list1.store(integer1) = S-list(list1, integer1)
ENDCLASS (* list *)

```

parameterised on variable parameters are, by definition, true for all values of these variables.

#### 2.3.4.6 Dynamic Structure

The `int-var` example shows how to record the attributes of an object which has constant state structure. The state is said to be constant because all the data fields are fixed at creation (although the values in the fields may not be fixed) by one structure operation. In example 7 we specify a behaviour which does not exhibit static internal structure.

The `list` class is a simple store of integers which has two external attributes: `store` and `check`. The `store` attribute is used to put integer values into the receiving object of the `list` class. The `check` attribute is used to test if a given integer value has been previously `stored`. The `S-List` operation is said to define a recursive structure.

#### 2.3.4.7 Dependencies Between Classes

Many of the previous examples have class headers of the following form: `CLASS A USING B, ... OPNS`. It is a requirement of an analysis language that pre-defined behaviours can be re-used in the specification of new behaviour. The `USING` construct provides the basis for such re-use. Example 7 is the first specification in which the classes `used` are not just passive data carriers but actually play a fundamental role in the behaviour of the new class being specified. The `check` attribute of class `list` makes comparisons between `integers` which have been `stored` and an input `integer` parameter. This comparison depends on the following behaviour being defined in the classes used by `list`:

- Class `integer` must have an external `ACCESSOR` attribute defined as `eq<integer> -> Bool`.
- Class `Bool` must have an external `ACCESSOR` attribute defined as `or<Bool> -> Bool`.

#### 2.3.4.8 Multiple Structure Operations

In object oriented analysis we may identify a class of behaviour which is made up of 2, or more, distinct groups. For example, students at university may be either single honours students or joint honours students. Although the external interface of these two groups must be the same when they

---

say what role a particular variable takes.

are part of the same class, we require a mechanism to distinguish between them. Such a mechanism already exists, namely **STRUCTURE** operations. In OO ACT ONE we record this type of behaviour as a class with more than one **STRUCTURE** operation (see example 8).

```
(* Example 8: Multiple Structures *)
CLASS Students USING Subject, Bool OPNS
STRUCTURES: Single<Subject>, Joint<Subject,Subject>
ACCESSORS: studies<Subject> -> Bool
EQNS
Single(Subject1)..Studies(Subject2) = Subject1..eq(Subject2);
Joint(Subject1,Subject2)..Studies(Subject3) =
(Subject1..eq(Subject3))..or(Subject2..eq(Subject3))
ENDCLASS (* Students *)
```

The **STRUCTURE** mechanism provides a natural way of describing this type of class partitioning. Note that a **TRANSFORMER** operation can be defined to model a single honours student changing to be a joint honours student (or vice-versa). Multiple **STRUCTURE** operations can be used to model objects with dynamic structure.

#### 2.3.4.9 DUALS: A means of combining ACCESSORS and TRANSFORMERS

The previous examples have defined classes with two different types of attribute — **ACCESSORS** and **TRANSFORMERS**. We identify the need for an attribute which is a combination of these. For example, a **stack** class may store **integers**. An attribute **pop** is required to model the removal of an **integer** from the stack (i.e. a change of state) and the return of this information to the **pop** requester. To model this type of service in OO ACT ONE, a **DUAL** attribute is defined. This is illustrated in example 9.

```
(* Example 9: DUAL attributes *)
CLASS stack USING integer, Bool OPNS
LITERALS: empty
STRUCTURES: Sstack<stack, integer>
DUALS: pop -> integer
TRANSFORMERS: push<integer>
EQNS
empty.pop = empty AND ~integer;
Sstack(stack1, integer1).pop = stack1 AND integer1;
stack1.push(integer1) = Sstack(stack1, integer1)
ENDCLASS (* stack *)
```

A **DUAL** equation is defined as the conjunction of an **ACCESSOR** equation and a **TRANSFORMER** equation. For example, **empty.pop = empty AND ~integer;** specifies that an empty stack ‘changes’ state to being empty in response to a **pop** request and returns the value **~integer** to the service requester.

Modelling **DUAL** behaviour is fundamental to object oriented analysis and requirements capture. It can be argued that such behaviour can be adequately represented by an **ACCESSOR** followed im-

mediately by a **TRANSFORMER**. However, this dual model depends on some ‘lock out’ facility between servicing the **ACCESSOR** and the **TRANSFORMER**. Such a facility is implementation oriented and as such does not provide a good model for analysing this type of behaviour. The **DUAL** mechanism abstracts away from the *how* to the *what*.

