Chapter 4

Formal Object Oriented Analysis: The Practical Issues

This chapter examines the more practical issues which arise during object oriented analysis and requirements capture (when using OO ACT ONE). The discussions which are undertaken assume that the underlying formality of OO ACT ONE is well understood. The object oriented-ness of OO ACT ONE, and its suitability to the task of requirements capture and analysis, is no longer being considered. Rather, we proceed to investigate more general object oriented analysis issues, using OO ACT ONE to illustrate the points being made.

The structure of this chapter is as follows:

- **Section 4.1: Subclassing**
  This section examines the role of subclassing during object oriented analysis. Two different types of subclassing hierarchy are identified, namely those which offer multiple inheritance features and those which do not. Polymorphism and dynamic binding are then considered. The need to differentiate between explicit and implicit subclassing relationships is emphasised. Then, the concept of abstract superclass is given a more rigorous formulation. Finally, the classic polymorphism problem of heterogenous structures is explored.
- **Section 4.2: Composition**
  This section examines the notion of composition and its fundamental role in object oriented analysis. Emphasis is given to distinguishing composition from configuration and interaction. Two different types of compositional structures are introduced: dynamic and static. Then a pure style of OO ACT ONE specification is defined to model the object oriented notion of persistency. Finally, this section investigates the modelling of shared objects and timing properties during analysis.
- **Section 4.3: Other Object Oriented Analysis Issues**
  This section examines a potpourri of other object oriented issues: concurrency, nondeterminism, communication models, exception handling, the active/passive categorisation of objects, persistency and class routines concerned with creation and configuration.

- **Section 4.4: Reviewing The OO ACT ONE Specification Language**
  Section 4.4 reviews the OO ACT ONE specification language and asks if it is a *good* analysis technique when judged by the criteria put forward in chapter 2.
- **Section 4.5: The Practicalities of Building a Formal Model**
  This section begins by listing a set of critreria which an analysis method (as opposed to set of models) must fulfil. Then it defines a *skeleton* method for building a formal object oriented requirements model using OO ACT ONE. This *skeleton* method is shown to place emphasis on: re-use of pre-defined behaviours, recording of problem domain structure and improving problem domain understanding. After the method is formulated, some more general questions concerning analysis decisions, which influence the style of a specification produced using this method, are put forward. Then guidelines are given for making changes to the requirements models. Finally, a list of general analysis principles are proposed, which analysts can rely on to help make difficult development decisions.
- **Section 4.6: figure=FormAnal-Chp4/Figures and Object Oriented Design**
  This section introduces the process of going from analysis to design. It argues that the structure of a requirements specification is fundamental to the initial design of a solution to the problem. It is argued that executable requirements models are advantageous to object oriented development. Finally, as a preview of chapter 5, the notion of correctness preserving transformation is introduced.

4.1 Subclassing

4.1.1 Categorising Class Hierarchies

When one class is defined as a subclass of another it is said to **inherit** features of its superclass. Object oriented languages support either single inheritance or multiple inheritance. We argue that multiple inheritance is necessary for modelling customer understanding.

4.1.1.1 Single Inheritance

In single inheritance models, classes are allowed at most one parent (direct superclass). The class hierarchy diagram in figure 4.1 illustrates the type of hierarchy which arises when such a restriction is enforced.

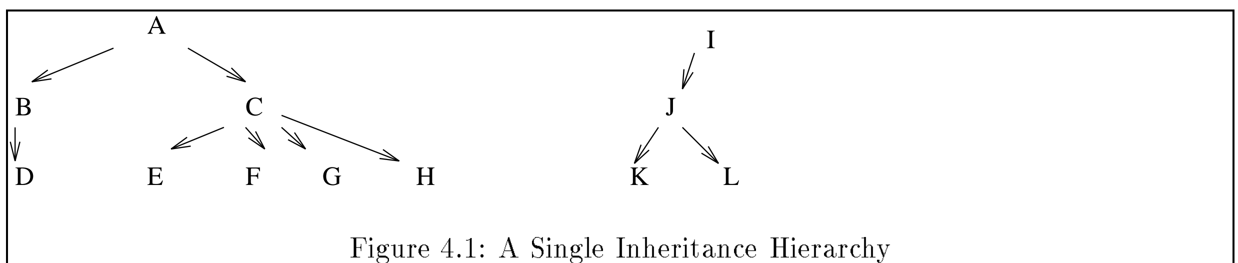

Figure 4.1: A Single Inheritance Hierarchy

In such models, inclusion polymorphism retains an important role. For example, a member of class `G` is also a member of classes `C` and `A`, and so can be treated as a `C` or an `A` when necessary. However, using single inheritance, it is not possible to represent the behaviour of a class which is a subclass of two different classes, which themselves are not related by a classification relationship. More precisely, in a single inheritance model, $(A \sqsubseteq B \text{ and } A \sqsubseteq C) \Leftrightarrow (B \sqsubseteq C \text{ or } C \sqsubseteq B)$. Often this restrictive view of inheritance is not desirable.

4.1.1.2 Multiple Inheritance

In multiple inheritance models, classes are not restricted in the number of parents they can have. The class hierarchy in figure 4.2 illustrates a typical multiple inheritance model.

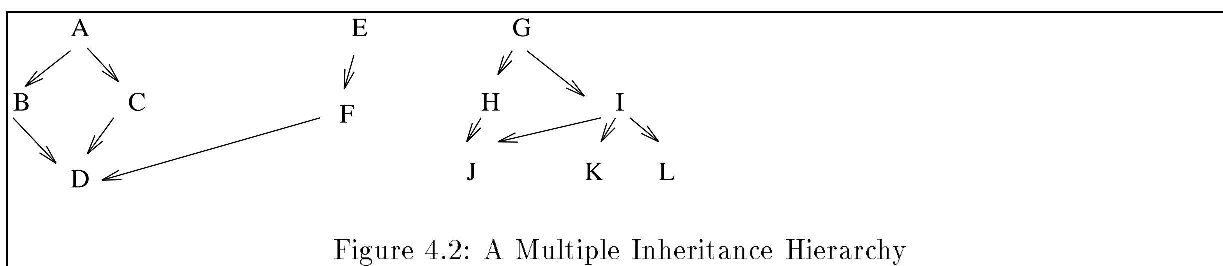

Figure 4.2: A Multiple Inheritance Hierarchy

Consider class `D`, in figure 4.2. Class `D` is a direct subclass of classes `B,C` and `F`. However, $C \not\sqsubseteq E$ and $E \not\sqsubseteq C$, for example. Class `D` offers the behaviour of both `C` and `F` even though these classes may not be related in any other way.

Multiple inheritance is such an important object oriented modelling concept that we must question why it is that some object oriented languages do not offer a multiple inheritance facility (see chapter 6). The answer is quite simply that in some cases, depending on the exact nature of the inheritance mechanism in the language in question, it is more trouble than it is worth. Fortunately, this is not the case with OO ACT ONE.

4.1.1.3 Multiple Inheritance is Problematic Only When Inheritance is not Subclassing

In object oriented programming languages, multiple inheritance can, and usually does, cause implementation difficulties. This is because these languages use inheritance as a code re-use facility rather than as a means of explicitly identifying behavioural compatibility. Object oriented programming languages make it difficult to distinguish between subclassing and composition. Using OO ACT ONE, the formal definition of these two relationships makes it much easier to distinguish between them. Multiple inheritance makes it appealing, when behaviour compatibility is not enforced, for programmers to use inheritance compositionally.

4.1.2 Inclusion Polymorphism and Dynamic Binding

The principle behind polymorphism is that a value (object) should not be constrained to being strictly typed (as a member of only one class). Polymorphic object oriented systems allow an object to be

treated as a member of more than one class. This type of property should set alarm bells ringing when it is first met: how are polymorphic systems type checked? Polymorphism seems to imply that no static type checking can be incorporated in an object oriented model which adheres to this flexible approach. This is true only for uncontrolled polymorphism.

In the most general case, uncontrolled polymorphism means that an object can be treated as if it is a member of any class. In other words, type checking is non-existent. Consequently, unless the programmer is very careful, errors can occur when an object is asked to provide a service which is not part of its external interface. Uncontrolled polymorphism is not a good feature for any language to exhibit.

Languages with hierarchical categorisation mechanisms are amenable to inclusion polymorphism. Two such types of language are those which include the notions of:

- types and subtyping.
- classes and subclassing.

In a typed language a value of one type can always be used in place of a value of any of its supertypes. In a classed language, an object of one class can always be used in place of an object in any of its superclasses. Inclusion polymorphism in typed languages can guarantee only the non introduction of syntax errors when a value of one type is replaced by the value of another. Inclusion polymorphism in a classed language can, and should, guarantee that a behavioural equivalence between a system before and after a class member in that system is replaced by the corresponding member in one of its subclasses.

The notion of substitutability is central to polymorphism. We must address the question of what it means to be able to accept a member of one class in place of a member of another class. It is difficult to address such a question without reference to a particular language. In OO ACT ONE there are two instances of substitutability:

- **Creating new objects from component parts**
  In OO ACT ONE, `STRUCTURE` operations are the means by which objects can be constructed from components. The parameter values of the `STRUCTURE` operations are the components of the new object being created. Inclusion polymorphism allows a new object to be created from component objects which are members of the class specified by the `STRUCTURE` operation defined, or members of a subclass of the class specified. For example, an object which is a pair of `integers` can also be created, using the same `STRUCTURE` operation, from a pair of `odd-integers` (provided `odd-integers`$\sqsubseteq$ `integers`).
- **Input parameters in external attributes**
  In OO ACT ONE, external attributes of a class can be parameterised on input values. A parameter value can be a member of the class specified in the operation definition, or a member of a subclass of the class specified. For example, an integer object with attribute `add <integer>` can be asked to add an `odd-integer` (again provided the appropriate subclassing relationship is explicitly defined in the OO ACT ONE code).

These two types of polymorphism are powerful mechanisms in an object oriented analysis model.

Dynamic binding is primarily an implementation concern. When an object is asked to service a request, the mechanism which it uses to service the request can be bound to the object at the time the request is placed. This is dynamic binding. It is not an analysis issue because in our object oriented requirements model we do not stipulate how the services between client and server should be provided.

4.1.3 OO ACT ONE: An Explicit Subclassing Approach

OO ACT ONE supports the specification of multiple inheritance hierarchies in a peculiar way. When one class, A say, is identified as being a subclass of two other classes, B and C say, there are two ways, in general, for this behaviour to be specified:

- i) First specify B, then explicitly define A to be a subclass of B and, finally, specify C as a superclass of A.
- ii) First specify A, then explicitly define B to be a superclass of A and, finally, specify C as a superclass of A.

In each of these specifications the resulting class hierarchy is the same.

Note that multiple inheritance is possible in OO ACT ONE because we provide explicit superclassing mechanisms, defined as inverses of the explicit subclassing mechanisms. In object oriented programming all explicit class relationships (like inheritance) allow a new class to be defined only as a subclass of an already existing class. OO ACT ONE allows the relationship to be defined in the other direction: a new class can be defined to be a superclass of an already existing class. This is a new approach to the definition of object oriented behaviour which facilitates multiple inheritance modelling in OO ACT ONE.

4.1.4 Abstract Classes

The term *abstract superclass* pervades object oriented programming languages. It arises in the following type of scenario:

> A class of shapes (for display on a screen, say) is a superclass of triangles, squares and pentagons. Each of these subclasses exhibits the behaviour of superclass shapes. It is not possible to instantiate a member of the shapes class which is not a triangle, square or pentagon. The shapes class is an abstract superclass.

This notion is more formally represented in OO ACT ONE by a class which is defined to generalise another class, or classes, without defining new literal or structure operations. In terms of the O-LSTS semantics, the definition of an **abstract superclass** is given below.

Definition: Abstract Superclass

Class C is an **abstract superclass** in $Env_D \Leftrightarrow$
$\forall c \in US(C), \exists C' \in visible(D)$ such that $c \in US(C')$ and $C' \sqsubseteq C$ in Env_D.

4.1.5 A Polymorphism Problem: Heterogeneous Data Stores

Consider a class of `items`, which is an abstract superclass with subclasses `integer`, `character` and `bool`. We wish to define a data store (a `stack` say) of `items` with external attributes `push` and `pop`. The main difficulty with this type of behaviour is the unidirectional aspect of polymorphism in object oriented languages: an object can be polymorphised 'up the class hierarchy but not down it'. In other words, a member of a class C can be treated as a member of a superclass of C but not a member of a subclass of C. Consequently, for example, if an `integer` *becomes* an `item`, when it is `push`ed onto an `item stack`, then when it is `pop`ped off the stack it remains a member of the `item` class and cannot be used as an `integer`.

The O-LSTS semantics conforms to the unidirectional polymorphic model. There are two reasons for defining the semantics in this way:

- It simplifies the understanding of O-LSTS behaviour since it is always clear how each object is classified.
- It simplifies the semantics since it is not necessary for every object to remember its polymorphic history. For example, an `item` does not need to know that it was once an `integer` which was once an `odd-integer` ....

4.2 Composition

The composition relationship is fundamental to object oriented analysis and requirements capture. Human understanding of systems is based on a divide-and-conquer deconstructionist approach, which can be summarised as follows:

> If we wish to understand an object A then by identifying the parts A is made from, and attempting to understand these, the original problem of understanding A is simplified.

There are three questions which we need to ask about this deconstructionist philosophy:

- 1) Is there always a unique set of components associated with a system being analysed?
- 2) Why should a component of a system be easier to understand than the system in which it is found?
- 3) How do we know when the deconstruction of understanding should end?

The answer to the first question is certainly *NO*, otherwise analysis would be 'trivial'. The second question is less easy clear since the rather paradoxical answer is that sometimes components of a system are more difficult to understand than the system in which they are found. This is especially true when a component is used, in the particular system being analysed, to provide only a small set of the behaviours which it is capable of exhibiting. In such a system it may be easy to understand the limited behaviour of such a component. However, if such a component is separately analysed then the usage constraints are no longer relevant and understanding its complete behaviour becomes very difficult. (This argument illustrates quite strongly why abstraction is such a powerful mechanism

for human understanding. Deconstructionism depends on the ability to be able to identify useful composition abstractions.) The final question is the bane of many an analysis process: too many systems are over-analysed leading to an increase in development costs and bad design. It is vital that an analysis method incorporates a means of deciding when the requirements capture process is complete.

4.2.1 Composition Structure

With respect to analysis alone it has already been stated that there are many ways of structuring problem understanding. An analyst must chose the structure best suited to communicating the requirements model with the customer. The designers should not be considered at this stage: they should be familiar with the object oriented analysis models and therefore be able to cope with any structure defined within the figure=FormAnal-Chp4/Figures semantic framework.

The compositional structure is fundamental to object oriented analysis. OO ACT ONE has an explicit mechanism for capturing structure properties: the `STRUCTURE` operation. This makes composition one of the most visible aspects of an OO ACT ONE specification.

Composition is a relationship between a container object and its contained parts. It tells us nothing about the relationship between the parts. Configuration and interaction are two relationships which arise out of one object being decomposed into distinct parts. Two objects may configure and interact only if they are components of a common container.

4.2.2 Configuration

`STRUCTURE` operations only brush the surface with respect to the composition analysis of problems. One can state that a car is composed from an engine, a chassis, a suspension, body, wheels, lights, etc ...(and each of these components can themselves be further decomposed) but this does say how the components are connected together (or if they are connected at all).