#### 2.3.4.10 Unspecified Class Members: handling exceptions

Example 9 illustrates the first instance of explicitly defining behaviour of a class to be unspecified. An empty stack cannot return a meaningful result in response to a **pop** request. Certainly, there are a number of different options for coping with such exceptions, but an analysis language must handle them in as abstract a way as possible. In our object oriented model it is necessary that **pop** is defined for the member **empty**, otherwise **empty** would not be a valid **stack**. Rather than adopting a particular implementation strategy to deal with exceptions (like ‘just return a 0’) we define a mechanism to enable analysts to defer exception handling to the designers and implementers.

In OO ACT ONE, all classes are defined to have an unspecified literal member, represented by the class name preceded by a ‘~’ character. This member is implicit in every class specification and is used to represent behaviour which the analyst may not wish to specify at this stage of development. By default, the external attributes of unspecified members are defined to result in unspecified behaviour of the appropriate class (see chapter 3).

#### 2.3.4.11 Distinguishing Between Accessors and Transformers

Example 10 is included to emphasise the importance of distinguishing between **TRANSFORMERS** and **ACCESSORS**. In some object oriented models this is not done (for example, see [14]). We define a linked list of integers (**Linked-List**) with transformer and accessor operations which seem to define identical behaviour. These operations are **tailT** and **tailA**. However, our object oriented interpretation of the behaviours offered by these attributes is very different. A **Linked-List** object, in response to a **tailT** request, updates its internal state by removing the last **integer** element which was **added**. It does not return any result to the service requester. Contrastingly, the same object, in response to a **tailA** request does not update its internal state, but it does return a result to the service provider. Ambiguous specifications arise if **TRANSFORMERS** and **ACCESSORS** are not distinguished.

#### 2.3.4.12 Invariant Properties

A class invariant is some property which every member of that class must fulfil. We require an object oriented analysis language to incorporate some sort of invariant mechanism.

OO ACT ONE provides two types of invariant mechanism: class invariants and structure invariants. These are illustrated by examples 11 and 12. Again, the precise meaning of these invariants is unimportant at this stage: it is the principle behind invariant properties which is important.

Class **StudentsB** is similar to class **Students** (see example 8) except that there is an additional invariant property which guarantees that a joint honours student studies two different subjects.

```
(* Example 10: Distinguishing between accessors and transformers *)
CLASS Linked-List USING integer, Bool OPNS
LITERALS: empty STRUCTURES: S-Linked-List<Linked-List, integer>
ACCESSORS: tailA -> Linked-List
TRANSFORMERS: add<integer>, tailT
EQNS
empty..tailA = ~list;
S-Linked-List(Linked-List1, integer1)..tailA = Linked-List1;
empty.tailT = ~list;
S-Linked-List(Linked-List1, integer1).tailT = Linked-List1;
Linked-List1.add(integer1) = S-Linked-List(Linked-List1, integer1)
ENDCLASS (* Linked-List *)
```

```
(* Example 11: Structure invariants *)
CLASS StudentsB USING Subject, Bool OPNS
STRUCTURES: Singles<Subject>, Joints<Subject, Subject >
ACCESSORS: studies<Subject> -> Bool
INVARIANTS: Joint(Subject1, Subject2) REQUIRES Subject1..neq(Subject2)
EQNS ...
ENDCLASS (* StudentsB *)
```

The class `MathsStudents` is constructed from the members of `Students` which study `Maths`. Chapter 3, section 3.4, examines the OO ACT ONE invariant mechanisms in much more detail.

### 2.3.4.13 Composition vs Subclassing: Introducing the Problem

Often, object oriented programmers use inheritance (a subclassing mechanism) as a code sharing technique rather than in recognition of an actual subclassing relationship between classes (Meyer [84] often uses inheritance in this way and Stein [104] argues that delegation is inheritance). This is problematic in all areas of object oriented development. The following `vector` example illustrates the problem from two different points of view.

#### A vector Class

Two different views are as follows:

- **Subclassing** — A vector can be defined to be a subclass of both a magnitude class and a direction class. A vector incorporates all the attributes of a magnitude and all the attributes of a direction. Consequently, a vector is both these things. Subclassing is a natural way of representing these behaviour characteristics.

```
(* Example 12: class invariants *)
CLASS MathsStudents USING Subject, Bool OPNS
STRUCTURES: Singles<Subject>, Joints<Subject, Subject >
ACCESSORS: studies<Subject> -> Bool
INVARIANTS: MathsStudents1..studies(Maths);
EQNS ...
ENDCLASS (* MathsStudents *)
```

- **Composition** — A vector is constructed from two components, namely a magnitude and a direction. A vector is not a magnitude, it is not a direction, it is some amalgamation of both behaviours into a new class of behaviour.

Depending on which object oriented method is being applied, either of these views is likely to be modelled during the analysis of vector behaviour. During object oriented analysis we must always ask which model is a true reflection of the customer's understanding of the behaviour being specified?