In section 3.3.5, configuration of components is formally defined as an interdependency during the fulfilment of a service request. Informally, two components configure if at least one of the services provided by their containing object depends on both the components to fulfil that service. This is a very abstract way of conceptualising configuration, but one which is appropriate to analysis. Analysis identifies *what* not *how*.

4.2.3 Interaction (Data Flow and Control Flow)

Interactions, with associated data and control flow, are much more concrete notions than configuration. They describe *how* behaviour is fulfilled rather than only saying *what* the behaviour requirements are. These notions are common to structured analysis methods but are not given prominence in the object oriented analysis advocated in this thesis: they are considered in greater detail as part of the object oriented design process. An interpetation of these concepts can be derived from an OO ACT ONE specification, but such an interpretation should be taken only in the following two circumstances:

- When the customer is more familiar with structured analysis methods than with object oriented techniques.
- When designers require a more traditional interpretation of a requirements model.

Appendix D details a formal interpretation of interaction, data-flow and control-flow in OO ACT ONE specifications.

4.2.4 Structures: Dynamic and Static

The notions of dynamic and static structure are important when attempting to provide an interpretation of composition properties. This section shows the difficulties that dynamic structures give rise to in an object oriented analysis model. They are open to abuse in the sense that, if used *wrongly*, they can make specifications hard to communicate with the customer. A *pure* style of specification is introduced to overcome this potential problem.

4.2.4.1 *Impure* Object Oriented Specification Practices

Impure Style: Example 1
The following OO ACT ONE specification of class `TwoStacks` is well defined but illustrates an *impure* style of specification.

```
CLASS TwoStacks USING Stack, Nat OPNS
STRUCTURES: S<Stack, Stack>
DUALS: pop -> Nat
TRANSFORMERS: push<Nat>, swap
EQNS
S(Stack1, Stack2).pop = S(Stack1, Stack2.pop) AND Stack2..pop;
S(Stack1, Stack2).push(Nat1) = S(Stack1.push(Nat1), Stack2.pop)
S(Stack1, Stack2).swap = S(Stack2, Stack1)
ENDCLASS (* TwoStacks *)
```

The behaviour specified in `TwoStacks` is straightforward to understand but on closer examination raises some interesting points for designers. For example, the `Stack` components may be implemented on different processors at different sites and conceptually the specification seems to suggest that a `swap` service results in all the data stored in one `Stack` being transferred to the other (and vice versa). Of course, this is an inefficient way of implementing such behaviour. It is more natural to define a pointer component which addresses either `Stack1` or `Stack2` and makes the state of the pointer change in response to a `swap` request. Such a behaviour is specified below in class `TwoStacksB`. (In O-LSTS semantics it is simple to show that these specifications fulfil each others behaviour.)

The `TwoStacksB` specification is written in a *pure* object oriented style — the components of a `TwoStacksB` object persist throughout the life-time of the object and are used only through their external interfaces. In the first class specification of `TwoStacks` the structure components do not persist, even though the structure is fixed.

Impure Style: Example 2

```
CLASS TwoStacksB USING Stack, Nat, Bool OPNS
STRUCTURES: S<Stack, Stack, Bool>
DUALS: pop -> Nat
TRANSFORMERS: push<Nat>, swap
EQNS
Bool1 => S(Stack1, Stack2, Bool1).pop = S(Stack1, Stack2.pop, Bool1) AND Stack2..pop
OTHERWISE S(Stack1.pop, Stack2, Bool1) AND Stack1..pop;
Bool1 => S(Stack1, Stack2, Bool1).push(Nat1) =
S(Stack1.push(Nat1), Stack2.pop, Bool1)
OTHERWISE S(Stack1.push(Nat1), Stack2.pop, Bool1);
S(Stack1, Stack2, Bool1).swap = S(Stack1, Stack2, Bool1..not)
ENDCLASS (* TwoStacksB *)
```

```
CLASS StacksAgain USING Stack, Nat OPNS
STRUCTURES: S-SA< Stack, Stack>
DUALS: pop -> Nat
TRANSFORMERS: push<Nat>, move
EQNS
S-SA(Stack1, Stack2).pop = S-SA(Stack1.pop, Stack2) AND Stack2..pop;
S-SA(Stack1, Stack2).push(Nat1) = S-SA(Stack1.push(Nat1), Stack2);
S-SA(Stack1, Stack2).move = S-SA(Stack1.pop, Stack2.push(Stack1..pop))
ENDCLASS (* StacksAgain *)
```

This `StacksAgain` specification is also well defined but written in an *impure* style. The `pop` operation is defined *impurely* — it uses the result of a dual operation on the second component without changing its state accordingly. In a *pure* object oriented style it is required that when dual operations are requested the accessor part of the attribute cannot be utilised without the transformer acting on the internal state of the serving object. In other words, although OO ACT ONE conceptually allows the two parts of a dual to be used separately, a *pure* specification style does not let an accessor part of a dual to appear in the right hand side of an equation definition without the new state of the serving component being updated to model the servicing of the whole dual. Note that such a check only makes sense in a class which is defined to have persistent components.

4.2.4.2 Object Oriented Interpretation of Dynamic Structure

It is not possible to extend these pure styling conventions to cope with objects which have dynamic structure. In particular, objects defined using recursive structure operations are difficult to reason about compositionally. For example an integer stack object is illustrated using an object structure diagram in figure 4.3.

In the same figure a new representation is introduced for more concise diagrammatic representation of recursive structures. The formal meaning of both diagrams is given by the OO ACT ONE `STRUCTURE` expression which is common to both.

The syntax of the new diagram is representative of the way in which linked list structures (common to all forms of programming) have an informal yet powerful means of representation (as nodes

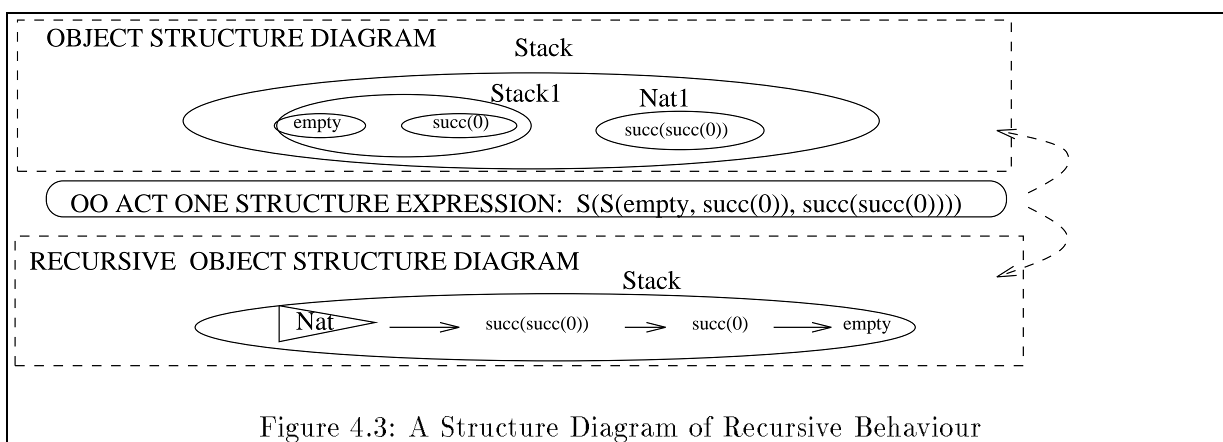

Figure 4.3: A Structure Diagram of Recursive Behaviour

and pointers). The recursive structure diagram emphasises that, although objects with recursive structure have a complex embedded structure, the consistent relationship between components and subcomponents (and subcomponents and their subsubcomponents, etc ...) means that such objects are often understood in a linear or tree like fashion.

Non-recursive Dynamic Structures

A non-recursive dynamic structure is specified in OO ACT ONE when the structure expression on the left hand side of an equation does not match the structure expression on the right hand side of the equation. This models an object (or objects) in a class which change their internal structural composition in response to a service request. Modelling such behaviour is very powerful but should be used only to represent special events in the life of an object. It should not be the normal means of defining behaviour, since if it is not done sparingly it can severly reduce the clarity of the intended meaning.

4.2.5 Shared Objects

In OO ACT ONE there is no notion of one object being shared between others. This thesis argues that such an idea is implementation oriented. It is not desirable for an analyst to worry about such things. A simple example, the `TwinFunction` class, illustrates this quite clearly.

The class structure diagram for this specification is given in figure 4.4.

In this example, a reasonable interpretation is that the two function components share the two queues[1]. For example, both function components **depend on** `Queue1`, and so `Queue1` can be implemented as a (persistent) object shared between `Function1` and `Function2`. This sharing interpretation is not an analysis issue. A designer could choose a solution architecture in which sharing is not evident.

[1]An alternative, but equally valid interpretation, is to say that the two queues share the two functions.

```
CLASS TwinFunction USING Function, Queue, Nat OPNS
STRUCTURES: S-TF<Queue, Queue, Function, Function>
DUALS: remove -> Nat
TRANSFORMERS: add<Nat>, process1, process2
EQNS
S-TF(Queue1,Queue2,Function1,Function2).remove =
S-TF(Queue1,Queue2.pop,Function1,Function2) AND Queue2..pop;
S-TF(Queue1, Queue2, Function1, Function2).add(Nat1) =
S-TF(Queue1.push(Nat1), Queue2, Function1, Function2);
S-TF(Queue1, Queue2, Function1, Function2).process1 =
S-TF(Queue1.pop, Queue2.push(Function1.in(Queue1..pop)), Function1, Function2);
S-TF(Queue1, Queue2, Function1, Function2).process2 =
S-TF(Queue1.push(Function2.in(Queue2...pop)), Queue2.pop, Function1, Function2)
ENDCLASS (* TwinFunction *)
```

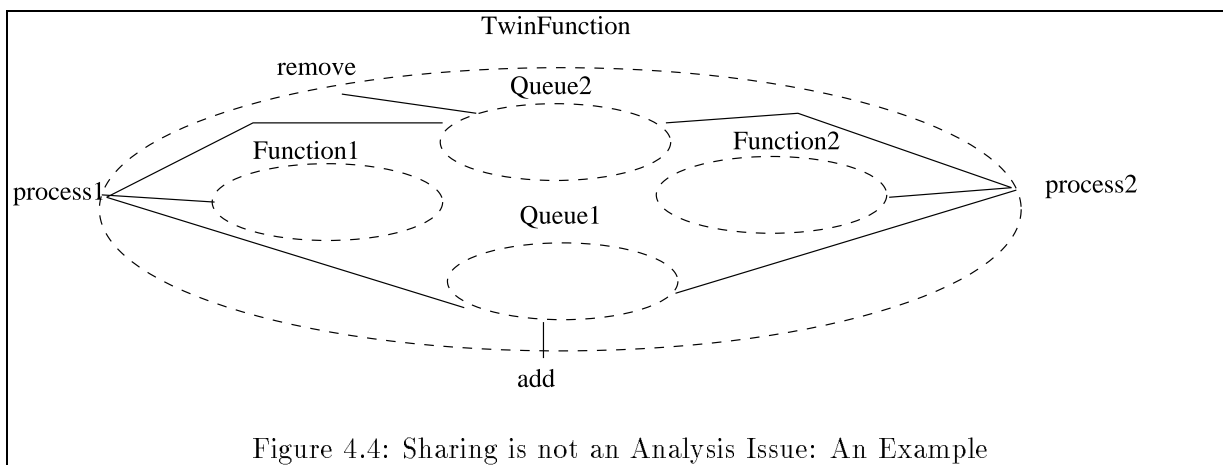

Figure 4.4: Sharing is not an Analysis Issue: An Example

4.2.6 Timing Properties

The formal specification of timing properties has been the basis of a wide range of research in recent years. Some semantic models are intended for general use (see [63], for example) whilst other semantics have been given for more specific formal models: LOTOS, for example, has a simple timing model for defining behaviour as a temporal ordering of events. There has been much work done to extend the LOTOS timing model (see [11], for example). It is useful to ask what sort of timing properties a formal object oriented requirements model should be able to exhibit, and whether OO ACT ONE is suitable for specifying these properties.

4.2.6.1 A Categorisation of Timing Properties

For discussion purposes, timing properties are categorised into four groups:

- **Event Sequencing**
  LOTOS exhibits such a timing model. The sort of behaviour that this model specifies is 'event A must occur before event B can occur'. In object oriented analysis, events correspond to service requests at the external interface of an object. Now, our object oriented semantics state that all

service requests in the external interface of an object can be serviced at any time during that object's lifetime. Consequently, external event sequencing properties are not relevant. Section 4.2.6.2 formulates an interpretation of internal event sequencing properties for structured objects specified in OO ACT ONE.

- **State Sequencing**
  State sequencing is logically equivalent to event sequencing. The sort of behaviour specified in state sequencing models is 'object A cannot be in state S before it has been in state T'. Such timing properties can be taken directly from the O-LSTS semantics for OO ACT ONE specifications. However, such behaviour properties are not explicit in the OO ACT ONE and can only be deduced after some appropriate analysis.
- **Synchronisation**
  One of the most common timing properties that is used in the specification of behaviour is synchronisation. For example, 'Object A must synchronise with object B on event C'. Other than for communication purposes, synchronisation constraints are predominantly design and implementation issues. It is beyond the scope of this work to extend the O-LSTS semantics to facilitate the explicit statement of such properties during object oriented analysis.
- **Quantitative Timing**
  We are not concerned in our object oriented semantic model with being able to specify timing properties with respect to some sort of global passing of time.

4.2.6.2 Interpretation of Timing Properties in an OO ACT ONE Specification

Two interpretations of internal timing properties are given for OO ACT ONE specifications of structured objects:

I) Ordering of Internal Service Requests

A static analysis of OO ACT ONE equation definitions is sufficient to identify 'timing properties' which are implicit in an object oriented requirements specification. For example, in `TwinFunction` (see 4.2.6) the equation:

```
S-TF(Queue1, Queue2, Function1, Function2).process1 =
S-TF(Queue1.pop, Queue2.push(Function1.in(Queue1..pop)), Function1, Function2);
```

can be given the following interpretation:

> When servicing a `process1` request, the `Queue1` component must have a `pop` serviced not after the `Function1` component has an `in` serviced, which in turn must then occur not after the `Queue2` component has a `push` serviced.

In an implementation, the component services cannot occur concurrently, since the result of one service is used as input to another. Therefore the phrase 'not after' can be read as 'before', when design and implementation issues arise. Designers can interpret the specification as saying that the internal `pop` occurs before the internal `in` which occurs before the internal `push`, which all result from one external `process1` request. This property is not an explicit part of the object oriented semantics

which do not specify how internal state transitions are achieved.