In example 13 we chose to define a `vector` class in a compositional fashion. Every member of the `vector` class is represented by the parameterised structure expression `a-vector(magnitude1, direction1)`. The `a-vector` operation is the only structure of the `vector` class. Consequently, we interpret this to mean that every `vector` object is composed from two component objects (of type `magnitude` and `direction`). We also say, without risk of ambiguity, that a `vector` class is composed from a `magnitude` class and a `direction` class.

```
(* Example 13: Composition is not subclassing *)
CLASS vector USING magnitude, direction OPMS
STRUCTURES: a-vector<magnitude, direction>
ACCESSORS: length -> magnitude, angle -> direction
TRANSFORMERS: newlength <magnitude>, newangle <direction>
EQNS
a-vector(magnitude1, direction1)..length = magnitude1;
a-vector(magnitude1, direction1)..angle = direction1;
a-vector(magnitude1, direction1).newlength(magnitude2) = a-vector(magnitude2, direction1);
a-vector(magnitude1, direction1).newangle(direction2) = a-vector(magnitude1, direction2)
ENDCLASS (* vector *)
```

#### 2.3.4.14 Structure and Implementation Independence

This representation of a vector is not the only way of expressing its external behaviour. It is possible to define a `vector` using Cartesian co-ordinates  $x$  and  $y$ , say. Then, the length can be calculated as  $\sqrt{x^2 + y^2}$ , and the angle can be calculated as  $\tan^{-1}(\frac{y}{x})$ . This Cartesian representation, rather than the polar form given earlier, is more appropriate when the 'addition' of vectors is prominent in the analysis. However, it is much easier to 'multiply' vectors in polar form. The structure of conceptualisation on which the vector class definition is based can thus be seen to be important with respect to possible extensions to the external attributes.

An analyst must choose one representation over the other. We must question whether it is useful to say that a vector is composed of a magnitude and direction when it is equally likely that it is identified during the analysis as a co-ordinate in Cartesian space? The way in which an analyst views a problem is reflected in the requirements specification. Of necessity, it seems that analysis cannot avoid a predisposition in the view that is presented of the problem in the statement of requirements. It is wrong to ask an analyst to represent all possible conceptualisations — it would result in overly complex specifications without guaranteeing that all 'reasonable' ways of viewing the behaviour had

been recorded. An analyst must always chose the representation which is the best reflection of the way in which the customer understands the behaviour being specified.

### 2.3.5 Explicit Subclassing Relationships

#### 2.3.5.1 Implicit vs Explicit Relationships

Before considering subclassing in our object oriented analysis models, it is necessary to make a statement concerning implicit and explicit subclassing relationships. In a large, complex system with many classes, irrespective of the precise nature of the subclassing definition, it is probable that there are a large number of subclassing relationships between classes. We distinguish between two different types of relation:

- **Implicit**

An implicit subclassing relationship is one which has no relevance in the specification. For example, consider a `wine` class which has two `ACCESSOR` attributes: `name` and `year`, and a `person` class which has three `ACCESSOR` attributes `name`, `year` and `age`. Depending on how these classes are defined, it is possible that `person` is a subclass of `wine`. In other words, all members of the `person` class are also members of the `wine` class. The consequences of this in an implementation of such a model are unthinkable! This type of relationship is referred to as **implicit** since it is inherent in the specification, but is not explicitly acknowledged or used.

- **Explicit**

An explicit subclassing relationship is one which is explicitly acknowledged within an object oriented specification. For example, if our analysis identifies that all maths students are students then the subclassing relationship `MathsStudent`  $\sqsubseteq$  `Students` should be recorded explicitly. Consequently, in object oriented analysis we require a mechanism for making such statements and for verifying that the relationship is well-defined. There are two approaches to this problem:

- i) Define classes in the normal fashion and separately include a list of subclassing relationships which are relevant in the specification.
- ii) Define explicit classification mechanisms for defining a new class to be a subclass (or superclass) of an already existing class.

In OO ACT ONE we chose the second approach because the explicit classification mechanisms can be defined in a way that guarantees a valid class relationship between the new and old class. The first approach requires a general mechanism for checking subclassing relationships between any two classes. The analysis behind such a mechanism is much more difficult to formulate than that which guarantees subclassing in specific cases. Also, we argue that there are advantages in having a limited number of subclassing primitives.

#### 2.3.5.2 Specialisation and Generalisation

*Specialisation* of a class's behaviour is a straightforward reduction in the number of member objects through the addition of some property which must be fulfilled by members of the new subclass but

which may not be fulfilled by every member of the original superclass. The new class members continue to provide the corresponding external behaviour as the corresponding members in the old class. The old (super)class behaviour can be said to contain the new (sub)class behaviour. For example, the even integers are a specialisation of the integers.