II) Synchronisation of Internal Requests

Consider a system of two stacks and one queue component. Informally, the behaviour of such a system is defined as follows. Natural numbers are pushed onto the queue. They can be moved synchronously to both the stacks and separately popped off either stack. This is precisely stated in the OO ACT ONE specification of class `QSS`.

```
CLASS QSS USING Queue, Stack, Nat OPNS
STRUCTURES: S-QSS<Queue, Stack, Stack>
DUALS: pop1 -> Nat, pop2 -> Nat
TRANSFORMERS: push<Nat>, move
EQNS S-QSS(Queue1, Stack1, Stack2).push(Nat) = S-QSS(Queue1.push(Nat), Stack1, Stack2);
S-QSS(Queue1, Stack1, Stack2).pop1 = S-QSS(Queue1, Stack1.pop, Stack2) AND Stack1..pop;
S-QSS(Queue1, Stack1, Stack2).pop2 = S-QSS(Queue1, Stack1, Stack2.pop) AND Stack2..pop;
S-QSS(Queue1, Stack1, Stack2).move =
S-QSS(Queue1.pop, Stack1.push(Queue1..pop), Stack2.push(Queue1..pop))
ENDCLASS (* QSS *)
```

The 'synchronisation' of the two `Stacks` on the `move` operation is represented by the **state label expression** `Queue1..pop` appearing twice in the **state label expression** on the right hand side of the `move` equation definition. The result of `Queue1` popping off a value is used by both the `Stacks`. In the OO ACT ONE specification there is no explicit statement that both `Stacks` synchronise. This synchronisation is one way the designer of a solution to the system can guarantee that the same value is given to both components: it is not the only solution.

4.2.6.3 Timing is a Design Concern

This section has shown that timing is not directly an analysis concern. It is necessary that an object oriented requirements model can be interpreted by designers and so an informal mechanism for extracting timing properties with respect to internal interaction between peer components has been formulated. However, OO ACT ONE does not contain explicit timing mechanisms. The timing aspects of an object oriented requirements model are deliberately abstracted away from.

4.3 Other Object Oriented Analysis Issues

4.3.1 Concurrency

In OO ACT ONE, the servicing of a request is defined as the evaluation of a **state label expression**. The subexpressions of a **state label expressions** may be evaluated independently, and this can be given a concurrent interpretation. Concurrency is examined in more detail in section 6.5.

4.3.2 Communication: Synchronous vs Asynchronous

All computing systems exhibit communication properties. There are fundamentally two different communication models: synchronous and synchronous. During object oriented analysis and requirements capture, it is important that the communication model is abstracted away from.

4.3.3 Exception Handling

`Unspecified` class members are used to model exceptions. When behaviour is required which cannot be modelled in the requirements without making implementation decisions then the analyst can chose to model this behaviour using an `unspecified` value. It is important that these exceptions are not dealt with prematurely by the analyst. The `unspecified` mechanism allows the analyst to abstract away from *how* exceptions are implemented to identify only *what* exceptions must be considered at later stages of development.

4.3.4 Nondeterminism and Probabilistic Behaviour

Nondeterminism is a powerful specification facility. Until now, the systems we have analysed have all been deterministic. The reason for ignoring nondeterministic behaviour until this stage is that it is not necessary to change the semantics to record this type of behaviour. Nondeterminism is recorded by commenting `TRANSFORMER` operations as being `INTERNAL`. An `INTERNAL` operation need not be initiated through the external interface of the object in question.

In the ACT ONE executable model, the `INTERNAL` transitions are treated no different from the others. However, in a concurrent model of the requirements, the `INTERNAL` transitions can occur independent of the environment in which the object in question is found. The commenting of transitions as `INTERNAL` keeps the semantics simple whilst ensuring that the designers are explicitly informed of the nondeterminism.

Nondeterminism is used in two ways:

- To model probabilistic behaviour.
- To specify implementation freedom.

Both these aspects are important in the analysis of a system.

4.3.4.1 Probabilistic Behaviour

Consider the `CoinToss` class, defined below.

```
CLASS CoinToss USING Bool OPNS
STRUCTURES: Coin<Bool>
ACCESSORS: Toss -> Bool
TRANSFORMERS: HorT<Bool> (*INTERNAL*)
EQNS CoinToss.HorT(Bool1) = Coin(Bool1); Coin(Bool1)..Toss = Bool1
ENDCLASS (* CoinToss *)
```

A valid implementation of this OO ACT ONE class can, for example: always respond `true` in response to a `Toss` request, or always respond `false`, or alternate between `true` and `false` responses (to name but three options). When nondeterminism is used to model random behaviour then the analyst must record the probabilisitic requirements outside the OO ACT ONE framework of specification. (It is beyond the scope of this thesis to examine how this can be done: [63] gives one particular view of probabilistic semantics which may be useful in this respect.)

4.3.4.2 Implementation Independence

Consider the behaviour specified by class `SysQSS`, below.

```
CLASS SysQSS USING Queue, Stack, Nat OPNS
STRUCTURES: SystemQSS<Queue, Stack, Stack, Bool> DUALS: pop1 -> Nat, pop2 -> Nat
TRANSFORMERS: push<Nat>, move, PickStack<Bool> (*INTERNAL*)
EQNS SystemQSS(Queue1,Stack1,Stack2,Bool1).push(Nat1) =
SystemQSS(Queue1.push(Nat1),Stack1,Stack2,Bool1);
SystemQSS(Queue1, Stack1, Stack2, Bool1).pop1 = SystemQSS(Queue1, Stack1.pop, Stack2, Bool1) AND
Stack1..pop;
SystemQSS(Queue1, Stack1, Stack2, Bool1).pop2 = SystemQSS(Queue1, Stack1, Stack2.pop, Bool1) AND
Stack2..pop;
Bool1 => SystemQSS(Queue1, Stack1, Stack2, Bool1).move = SystemQSS(Queue1.pop,
Stack1.push(Queue1..pop), Stack2, Bool1)
OTHERWISE SystemQSS(Queue1.pop, Stack1, Stack2.push(Queue1..pop), Bool1);
SystemQSS(Queue1,Stack1,Stack2,Bool1).PickStack<Bool2> = SystemQSS(Queue1,Stack1,Stack2,Bool2)
ENDCLASS (* SysQSS *)
```

This class models implementation freedom: when a `move` request is serviced, the implementer is free to decide how the system chooses which `Stack` component the `Queue` should transfer its data to.

4.3.5 Active and Passive Objects

In most object oriented systems it is common to distinguish between active and passive objects:

- An active object is likened to a process whose existence persists over a sequence of events.
- A passive object is likened to a piece of data which flows between active objects.

In this thesis, this potentially confusing distinction is not made: the active and passive concepts seem to have arisen from implementers attempting to conceptualise objects in non-object oriented terms.

When using OO ACT ONE, there is a more useful division than active and passive: static and dynamic. Static objects are those whose classes do not offer transformer attributes. Dynamic objects are those whose classes do offer transformer attributes. Static objects are important because they can be implemented as shared objects, without risk of the principle of encapsulation being broken.

4.3.6 Persistency

When an object oriented system is created, a component of the system is said to *persist* if it is identifiable throughout the lifetime of the system. When a transformer request is serviced, the new state of an object may be constructed from the same components. This interpretation can be taken when the OO ACT ONE specification is defined *purely* (see 4.2.4.2).

Dynamic objects which are *purely* defined can be implemented as a fixed set of components. History dependent behaviour is then realised by the set of persistent component objects changing their internal state through services delegated to them (by their containing object) at their external interfaces.

The notion of persistency is not explicit in the OO ACT ONE requirements model. We recommend that a need for persistency should be recorded in the informal parts of the requirements document. Then, a simple static analysis of the OO ACT ONE code can check to ensure that the class in question is *purely* defined. It is not an error if the specification is *impure*, but a warning message can be given to state that persistency cannot be guaranteed.

4.3.7 Class Routines: Configuration and Creation

Creation and configuration routines provide a means of restricting the set of initial states which an object of a specified class can attain. In OO ACT ONE, an `(* INITIAL *)` comment can be used to identify the literals and structures (and possible invariants on the structures) which can be used to define a newly created object. Such a comment can be used to stimulate a static analysis to check that all dynamically created objects are correctly initialised. (Such an analysis is beyond the scope of this thesis.)

4.4 Reviewing the OO ACT ONE Specification Language

4.4.1 Does It Meet Our Expressional Requirements?

The fundamental requirements for an object oriented analysis language are that it provides a means of recording relevant information which: increases problem understanding, is amenable to customer validation, and results in a formal requirements model useful to designers. OO ACT ONE fulfils all these requirements.

In section 2.2.3 we identified 'features of good analysis methods' as a list of criteria by which analysis methods can be judged. This section re-examines these criteria with respect to OO ACT ONE.

- **i) Amenability to change within a stable structure**
  The classes, objects and object compositions, within a system, provide a stable base upon which changes to requirements can be easily made.
- **ii) Encouragement of Re-Use**
  Three types of re-use are evident in OO ACT ONE:

  - **Compositional** — re-use of predefined components and structures.
  - **Categorisational** — re-use of behavioural characteristics as defined in a class hierarchy.
  - **Experience** — OO ACT ONE is simple and straightforward since it is built around a small set of well understood concepts and precisely defined mechanisms. Consequently, this thesis argues that analysts can quickly gain experience in developing technique founded on experience, rather than learning the underlying model, when coding in OO ACT ONE.

- **iii) Interfacing between customers and designers**

  The 5-model approach to specification is prominent in OO ACT ONE. This approach is *customer oriented* in the sense that it arose from a model of customer understanding and is amenable to customer validation. The diagrammatic notations are accessible to customers, analysts and designers alike, whilst the underlying mathematical model provides the formal basis upon which one coherent framework of understanding is built.

- **iv) Incorporating Standard Modelling Techniques**

  We consider five standard modelling practices to be inherent in OO ACT ONE specifications:

  - Abstraction (and encapsulation) are fundamental to the object oriented model.
  - (De)composition is provided by `STRUCTURE` operations
  - Class hierarchy constructs are provided by explicit subclassing mechanisms.
  - Communication between objects is modelled as one object requesting a service of another. The service is requested so that the **client** can utilise the encapsulated behaviour offered by the **server**. The information that flows between the objects is defined by the input and output parameters.
  - Model co-ordination is prominent in an OO ACT ONE specification, where there are many different views of the behaviour defined, which combine in a consistent and coherent fashion.

- **v) Having a formal basis**

  The OO ACT ONE language is rigorously defined using the O-LSTS semantics.

4.4.2 Is OO ACT ONE Purely an Analysis Language?

Sections 4.2 and 4.3 emphasise the number of object oriented issues which are abstracted away from when using OO ACT ONE during analysis and requirements capture. The most design-like features of OO ACT ONE specifications are the `STRUCTURE` operations. The composition relationship is a characteristic which arises from problem domain structure, rather than from some arbitrary structure imposed by the analysts to aid their understanding. This thesis argues that it is necessary for such structure to be prominent in a set of object oriented requirements.

4.5 The Practicalities of Building a Formal Model

There are three aspects of analysis which group together under the heading ‘practicalities’:

- **Models:** syntax and semantics.
- **Development Method:** how to generate, test and change the models to achieve the best requirements model.
- **Tools:** automated mechanisms for support of model building, execution, validation and verification

Chapters 2 and 3 have developed a number of different analysis models. Section 4.5.1 defines a method[2] (called the *skeleton* approach) for the development of OO ACT ONE requirements models.

4.5.1 The Skeleton Method to Object Oriented Analysis

The whole *skeleton* strategy is based around the application of seven different processes, each of which can be applied at any time during the analysis and, with careful control, can often be applied in parallel. These processes are shown in figure 4.5.

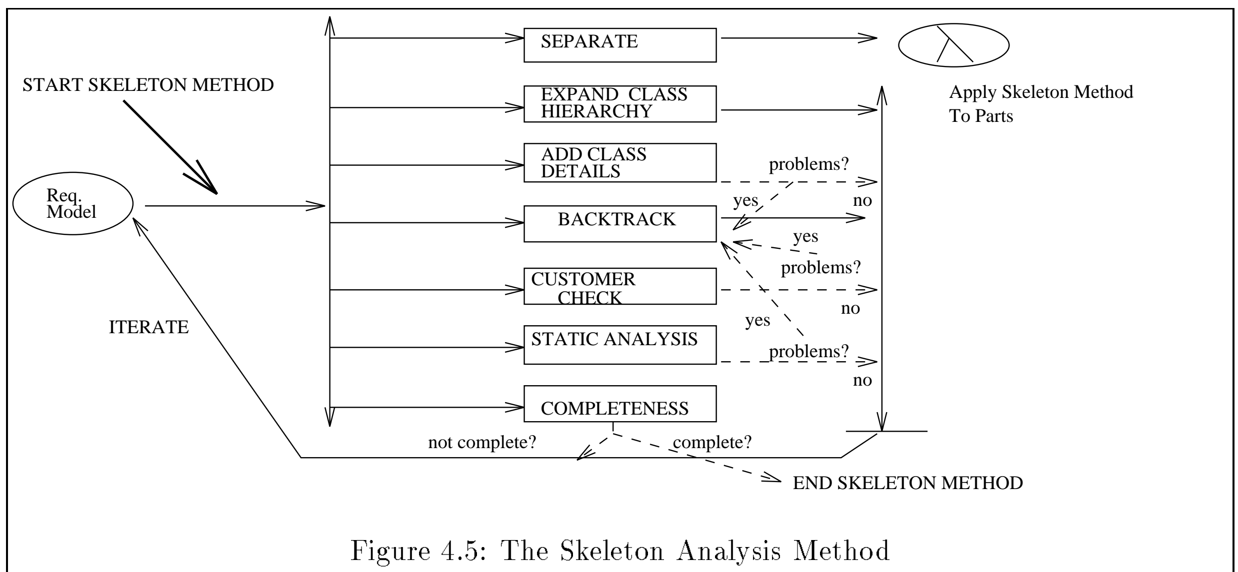

Figure 4.5: The Skeleton Analysis Method

4.5.1.1 The Opportunistic Algorithm

The following simple algorithm provides a framework upon which a complex analysis method can evolve.

BEGIN Skeleton Method
REPEAT
Choose a process (or processes) from **(i) to (vii)**

- **(i) Separation**
  Recognise distinct and separate parts of the system and apply the skeleton approach to each of these. Reconnect the parts after their completion.

[2]The method is no more than a simple algorithm which can be followed when learning how to use OO ACT ONE. A proper analysis method should evolve from this very primitive starting point.

- **(ii) Expand Class Hierarchy**: either
  - Identify a new class in the system and add to hierarchy without defining it as a subclass or superclass of any other classes in the hierarchy.
  - Identify a new class explicitly as a (sub/super)class of an already identified class.
  - Change a class which is separate from the other classes by explicitly defining it as a (super/sub)class of an already identified class.
- **(iii) Add Details to some class in the hierarchy**: either
  - Identify literal or structure members.
  - Identify invariant properties.
  - Specify an external attribute of a class as an accessor, transformer or dual operation.
  - Specify the behaviour associated with an external attribute in an equation definition (provided it has not been previously defined).
- **(iv) Backtrack**: either
  - Disconnect a (sub/super)class link in the hierarchy.
  - Delete a class which is no longer needed (perhaps because it has been superseded by another).
  - For some class in the hierarchy, remove literal or structure operations.
  - Delete an external attribute.
  - Delete an equation associated with an external attribute.
- **(v) Check with the customer**: either
  - Execute, using the ACT ONE dynamic model, one or more of the classes whose behaviour is complete. In other words, validate customer expectations of the dynamic behaviour of the system (or a component of the system) being analysed.
  - Present the customer with graphical views of the OO ACT ONE code: class hierarchies, structure diagrams and O-LSTSD(s). This helps to check analyst understanding against customer understanding of the problem domain.