*Generalisation* is the inverse of specialisation. For example, the class of integers is a generalisation of the class of even integers and the class of odd integers.

### 2.3.5.3 Extension and Restriction

*Extension* involves an addition of new attributes to an already existing class of behaviour. For example, a queue which can be reset to empty is an extension of a queue which cannot be reset.

*Restriction* is defined as the inverse of extension. Rather than extending an existing class with new attributes, restriction defines a subset of attributes in the existing (sub)class which are offered by the new (super)class.

### 2.3.5.4 Subclassing: a look ahead

The four explicit class relationships are formally defined in chapter 3, together with a mechanism for combining specialisation and extension. By their nature the explicit class relationships are difficult to analyse using informal examples and thus it is necessary to develop a formal framework for modelling object oriented requirements before we can pursue a rigorous formulation. There are many types of subclassing but, in our formal object oriented analysis, specialisation and extension (and their inverses) are the only two relationships which are deemed a necessary part of an object oriented analysis language.

## 2.3.6 Reviewing Object Oriented Analysis Language Requirements: A Five Model Approach

### 2.3.6.1 Five Object Oriented Models

We propose that object oriented methods are dependent on five central relationships. These are as follows:

- **Classification**

This is a relationship between an object and a class. All objects in a system are classified. Classes correspond to a group of objects which share a particular classification. Classification is fundamental to human understanding.

- **Subclassing**

This is a relationship between classes. If  $A$  is a subclass of  $B$  then all members (objects) of  $A$  are also members of  $B$ . The subclassing relationship is also prominent in human understanding.

- **Composition**

This is a relationship between objects. One object is said to be composed from its component parts (themselves objects). The classification of an object's components and the relationships

between these components define the internal structure of an object. Often, all the objects in a class exhibit the same internal structure. In this case a class can be said, without risk of ambiguity, to be composed from its component classes. Class composition is a concise way of defining a set of object composition relationships which hold for every class member.

- **Configuration**

This is also a relationship between objects. Two objects which are components of the same containing object may, or may not, be ‘connected together’. When two objects are connected we say that they configure. More specifically, there is some link between their external interfaces. The car example in figure 2.2 helps to clarify this notion.

- **Interaction**

Interaction is the only dynamic relationship considered during formal object oriented analysis. All the previous four relationships make up a static view of an object oriented system. An interaction represents an event (and the consequences of the event) that occurs in the lifetime of an object oriented system. Interactions occur between objects which have been configured. Note that two objects which are configured do not necessarily interact in the lifetime of the object. The external interactions of a system are defined as those which occur between the system and its environment. Internal interactions occur between the components of a system. Object oriented behaviour defines possible sequences of interactions between an object and its environment.

### 2.3.6.2 A Five-model Example

Object oriented analysis is the identification of objects in a system, and the subsequent modelling of these five relationships. For example, consider a system which is very well understood, namely the behaviour of a car. The five relationships are illustrated in figure 2.2.

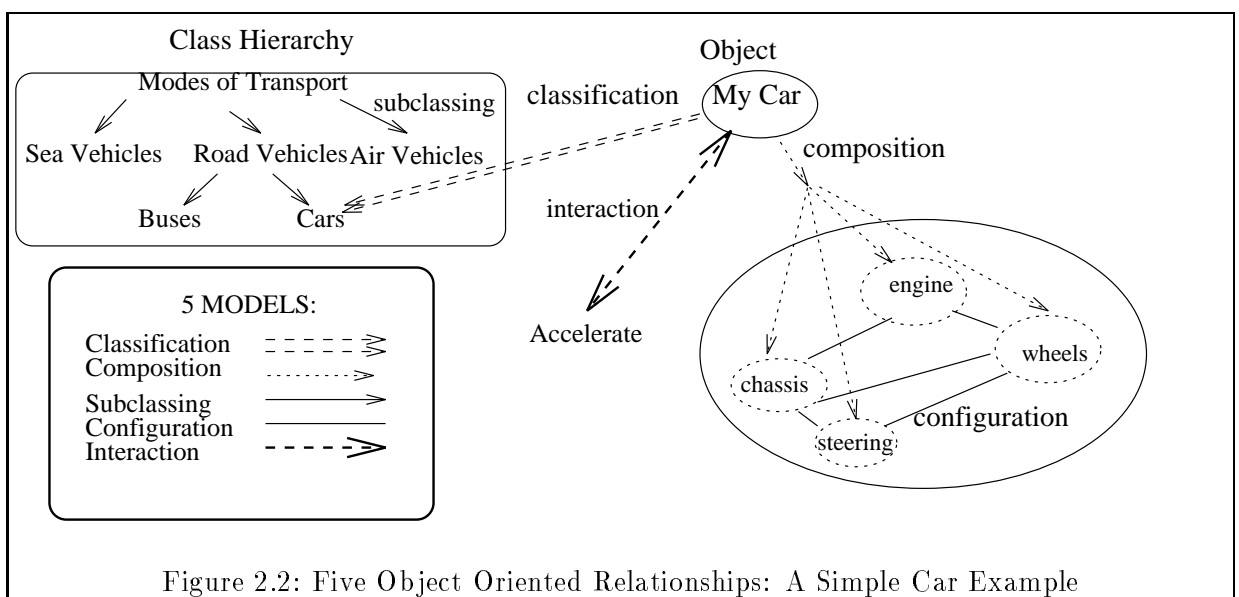


Figure 2.2: Five Object Oriented Relationships: A Simple Car Example

Distinguishing between the five relationships in this analysis is fundamental to human understanding and recording such relationships is therefore crucial in object oriented analysis and requirements capture. These relationships are the basis for the modelling techniques advocated within our object oriented development strategy. Using these models ensures that our object oriented approach is *customer oriented*.