  Then, if the checks have identified 'problems' these must be noted and resolved (either by **backtracking** or by adding the extra understanding gained from customer interaction). Otherwise, the classes checked are noted as being validated and marked as such in the class hierarchy.
- **(vi) Statically Analyse Requirements Specification**
  The whole of the system or distinct parts (classes) can be checked for correctness using the static analysis defined in section 3.4. After a successful analysis, the ACT ONE execution model can be used by the analyst alone to test their understanding of the specification. Problems identified during the static analysis stage or the execution stage must be noted and corrected in the model.
- **(vii) Check The Analysis For Completion**
  The following tests must be made before the analysis can be declared complete:
  - Check that all customer classes (i.e. those classes identified by the customer in the problem domain) have been specified in the class hierarchy.
  - Statically Analyse the whole system (i.e. all the separate class trees) for correctness properties.
  - Validate that the customer is happy with the dynamic behaviour exhibited by every class in the specification (this should be done as the specification is developed so that the final check is straightforward) and the relationships between them. In particular check the classification, composition and configuration properties.

UNTIL analysis is checked as complete
END Skeleton Method

It is not practical for every single change to the analysis model to be formally tested, or validated by the customer, as they are made. It is up to the analysis manager to decide the *best* strategy for making intermediate checks.

During application of the *skeleton* approach, different parts of the system will be better understood and more completely specified than others. As analysis proceeds a skeleton class hierarchy gradually appears and this skeleton is held together by the classification and composition relationships. The meat on the bones is provided by the operation and equation definitions in each class.

Central to the skeleton method is some means of guaranteeing that one class of behaviour is not specified many times as a result of splitting the system up into separate analysis parts. Further, there must be some standard way of re-using already existing components, and for making newly defined component classes available for re-use.

4.5.1.2 Re-use and Costing

Project managers must take into account the need for different costing strategies when re-use is prominent in a development method. In an ideal environment one could pay for predefined components with the formal specification acting as some sort of contract between vender and consumer. However, it is more likely that companies will develop their own libraries of classes for use in-house. Software re-use is appealing in principle but does lead to many difficult questions in practice. It is beyond the scope of this thesis to examine the consequences of re-use in the environment in which the development is taking place.

4.5.1.3 Re-use: A Note On Libraries

In a proper object oriented development environment pre-defined classes of behaviour should be as easy to re-use as the application of any other language mechanism. The library classes should be looked upon as part of the semantics of the language. Object oriented programming languages are constructed from three parts:

- The language primitives.
- The re-use mechanisms.
- The libraries of re-usable components.

It is these three things together which combine to produce an object oriented language. Most object oriented languages come already with library class hierarchies to provide behaviour comparable to that provided as language primitives in larger imperative languages. Object orientation places more responsibility on the user to control the library facilities. Unfortunately, OO ACT ONE does not presently have a large set of library classes.

4.5.2 Validation

Other than the static analysis of an object oriented specification the only other means of testing the requirements model is by running event sequences on the ACT ONE executable model. The event traces produced (in response to a sequence of evaluations) must be validated against customer and/or analyst understanding of the requirements. In the execution of some behaviour three types of result occur:

- 1) The ACT ONE expression is evaluated (according to the well-defined re-write rules) to represent a value which has a valid object oriented interpretation:
  - When the operation is a transformer, the expression evaluates to a member of the **server class** represented as a literal a structure expression of the **server class** sort.
  - When the operation is a dual, or an accessor, the expression evaluates to a dual expression of the **server class**
- 2) The ACT ONE expression evaluates to an **unspecified** literal value of the appropriate class. This corresponds to behaviour which must be determined at a later stage in the development.
- 3) The expression does not evaluate to either of the cases above. This occurs only when the dynamic behaviour of an object results in an invariant property being broken. It is not possible to guarantee that all invariants are upheld during the lifetime of a class without severly restricting their use. For example, invariants could be permitted only in static classes (i.e those without transformer attributes). However, this restriction limits the power of expression in OO ACT ONE. It is up to the analysts to test the correctness of their specifications with respect to invariants. Consequently, the static analysis of OO ACT ONE warns the analyst when an invariant is used in a dynamic class. Analysts must prove that invariants are never broken otherwise correct behaviour cannot be guaranteed.

There are many different ways, in theory, that invariant properties can be guaranteed. This section identifies two solutions that work in practice.

- **Solution 1: Test Initial States and Validate Transitions**
  Given a list of initial states that an object in a particular class can be initialised to, it is necessary to first check that these states fulfil the invariants. Once this is done, it is necessary only to check that all invariant properties are upheld across the state transitions defined in the transformer and dual equations.
- **Solution 2: Ignore Transformers Which Result In Broken Invariant Properties**
  All transformer operations are defined to result in no change of state when an invariant would be broken as a consequence of their being fulfilled in the normal way. Dual operations are similarly defined, with the additional property that the unspecified value of the appropriate class is returned as the result of an operation which results in a broken invariant. This convention guarantees that the state of an object always fulfils the invariants defined on it.

There are certainly *better* approaches to dealing with invariant validation but this is a general problem which was not examined in this thesis.

4.5.3 Tools

This thesis has presented a set of models for developing object oriented requirements specifications and a method for utilising these models. However, the thesis does not present any tools for aiding this process. Clearly, tool support is vital to all development processes. Rather than creating these tools, this thesis identifies the type of tools which can and should be developed as future work:

- A customer friendly, animation tool, is required. This tool can be supported by the ACT ONE evaluation mechanism.
- A tool to aid the analyst in the synthesis of OO ACT ONE specifications is required. In particular, some sort of library browsing facility is required whereby components can be cut and pasted in and out of a system hierarchy.
- There is a need for tool support in the areas of static analysis and verification.
- Finally, a range of tools are required to help managers get the best out of their resources when developing large requirements models.

4.5.4 Analysis Style: High Level Decisions

OO ACT ONE is still in its infancy. However, already the research has identifed different ways of applying the *skeleton* method, which collectively can be considered to define different object oriented analysis styles. These styles are characterised by a sequence of high level decisions which are taken when the *skeleton* method is applied.

The analyst's job is to achieve a mutual understanding of problem domain structure with the customer and to record it in a meaningful way. Analysts can influence the representation of problem domain structure through interaction with the customer.

4.5.4.1 Achieving a Mutual Understanding of the Problem Domain

The structure of the problem domain should never be compromised to make the recording of the requirements suit the analyst. An analyst, on identification of a complex compositional problem domain, may suggest a better way for customers to structure their understanding. This better representation may reflect a simplification, but if the customer does not agree with the suggested (de)composition then no changes should be made. Problems arise only when customers view a problem in a very convoluted way and therefore make it difficult to express their requirements in a comprehensible fashion. In such a case the analysts must 'educate' the customer and attempt to relay a better understanding of the problem domain (if there is one). When the customer and analyst cannot find a mutually agreeable way of understanding the problem then there is no simple way to express the requirements. Requirements capture is not complete until both customer and analyst are sure that they have a common understanding of the specification produced.

Analysts are free to chose any means of capturing customer requirements. Within the *skeleton* approach there is enough analyst freedom to give rise to specification styles. These styles are predominantly related to achieving a balance between:

- Inheritance and delegation.
- Re-use and coding from scratch.
- Extending class hierarchies and redefining existing classes.
- Deep-and-narrow and shallow-and-wide structures.
- Bottom-up and top-down development.

4.5.4.2 Inheritance and Delegation

Although subclassing and composition are very different concepts, it is common within object oriented analysis to confuse the two notions. It is the analyst's job to remove this confusion. In the process of recording requirements an analyst is often faced with the choice of specifying new behaviour by inheriting from an already existing class, or by delegating tasks to an already existing class, as a component of the new class. Clearly, if the customer can distinguish between these two mechanisms of re-use, and can identify which mechanism is appropriate in each case, then there is no problem. However, when a customer is unsure it is the analyst who choses the solution.