### 2.3.6.3 FOOA: a review of our model requirements

This section has identified a number of aspects which must be present in a FOOA notation. These are:

- A means of defining classes of parameterised behaviour in terms of an abstract interface.
- A means of uniquely referencing each instance of such a parameterised class.
- A means of modelling a dynamic object as it changes its behaviour over time.
- A subclassing relationship between classes of behaviour.
- A composition facility for defining behaviours in terms of component behaviours.
- A facility for differentiating between three types of external attribute — accessors, transformers and duals.
- An interpretation of the internal structure of an object.

All these requirements are met in an approach based on the generation of the five object oriented models defined above.

## 2.4 Formal Object Oriented Analysis Using Abstract Data Types (ADTs)

### 2.4.1 Background to Abstract Data Types

Abstract concepts can be represented in a number of different ways. The means of representation, which is often referred to as the notation, is arbitrary in the sense that there are an infinite number of ways (syntactically) of labelling entities and representing the relationships between them. Natural languages illustrate the diverse range of notations that exist to provide, in general terms, the same representational ability.

A more formal example of a notation, which most everyone is familiar with, is the abstract concept of a counting mechanism or representation; or, to put it in more concrete terms, the concept of a positive integer. The arabic system of enumeration (1, 2, 3, 4, ...) identifies the same abstractions as the roman numerals (I, II, III, IV, ...), and the binary patterns (1, 10, 11, 100, ...). Furthermore, the arithmetic operations on these entities (objects) can be expressed in many different ways (e.g. prefix, infix or suffix notation).

In programming environments we are familiar with the idea of abstract behaviour being represented in different ways. For example, a string of characters can be represented as a fixed array or

as a linked list. Abstract Data Types (ADTs) are useful in computing because they can capture behavioural properties of entities in a manner which allows different implementations to be valid realisations of the same behaviour. In an ADT it is the abstract relationship between entities (objects) which is important. The concrete syntax is inconsequential (except that it should promote understanding of the underlying meaning and be amenable to manipulation within the conceptual framework to which it is being applied).

### 2.4.2 ADTS in an Object Oriented Semantic Framework

The concrete syntax which we employ in formal object oriented analysis and requirements capture must incorporate the following:

- A means of categorising entities into classes of behaviour.
- A mechanism for representing a set of operations associated with each class, where each operation associates one or more classes of entity with a resulting class of entity. In other words, a means of recording the external interface of a class so that all operations (on class members) can be statically ‘type checked’ for correctness.
- A means of defining the behaviour associated with each operation. In other words, a set of equations or axioms which give meaning to the operations.
- A facility for defining one class of behaviour in terms of other component classes of behaviour.
- An explicit means of representing the structure of the problem domain.
- Parameterised classes of behaviour (genericity)
- Inclusion polymorphism (subclassing).

Abstract data typing languages provide a suitable formal framework in which these types of property can be expressed. However, the relationship between type and class is complex. ADTs provide a good framework onto which object oriented requirements can be mapped.

An ADT provides us with a means of specifying ‘implementation free’ behaviour. This is ideal for requirements capture: analysts must try to identify and record *what* is required rather than *how* these requirements are to be met. However, as is argued in the previous section, a set of requirements must always contain some structure otherwise it would be impossible to record or understand them. The object oriented method of analysis and requirements capture encourages the recording of certain structural aspects of the problem domain. This aids understanding and gives the designers an initial structure upon which the design can be developed. In this way a formal statement of object oriented requirements is useful in later stages of development on two accounts: it unambiguously defines what is needed and it provides a structure for understanding the needs.