4.5.4.3 Re-Use and New Class Production

When behaviour is required that is 'similar to' or 'related to' an already existing class, the temptation is to work around the re-usable component. This can lead to additional problems if the work needed to include the predefined component is more than the work which would have been needed to generate the required behaviour from scratch. We recommend that existing classes be re-used only in two cases:

- the component to be re-used is identified as being a (sub/super)class of another which is already part of the specification
- the class to be re-used has been identifed as a structure component of a class already in the specification

This is a conservative form of re-use since we do not advocate adapting components to fulfil ones needs. A less conservative (adaptive) style is one in which nearly all OO ACT ONE code is produced from already existing components. In other words, pick a class as close to what is required as is possible and adapt it until it suits. Analysts can develop a style which reflects their attitude to re-use.

4.5.4.4 Class Changes and New Subclasses

A high level decision is often required when deciding whether to define a new class as a subclass of an already existing class or to directly extend the existing class. Each choice has its advantages and disadvantages:

- i) Using the subclass relationship defines an explicit relationship which is guaranteed by the subclassing mechanism; but it extends the class hierarchy which, in general, should be as simple and uncluttered as possible.

- ii) Changing the already existing class keeps the class hierarchy simple but there are no automatic guarantees that the class maintains its old behaviour. Thus, the change may affect other clients of the class (those classes which inherit behaviour from it or use it compositionally).

Adopting the first choice has many consequences for re-use and implementation. Large complex hierarchies make re-use much more difficult since it makes it harder to find the behaviour which one requires. Also, implementing such hierarchies often has major overheads [101].

4.5.4.5 Structures: Deep and Narrow or Shallow and Wide?

Structure appears in the OO ACT ONE composition and classification hierarchies.

I) Composition

OO ACT ONE `STRUCTURES` define the composition properties of groups of objects whose components are members of the same class. There are two types of style for simplifying the recording and presentation of compositional information:

- **Flat and Wide Compositions:**
  The number of levels of (de)composition is kept to a minimum.
- **Deep and Narrow Compositions:**
  The number of components in each `STRUCTURE` is kept to a minimum.

As with all analysis problems, the exact style of specification depends on the customer and their problem domain. In all cases, independent of style, OO ACT ONE compositions should never be too wide or too deep.

II) Classification Structure

Structure in the class hierarchy is amenable to the same kind of reasoning as the compositional structure. To simplify understanding of a class hierarchy it is recommended that a class should not have too many direct subclasses (children) or direct superclasses (parents). Further, a class should not be too many subclass relationships away from its root superclass.

In an OO ACT ONE specification a large number of separate class trees commonly make up a system class hierarchy. There is no benefit, in our experience, of restricting the number of trees in each case. In fact, there are many advantages in having many smaller disconnected trees rather than connecting together classes into larger trees of class relationships.

As with the compositional structure, it is up to the analyst to interact with the customer to achieve the best mutual understanding of the system being analysed. The analyst's own particular style of seeing the problem is sure to influence the way in which the customer views the classification requirements.

4.5.4.6 Bottom-up vs Top-down

Object oriented analysis is both bottom-up and top-down. Already existing classes are synthesised into the requirements model, whilst parts of the problem domain which are not well understood are deconstructed in an attempt to improve understanding.

An object oriented analysis technique must be flexible so that problem domains which are well understood can be analysed compositionally whilst those less well understood can be analysed decompositionally. A certain style of specification will result when an analyst is involved in the same sort of problems over and over again. Certain components will be used repeatedly and the way they are incorporated in the new specification will be standardised by a particular analyst. Contrastingly, analysts who face a wide range of problems will build up a style of specification built around method rather than components. These analysts will, with experience, acquire general techniques.

4.5.5 General Analysis Principles

The following is a list of principles which analysts should adhere to:

- The customer is central to analysis. The requirements model must be customer led.
- Keep things simple.
- Specify for re-use, and re-use pre-defined components wherever practical.
- Test any changes to the specification which alter customer and/or analyst understanding.
- Comment all non-functional requirements:
  - Unspecified behaviour and the reason for its appearance.
  - Nondeterministic behaviour.
  - Persistency within a pure OO ACT ONE specification.
- Record all analysis decisions which re-structure customer understanding, together with customer reaction to such changes.
- Never rush analysis: it is the most important part of software development.

4.6 FOOA and Object Oriented Design

4.6.1 Importance of Structure

Explicit structure does not make a specification into a design. The structures in an operational specification are independent of specific resources in an implementation environment, whereas designs actually refer to specific resource allocation in a final implementation. Furthermore, the identification of the structures, and the interaction and relation between them, is problem domain dependent: they are chosen for the way in which they model customer understanding.

This thesis argues that it is impossible within an object oriented framework to acquire problem domain understanding of requirements free of structural bias. Even if this was possible, it would

be very difficult to specify them formally since all formalisms introduce internal structure to decompose complexity[3]. Structure is the only solution to capturing the requirements of complex systems. The structure explicit in an OO ACT ONE specification is not a necessary part of the design or implementation, but it does act as a good basis upon which a solution can be developed.

4.6.2 Executable Models

Executable formal requirements models are easier to communicate with the customer. Executable specifications make rapid prototyping straightforward and automatic. They improve the process of customer validation by allowing walk throughs of dynamic system behaviour. Test traces have their limitations but program proving, the other main approach to validation, has other well known inadequacies too [3, 8]. Executable specifications are necessary, at least until the state-of-the-art in proving specification correctness is further developed. From a project manager's point of view, executable models provide additional advantages with respect to early results and accountability. An executable specification is a clear statement of the progress being made during analysis.

4.6.3 Constructive vs Unconstructive Specifications

The debate for and against constructive requirements models has been proceeding in a wide range of publications without any agreement between the two factions. The non-executable advocates concede the difficulties of relating specifications to a customer, but argue that better solutions should be sought other than presenting the customer with an executable model (see [64], for example). It is argued that, in general, a specification written in a notation that is not directly executable contains less implementation detail than an executable model. Matching such a specification to user requirements is, it is claimed, more straightforward since there are no additonal algorithmic details necessary for executability[4]. It is also claimed that executable specifications unnaturally hinder designers by constraining the possible choice of implementations [78]. This thesis argues that it is useful to provide a concrete structure in which designers can begin their work.

This thesis takes an executable approach to requirements capture. It is the only approach, currently, of making a formal object oriented analysis model which is amenable to customer validation. Executing a requirements model is a vital part of customer validation [131, 107].

4.6.4 Design and Design Transformations: A Preview

The structure of an OO ACT ONE specification is problem oriented, not necessarily implementation based. During design, the specification is subjected to transformations which preserve external behaviour characteristics but alter or extend the internal structure to yield an implementation oriented

[3]Axiomatic methods do not appear to bias structure in as extreme a manner as other specification methods, but these have proved to be limiting in the type of behavioural characteristics they can define.

[4]In the object oriented paradigm, this argument seems less convincing than when applied to traditional approaches to software development.

architecture for the system being modelled. Much work has been carried out in developing and automating these *correctness preserving transformations* (CPTs). The work by Partsch [96] argues the case for CPTs in some detail. In chapter 5, we define a set of CPTs which work on full LOTOS specifications (with the ACT ONE as it is generated from the OO ACT ONE requirements model). We develop only a small set of CPTs, but these suffice to illustrate the general principles.

Using CPTs means that problem domain structure need not be compromised in a requirements model. The initial requirements can be optimized for clarity, re-use, maintainability and, above all, customer accessibility. Design transformations manipulate this structure to achieve implementation ideals based on efficiency, use of available resources and current programming practices. For an object oriented formal development approach to become widely used, CPTs must become standardised elements of design, and tool support must be provided to control the sequential application of such transformations.