### 2.4.3 ADTs in the Initial Stages of Object Oriented Development

The idea of using ADTs at the beginning of an object oriented development strategy is not new. Meyer [84] states that:

“In object oriented design, every module is organised around a class of data structures ...To avoid implementation dependencies, the underlying description should be that of an ADT.”

Meyer goes on to relate design and implementation:

“Object oriented design is the construction of software systems as structured collections of ADT implementations.”

It is clear that Meyer believes that ADTs have an important role to play in the object oriented development of a system. But, Meyer does not further their cause in any way. He gives no indication as to how ADT specifications arise from analysis, or even to show how they can direct the design of an object oriented implementation. Perhaps the use of ADTs is so straightforward that no further instruction is necessary? This thesis shows otherwise.

The relationships between object oriented concepts and parts of an ADT specification are mentioned by Meyer. This informal interpretation is incomplete and imprecise; however, it has enough similarities to the work in this thesis to warrant inclusion below.

- An ADT expression corresponds to an object.
- The type of the expression corresponds to the class of the object.
- The valid operations on a type correspond to the services (external attributes) which that class of object can provide.
- The algebraic simplification of an expression (as defined in the equations for each operation) can be viewed as equivalent to the internal execution of instructions in an object.
- The value of an object (as an accumulation of its internal state) corresponds to the equivalence group of expressions of which that object is defined to belong in the equations. For example, the equivalence group  $\{‘3’, ‘1+2’, ‘2+1’, ‘1+1+1’, \dots\}$  is represented by the object ‘3’.

Other object oriented concepts which have not been mentioned in this informal list are primarily structural. The structure of a system which arises from the class hierarchy identified in the analysis is not evident in Meyer’s interpretation of ADT specifications. Also, the structure arising from the decomposition of behaviour into component behaviours has been acknowledged only in a very loose way.

#### 2.4.4 A Formal Object Oriented Development Method

It is clear that Meyer’s ideas on the role of ADTs in object oriented development need clarification. Before we examine the ADT ACT ONE, and show how it can be used for object oriented requirements capture, it is necessary to re-define the roles of the different stages of development. This acts to put the object oriented requirements capture process in a more concrete context and emphasises the role of Formal Object Oriented Analysis (FOOA). We define the stages of development as follows:

- **Analysis** is the process of understanding a system.

- **Requirements Capture** is the recording of the system understanding as a set of requirements.
- **Design** is the restructuring of the requirements towards an implementation architecture.
- **Implementation** is the realisation of behaviour specified in the design.

This thesis proposes a formal object oriented approach to development. The formal object oriented requirements capture method is defined as follows:

**Formal object oriented analysis and requirements capture** is the recording of the requirements of a system in terms of a set of ADTs, the structure of which corresponds to the structure of the problem domain. In effect, every object discovered in the analysis has a corresponding ADT specification.

Formal object oriented design is now defined as:

**Formal object oriented design** is the restructuring of the ADT specifications so that they can be re-used, within a less abstract model, to express the requirements in terms of solution domain objects and architecture.

In object oriented development there is a higher degree of correspondence between problem domain and solution domain structures than with traditional development methods. Designers should be encouraged to re-use analysis structure as much as possible. But, there will always be a conflict of interest between the way in which a problem is recorded and the way it is solved.

## 2.5 Classes and Types

### 2.5.1 Typing in Object Oriented Languages: An Introduction

There has been much interest in the relationship between static type checking and dynamic binding in object oriented programming [85, 22, 38]:

- **Static Type Checking:** when the code is statically checked to ensure that all service requests in a system can be fulfilled by the system component receiving objects during the system lifetime.
- **Dynamic Binding:** when the particular methods (code) are bound to service requests at run time.

Object oriented languages which facilitate dynamic binding can give rise to run time errors when objects are asked to provide services which are not part of their interfaces. This is not a desirable feature of any system; in particular critical (real time) systems should not produce ‘message-not-known’ results. Static type checking can help to prevent such errors. However, as pointed out in [85], such checking can also inhibit a dynamic binding facility. What is required is some means of combining static type checking and dynamic binding which guarantees that no run time errors arise from objects being unable to fulfil requests made of them, whilst also allowing service requests to be dynamically bound to services.

Object oriented programmers have identified the advantages of using abstract data types (ADTs) to support a type checking facility in dynamically bound object oriented languages [38, 22]. This work proposes using ADTs in the analysis and requirements capture stages of object oriented development. Before proceeding to relate the notions of type and class through the concept of data abstraction, it is necessary to examine the notion of type.

### 2.5.2 Types

A type is a description in the abstract of a related group of entities. Without types it is impossible to reason about all the different objects in a complex system as they would appear to have unrelated behaviour. Typing facilitates the grouping together of values in such a way that the shared behaviour is emphasised whilst the differences are abstracted away from. Typing has three roles:

- **Abstraction:** Values of the same type (in a programming language) share structure and semantics. The structure is used to represent the internal organisation of the value. The semantics represent the external behaviour of the type values. The way in which the values can be interpreted is given by the set of operations applicable to the type and the ‘meaning’ of such operations.
- **Re-use:** New abstractions can be created from existing ones. Types provide a natural way of structuring libraries into well defined packages of behaviour.
- **Validation:** Types provide a means of guaranteeing the validity of operations on given values through a static analysis of the system in question.

### 2.5.3 Type Systems

There are many different typing systems which fall into two distinct categories:

- **Monomorphic** systems require all data values to belong to only one type.
- **Polymorphic** systems allow values to belong to more than one type.

Polymorphism, together with dynamic binding, is a key feature of object oriented systems. In languages where functions are treated as types, the notion of a polymorphic function is widely accepted (see [111, 122], for example). For example, addition is applicable to both integers and reals, and consequently ‘+’ is an **overloaded operation**. It is also possible that ‘+’ can be used to calculate the sum of an integer and a real (with a real result). This is an example of **coercion** — the integer is coerced into being a real value<sup>11</sup>. These polymorphic techniques are available in a wide range of programming languages (imperative, functional and object oriented). They work on only a specific number of types in an unprincipled way. More universal techniques are **genericity** and **subtyping** and these are sometimes referred to as **universal polymorphism** [22].

---

<sup>11</sup>In object oriented systems, this is similar to a member of one class being dynamically bound to being a member of one of its superclasses.

- **Genericity:** a generic function works universally on a range of types (e.g. a swap function). Unconstrained genericity places no restrictions on the properties exhibited by these types. Constrained genericity is necessary in more complex behaviours. For example, a generic ordered list of values (of the same type) requires the type to have some partial ordering property. Generic types are said to be **parameterised**.
- **Subtyping:** the range of types a function can operate on is determined by a subtyping relationship. A function defined on a type can also operate on any subtypes. In object oriented terms this corresponds to the external attributes of a class including all the attributes of its superclasses. This is also known as **inclusion polymorphism**.

Genericity and subtyping are very different in principle and each have their own place in an object oriented framework [82]. Most abstract data type languages (including ACT ONE) incorporate a facility for defining parameterised types. Parameterised types can be statically instantiated at specification time and as such they give rise to a distinct group of behaviours (which just happen to have a similar structure). Subtyping relationships are much more interesting (in our object oriented framework) because they have an informal correspondence to our notion of subclassing.

#### 2.5.4 Mapping Classes to ADT Specifications

We propose to show that it is beneficial to distinguish the notions of class and type (in the sense of a syntactic interface offered by some element in an implementation language). However, we relate the more formal notion of ADT specification (a well-defined semantic notion of type) with the object oriented concept of class by defining a mapping from object oriented requirements to ADT specification. It is clear that type and class should not be confused [28], but we do believe that types can be used to implement the semantics of the class notion.

Types are more general than classes. In this thesis we generate type specifications from a formal model of object oriented requirements. The set of behaviours that can be specified in this way is much smaller than the set of all behaviours which can be specified using ADTs.

The differences between types and classes (subtypes and subclasses) arise from the way in which the terminology is applied rather than from differences in the underlying principles. The three roles of types, namely abstraction, re-use and validation, are equally applicable to classes:

- **Abstraction:** classes define an abstract interface behind which all the properties of objects in the class are encapsulated.
- **Re-use:** classes provide a fundamental package of re-usable behaviour.
- **Validation:** object oriented systems can be statically analysed to guarantee that all service requests to each object in the system, which may occur in the system lifetime, are available as part of the interface of the class to which the object belongs.

Problems arise in conceptually relating class with type when type is taken to represent a purely static syntactic interface. It is necessary to consider the behaviour offered by type ‘members’ through their

interfaces. Abstract data types provide both syntactic and semantic views of interface. Consequently, this thesis supports the view that classes and ADTs can be usefully related in a formal framework.

Abstraction is necessary in object oriented analysis since the view of a class as an ‘implementation body’ is wrong:

- Class defines behaviour — a stack provides LIFO behaviour no matter whether it is implemented in C++ or Eiffel, or whether it is represented (internally) as a linked list or an ordered bag. The notion of class as behaviour is vital when re-use and ‘correctness’ are considered. When designing a system it is desirable to be able to reason about components of a system without reference to implementation details. This is possible only if the notion of class is implementation independent. Re-use is a behavioural concern: it is wrong to limit re-use to the level of code integration.
- It is not easy to make a distinction between specification and implementation. An abstract specification may have many different valid implementations — an implementation resolves all (or some) of the abstraction. In many cases an implementation of one specification can itself be viewed as a specification of a less abstract set of implementations. There is no clear level of abstraction at which we can distinguish implementation from specification.

We have argued that class is not an implementation concern alone<sup>12</sup>. Implementation classes, as defined in object oriented programming languages, are not to be confused with the notion of class as an abstract statement of behaviour as defined by a particular ADT specification. ADTs provide the foundation upon which object oriented behaviour can be formally modelled. The notions of type and subtype need to be strengthened to provide a formal object oriented interpretation of class and subclass.

ADTs provide an abstraction over data structures in terms of well defined (procedural) interfaces. It is important that classes are not defined solely on the syntax of the interface in their resulting ADT specification. The semantics of behaviour provided at the interface is fundamental in the definition of class relationships. The notion of type as defined by interface is useful only for static type checking in the traditional sense: the non-introduction of syntax errors in code when a type is replaced by a subtype.

Consider the specifications of a queue and a stack. Both specifications could have the same external interface (defined by the operations ‘add’ and ‘remove’, say), but it is confusing to say that they have the same type. Type has three roles — abstraction, re-use and correctness. Type as a syntactic interface definition does not fulfil the second role and only partially fulfils the third role. We are not arguing that the notion of typing is without merit. However, within object oriented languages, it is more beneficial to incorporate the type concept in a more powerful means of categorisation, namely classification. This is particularly important when defining inclusion polymorphism. In the following chapters we retain the concept of *type* when referring to purely syntactic properties or relationships.

---

<sup>12</sup>Classes can be implemented but these implementations define class behaviour in a very constructive fashion — *how* not *what*.

## 2.6 A Formal Object Oriented Requirements Model in ACT ONE: A Preview

### 2.6.1 Modelling Object Oriented Requirements in ACT ONE

Type is more general a concept than class. Consequently, we have two options if we wish to use ACT ONE to model object oriented requirements:

- Restrict the ACT ONE syntax (i.e. enforce an object oriented style) and/or incorporate additional static analysis checks to ensure all specifications have a valid object oriented interpretation.
- Define a new object oriented analysis language and provide a mapping from specifications written in this new language to ACT ONE.

We chose the second option because:

- The ACT ONE syntax does not have an object oriented ‘flavour’. Although ACT ONE specifications can be given an object oriented interpretation, we feel that it is necessary to have the object oriented concepts prominent in an object oriented statement of requirements.
- ACT ONE is only one particular abstract data typing language. By defining a new language, we have an approach which can be generalised to modelling requirements in any given ADT (or any other formal language).

### 2.6.2 An Overview of the Class $\rightarrow$ ADT Mapping

The mapping from object oriented requirements specification to ACT ONE is similar to the mapping suggested by Meyer and others (see section 2.4.3). The mapping transfers the structural and hierarchical aspects of an object oriented model specified in OO ACT ONE to the ACT ONE code. Chapter 3, section 3.5, formalises the mapping from OO ACT ONE to ACT ONE. The fundamental relationships between these two different languages are:

- CLASS  $\rightarrow$  sort.
- LITERALS  $\rightarrow$  literals.
- STRUCTURES  $\rightarrow$  operations which are used to generate the terms which represent members of the class.
- INVARIANTS  $\rightarrow$  global preconditions on sort equations.
- ACCESSOR, DUAL and TRANSFORMER attributes  $\rightarrow$  operations which are term generators.
- Service requests and service responses (i.e. interactions)  $\rightarrow$  evaluation of an ACT ONE expression.
- Composition  $\rightarrow$  parameterisation of structure operations.
- Subclassing  $\rightarrow$  well defined relationship between the classes from which the sorts are generated.
- Inclusion Polymorphism  $\rightarrow$  a form of value coercion between classes and superclasses.

Although the mappings above are only informally introduced, the flavour of the ACT ONE object oriented model is evident. Chapter 3 adds precision and formality to these informal correspondences.

### 2.6.3 Using the ACT ONE Object Oriented Model

The ACT ONE generated from OO ACT ONE is used in three ways:

- It helps in the static analysis of object oriented properties in the system being specified.
- It provides an execution model for the testing of dynamic behaviour.
- It provides a natural mode of expression to bridge the gap between analysis and design.

The ACT ONE code is not intended to be explicitly presented to the customer. There are diagrammatic representations of object oriented properties which are more *customer oriented* (see sections 3.2 and 3.3). The structural information in these diagrams corresponds to much of the structural information recorded in the ACT ONE model. The formality underlying the meaning of these object oriented analysis diagrams does not make them any less practical than the widely accepted models advocated in other, less formal, analysis methods.